

Manuel Clavel
Francisco Durán
Steven Eker
Santiago Escobar
Patrick Lincoln
Narciso Martí-Oliet
José Meseguer
Carolyn Talcott

All About Maude

A High-Performance Logical Framework

Springer

Dedicated to the memory of
JOSEPH A. GOGUEN

Preface

This book gives a comprehensive account of Maude, a language and system based on rewriting logic. Many examples are used throughout the book to illustrate the main ideas and features of Maude, and its many possible uses. Maude modules are rewrite theories. Computation with such modules is efficient deduction by rewriting. Because of its logical basis and its initial model semantics, a Maude module defines a precise mathematical model. This means that Maude and its formal tool environment can be used in three, mutually reinforcing ways:

- as a declarative programming language;
- as an executable formal specification language; and
- as a formal verification system.

Maude’s rewriting logic is simple, yet very expressive. This gives Maude good representational capabilities as a *semantic framework* to formally represent a wide range of systems, including models of concurrency, distributed algorithms, network protocols, semantics of programming languages, and models of cell biology. Rewriting logic is also an expressive *universal logic*, making Maude a flexible *logical framework* in which many different logics and inference systems can be represented and mechanized. This makes Maude a useful *metatool* to build many other tools, including those in its own formal tool environment. Thanks to the logic’s simplicity and the use of advanced semi-compilation techniques, Maude has a high-performance implementation, making it competitive with other declarative programming languages.

The introduction (Chapter 1) gives a high-level overview of Maude’s main concepts and underlying philosophy, and of its various applications. Since this book gives a very complete account of Maude in its various aspects, Section 1.7 gives specific suggestions on different reading “paths” within the book, that can be chosen depending on the degree of prior familiarity with the main ideas and the various uses intended, for example, as a programming language or as a formal specification and verification tool.

© Maude 2 is copyright 1997-2015 SRI International, Menlo Park, CA 94025, USA.

The Maude system is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version. The Maude system is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose. See the GNU General Public License for more details.

Acknowledgements

As explicitly indicated in the heading of the appropriate chapters, several colleagues, including Christiano Braga, Azadeh Farzan, Joe Hendrix, Peter Ölveczky, Miguel Palomino, Adrián Riesco, Ralf Sasse, Mark-Oliver Stehr, and Alberto Verdejo, have contributed in a substantial way to this book project by developing illuminating tutorial examples, and/or by providing summary accounts of various Maude-based formal tools. We thank all of them most warmly for their important contributions.

Languages are living organisms. The lifeblood provided by experienced users is key to their growth and their improvement. We have benefited much from colleagues who have used different alpha versions of Maude; we cannot mention them all, but Christiano Braga, Feng Chen, Grit Denker, Azadeh Farzan, Joe Hendrix, Merrill Knapp, Nirman Kumar, Peter Ölveczky, Miguel Palomino, Adrián Riesco, Dilia Rodriguez, Grigore Roşu, Ralf Sasse, Koushik Sen, Ambarish Sridharanarayanan, Mark-Oliver Stehr, Prasanna Thati, and Alberto Verdejo deserve special thanks for their creative uses of Maude and their suggestions for improving the language.

This book project has gone through many drafts as new features were added to Maude and new chapters and examples were added to the book. We have benefited much from valuable comments and constructive criticism provided by several colleagues at different stages, including in some cases detailed comments to the latest drafts when the book was nearing its completion. For all these comments we cordially thank Christiano Braga, Peter Mosses, Peter Ölveczky, Miguel Palomino, Sylvan Pinsky, Isabel Pita, Adrián Riesco, Dilia Rodriguez, Manuel Roldán, Mark-Oliver Stehr, Antonio Vallecillo, and Alberto Verdejo.

We are grateful to José Quesada, who developed MSCP, the Maude parser. MSCP is implemented using SCP [309] as the formal kernel, and provides a basis for flexible syntax definition, and an efficient treatment of what might be called *syntactic reflection*.

Maude's historical precursor is the OBJ3 language [174]. The OBJ3 experience has greatly influenced the Maude design and philosophy, and we are

grateful to all our former OBJ colleagues for this. Joseph Goguen, to whose memory this book is dedicated, should be mentioned in particular, because of his enormous influence in all aspects of OBJ; and Tim Winkler for having implemented a state-of-the-art OBJ3 system with such great skill.

Two other rewriting logic languages, ELAN [27] and CafeOBJ [165], have provided a rich stimulus to the design of Maude. Although our language design solutions have often been different, we have all been wrestling with a similar problem: how to best obtain efficient language implementations of rewriting-based languages. We have benefited much from the ELAN and CafeOBJ experience, and from many discussions with their main designers and implementers: Claude and Hélène Kirchner, Marian Vittek, Pierre-Etienne Moreau, Kokichi Futatsugi, Răzvan Diaconescu, Ataru Nakagawa, Toshimi Sawada, and Makoto Ishisone.

Bringing a new language design to maturity requires a long-term research effort and substantial resources. Perhaps the longest, most sustained support has come from the US Office for Naval Research (ONR) through a series of contracts and grants. We are most grateful to Dr. Ralph Wachter at ONR for his continued encouragement at every step of the way. The US Defense Advance Research Projects Agency (DARPA), the US National Science Foundation (NSF), Japan's Information Technology Promotion Agency, the Spanish Ministry for Education and Science (MEC), and the Comunidad Autónoma de Madrid have also contributed important resources to the development of Maude, its foundations, and its applications. In particular, the latest language developments and applications, and the final phases of the book project, have been partially supported by: NSF grant CCR-023446; ONR grant N00014-02-1-0715; the Spanish Ministry for Education and Science under grants for projects: MIDAS (TIC 2003-01000), SELF (TIN 2004-07943-C04-01), and Desarrollo de software para sistemas distribuidos P2P (TIN 2005-09405-C02-01); and funding from the Comunidad Autónoma de Madrid for project PROMESAS (S-0505/TIC/0407).

Finally, we also thank the people at Springer; in particular, Alfred Hofmann as executive editor for his encouragement to publish this book since its early stages, and Frank Holzwarth for his very helpful technical advice during the book's preparation.

Contents

1	Introduction	1
1.1	Simplicity, expressiveness, and performance	1
1.1.1	Simplicity	2
1.1.2	Expressiveness	5
1.1.3	Performance	9
1.2	The logical foundations of Maude.....	11
1.3	Programming, specification, and verification	14
1.4	A high-performance logical framework.....	17
1.5	Core Maude vs. Full Maude.....	20
1.6	Book structure	21
1.7	How to read this book	23

Part I Core Maude

2	Using Maude	33
2.1	Getting Maude	33
2.2	Running Maude	33
2.3	Getting support and more information	37
2.4	Reporting bugs in Maude.....	38
3	Syntax and Basic Parsing	41
3.1	Identifiers	41
3.2	Modules	42
3.3	Sorts and subsorts	43
3.4	Operator declarations	46
3.5	Kinds	48
3.6	Operator overloading	50
3.7	Variables	51
3.8	Terms and preregularity	51
3.9	Parsing	53

3.9.1	Default precedence values	55
3.9.2	Default gathering patterns	56
3.9.3	The extended signature of a module	57
3.9.4	Parsing examples	58
4	Functional Modules	63
4.1	Unconditional equations	64
4.2	Unconditional memberships	65
4.3	Conditional equations and memberships	66
4.4	Operator attributes	70
4.4.1	Equational attributes	70
4.4.2	The <code>iter</code> attribute	73
4.4.3	Constructors	73
4.4.4	Polymorphic operators	77
4.4.5	Format	78
4.4.6	Ditto	81
4.4.7	Operator evaluation strategies	82
4.4.8	Memo	86
4.4.9	Frozen arguments	89
4.4.10	Special	90
4.5	Statement attributes	91
4.5.1	Labels	91
4.5.2	Metadata	91
4.5.3	Nonexec	92
4.5.4	Otherwise	92
4.5.5	Print	97
4.6	Admissible functional modules	98
4.7	Matching and equational simplification	99
4.8	More on matching and simplification modulo	103
4.9	The <code>reduce</code> , <code>match</code> , <code>trace</code> , and <code>show</code> commands	109
4.10	A number hierarchy	114
4.11	Partial operations, subsorting, and errors	118
5	A Hierarchy of Data Types: From Trees to Sets	123
5.1	Nonempty binary trees	124
5.2	Nonempty lists	126
5.3	Lists	127
5.4	Multisets	128
5.5	Sets	130
5.6	Idempotent semigroups	132

6	System Modules	135
6.1	Unconditional rules	136
6.2	Conditional rules	138
6.3	Admissible system modules	139
6.4	The <code>rewrite</code> , <code>frewrite</code> , and <code>search</code> commands.....	143
6.4.1	The <code>rewrite</code> command	143
6.4.2	The <code>frewrite</code> command	146
6.4.3	The <code>search</code> command	147
6.5	More examples of system modules	152
6.5.1	Petri nets	153
6.5.2	Blocks world.....	158
7	Playing with Maude	163
7.1	Writing on a blackboard.....	164
7.2	The hopping rabbits game	167
7.3	The Josephus problem	169
7.4	The three basins puzzle	171
7.5	Crossing the bridge	173
7.6	The looping chips game	176
7.7	The Khun Phan puzzle.....	177
7.8	Crossing the river	180
7.9	Dominoes on a chessboard	184
7.10	Black or white	186
7.11	The game is not over	187
8	Module Operations	189
8.1	Module importation	190
8.1.1	Protecting	191
8.1.2	Extending	192
8.1.3	Including	193
8.1.4	Default conventions in module importations	194
8.1.5	Some module hierarchy examples	195
8.2	Module summation and renaming	197
8.2.1	The summation module expression	197
8.2.2	Module renaming.....	198
8.3	Parameterized programming	202
8.3.1	Theories	202
8.3.2	Views.....	208
8.3.3	Parameterized modules.....	214
8.3.4	Module instantiation.....	218
8.3.5	A specification of sorted lists.....	226
8.3.6	The lambda calculus	228

9	Predefined Data Modules	235
9.1	Boolean values	236
9.2	Natural numbers	241
9.3	Random numbers and counters	245
9.4	Integer numbers	248
9.5	Machine integers	252
9.6	Rational numbers	255
9.7	Floating-point numbers	260
9.8	Strings	264
9.9	String and number conversions	268
9.10	Quoted identifiers	270
9.11	Basic theories and standard views	271
	9.11.1 TRIV	272
	9.11.2 DEFAULT	273
	9.11.3 STRICT-WEAK-ORDER and STRICT-TOTAL-ORDER	274
	9.11.4 TOTAL-PREORDER and TOTAL-ORDER	276
9.12	Containers: lists and sets	278
	9.12.1 Lists	278
	9.12.2 Sets	281
	9.12.3 Relating lists and sets	284
	9.12.4 Generalized lists	285
	9.12.5 Generalized sets	288
	9.12.6 Sortable lists	291
	9.12.7 Making lists out of sets	298
9.13	Maps and arrays	300
	9.13.1 Maps	301
	9.13.2 Arrays	303
9.14	A linear Diophantine equation solver	305
10	Specifying Parameterized Data Structures in Maude	311
10.1	Stacks	312
10.2	Queues	314
10.3	Priority queues	315
10.4	Lists	317
10.5	Sorted lists	319
10.6	Multisets	322
10.7	Binary trees	324
10.8	General trees	326
10.9	Binary search trees	328
10.10	AVL trees	333
10.11	2-3-4 trees	337
10.12	Red-black trees	340
10.13	Efficiency concerns	342

11 Object-Based Programming	343
11.1 Configurations	344
11.2 Object-message fair rewriting	355
11.3 Example: data agents	357
11.4 External objects	365
11.4.1 Sockets	365
11.4.2 Buffered sockets	373
12 Model Checking Invariants Through Search	377
12.1 Invariants	377
12.2 Model checking of invariants	378
12.3 Bounded model checking of invariants	382
12.4 Verifying infinite-state systems through abstractions	384
13 LTL Model Checking	389
13.1 LTL formulas and the LTL module	389
13.2 Associating Kripke structures to rewrite theories	391
13.3 LTL model checking	396
13.4 Equational abstractions	403
13.5 The LTL satisfiability and tautology checker	412
13.6 Verifying LTL properties of imperative concurrent programs ..	414
13.6.1 The semantics of a simple parallel language	414
13.6.2 Model checking Dekker’s algorithm	416
13.7 Crossing the river (revisited)	420
13.8 Other model-checking examples	422
14 Reflection, Metalevel Computation, and Strategies	423
14.1 Reflection and metalevel computation	423
14.2 The META-TERM module	425
14.2.1 Metarepresenting sorts and kinds	425
14.2.2 Metarepresenting terms	426
14.3 The META-MODULE module: Metarepresenting modules	427
14.4 The META-VIEW module: Metarepresenting views	432
14.5 The META-LEVEL module: Metalevel operations	433
14.5.1 Moving between reflection levels: <code>upModule</code> , <code>upTerm</code> , <code>downTerm</code> , and others	434
14.5.2 Simplifying: <code>metaReduce</code> and <code>metaNormalize</code>	438
14.5.3 Rewriting: <code>metaRewrite</code> and <code>metaFrewrite</code>	440
14.5.4 Applying rules: <code>metaApply</code> and <code>metaXapply</code>	441
14.5.5 Matching: <code>metaMatch</code> and <code>metaXmatch</code>	445
14.5.6 Searching: <code>metaSearch</code> and <code>metaSearchPath</code>	448
14.5.7 Parsing and pretty-printing: <code>metaParse</code> and <code>metaPrettyPrint</code>	450
14.5.8 Sort operations	453
14.5.9 Other metalevel operations: <code>wellFormed</code>	460

14.6	Internal strategies	461
15	Unification and Variant Generation	465
15.1	Introduction	465
15.2	Order-sorted unification modulo a set of axioms	466
15.3	Theories currently supported	468
15.4	The <code>unify</code> command	469
15.4.1	Associative-commutative (<i>AC</i>) unification examples	471
15.4.2	Unification examples with the <code>iter</code> attribute	473
15.4.3	Associative-commutative with identity (<i>ACU</i>) unification examples	474
15.4.4	Unification examples with an identity symbol	476
15.5	Unification at the metalevel: <code>metaUnify</code> and <code>metaDisjointUnify</code>	479
15.6	Some applications of unification	482
15.6.1	Narrowing-based unification	483
15.6.2	Symbolic reachability analysis in rewrite theories	484
15.6.3	Other automated deduction applications	486
15.7	Endogenous vs. exogenous order-sorted unification algorithms	487
15.8	Some notes on the implementation of unification	489
15.8.1	Combining unification algorithms	489
15.8.2	Diophantine basis element selection	491
15.9	Variants	492
15.9.1	Theories supported for variant generation	494
15.9.2	The <code>get variants</code> command	495
15.9.3	Variant generation at the metalevel: <code>getVariants</code>	501
15.10	Variant-based equational order-sorted unification	502
15.10.1	The <code>variant unify</code> command	503
15.10.2	Variant unification at the metalevel	505
16	Metaprogramming Applications	507
16.1	Commutative order-sorted unification	508
16.2	Rule instrumentation	520
16.3	A deadlock-freedom transformation	526
17	Mobile Maude	533
17.1	Processes and mobile objects	534
17.1.1	Processes and root objects	534
17.1.2	Mobile objects	536
17.1.3	Message forwarding	537
17.2	Mobile Maude additional definitions	539
17.3	Mobile Maude's rewriting semantics	540
17.3.1	Letting mobile objects do something	541
17.3.2	Object communication	542
17.3.3	Object mobility	546

- 17.3.4 The creation of mobile objects 551
- 17.3.5 Mobile object destruction 553
- 17.4 Mobile Maude Architecture 554
- 17.5 A buying printers example 558
- 17.6 Model checking Mobile Maude applications 566
 - 17.6.1 Redefinition of the `SOCKET` module 567
 - 17.6.2 Two-level atomic propositions for the buying printers
example 568
- 18 User Interfaces and Metalanguage Applications 571**
 - 18.1 The `LOOP-MODE` module 571
 - 18.2 User interfaces 572
 - 18.3 Using the loop 576
 - 18.4 Metalanguage applications: tokens, bubbles, and metaparsing . . 577
 - 18.5 Interactive Maude 585
 - 18.5.1 `IMaude` 586
 - 18.5.2 `IOP` 596

Part II Full Maude

- 19 Full Maude: Extending Core Maude 607**
 - 19.1 Running Full Maude 608
 - 19.2 Using Core Maude modules in Full Maude 613
 - 19.3 Additional module operations in Full Maude 615
 - 19.3.1 The tuple and power module expressions 617
 - 19.3.2 Parameterized views 619
 - 19.3.3 Example: leftist trees 622
 - 19.4 Moving up and down between reflection levels 626
 - 19.4.1 Up 626
 - 19.4.2 Down 628
 - 19.5 *Ax*-coherence completion 629
 - 19.6 Differences between Full Maude and Core Maude 632
 - 19.7 Adding new features to Full Maude 634
 - 19.7.1 A `unifyc` unification command 634
 - 19.7.2 A new module expression: `DEADLOCK-FREE` 642
- 20 Narrowing 649**
 - 20.1 Introduction 649
 - 20.1.1 Applications 651
 - 20.1.2 Completeness of narrowing 652
 - 20.1.3 Narrowing with simplification 653
 - 20.2 Theories supported for narrowing reachability 653
 - 20.3 The narrowing `search` command 654
 - 20.3.1 Narrowing with extra variables in righthand sides of rules 657

20.3.2	Reachability at the metalevel: <code>metaNarrowSearch</code> and <code>metaNarrowSearchPath</code>	660
20.3.3	Paramodulation	662
21	Object-Oriented Modules	665
21.1	Object-oriented systems	666
21.1.1	Objects and messages	666
21.1.2	Classes	667
21.1.3	Inheritance	668
21.1.4	Object-oriented rules	669
21.2	More examples of object-oriented modules	673
21.2.1	A puzzle	673
21.2.2	A simple spreadsheet	675
21.2.3	Blocks world	678
21.3	A bigger example: a rent-a-car store	680
21.4	Object-oriented parameterized programming	684
21.4.1	Theories	684
21.4.2	Views	685
21.4.3	Parameterized object-oriented modules	685
21.5	Module operations on object-oriented modules	688
21.5.1	Module summation and renaming	688
21.5.2	Module instantiation	689
21.6	Example: extended rent-a-car store	690
21.7	A strategy for sequential rule execution	696
21.8	Model checking a round-robin scheduling algorithm	700
21.9	From object-oriented modules to system modules	704

Part III Applications and Tools

22	A Sampler of Application Areas	711
22.1	Models of computation	711
22.2	Semantics of programming languages and software analysis	713
22.3	Maude as a metalanguage	716
22.3.1	Representing, mapping, and reasoning about logics	716
22.3.2	Specifying and building formal tools	718
22.4	Modeling and analysis of networks and distributed systems	719
22.4.1	Distributed architectures and components	719
22.4.2	Specification and analysis of communication protocols	721
22.4.3	Modeling and analysis of security protocols	722
22.5	Real-time systems	723
22.6	Probabilistic systems	724
22.7	Modeling and analysis of biological systems	727

23 Some Tools	733
23.1 Maude tools.....	733
23.1.1 The ITP: an Inductive Theorem Prover.....	733
23.1.2 The Maude Termination Tool	735
23.1.3 The Church-Rosser Checker	736
23.1.4 The Maude Coherence Checker	738
23.1.5 The Sufficient Completeness Checker	739
23.1.6 The Real-Time Maude tool	741
23.1.7 Predicate abstraction in Maude.....	742
23.2 Other tools.....	743
23.2.1 The Open Calculus of Constructions	743
23.2.2 The Maude MSOS tool	748
23.2.3 The CCS and LOTOS tools	749
23.2.4 The MSR cryptoprotocol specification language.....	750
23.2.5 JavaFAN	753
23.2.6 Java+ITP	754
23.2.7 The ITP/OCL tool	756
23.2.8 The Pathway Logic Assistant	758

Part IV Reference

24 Debugging and Troubleshooting	763
24.1 Debugging approaches	763
24.1.1 Tracing	763
24.1.2 Term coloring	772
24.1.3 The debugger	774
24.1.4 The profiler	777
24.1.5 Performance note	788
24.2 Traps and known problems	789
24.2.1 Associativity and idempotency	789
24.2.2 Segmentation fault (core dumped)	790
24.2.3 Bare variable lefthand sides	790
24.2.4 Operator overloading and associativity	791
24.2.5 Preregularity and equational attributes	791
24.2.6 Collapse theories.....	794
24.2.7 One-sided identities and associativity.....	795
24.2.8 Memberships for associative operators	797
24.2.9 Memberships for iterated operators	800
24.2.10 Ambiguity in <code>print</code> attribute items	801

25 Complete List of Maude Commands	803
25.1 Command line flags	803
25.2 Rewriting commands	804
25.3 Matching commands	806
25.4 Searching commands	807
25.5 Unification and variants commands	808
25.6 Tracing commands	809
25.7 Print attribute commands	810
25.8 Print option commands	810
25.9 Show option commands	811
25.10 Show commands	812
25.11 Profiler commands	813
25.12 Debugger commands	813
25.13 Miscellaneous commands	814
25.14 System level commands	815
26 Core Maude Grammar	817
26.1 The grammar	817
26.2 Synonyms	821
26.3 Lexical Issues	823
References	825
Subject Index	853
Index of Maude Modules	861
Index of Maude Theories	865
Index of Maude Views	867

List of Figures

1.1	Diagram of reading suggestions	29
2.1	Maude home page at <code>maude.cs.uiuc.edu</code>	34
2.2	Running Maude inside Emacs	38
4.1	Confluence diagram	101
6.1	Coherence diagram	141
6.2	Graphical representation of search graph in example	150
6.3	Petri net of the vending machine	153
6.4	A Petri net model of a library	154
6.5	Initial and final states in a world with three blocks	159
7.1	Rabbits ready to jump	167
7.2	The Khun Phan puzzle	178
8.1	Importation (protecting) graph of number hierarchy modules	197
8.2	Hierarchy of order theories	208
8.3	Structure of LEX-PAIR	218
9.1	Importation (protecting) graph of predefined modules	236
9.2	Importation graph of parameterized list and set modules	278
9.3	From lists to weakly sortable lists	292
9.4	From weakly sortable lists to sortable lists	294
9.5	Another version of sortable lists	296
10.1	Left-right rotation in an AVL tree	334
10.2	An example of an AVL tree	337
11.1	Importation graph of bank modules	349
11.2	Importation graph of ticker modules	352
11.3	Importation graph of data-agents modules	363

XXII List of Figures

13.1	Importation graph of model-checking modules	401
13.2	Graphical representation of a Kripke structure	413
14.1	Importation graph of metalevel modules	425
17.1	Object and message mobility	536
17.2	Buyers and sellers configuration	562
18.1	An IOP process configuration	598
18.2	A simple vending machine interface	602
19.1	An example of a leftist tree	625
21.1	Possible initial and final states for the 8-puzzle	673
23.1	Embedding of MSR into rewriting logic	751
23.2	Architecture of JavaFAN	754
23.3	A screen shot of the PLA Viewer	759
24.1	Number of rewrites and cpu time for different versions of the sorting algorithms	787

Introduction

This introduction tries to give the big picture on the goals, design philosophy, logical foundations, applications, and overall structure of Maude. It is written in an impressionistic, conversational style, and should be read in that spirit. The fact that occasionally some particular technical concept mentioned in passing (for example, “the Church-Rosser property”) may be unfamiliar should not be seen as an obstacle. It should be taken in a relaxed, sporting spirit: those things will become clearer in the body of the book; here it is just a matter of gaining a first overall impression.

1.1 Simplicity, expressiveness, and performance

Maude’s language design can be understood as an effort to simultaneously maximize three dimensions:

- *Simplicity*: programs should be as simple as possible and have clear meaning.
- *Expressiveness*: a very wide range of applications should be naturally expressible: from sequential, deterministic systems to highly concurrent nondeterministic ones; from small applications to large systems; and from concrete implementations to abstract specifications, all the way to *logical frameworks*, in which not just applications, but entire formalisms, other languages, and other logics can be naturally expressed.
- *Performance*: concrete implementations should yield system performance competitive with other efficient programming languages.

Although simplicity and performance are natural allies, maximizing expressiveness is perhaps the key point in Maude’s language design. Languages are after all *representational devices*, and their merits should be judged on the degree to which problems and applications can be represented and reasoned about generally, naturally, and easily. Of course, *domain-specific* languages

also have an important role to play in certain application areas, and can offer a useful “economy of representation” for a given area. In this regard, Maude should be viewed as a high-performance *metalanguage*, through which many different domain-specific languages can be developed.

1.1.1 Simplicity

Maude’s basic programming statements are very simple and easy to understand. They are *equations* and *rules*, and have in both cases a simple *rewriting semantics* in which instances of the lefthand side pattern are replaced by corresponding instances of the righthand side.

A Maude program containing only equations is called a *functional module*. It is a functional program defining one or more functions by means of equations, used as simplification rules. For example, if we build lists of quoted identifiers (which are sequences of characters starting with the character ‘`’` and belong to a sort¹ `Qid`) with a “cons” operator denoted by an infix period,

```
op nil : -> List .
op _._ : Qid List -> List .
```

then we can define a length function and a membership predicate by means of the operators and equations

```
op length : List -> Nat .
op _in_ : Qid List -> Bool .

vars I J : Qid .
var L : List .

eq length(nil) = 0 .
eq length(I . L) = s length(L) .

eq I in nil = false .
eq I in J . L = (I == J) or (I in L) .
```

where `s_` denotes the successor function on natural numbers, `_==_` is the equality predicate on quoted identifiers, and `_or_` is the usual disjunction on Boolean values. Such equations (specified in Maude with the keyword `eq` and ended with a period) are used from left to right as *equational simplification rules*. For example, if we want to evaluate the expression

```
length('a . 'b . 'c . nil)
```

we can apply the second equation for `length` to simplify the expression three times, and then apply the first equation once to get the final value `s s s 0`:

¹ In Maude, types come in two flavors, called *sorts* and *kinds* (see Section 3, and the discussion of user-definable data in Section 1.1.2 below).

```
length('a . 'b . 'c . nil)
= s length('b . 'c . nil)
= s s length('c . nil)
= s s s length(nil)
= s s s 0
```

This is the standard “replacement of equals by equals” use of equations in elementary algebra and has a very clear and simple semantics in equational logic. Replacement of equals by equals is here performed only from left to right and is then called *equational simplification* or, alternatively, *equational rewriting*. Of course, the equations in our program should have good properties as “simplification rules” in the sense that their final result exists and should be unique. This is indeed the case for the two functional definitions given above.

In Maude, equations can be *conditional*; that is, they may only be applied if a certain condition holds. For example, we can simplify a fraction to its irreducible form using the conditional equation

```
vars I J : NzInt .
ceq J / I = quot(J, gcd(J, I)) / quot(I, gcd(J, I))
  if gcd(J, I) > s 0 .
```

where `ceq` is the Maude keyword introducing conditional equations, `NzInt` is the sort of nonzero integers, and where we assume that the integer quotient (`quot`) and greatest common divisor (`gcd`) operations have already been defined by their corresponding equations.

A Maude program containing rules and possibly equations is called a *system module*. Rules are also computed by rewriting from left to right, that is, as *rewrite rules*, but they are *not* equations; instead, they are understood as *local transition rules* in a possibly concurrent system. Consider, for example, a distributed banking system in which we envision the account objects as floating in a “soup,” that is, in a multiset or bag of objects and messages. Such objects and messages can “dance together” in the distributed soup and can interact locally with each other according to specific rewrite rules. We can represent a bank account as a record-like structure with the name of the object, its class name (`Account`) and a `bal(ance)` attribute, say, a natural number. The following are two different account objects in our notation:

```
< 'A-001 : Account | bal : 200 >
< 'A-002 : Account | bal : 150 >
```

Accounts can be updated by receiving different *messages* and changing their state accordingly. For example, we can have `debit` and `credit` messages, such as

```
credit('A-002, 50)
debit('A-001, 25)
```

We can think of the “soup” as formed just by “juxtaposition” (with empty syntax) of objects and messages. For example, the above two objects and two messages form the soup

```
< 'A-001 : Account | bal : 200 >
< 'A-002 : Account | bal : 150 >
credit('A-002, 50)
debit('A-001, 25)
```

in which the order of objects and messages is immaterial. The local interaction rules for crediting and debiting accounts are then expressed in Maude by the rewrite rules

```
var I : Qid .
vars N M : Nat .

r1 < I : Account | bal : M > credit(I, N)
=> < I : Account | bal : (M + N) > .

crl < I : Account | bal : M > debit(I, N)
=> < I : Account | bal : (M - N) >
if M >= N .
```

where rules are introduced with the keyword `r1` and conditional rules (like the above rule for `debit` that requires the account to have enough funds) with the `crl` keyword.

Note that these rules *are not equations at all*: they are local transition rules of the distributed banking system. They can be applied concurrently to different fragments of the soup. For example, applying both rules to the soup above we get the new distributed state:

```
< 'A-001 : Account | bal : 175 >
< 'A-002 : Account | bal : 200 >
```

Note that the rewriting performed is *multiset rewriting*, so that, regardless of where the account objects and the messages are placed in the soup, they can always come together and rewrite if a rule applies. In Maude this is specified in the *equational part* of the program (system module) by declaring that the (empty syntax) multiset union operator satisfies the associativity and commutativity equations:

```
X (Y Z) = (X Y) Z
X Y = Y X
```

This is not done by giving the above equations explicitly. It is instead done by declaring the multiset union operator with the `assoc` and `comm` equational attributes (see Section 4.4.1 and Section 1.1.2 below), as follows, where `Configuration` denotes the multisets or soups of objects and messages.

```
op __ : Configuration Configuration -> Configuration [assoc comm] .
```

Maude then uses this information to generate a *multiset matching algorithm*, in which the multiset union operator is matched *modulo* associativity and commutativity.

Again, a program involving such rewrite rules is intuitively very simple, and has a very simple rewriting semantics. Of course, the systems specified by such rules can be highly concurrent and *nondeterministic*; that is, unlike for equations, there is no assumption that all rewrite sequences will lead to the same outcome. For example, depending on the order in which `debit` or `credit` messages are consumed, a bank account can end up in quite different states, because the rule for debiting can only be applied if the account balance is big enough. Furthermore, for some systems there may *not* be any final states: their whole point may be to continuously engage in interactions with their environment as *reactive* systems.

1.1.2 Expressiveness

The above examples illustrate a general fact, namely, that Maude can express with equal ease and naturalness *deterministic* computations, which lead to a unique final result, and *concurrent, nondeterministic* computations. The first kind is typically programmed with equations in functional modules, and the second with rules (and perhaps with some equations for the “data” part) in system modules.

In fact, functional modules define a *functional sublanguage*² of Maude. In a functional language true to its name, functions have unique values as their results, and it is neither easy nor natural to deal with highly concurrent and nondeterministic systems while keeping the language’s functional semantics. It is well known that such systems pose a serious expressiveness challenge for functional languages. In Maude this challenge is met by system modules, which extend the purely functional semantics of equations to the concurrent rewriting semantics of rules.³ Although certainly declarative in the sense of having a clear logical semantics, system modules are of course *not* functional: that is their entire *raison d’être*.

Besides this generality in expressing both deterministic and nondeterministic computations, further expressiveness is gained by the following features:

- equational pattern matching,
- user-definable syntax and data,
- types, subtypes, and partiality,
- generic types and modules,
- support for objects, and

² This sublanguage is essentially an extension of the OBJ3 equational language [174], which has greatly influenced the design of Maude.

³ As explained in Section 1.2, mathematically this is achieved by a *logic inclusion*, in which the functional world of equational theories is conservatively embedded in the nonfunctional, concurrent world of rewrite theories.

- reflection.

We briefly discuss each of these features in what follows.

Equational pattern matching

Rewriting with both equations and rules takes place by *matching* a lefthand side term against the subject term to be rewritten. The most common form of matching is *syntactic matching*, in which the lefthand side term is matched as a tree on the (tree representation of the) subject term (see Section 4.7). For example, the matching of the lefthand sides for the equations defining the `length` and `_in_` functions above is performed by syntactic matching. But we have already encountered another, more expressive, form of matching, namely, *equational matching* in the bank accounts example: the lefthand side

```
< I : Account | bal : M > credit(I, N)
```

has the (empty syntax) multiset union operator `__` as its top operator, but, thanks to its `assoc` and `comm` equational attributes, it is matched not as a tree, but as a multiset. Therefore, the match will succeed provided that the subject multiset contains instances of the terms `< I : Account | bal : M >` and `credit(I, N)` in which the variable `I` is instantiated the same way in both terms, *regardless of where those instances appear in the multiset*, that is, *modulo* associativity and commutativity.

In general, a binary operator declared in a Maude module can be defined with any⁴ combination of equational attributes of: associativity, commutativity, left-, right-, or two-sided identity, and idempotency. Maude then generates an *equational matching algorithm* for the equational attributes of the different operators in the module, so that each operator is matched *modulo* its equational attributes. This book will illustrate with various examples the expressive power afforded by this form of equational matching (see Section 4.8).

User-definable syntax and data

In Maude the user can specify each operator with its own syntax, which can be prefix, postfix, infix, or any “mixfix” combination. This is done by indicating with underscores the places where the arguments appear in the mixfix syntax. For example, the infix list *cons* operator above is specified by `_._`, the (empty syntax) multiset union operator by `__`, and the if-then-else operator by `if_then_else-fi`. In practice, this improves readability (and therefore understandability) of programs and data. In particular, for *metalanguage uses*, in which another language or logic is represented in Maude, this can make a big difference for understanding large examples, since the Maude representation can keep essentially the original syntax. The combination of user-definable syntax with equations and equational attributes for matching leads to a very expressive capability for specifying any *user-definable data*. It is well known

⁴ Except for any combination including both associativity and idempotency, which is not currently supported.

that any computable data type can be equationally specified [21]. Maude gives users full support for this equational style of defining data which is not restricted to syntactic terms (trees) but can also include lists (modulo associativity), multisets (modulo associativity and commutativity), sets (adding an idempotency equation), and other combinations of equational attributes that can then be used in matching. This great expressiveness for defining data is further enhanced by Maude’s rich type structure, as explained below.

Types, subtypes, and partiality

Maude has two varieties of types: *sorts*, which correspond to well-defined data, and *kinds*, which may contain error elements. Sorts can be structured in *subsort hierarchies*, with the subsort relation understood semantically as subset inclusion. For example, for numbers we can have subsort inclusions

```
Nat < Int < Rat
```

indicating that the natural numbers are contained in the integers, and these in turn are contained in the rational numbers. All these sorts determine a *kind* (say the “number kind”) which is interpreted semantically as the set containing all the well-formed numerical expressions for the above number systems as well as error expressions such as, for example, $4 + 7/0$. This allows support for *partial functions* in a total setting, in the sense that a function whose application to some arguments has a kind but not a sort should be considered *undefined* for those arguments (but notice that functions can also map undefined to defined results, for example in the context of error recovery). Furthermore, operators can be *subsort-overloaded*, providing a useful form of *subtype polymorphism*. For example, the addition operation `_+_` is subsort overloaded and has typings for each of the above number sorts. A further feature, greatly extending the expressive power for specifying partial functions, is the possibility of *defining sorts by means of equational conditions*. For example, a sequential composition operation `_;` concatenating two paths in a graph is defined if and only if the target of the first path coincides with the source of the second path. In Maude this can be easily expressed with the “conditional membership” (see Section 4.3):

```
vars P Q : Path .
cmb (P ; Q) : Path if target(P) = source(Q) .
```

Generic types and modules

Maude supports a powerful form of *generic programming* that substantially extends the *parameterized programming* capabilities of OBJ3 [174]. The analogous terminology to express these capabilities in higher-order type theory would be *parametric polymorphism* and *dependent types*. But in Maude the parameters are not just types, but *theories*, including operators and equations that impose semantic restrictions on the parameterized module instantiations. Thus, whereas a parametric LIST module can be understood just at the level

of the parametric type (sort) of list elements, a parameterized **SORTING** module has the theory **TOSET** of totally ordered sets as its parameter, including the axioms for the order predicate, that must be satisfied in each correct instance for the sorting function to work properly. Types analogous to dependent types are also supported by making the parameter instantiations depend on specific parametric constants in the parameter theory, and by giving membership axioms depending on such constants. For example, natural numbers modulo n (see Section 21.8), and arrays of length n , can be easily defined this way. The fact that entire modules, and not just types, can be parametric provides even more powerful constructs. For example, **TUPLE**[n] (see Section 19.3.1) is a “dependent parameterized module” that assigns to each natural number n the parameterized module of n -tuples (together with the tupling and projection operations) with n parameter sorts.

Support for objects

The bank accounts example illustrates a general point, namely, that in Maude it is very easy to support objects and distributed object interactions in a completely declarative style with rewrite rules. Although such object systems are just a particular style of system modules in which object interactions (through messages or directly between objects) are expressed by rewriting, Maude provides special support for object-based programming and for fair execution of object-based applications (see Chapter 11). Furthermore, the Full Maude extension provides special syntax in *object-oriented modules* (see Chapter 21). Such modules directly support object-oriented concepts like objects, messages, classes, and multiple class inheritance. Moreover, the support for *communication with external objects* (see Section 11.4) allows Maude objects to interact by message passing with internet sockets and, through them, with all kinds of other external objects, such as files, databases, graphical user interfaces, sensors, robots, and so on. All this is achieved without compromising Maude’s declarative nature: interaction with normal Maude objects and with external objects can both be programmed with rewrite rules. Using internet sockets as external objects, it is also easy to develop *distributed implementations* in Maude, where a “soup” of objects and messages is not realized just as a multi-set data structure in a single sequential machine, but as a “distributed soup,” with objects and messages in different machines or in transit (see Chapter 17 for a mobile language application of this kind).

Reflection

This is a very important feature of Maude. Intuitively, it means that Maude programs can be metarepresented as *data*, which can then be manipulated and transformed by appropriate functions. It also means that there is a systematic *causal connection* between Maude modules themselves and their metarepresentations, in the sense that we can either first perform a computation in a module and then metarepresent its result, or, equivalently, we can first metarepresent the module and its initial state and then perform the entire

computation *at the metalevel*. Finally, the metarepresentation process can itself be iterated giving rise to a very useful *reflective tower*. Thanks to Maude’s logical semantics (more on this in Section 1.2), all this is not just some kind of “glorified hacking,” but a precise form of *logical reflection* with a well-defined semantics (see Chapter 14 and [77, 78]). There are many important applications of reflection. Let us mention just three:

- *Internal strategies*. Since the rewrite rules of a system module can be highly nondeterministic, there may be many possible ways in which they can be applied, leading to quite different outcomes. In a distributed object system this may be just part of life: provided some fairness assumptions are respected, any concurrent execution may be acceptable. But what should be done in a sequential execution? Maude does indeed support two different fair execution strategies in a built-in way through its `rewrite` and `frewrite` commands (see Section 6.4). But what if we want to use a different strategy for a given application? The answer is that Maude modules can be executed at the metalevel with user-definable *internal strategies*⁵ (see Section 14.6). Such internal strategies can be defined by rewrite rules in a metalevel module that guides the possibly nondeterministic application of the rules in the given “object level” module. This process can be iterated in the reflective tower. That is, we can define meta-strategies, meta-meta-strategies, and so on.
- *Module algebra*. The entire *module algebra* in which parameterized modules can be composed and instantiated becomes expressible within the logic, and *extensible* by new module operations that transform existing modules metarepresented as data. This is of more than theoretical interest: Maude’s module algebra is realized exactly in this way by Full Maude, a Maude program defining all the module operations and easily extensible with new ones (see Part II of this book).
- *Formal tools*. The verification tools in Maude’s formal environment must take Maude modules as arguments and perform different formal analyses and transformations on such modules. This is again done by reflection in tools such as Maude’s inductive theorem prover, the Church-Rosser checker, the Maude termination tool, the Real-Time Maude tool, and so on (see Chapter 23).

1.1.3 Performance

Achieving expressiveness in all the ways described above without sacrificing performance is a nontrivial matter. Successive Maude implementations have been advancing this goal while expanding the set of language features. More work remains ahead, but it seems fair to say that Maude, although still an interpreter, is a high-performance system that can be used for many non-toy

⁵ That is, internal to Maude’s logic, in the sense of being definable by logical axioms.

applications with competitive performance and with many advantages over conventional code. Without in any way trying to extrapolate a specific experience into a general conclusion, a concrete example from the Maude user’s trenches may illustrate the point. A formal tool component to check whether a trace of events satisfies a given linear temporal logic (LTL) formula was written in Maude at NASA Ames by Grigore Roşu in about one page of Maude code. The component had a trivial correctness proof—the Maude module was based on the equational definition of the LTL semantics for the different connectives. This replaced a similar component having about 5,000 lines of Java code that had taken over a month to develop by an experienced colleague. The Java tool used a translation of LTL formulas into Büchi automata (the usual method to efficiently model check an LTL formula) and run about three times more slowly than the Maude code. It would have been very difficult to prove the correctness of the Java tool and, having a better and clearly correct alternative in the Maude implementation, this was never done.

Generally and roughly speaking, the current Maude implementation can execute syntactic rewriting with typical speeds from half a million to several million rewrites per second, depending on the particular application and the given machine. Similarly, associative and associative-commutative equational rewriting with term patterns used in practice⁶ can be performed at the typical rate of one hundred thousand to several hundred thousand rewrites per second.

These figures must be qualified by the observation that, until recently, the cost of an associative or associative-commutative rewriting step *depended polynomially on the size of the subject term*, even with the most efficient algorithms. In practice this meant that this kind of rewriting was not practical for large applications, in which the lists or multisets to be rewritten could have millions of elements. This situation has been drastically altered by a recent result of Steven Eker [136] providing new algorithms for associative and associative-commutative rewriting that, for patterns typically encountered in practice, can perform one step of associative rewriting in constant time, and one associative-commutative rewriting step in time proportional to the logarithm of the subject term’s size. Maude supports equational rewriting with these new algorithms.

The reason why the Maude interpreter achieves high performance is that the rewrite rules are carefully analyzed and are then *semicompiled* into efficient matching and replacement automata [134] with efficient matching algorithms. One important advantage of semicompilation is that it is possible to trace every single rewriting step. More performance is of course possible

⁶ In its fullest generality, it is well known that associative-commutative rewriting is an NP-complete problem. In programming practice, however, the patterns used as lefthand sides allow much more efficient matching, so that the theoretical limits only apply to “pathological” patterns not encountered in typical programming practice.

by full compilation. Maude has an experimental compiler for a subset of the language which can typically achieve a fivefold speedup over the interpreter.

Four other language features give the user different ways of optimizing the performance of his/her code. One is *profiling*, allowing a detailed analysis of which statements are most expensive to execute in a given application (see Section 24.1.4). Another is *evaluation strategies* (see Section 4.4.7), giving the user the possibility of indicating which arguments to evaluate and in which order before simplifying a given operator with the equations. This can range from “no arguments” (a lazy strategy) to “all arguments” (an eager bottom-up strategy) to something in the middle (like evaluating the condition before simplifying an if-then-else expression). Evaluation strategies control the positions in which *equations* can be applied. But what about rules? The analogous feature for rules is that of *frozen argument positions*; that is, declaring certain argument positions in an operator with the **frozen** attribute (see Section 4.4.9) blocks rule rewriting anywhere in the subterms at those positions. A fourth useful feature is *memoization* (see Section 4.4.8). By giving an operator the **memo** attribute, Maude stores previous results of function calls to that symbol. This allows trading off space for time, and can lead in some cases to drastic performance improvements.

One nagging question may be reflection. Is reflection really practical from a performance perspective? The answer is yes. In Maude, reflective computations are performed by *descent functions* that move metalevel computations to the object level whenever possible (see Section 14.5). This, together with the use of caching techniques, makes metalevel computations quite efficient. A typical metalevel computation may perform millions of rewrites very efficiently at the object level, paying a cost (linear in the size of the term) in changes of representation from the metalevel to the object level and back only at the beginning and at the end of the computation.

1.2 The logical foundations of Maude

The foundations of a house do not have to be inspected every day: one is grateful that they are there and are sound. This section describes the logical foundations of Maude in an informal, impressionistic style, not assuming much beyond a cocktail party acquaintance with logic and mathematics. The contents of this section may be read in two ways, and at two different moments:

- before reading the rest of the book, to obtain a bird’s-eye view of the mathematical ideas underlying Maude’s design and semantics; or
- after reading the rest of the book, to gain a more unified understanding of the language’s design philosophy and its foundations.

Readers with a more pragmatic interest may safely skip this section, but they may miss some of the fun.

Maude is a declarative language in the strict sense of the word. That is, a Maude program is a *logical theory*, and a Maude computation is *logical deduction* using the axioms specified in the theory/program. But which logic? There are two, one contained in the other. The seamless integration of the functional world within the broader context of concurrent, nondeterministic computation is achieved at the language level by the inclusion of functional modules as a special case of system modules. At the mathematical level this inclusion is precisely the sublogic inclusion in which *membership equational logic* [248, 31] is embedded in *rewriting logic* [243, 36].

A functional module specifies a *theory* in membership equational logic. Mathematically, we can view such a theory as a pair $(\Sigma, E \cup A)$. Σ , called the *signature*, specifies the type structure: sorts, subsorts, kinds, and overloaded operators. E is the collection of (possibly conditional) equations and memberships declared in the functional module, and A is the collection of equational attributes (`assoc`, `comm`, and so on) declared for the different operators. Computation is of course the efficient form of equational deduction in which equations are used from left to right as simplification rules.

Similarly, a system module specifies a *rewrite theory*, that is, a theory in rewriting logic. Mathematically, such a rewrite theory is a 4-tuple $\mathcal{R} = (\Sigma, E \cup A, \phi, R)$, where $(\Sigma, E \cup A)$ is the module's equational theory part, ϕ is the function specifying the frozen arguments of each operator in Σ , and R is a collection of (possibly conditional) rewrite rules. Computation is rewriting logic deduction, in which equational simplification with the axioms $E \cup A$ is intermixed with rewriting computation with the rules R .

We can of course view an equational theory $(\Sigma, E \cup A)$ as a degenerate rewrite theory of the form $(\Sigma, E \cup A, \phi_\emptyset, \emptyset)$, where $\phi_\emptyset(f) = \emptyset$, that is, no argument of f is frozen, for each operator f in the signature Σ . This defines a sublogic inclusion from membership equational logic (MEqLogic) into rewriting logic (RWLogic) which we can denote

$$\text{MEqLogic} \hookrightarrow \text{RWLogic}.$$

In Maude this corresponds to the inclusion of functional modules into the broader class of system modules. However, Maude's inclusion is more general: the user can give the desired freezing information for each operator in the signature of a functional module, not just the ϕ_\emptyset above.

Another important fact is that each Maude module specifies not just a theory, but also an *intended mathematical model*. This is the model the user has intuitively in mind when writing the module. For functional modules such models consist of certain sets of data and certain functions defined on such data, and are called *algebras*. For example, the intended model for a NAT module is the natural numbers with the standard arithmetic operations. Similarly, a module LIST-QID may specify a data type of lists of quoted identifiers, and may import NAT and BOOL as submodules to specify functions such as `length` and `_in_`. Mathematically, the intended model of a functional module specifying an equational theory $(\Sigma, E \cup A)$, with Σ the signature defining the

sorts, subsorts, and operators, E the equations and memberships, and A the equational attributes like `assoc`, `comm`, and so on, is called the *initial algebra* of such a theory and is denoted $T_{\Sigma/E \cup A}$.

In a similar way, a system module specifying a rewrite theory $\mathcal{R} = (\Sigma, E \cup A, \phi, R)$ has an *initial model*, denoted $\mathcal{T}_{\mathcal{R}}$, which in essence is an algebraic (*labeled*) *transition system*.⁷ The states and data of this system are elements of the underlying initial algebra $T_{\Sigma/E \cup A}$. The state transitions are the (possibly complex) *concurrent rewrites* possible in the system by application of the rules R . For our bank accounts example, these transitions correspond to all the possible concurrent computations that can transform a given “soup” of account objects and messages into another soup. Again, this is the model the programmer of such a system has in mind.

How do the mathematical models associated with Maude modules and the computations performed by them fit together? Very well, thanks. This is the so-called agreement between the *mathematical semantics* (the models) and the *operational semantics* (the computations). In this introduction we must necessarily be brief; see Sections 4.6 and 4.7 and [31] for the whole story in the case of functional modules, and Section 6.3 and [377] for the case of system modules. Here is the key idea: under certain executability conditions required of Maude modules, both semantics coincide. For functional modules we have already mentioned that the equations should have good properties as simplification rules, so that they evaluate each expression to a single final result. Technically, these are called the *Church-Rosser* and *termination* assumptions. Under these assumptions, the final values, called the *canonical forms*, of all expressions form an algebra called the *canonical term algebra*. By definition, the results of operations in this algebra are exactly those given by the Maude interpreter: this is as computational a model as one can possibly get. For example, the results in the canonical term algebra of the operations

```
length('a . 'b . 'c . nil)
'b in ('a . 'b . 'c . nil)
```

are, respectively,

```
s s s 0
true
```

Suppose that a functional module specifies an equational theory $(\Sigma, E \cup A)$ and satisfies the Church-Rosser and termination assumptions. Let us then denote by $Can_{\Sigma/E \cup A}$ the associated canonical term algebra. The coincidence of the mathematical and operational semantics is then expressed by the fact that we have an isomorphism

$$T_{\Sigma/E \cup A} \cong Can_{\Sigma/E \cup A}.$$

⁷ With additional operations, including a sequential composition operation for labeled transitions.

In other words, except for a change of representation, both algebras are identical.

For system modules, the executability conditions center around the notion of *coherence* between rules and equations (see [377] and Section 6.3). The equational part $E \cup A$ should be Church-Rosser and terminating as before. A reasonable strategy (the one adopted in Maude by the `rewrite` command, see Chapter 6) is to first apply the equations to reach a canonical form, and then do a rewriting step with a rule in R . But is this strategy *complete*? Couldn't we miss rewrites with R that could have been performed if we had not insisted on first simplifying the term to its canonical form with the equations? Coherence guarantees that this kind of incompleteness cannot happen (see Section 6.3).

1.3 Programming, specification, and verification

The observations in the previous section about the agreement between mathematical and operational semantics in Maude programs are of enormous importance for reasoning about them and verifying their correctness. The key point is that there are three different uses of Maude modules:

1. As *programs*, to solve some application. In principle we could have programmed such an application in some other programming language, but we may have chosen Maude because its features make the programming task easier and simpler.
2. As *formal executable specifications*, that provide a rigorous mathematical model of an algorithm, a system, a language, or a formalism. Because of the agreement between operational and mathematical semantics, this mathematical model is at the same time *executable*. Therefore, we can use it as a precise prototype of our system to *simulate* its behavior. The system itself could be implemented in a conventional language, or perhaps in Maude itself (as in (1) above) as a more detailed Maude program, or maybe our specification is already detailed and efficient enough to be directly used as *its own implementation*.
3. As models that can be *formally analyzed and verified* with respect to different *properties* expressing various *formal requirements*. For example, we may want to prove that our Maude module terminates; or that its equations have the Church-Rosser property; or that a given function, equationally defined in the module, satisfies some properties expressed as first-order formulas. Similarly, given a system module we may want to *model check* some properties about it, such as the satisfaction of some invariants or, more generally, of some temporal logic formulas.

Note that the distinction between uses (1) and (2) is, for the most part, in the eyes of the beholder. In fact, there is a seamless integration of specifications and code. The same Maude module can simultaneously be viewed as an executable formal specification and as a program. Furthermore, certain

kinds of formal requirements needed for verification in (3) can be expressed at the Maude level, either in Maude *theories* (see Section 8.3.1), or by including some *nonexecutable* statements in a Maude module giving them the `nonexec` attribute (see Section 4.5.3). This can be very useful in several ways. For example, we may include *lemmas* that we have proved about a module, either in theories or as nonexecutable statements in the module itself. Similarly, we may begin with some nonexecutable specifications in a Maude theory, and then refine them using *views* (see Section 8.3.2) into the desired Maude module satisfying them.

There is, however, no need for all the properties that we wish to formally verify in (3) to be in the logic of Maude, that is, to be statements in membership equational logic or in rewriting logic. More generally, properties can be expressed, for example, as arbitrary first-order logic formulas, or as temporal logic formulas. An interesting issue is then to explain precisely what it means for a Maude module, defined in membership equational logic or in rewriting logic, to *satisfy* a formula in one of those logics. Here is where the Maude initial model semantics explained in Section 1.2 becomes crucial. Such a semantics means that what a Maude module *denotes* is a specific *mathematical model*, namely, the initial one. Satisfaction of any property, expressed as some kind of formula, means satisfaction of that formula in the initial model. This is an important observation, even when the formula in question is expressed in Maude's native logic. Let us explain this idea in more detail.

Consider, for example, that we have defined natural number addition in a Maude functional module with Peano notation, so that zero is represented as the constant `0`, and there is a successor function `s_` so that, for example, 2 is represented as `s s 0`. Natural number addition can then be defined by the equations

```
op _+_ : Nat Nat -> Nat .
vars N M K : Nat .
eq N + 0 = N .
eq N + (s M) = s (N + M) .
```

The *initial model* of these equations is precisely the algebra of the natural numbers with zero, successor, and the usual addition function. For example, using the canonical term algebra representation (see Section 1.2), when we add `s s 0` and `s s 0` in this algebra we obtain the result `s s s s 0`.

Consider now two relevant *properties* of natural number addition, namely, associativity and commutativity. These properties are precisely described by the respective equations

```
eq N + M = M + N [nonexec] .
eq N + (M + K) = (N + M) + K [nonexec] .
```

where we have used the `nonexec` attribute to emphasize that these equations are not part of our natural number addition module, and are not meant to be executed (in fact, if executed the first equation would loop). They may,

for example, be stated in a separate Maude theory as properties we wish to verify.

The first thing to observe is that the above associativity and commutativity equations *are not provable* by equational deduction, that is, they do not follow by replacing equals by equals from the two equations defining the addition function. They are in fact *inductive* properties of the addition function. Therefore, in order to prove them, using for example Maude's inductive theorem prover (ITP, see Section 23.1.1), we need to use a stronger proof method, namely, Peano induction. But for any equational specification, being an inductive property and being a property satisfied by its initial model are one and the same thing [253]. Therefore, what we mean when we say that our natural number addition module *satisfies* the associativity and commutativity equations is *precisely* that its initial model does.

Of course, associativity and commutativity are properties expressible in Maude's native logic (in fact, just in its equational sublogic). But the case of arbitrary first-order formulas is entirely similar. Consider, for example, the property that any even number is the sum of two odd numbers, which can be expressed as the first-order formula

$$\forall n : Nat (even(n) \implies \exists x, y : Nat (odd(x) \wedge odd(y) \wedge n = x + y)).$$

Let us assume, for argument's sake, that we had also defined the *odd* and *even* predicates in our Maude natural number module. What does it mean for our module to satisfy the above formula? Just as before, it exactly means that the initial model denoted by our Maude specification satisfies the formula. The point is that membership equational logic is a sublogic of many-kinded first-order logic with equality (MKFOL⁼) that we can represent with a sublogic inclusion

$$MEqLogic \hookrightarrow MKFOL^=.$$

Therefore, our initial model is also a first-order logic model, and it is perfectly clear what it means for it to satisfy a first-order formula.

In a similar way, if we have a Maude system module and choose an initial state for it, we may be interested in verifying that it satisfies a given *temporal logic* formula. Defining satisfaction in this case is not as direct as for first-order formulas, because we do not have a sublogic inclusion from rewriting logic into temporal logic. However, the meaning of satisfaction in this case is also fairly straightforward. The point is that to such a system module, that is, to a rewrite theory in which we have defined some atomic state predicates equationally, we can naturally associate a *Kripke structure* (see Section 13.2). Since Kripke structures are the standard models of temporal logic, satisfaction of the given temporal logic formula exactly means that the Kripke structure associated to the module satisfies the formula. In fact, such a Kripke structure and the initial model of the rewrite theory are intimately related, so that the initial model can be used to define the corresponding Kripke structure. As explained in Chapters 12 and 13, if our system module is such that the

set of states reachable from the initial state is finite, we can use Maude's `search` command and Maude's model checker for linear temporal logic (LTL) as *decision procedures* to verify, respectively, the satisfaction of invariants and of LTL properties.

Besides being able to use Maude's inductive theorem prover (ITP) to verify inductive properties of functional modules, and the above-mentioned built-in support for verifying invariants and LTL formulas through the `search` command and Maude's LTL model checker, we can use the following Maude tools to formally verify other properties:

- the Maude Termination Tool (MTT) [117] can be used to prove termination of functional modules (see Sections 23.1.2, 12.4, and 13.4);
- the Maude Church-Rosser Checker (CRC) [73, 120] can be used to check the Church-Rosser property of unconditional functional modules (see Sections 23.1.3, 12.4, and 13.4);
- the Maude Coherence Checker (ChC) can be used to check the coherence (or ground coherence) of unconditional system modules (see Sections 23.1.4, 12.4, and 13.4); and
- the Maude Sufficient Completeness Checker (SCC) [190] can be used to check that defined functions have been fully defined in terms of constructors (see Sections 23.1.5, 4.4.3, 12.4, and 13.4).

Furthermore, if we are dealing with rewriting logic specifications of real-time and hybrid systems, we can use the Real-Time Maude tool (see Section 23.1.6) to both simulate such specifications and to perform search and model-checking analysis of their LTL properties.

In summary, therefore, Maude supports three seamlessly integrated tasks: programming, executable formal specification, and formal analysis and verification. For analysis and verification purposes, the Maude interpreter itself is the first and most obvious tool. It is in fact a high-performance *logical engine* that can be used to prove certain kinds of logical facts about our theories. For example, we can use the Maude interpreter as a decision procedure for equational deduction if the desired theory has good properties (see the example in Section 5.6). Similarly, as already mentioned, we can use it also to verify invariants and LTL properties of finite-state system modules. More generally, we can use other tools in Maude's formal environment, such as the ITP, MTT, CRC, ChC, and SCC tools (or Real-Time Maude for real-time systems) to formally verify a variety of other properties.

1.4 A high-performance logical framework

Our previous discussion of the programming, executable specification, and formal verification uses of Maude makes clear that we can distinguish two different levels of formal specification: a *system specification* level, and a *property*

specification one. In a system specification we are after an unambiguous specification of a given system and how it actually *works*. Ideally this specification should be both formal and executable, and should therefore provide an *executable mathematical model* of the system we are interested in. This is exactly what Maude modules provide.

By contrast, when specifying *properties* of a system we are not necessarily after an executable model of our system. Instead, we *assume it*, as either already given or to be developed later, and specify such properties in a typically nonexecutable manner: for example in first-order logic, higher-order logic, or some temporal logic. That is, the properties we specify have an *intended model*, namely the system design captured by a system specification, and we are interested in *verifying* by different methods that the intended model *satisfies* the properties stated in our property specification. In the context of Maude, such property specifications can be given in a variety of ways:

- as nonexecutable equations, memberships, and rules in Maude’s native logics;
- as first-order logic formulas; or
- as invariants or, more generally, linear temporal logic formulas.

We can then use Maude itself and its formal tool environment to try to verify that a given system specified as a Maude module satisfies the desired properties.

Since Maude system specifications should be both formal and executable, Maude native logics, namely, membership equational logic and its rewriting logic extension, should be *computational* logics, that is, logics in which computation and deduction coincides, and simple enough to allow a *high-performance* implementation as a declarative programming language. This is what the Maude implementation provides. Of course, as mentioned in Section 1.2 and further explained in Sections 4.6 and 6.3, Maude modules should be theories that satisfy some reasonable *executability requirements*, making possible not only their efficient execution, but also the already-mentioned coincidence between mathematical and operational semantics.

However, not all computational logics are equally *expressive*. For example, equational logics (in either first-order or higher-order versions) are very well suited to specify *deterministic systems* under the Church-Rosser assumption, but poorly equipped to specify concurrent and highly nondeterministic systems. The whole point of extending membership equational logic to rewriting logic is to seamlessly integrate the specification of deterministic systems, through equational specifications in functional modules, and of concurrent and nondeterministic systems, through rewriting logic specifications in system modules, within the same language. Experience has shown that this makes rewriting logic a very expressive *semantic framework* for system specification. Chapter 22 explains in detail many semantic framework applications of Maude. Here we can only hint at them by mentioning some of the relevant areas:

- *Models of computation.* As explained in Section 22.1, many models of computation, including a very wide range of concurrency models, can be naturally specified as different theories within rewriting logic, and can be executed and analyzed in Maude.
- *Programming languages.* As explained in Section 22.2, and illustrated for a simple language in Section 13.6.1, rewriting logic has very good properties—combining in a sense the best features of denotational semantics’ equational definitions with the strengths of structural operational semantics—to give formal semantics to a programming language. Furthermore, in Maude such semantics definitions become the basis of interpreters, model checkers, and other program analysis tools for the language in question.
- *Distributed algorithms and systems.* As explained in Section 22.4, because of its good features for concurrent, object-based specification, many distributed algorithms and systems, including, for example, network protocols and cryptographic protocols, can be easily specified and analyzed in Maude. Furthermore, making use of Maude’s external object facility to program interactions with internet sockets, one can not just specify but also program various distributed applications in a declarative way (see Section 11.4 and Chapter 17).
- *Biological systems.* Cell dynamics is intrinsically concurrent, since many different biochemical reactions happen concurrently in a cell. By modeling such biochemical reactions with rewrite rules, one can develop useful symbolic mathematical models of cell biology. Such models can then be used to study and predict biological phenomena such as, for example, biopaths (see Section 22.7).

Furthermore, other application areas can be naturally supported in appropriate *extensions* of rewriting logic and Maude. For example, real-time and hybrid systems can be specified as *real-time rewrite theories*. Such specification can be executed and analyzed in the Real-Time Maude tool (see Sections 22.5 and 23.1.6). Similarly, probabilistic systems can be specified as *probabilistic rewrite theories*, and can be simulated in PMaude and analyzed in the VeStA tool (see Section 22.6).

The fact that in a computational logic computation and deduction coincide, so that they are like two sides of the same coin, can be used in two ways: we can use the logic as a semantic framework to specify different computational entities as just explained; or we can use it as a *logical framework* to represent many other logics in it. That is, if our computational logic has good representational features, it can be used as a *universal logic* which can faithfully express the inference systems of many other logics.

Since the logic is computational and presumably has an efficient implementation, this is not just a purely theoretical exercise: we can use such an implementation to *mechanize* deduction in any logic that we can faithfully represent inside our logical framework. Experience has shown that rewriting logic

has very good properties as a logical framework in precisely this sense. This experience is summarized in Section 22.3. An important practical consequence is that it becomes quite easy to use Maude to develop a variety of *formal tools* for different logics. The point is that any such tool has an associated inference system, so it is just a matter of representing such an inference system as a rewrite theory and guiding the application of the inference rules with suitable strategies (see Section 14.6). In addition, since such formal tools often manipulate and transform not only formulas but also *theories*, Maude’s reflective capabilities, which allow manipulating theories as data, become enormously useful. Section 22.3 and Chapter 23 explain and illustrate with examples how Maude can be used in this way as a *metatool* to build many other tools for Maude itself and for many other logics. For an example illustrating this idea in full detail in the case of a unification tool see Sections 16.1 and 19.7.

Reflection and the existence of initial models (and therefore of induction principles for such models) have one further important consequence, namely, that rewriting logic has also good properties as a *metalogical framework*. A metalogical framework is a logical framework in which we can not only represent and simulate many other logics: we can also *reason* within the framework about the metalogical properties of the logics thus represented. As explained in [19], this is exactly what can be done in rewriting logic using Maude and Maude’s inductive theorem prover (ITP).

1.5 Core Maude vs. Full Maude

We call *Core Maude* the Maude 2 interpreter implemented in C++ and providing all of Maude’s basic functionality. Part I explains in detail all the aspects of Core Maude, including its syntax and parsing, functional and system modules, module hierarchies, module parameterization with theories and module instantiation with views, its suite of predefined modules, the model-checking capabilities, object-based programming, reflection, and metalanguage uses.

Full Maude is an extension of Maude, written in Maude itself, that endows the language with an even more powerful and extensible *module algebra* than that available in Core Maude. As in Core Maude, modules can be parameterized and instantiated with views, but in addition views can also be parameterized. Full Maude also provides generic modules for n -tuples. Object-oriented modules (which can also be parameterized) support notation for objects, messages, classes, and inheritance.

Full Maude itself can be used as a basis for further extensions, by adding new functionality. It is possible both to change the syntax or the behavior of existing features, and to add new features; for example, we can add new commands, as we explain in Section 19.7. In this way Full Maude becomes a common infrastructure on top of which one can build tools, such as, e.g., the Church-Rosser and coherence checkers (see Sections 23.1.3 and 23.1.4), as well as environments for other languages, such as, e.g., the Real-Time Maude tool

for specifying and analyzing real-time systems [289, 291] (see Section 23.1.6), and the Maude MSOS tool for modular structural operational semantics [55] (see Section 23.2.2).

1.6 Book structure

This book documents Maude 2, and explains Maude’s basic concepts in a leisurely and mostly informal style. The material is basically presented following a “grammatical” order; for example, all features related with operators are discussed together. Concepts are introduced by concrete examples, that in general are complete modules, although in some cases may be fragments of modules. Chapters describing the features of the language are interleaved with chapters containing additional examples and explanations that should be useful for readers interested in more details about a particular subject. The complete source code of all the examples is available in the companion cd-rom and also in <http://maude.cs.uiuc.edu>.

The book is divided in four parts: Part I is devoted to Core Maude, Part II is devoted to Full Maude, Part III summarizes applications and tools, and Part IV is a reference manual. Here is a brief summary of what can be found in the remaining chapters:

Part I. Core Maude.

- Chapter 2** explains how to get Maude, how to install the system on the different platforms supported, and how to run it. It also includes pointers on how to get additional information and support.
- Chapter 3** describes the basic syntactic constructs of the language, including what is an identifier, a sort, and an operator. The different kinds of declarations that can be included in the different types of modules are explained here, in addition to fundamental concepts such as kinds or terms, and a discussion on parsing.
- Chapter 4** introduces *functional modules*, and the different statements that can be found in this kind of modules, namely equations and membership axioms. Operator and statement attributes are also introduced. The final part of this chapter is devoted to the use of functional modules for equational simplification, for which matching modulo axioms is a fundamental feature.
- Chapter 5** describes a hierarchy of data types (non-empty binary trees, lists, multisets, and sets) obtained by imposing on a binary operator various equational properties such as associativity, identity, commutativity, and idempotency.
- Chapter 6** introduces *system modules*, and is mainly devoted to rules, term rewriting, and the `search` command.
- Chapter 7** is an introduction to rule-based programming in Maude by means of a collection of puzzles that are solved by rewriting and searching over system modules.

- Chapter 8** explains the support for modularity provided by Core Maude. It describes first the different modes of module importation, namely *protecting*, *extending*, and *including*. Then it introduces the module summation and renaming operations. Finally, this chapter explains the powerful form of parameterized programming available in Core Maude, based on theories and views.
- Chapter 9** provides detailed descriptions of the different predefined data types available, including Booleans, natural numbers, integers, rationals, floating-point numbers, strings, and quoted identifiers. It also describes the generic containers provided by Maude, namely lists, sets, maps, and arrays. The chapter finishes with a description of a built-in linear Diophantine equation solver.
- Chapter 10** describes the equational specification in Maude of typical *generic* (i.e., parameterized) data structures, such as stacks, queues, lists, multisets, binary trees, and general trees, and also more complex data types, such as sorted lists and search, AVL, 2-3-4, and red-black trees, whose definitions require membership axioms.
- Chapter 11** explains the basic support for object-based programming, with special emphasis on the standard notation for object systems. It also describes how communication with external objects is supported in Core Maude through sockets.
- Chapter 12** explains how to use the `search` command to model check invariant properties of concurrent systems specified as system modules in Maude.
- Chapter 13** introduces linear temporal logic (LTL) and describes the facilities for LTL model checking provided by the Maude system. This procedure can be used to prove properties when the set of states reachable from an initial state in a system module is finite. When this is not the case, it may be possible to use an equational abstraction technique for reducing the size of the state space.
- Chapter 14** presents the reflective capabilities of the Maude system. The concept of *reflection* is introduced, and the effective way of supporting metalevel computation is discussed. The predefined module `META-LEVEL` and its submodules are presented, with special emphasis on the descent functions provided. The chapter ends with an introduction to the notion of internal strategies.
- Chapter 15** describes Core Maude support of order-sorted unification modulo axioms such as either commutativity or associativity and commutativity. The importance of this feature is made explicit in an overview of several interesting applications of unification, including narrowing and symbolic reachability analysis. This chapter also includes a discussion on endogenous vs. exogenous order-sorted unification algorithms.
- Chapter 16** shows the power of programming at the metalevel by means of several metaprogramming applications, including a metalevel imple-

mentation of commutative order-sorted unification and theory transformations for adding instrumentation and for making a system module deadlock free.

Chapter 17 introduces Mobile Maude, a mobile agent language extending Maude and supporting mobile computation, and then describes its distributed implementation based on sockets.

Chapter 18 explains the way of using the facilities provided by the modules `META-LEVEL` and `LOOP-MODE` for the construction of user interfaces and metalanguage applications. It also explains how to endow Maude with interactive capabilities.

Part II. Full Maude.

Chapter 19 explains the nature of Full Maude, and how to use it. This chapter includes information on how to load Core Maude modules into Full Maude, on the additional module operations (supported by tuple generation and parameterized views), and on the facilities available in Full Maude for moving up and down between reflection levels.

Chapter 20 describes an implementation of narrowing in Full Maude based on the unification facilities introduced in Chapter 15.

Chapter 21 introduces object-oriented modules, which provide a syntax more convenient than that of system modules for object-oriented applications, with direct support for the declaration of classes, inheritance, and useful default conventions in the definition of rules. Such object-oriented modules can also be parameterized. This chapter includes several extended examples that illustrate the power of combining the additional features available in Full Maude.

Part III. Applications and tools.

Chapter 22 gives an overview of some areas of application of rewriting logic and Maude, emphasizing the applications for which Maude seems particularly well suited.

Chapter 23 describes some existing Maude-based tools, either for the analysis of Maude specifications, or for formal analysis in several domains but making use of Maude in some way.

Part IV. Reference.

Chapter 24 discusses debugging and troubleshooting, considering the different debugging facilities provided: tracing, term coloring, the debugger, and the profiler. A number of traps and known problems are also commented.

Chapter 25 gives a complete list of the commands available in Maude.

Chapter 26 includes the grammar of Core Maude.

1.7 How to read this book

This book is quite true to its title: it gives a fairly complete account of Maude in *all* its aspects. It is almost a Maude *encyclopedia*. This is in principle a good

thing, but must be borne in mind when reading the book. Usually, one does not read an encyclopedia *cover to cover*. Instead, one is interested in certain topics and goes directly to the entries for those topics. This style of reading should, up to a point, be also possible here. However, the complete independence between voices such as *Byzantium* and *topology* in an encyclopedia does not hold to the same extent between the different parts of this book: there are, indeed, some obvious inter-relations. Nevertheless, the chapters are written in as self-contained a manner as possible. Furthermore, they contain cross references to specific topics in other chapters that are either needed for a better understanding of the given chapter, or useful to further extend one's understanding of it. Therefore, it should often be possible to dive into a given chapter, using those cross references to fill in just the minimum background information needed to understand it.

How then should this book be read? The answer should be: *as it best fits your preferences and needs*. Since there are many such preferences, we cannot give an exhaustive list of suggestions for each of them. We can, however, give some concrete suggestions for three possible, perhaps fairly common, types of readers:

1. a reader just *getting acquainted* with Maude may wish to get a first introduction to its main ideas and to get started using Maude as soon as possible;
2. a reader already familiar with Maude's basic ideas may then be primarily interested in using Maude as a *programming language*, and only secondarily interested in its formal specification and verification aspects;
3. such a reader may instead have the opposite set of preferences: may be primarily interested in the *formal specification and verification* aspects of Maude, and only secondarily in using Maude as a programming language.

Of course, a type (2) or (3) reader may first have been a type (1) reader. Also, types (2) and (3) may be only caricatures, or just initial biases, since the whole Maude philosophy is to seamlessly integrate the programming and formal specification and verification aspects. Therefore, we very much hope that, no matter what your initial interests were, in the end you will come to appreciate and use the intimate interplay between programming and formal reasoning that Maude supports. However, we think that the best way to help you get there is precisely to help you use this book in a way that addresses your current needs and interests first. For this purpose, we give below some reading suggestions corresponding to these three types of readers. These suggestions are summarized in the diagram of Figure 1.1 in page 29.

Beginners

We try to give you in what follows what we think may be the *shortest* way for you to gain a first acquaintance with Maude and to become a Maude user. You

may want to begin by taking a first quick look at the introduction (this chapter) and the applications chapter (Chapter 22) to get a first bird's eye view. Then, we recommend a quick first look at Chapter 2 to get Maude running on your computer. This could be followed by a quick reading of Sections 3.1–3.8 in Chapter 3 to get a first acquaintance with Maude syntactic entities. In a first reading you may skip Section 3.9 on parsing. You may compensate for your temporary ignorance of parsing details by using two simple rules of thumb when you run into parsing troubles:

- make sure that each declaration inside a Maude module is ended with a space followed by a period; and
- add extra parentheses to terms in equations, memberships, and rules (or to terms for evaluation) to avoid parsing ambiguities.

After reading Sections 4.1–4.3 on functional modules and taking a quick look at uses of the `reduce` command in Section 4.9, you should be able to write some simple functional modules on your own. Try defining some simple data structures such as lists or trees yourself, and defining also various functions manipulating such data structures by giving the appropriate equations in a functional module. Then, after getting Maude to accept your module and solving any parsing errors, test the correctness of your function definitions by evaluating appropriate expressions with the `reduce` command. In a first reading you may skip most of Section 4.4, except for Section 4.4.1 on equational attributes. Then, you could read Sections 4.7 and 4.8 to gain a better understanding of the matching and simplification processes involved in evaluating terms with the `reduce` command, particularly when they are performed modulo axioms such as associativity, commutativity, and identity. Again, you should then experiment defining new functional modules of your own that use some of the `assoc`, `comm`, and `id`: attributes. You may then wish to read Section 4.11 to gain a better feeling for the use of subsorts.

After this, you may want to read Sections 6.1, 6.2, and 6.4 to learn the basics about system modules and their computations with the `rewrite`, `frewrite`, and `search` commands. You may follow this up with a look at some more examples in Section 6.5. You should again develop several system modules of your own, and test them by entering them in Maude and performing several `rewrite`, `frewrite`, and `search` commands.

Chapter 9 should be read as a reference: sooner or later you will need to use some of Maude's predefined modules, so you can read about each of them when the need arises.

If you have done all this, something achievable in a relatively short time, you will already be a Maude user, and you will have gained a pretty good basic understanding of what Maude is and how to use it. Hopefully, you will also be itching to learn many other things about it.

Maude programmers

Let us assume that you passed the beginner level, either by following our suggestions above or in some other way. Let us also assume that your main interest is in using Maude as a programming language. What should you read next? You may now want to go back and read Section 3.9 on parsing. You may also want to read all the sections of Chapter 4 that you skipped in a first reading. In fact, Chapter 4 is very important and you should understand it inside and out. This can be followed by a look at Chapter 5; this may give you a chance to compare your own solutions to develop Maude programs for common data structures with some solutions presented there. Likewise, you may want to read all the sections you skipped in a first reading of Chapter 6 (presumably this was just Section 6.3) to gain a full understanding of the basics of system modules. This can be followed by a reading of Chapter 7, which may hopefully stimulate you to develop some other games of your own.

After this, a careful reading of Chapter 8 on module operations should be quite profitable. In particular, the parameterized programming supported in Maude—which from a type-theoretic perspective combines so-called parametric polymorphism and the parametricity of dependent types, and from a software perspective corresponds to so-called generic programming—is a very powerful feature and can be used to greatly increase program modularity and reuse. This can be followed by a reading of Sections 9.11–9.13, to become familiar with a useful library of predefined parameterized modules. This may be a good moment to try defining various parameterized modules on your own, for example for other advanced data structures, comparing your solutions with those in Chapter 10, which may also help your appreciation of using subsorts and membership axioms to handle partial operations. Chapter 11 is a must, both to understand how Maude supports object-based programming, and to become familiar with the programming of external objects such as internet sockets. Again, developing some object-based modules on your own should help you consolidate your understanding.

The next big topic is Chapter 14 on reflection. From the programming point of view this is one of Maude’s most powerful features, but also one of the most demanding. Therefore this chapter, and the examples in it and in the followup Chapter 16 should be studied carefully; then you can experiment developing powerful reflective applications yourself. To complete your training as an advanced Maude programmer, we recommend studying Chapter 17, which combines techniques of distributed programming in Chapter 11 with reflection techniques in Chapter 14, and Chapter 18 on how to build interactive applications.

Formal methods users

If you have already passed the beginner level but your main interest is in formal specification and verification, you may want to take a look at Chapter 23

to get a first impression of the tools available in the Maude environment and also to get a feeling for other tools that you might want to develop yourself, not just for Maude, but for other languages and logics as well. Then you should go back to Chapter 4 and read it in full detail, making sure you understand the more theoretical sections. Likewise, you should go back to Chapter 6 and understand more theoretical sections like 6.3 that you may have skipped in a first reading.

Chapter 8 on module operations is also important. Specifications should be as modular and as generic as possible, because this can greatly reduce the verification effort. In particular, you should become familiar with parameterized modules, and with the use of theories and views to assert possibly nonexecutable specifications. After this, you may wish to study how object-oriented specification is supported in Maude. For this, we recommend that you read Chapter 11, complementing it with a reading of Chapter 21 on Full Maude’s object-oriented modules.

After this, a next possible step would be studying Chapters 12 and 13 on model checking of invariants and of general LTL properties. Of course, all along you should have developed examples of your own: both executable specifications of interesting systems, and verification of properties such as invariants and LTL properties. If you are interested in the verification of programs in conventional languages, you might take Section 13.6 as an invitation and starting point for developing the semantics of a programming language of your choice in Maude yourself. In this way, you can get powerful program analysis tools for such a language essentially *for free*. You may wish to benefit from the experience already gained for Java and the JVM, which is reported in Sections 23.2.5 and 23.2.6. The next natural step would be to get familiar with other tools in Maude’s formal environment by re-reading Section 23.1, downloading some of the tools described there and their documentation, and performing various verification efforts supported by those tools.

If you are interested not only in using formal tools, but also in developing new formal tools of your own using Maude, you may want to become a Maude programmer. For tool-building purposes reflection is likely to be a very useful and powerful technique to master; therefore, studying carefully Chapter 14 on reflection and Chapter 16 on metaprogramming applications should be a must. You will also need to learn about developing user interfaces by studying Chapter 18. Other than that, all we have already suggested for Maude programmers should also be useful to you sooner or later.

Teaching

The reading paths suggested above can also be a good basis for using this book as a textbook in various courses at the undergraduate or graduate levels. For example, a declarative programming course could be organized by selecting material from the “Maude programmers” path. Similarly, the “formal methods” path can be used as a basis for courses on program verification and

formal methods. Indeed, one of us (Meseguer) has essentially followed this path—complemented with more theoretical material on the deductive and model-theoretic aspects—in one-semester graduate courses on these subjects at the University of Illinois at Urbana-Champaign, and for shorter courses at the University of Pisa and various summer schools.

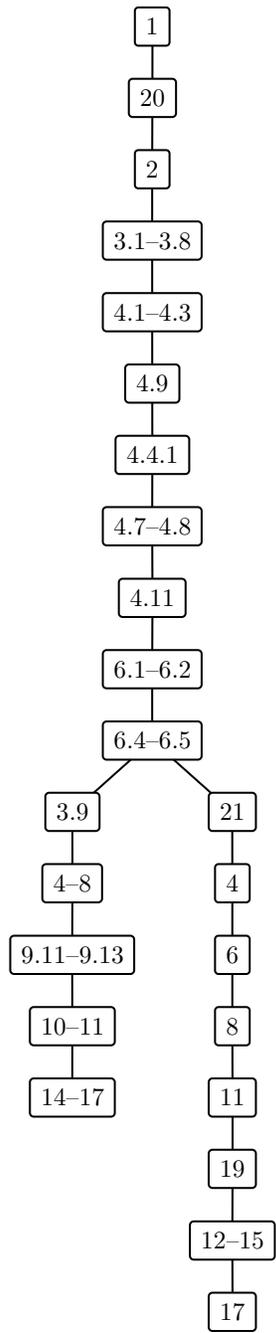


Fig. 1.1. Diagram of reading suggestions

Part I

Core Maude

Using Maude

2.1 Getting Maude

The Maude system is available, free of charge, under the terms of the GNU General Public License as published by the Free Software Foundation, at the Maude home page (a snapshot is shown in Figure 2.1)

`http://maude.cs.uiuc.edu`

There you can also find documentation about Maude, including a Maude primer, some papers on Maude and rewriting logic, and several Maude applications, including a set of proving tools for Maude specifications and Maude case studies.

Maude binaries are provided for selected architectures and operating systems, including Linux and MacOS X. Detailed information on this can be found in the Maude web site, where installation instructions are also available.

The companion cd-rom contains the latest version of Maude available at the time of writing, together with the complete source code of all the examples in this book.

2.2 Running Maude

A Maude session can be started by calling the `maude.linux` binary in the `maude-linux/bin` directory in a Linux shell window (and similarly for other platforms). For example, we can move into that directory and then invoke Maude, obtaining a banner similar to the following, where we can see the version of the system being executed, the date it was built, copyright information, and the current date.



Fig. 2.1. Maude home page at `maude.cs.uiuc.edu`

```
~/maude-linux/bin$ ./maude.linux
\|/
--- Welcome to Maude ---
/|/
Maude 2.7 built: Mar  3 2014 18:07:27
Copyright 1997-2014 SRI International
Tue Jul 15 22:57:15 2014

Maude>
```

The Maude system is now ready to accept Maude modules and commands (see Chapter 25 for a complete list of Maude commands). During a Maude

session, the user interacts with the system by entering a request at the Maude prompt. For example, one can quit:

```
Maude> quit
```

`q` may be used as an abbreviation of the `quit` command. But please, do not leave us so soon! One can also enter modules and use other commands. For example, we can enter the following module `SIMPLE-NAT`, which specifies the natural numbers in Peano notation with a plus operation `_+_` on them.¹

```
Maude> fmod SIMPLE-NAT is
  sort Nat .
  op zero : -> Nat .
  op s_ : Nat -> Nat .
  op _+_ : Nat Nat -> Nat .
  vars N M : Nat .
  eq zero + N = N .
  eq s N + M = s (N + M) .
endfm
```

Fortunately, we do not need to write our modules in the prompt. We can input one or several modules by saving them in a file and then entering the file with the `in` or `load` commands (the only difference between both commands is that the latter does not produce detailed output as modules are entered). Assuming that the file `my-nat.maude` contains the module `SIMPLE-NAT` above, we can do the following to enter it:

```
Maude> load my-nat.maude
```

After entering the module `SIMPLE-NAT` into Maude, we can, for example, reduce the term `s s zero + s s s zero` (which is the equivalent in Peano notation of the more usual $2 + 3$) as follows:

```
Maude> reduce in SIMPLE-NAT : s s zero + s s s zero .
reduce in SIMPLE-NAT : s s zero + s s s zero .
rewrites: 3 in 0ms cpu (0ms real) (~ rews/sec)
result Nat: s s s s zero
```

It is not necessary to give the name of the module in which to reduce a term explicitly. All commands that require a module refer to the current module by default, unless a module is explicitly given. The current module is usually the last module entered or used, although we can use the `select` command to select a module to be the current one (see Section 25.13).

```
Maude> reduce s s zero + s s s zero .
reduce in SIMPLE-NAT : s s zero + s s s zero .
rewrites: 3 in 0ms cpu (0ms real) (~ rews/sec)
result Nat: s s s s zero
```

¹ We do not display the `>` symbol that Maude adds at the beginning of each line.

Any action happening in the Maude system can be interrupted by typing control-C. In particular, by hitting control-C during a reduction in progress, such reduction is interrupted and the system gets into debugging mode (see Section 24.1.3).

Although it is not the case in the simple examples above, sometimes we get a very big term as output from Maude. In some cases, in order to make it easier to read and understand, we edit the presentation of the outputs given by Maude.

When you execute `maude.linux`, the file `prelude.maude`, which includes several predefined modules (see Chapter 9), is automatically loaded. To find `prelude.maude`, the Maude interpreter checks for it in several directories, in the following order:

1. the directories specified in the `MAUDE_LIB` environment variable,
2. the directory containing the executable, and
3. the current directory.

It is a good idea to include the path to `prelude.maude` in the `MAUDE_LIB` environment variable to be sure that it will always be found, because the executable finding code may not find the directory containing the executable.

Among the predefined modules included in `prelude.maude` we find a module `STRING` that declares sorts and operations for manipulating strings. In particular, `STRING` introduces the operation `_+_` to concatenate two strings. Then, to concatenate the strings “hello”, “ ”, and “world”, you can type at the Maude prompt the following `red` (which is the abbreviated form of `reduce`) request:

```
Maude> red in STRING : "hello" + " " + "world" .
reduce in STRING : "hello" + " " + "world" .
rewrites: 2 in 0ms cpu (0ms real) (~ rews/sec)
result String: "hello world"
```

Actually, although `STRING` is not the current module right after starting the system, it is imported by the current one, `CONVERSION`. Thus, we can type the following, just after starting Maude:

```
Maude> red "hello" + " " + "world" .
reduce in CONVERSION : "hello" + " " + "world" .
rewrites: 2 in 0ms cpu (0ms real) (~ rews/sec)
result String: "hello world"
```

Notice that Maude makes explicit the module in which the term is reduced, even when no module name is given by the user.

As said above, to load for example a user-defined module `HELLO-WORLD` for a Maude session, you can either type at the Maude prompt the whole module or simply type the following `in-roduce` request:

```
Maude> in hello-world
```

where `hello-world` is a text file in the current directory containing the module `HELLO-WORLD`.

For files specified by a bare file name, Maude also checks for files with `.maude`, `.fm`, and `.obj` extensions. Maude can also be told using the `MAUDE_LIB` environment variable about other directories to use to search for files. Thus to find a file specified in the `in` command, Maude searches, in order:

1. the current directory,
2. the directories in the `MAUDE_LIB` environment variable, and
3. the directory containing the executable.

If the desired file is in none of these places you must type its full path name.

As for user-defined modules, user requests such as the above can either be typed at the Maude prompt or simply `in`-troduced with a text file containing them. In fact, many users run Maude inside an Emacs-like editor, since this provides both text editing facilities for creating Maude modules and saving them in files, and also UNIX shell interaction to start a Maude session and to `in`-troduce during the session modules and commands created and saved in files, as shown in Figure 2.2.

Note that text files entered in Maude can contain not only modules, but also any command. Actually, a file can contain as many modules and commands as one wishes. When entering it with an `in` or `load` command, Maude will read them one after another as if they were written at the prompt of the system. Another important issue worth pointing out is that we can write single line and multiline comments anywhere inside a module or a file. Single line comments are started by either `***` or `---`, and ended by the end of line. Multiline comments are started by `***(` and ended by `)`. Parentheses must balance within multiline comments.

2.3 Getting support and more information

We maintain the following mailing lists related to Maude:

- `maude-users@maude.cs.uiuc.edu`. A moderated list for the discussion of topics of general interest to all Maude users. This list is typically low-traffic, and contains items such as calls for papers, announcements of new Maude related papers, and notifications of new releases of Maude. It is important that you subscribe to this list if using Maude, as this is the mechanism by which we will make important announcements about the system. To subscribe, or to view the archived messages, please go to

<http://maude.cs.uiuc.edu/mailman/listinfo/maude-users/>

- `maude-help@maude.cs.uiuc.edu`. This is an alias for submitting questions about any aspect of the use of Maude. Messages are distributed to a group of experienced users who have offered to provide help. This list is

```

Emacs - *shell*
aeris:~/Desktop miguelpt$ maude
\|/
  --- Welcome to Maude ---
  /|/
Maude 2.2 built: Nov 28 2005 16:08:20
Copyright 1997-2005 SRI International
Mon Jul 10 14:23:53 2006
Maude> load my-nat.maude
Maude> red s s zero + s s s zero .
reduce in MY-NAT : s s zero + s s s zero .
rewrites: 3 in 0ms cpu (0ms real) (~ rewrites/second)
result Nat: s s s s zero
Maude>

-1:** *shell* All (13,7) (Shell:run)-----
fmod MY-NAT is
  sort Nat .
  op zero : -> Nat .
  op s_ : Nat -> Nat .
  op _+_ : Nat Nat -> Nat .
  vars N M : Nat .
  eq zero + N = N .
  eq s N + M = s (N + M) .
endfm
-:-- my-nat.maude All (10,0) (Text)-----

```

Fig. 2.2. Running Maude inside Emacs

not open for subscription, but you can send messages to this list at any time. Questions posted here will be automatically archived at

<http://maude.cs.uiuc.edu/pipermail/maude-help/>

- maude-bugs@maude.cs.uiuc.edu. A list for reporting any problems you experience with Maude (see below), and also any suggestions for enhancements and improvements.

2.4 Reporting bugs in Maude

As already mentioned, bug reports should be sent to

maude-bugs@maude.cs.uiuc.edu

When submitting a bug report, please include the following information:

1. *Example to reproduce the bug*. Ideally this should be a single file that reproduces the bug by loading it. If your example is large and spread

out in multiple files, have a file `top.maude` that loads files and executes commands as necessary to reproduce the bug. Send all the files as a tar archive, optionally compressed with `gzip`.

If Maude's output is not obviously wrong (for example, an "internal error" message), include an explanation of why the output is wrong.

If you choose to simplify the example, note that a short runtime to expose the bug is desirable. A small example text is mostly unimportant unless it is necessary to understand such example text in order to understand why Maude's output is incorrect.

2. *Version of Maude used.* Make sure you provide information of the concrete release of Maude (and Full Maude if it is the case). If you are not using one of the ready-made binaries released by the Maude team, also give the versions of the compiler and tools used to build it and the libraries linked against.
3. *Platform.* Include the operating system type and version number, as well as the processor type.

Syntax and Basic Parsing

This chapter introduces the basic syntactic ingredients of all Maude specifications: identifiers, module names, sort names, and operator declarations. Other syntactic parts of Maude specifications, like equations and rules, will appear in the following chapters.

Some syntax is presented in an informal way by means of general schemes; a formal BNF grammar of the language can be found in Chapter 26.

The chapter finishes explaining some features that can be used to reduce parsing ambiguities in the user-definable syntax, including mixfix operator declarations, supported by Maude.

3.1 Identifiers

In Core Maude, identifiers are the basic syntactic elements, used to name modules and sorts, and to form operator names. For example, `NAT`, `Nat`, and `hello-world` are identifiers. In general, an identifier in Maude is any finite sequence of ASCII characters such that:

- It does not contain any white space. For example, the sequence `'abc def'` is not one identifier, but two.
- The characters `'{', '}', '(', ')', '[', ']'` and `'.'` are *special*, in that they break a sequence of characters into several identifiers. For example, the sequence `ab{c,d}ef` counts as *seven* identifiers, namely, `ab`, `{`, `c`, `,`, `d`, `}`, and `ef`.
- The backquote character `'`'` is used as an *escape character* to indicate that a blank space or the special characters do not break the sequence. Consequently, backquotes can *only* appear immediately *before* any of the special characters, or *between* two non-empty strings of characters—with neither the ending of the first string nor the beginning of the second string being another backquote—for exactly these purposes. For example, `1'ab'c',d'ef` is a single identifier. Maude's pretty printer will display such an identifier in the form `1 ab{c,d}ef`.

Nonprinting characters in strings use C backslash conventions [206, Section A2.5.2].

3.2 Modules

In Maude the basic units of specification and programming are called *modules*.¹ A module consists of syntax declarations, providing appropriate language to describe the system at hand, and of statements, asserting the properties of such a system. The syntax declaration part is called a *signature* and consists of declarations for:

- *sorts*, giving names for the types of data,
- *subsorts*, organizing the data types in a hierarchy,
- *kinds*, that are implicit and intuitively correspond to “error supertypes” that, in addition to normal data, can contain “error expressions,” and
- *operators*, providing names for the operations that will act upon the data and allowing us to build expressions (or terms) referring to such data.

We use symbols Σ, Σ' , etc. to denote signatures.

In Core Maude there are two kinds of modules: *functional modules* and *system modules*. Signatures are common for both of them. The difference between functional and system modules resides in the statements they can have:

- functional modules admit *equations*, identifying data, and *memberships*, stating typing information for some data, while
- system modules also admit *rules*, describing transitions between states, in addition to equations and memberships.

We use E, E' , etc. to denote sets of equations and memberships, and R, R' , etc. to denote sets of rules.

From a programming point of view, a *functional module* is an equational-style functional program with user-definable syntax in which a number of sorts, their elements, and functions on those sorts are defined. From a specification viewpoint, a functional module is an *equational theory* (Σ, E) with initial algebra semantics. Functional modules are described in detail in Chapter 4, here we just discuss some of their top-level syntax. Each functional module has a *name*, which is a Maude identifier. Any Maude identifier can be used, but the preferred style for module names is an all capitalized identifier, and in the case of a compound name the different parts are linked with hyphens. For example, a module defining numbers and operations on them can be called `NUMBERS`. The top-level syntax will then be

¹ As explained in Section 8.3.1, specifications can also be given in *theories*, with a syntax entirely similar to that of modules, but theories, unlike modules, need not be executable.

```
fmod NUMBERS is
...
endfm
```

with ‘...’ corresponding to all the declarations of submodule importations, sorts, subsorts, operators, variables, equations, and so on.

From a programming point of view, a *system module* is a declarative-style concurrent program with user-definable syntax. From a specification viewpoint, it is a *rewrite theory* (Σ, E, ϕ, R) (where ϕ specifies the frozen arguments of operators in Σ ; see Section 4.4.9) with initial model semantics. Again, each system module has a *name*, which is a Maude identifier. And as for functional modules, the preferred style is an all capitalized name, with consecutive parts linked with hyphens in the case of compound names. For example, a module specifying the behavior of a vending machine may be called VENDING-MACHINE. It will then be introduced with the following top-level syntax:

```
mod VENDING-MACHINE is
...
endm
```

where again ‘...’ corresponds to all the declarations of submodule importations, sorts, subsorts, operators, variables, equations, rules, and so on. System modules are described in detail in Chapter 6.

In the rest of the chapter we will describe the ingredients of signatures, that is, the syntactic elements common to both functional and system modules, such as sorts, subsorts, kinds, operators, variables, and the terms that can be built on a signature, postponing the discussion about the syntax specific to functional and system modules to Chapters 4 and 6, respectively.

3.3 Sorts and subsorts

The first thing a specification needs to declare are the types (that in the algebraic specification community are usually called *sorts*) of the data being defined and the corresponding operations. Sorts can be partially ordered via a *subsort* relation.

A sort is declared using the `sort` keyword followed by an identifier (the sort name), followed by white space and a period, as follows:

```
sort <Sort> .
```

and multiple sorts may be declared using the `sorts` keyword, as follows:

```
sorts <Sort-1> ... <Sort-k> .
```

The period at the end of the sort declaration, as for the other types of declarations, is crucial. Note that if either the period is missing or no space is left before and after the period, there can be parsing problems or unintended behavior. For example, the following declaration is syntactically correct but causes an unintended interpretation because of a missing ‘.’, since this way sorts A, B, `sort`, and C are declared.

```
sorts A B
sort C .
```

Note also that the keywords `sort` and `sorts` are synonyms. One may use `sort` for multiple sort declarations and `sorts` for single ones, although we do not encourage this style.

For example, we can declare sorts `Zero`, `NzNat`, and `Nat` in the `NUMBERS` module, either one at a time

```
sort Zero .
sort NzNat .
sort Nat .
```

or all at once

```
sorts Zero Nat NzNat .
```

The identifiers `<`, `->`, and `~>` cannot be used as sort names. Moreover, identifiers used for sorts cannot contain any of the characters ‘:’, ‘.’, ‘[’, or ‘]’. The reasons for these restrictions will become clear below in this section and in Sections 3.4, 3.5, and 14.2.1. The use of ‘{’, ‘}’, and ‘,’ is only allowed in structured sort names (see below). Although any so restricted identifier is a legal sort name, the preferred style is to capitalize the *first* letter of the name. Furthermore, in the case of a compound name, such as a sort of nonzero naturals, the names (each with the first letter capitalized) or suitable abbreviations will be *juxtaposed* without spaces or hyphens, like, for example, `NzNat`.

A sort name can also be structured. Structured sort names are used in parameterized modules; for example, we may use `List{X}` for a parameterized list sort with parameter `X` and `List{Nat}` for its instantiation to lists of natural numbers (see Section 8.3.3). A *structured sort name* contains at least one pair of curly brace symbols ‘{’ and ‘}’, and is constructed according to the following BNF grammar, without any white space between terminals:

$$\begin{aligned} \langle \textit{Sort} \rangle & ::= \langle \textit{sort identifier} \rangle \\ & \quad | \langle \textit{Sort} \rangle \{ \langle \textit{SortList} \rangle \} \\ \langle \textit{SortList} \rangle & ::= \langle \textit{Sort} \rangle \\ & \quad | \langle \textit{SortList} \rangle , \langle \textit{Sort} \rangle \end{aligned}$$

Notice that structured sorts *are* allowed to contain ‘{’, ‘,’ and ‘}’ but only in accordance with the above grammar. Thus all the following are structured sort names:

```

a{X}
a{X, Y}
a{b, c{d}}{e}
a{ }

```

while the following are *not* legal sort names:

```

{X}           (lacks sort identifier prefix)
a(X, Y)      (',' not inside braces)
a{b, {d}}{e} ({d} lacks sort identifier prefix)
a({)        ('{' without closing '}')

```

Structured sort names can be written in an equivalent *single-identifier form* by using backquotes. For example, the sort `a{b, c{d}}{e}` may be written as `a`{b`,c`{d}`}`{e`}``. Hybrid notation such as `a{b`,c}` is not allowed. When backquotes are omitted, the sort name becomes a sequence of tokens according to Maude's usual tokenization rules and arbitrary white space may be inserted between tokens. For example, `Foo`{X`,Y`}``, `Foo{X,Y}`, and `Foo{X, Y}` are three equivalent forms for the same structured sort name.

Structured sort names must be written in their equivalent single-identifier form inside operator hooks (see Chapter 9) or in metasyntax (see Chapter 14).

Apart from their special syntax and their use as parameterized sorts in parameterized modules (see Section 8.3.3), structured sort names behave just like sort identifiers.

The subsort relation on sorts parallels the subset relation on the sets of elements in the intended model of these sorts. Subsort inclusions are declared using the keyword `subsort`. The declaration

```
subsort <Sort-1> < <Sort-2> .
```

states that the first sort is a subsort of the second. For example, the declarations

```
subsort Zero < Nat .
subsort NzNat < Nat .
```

specify that the sorts `Zero` (containing only the constant 0) and `NzNat` (the nonzero natural numbers) are subsorts of `Nat`, the natural numbers. More than one subsort relationship can be declared using the keyword `subsorts`, as follows:

```
subsorts <Sort-1> ... <Sort-j> < ... < <Sort-k> ... <Sort-l> .
```

Then, the above declarations can be given in a single declaration as follows:

```
subsorts Zero NzNat < Nat .
```

If we extend `NUMBERS` with sorts `Int` and `NzInt` we can express the additional subsort relationships compactly by

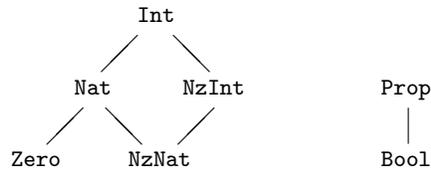
```

sorts NzInt Int .
subsorts NzNat < NzInt Nat < Int .

```

A set of subsort declarations must define a *partial order* among the set of sorts. For this to be true, the user is required to avoid *cycles* in the subsort declarations. For example, if a sort **A** is declared as a subsort of **B**, and **B** is declared as a subsort of **A**, we would have a cycle.

Note that the partial order of subsort inclusions partitions the set of sorts into *connected components*, that is, into sets of sorts that are directly or indirectly related in the subsort ordering. For example, all the above sorts **Zero**, **Nat**, **NzNat**, **NzInt**, and **Int** belong to the same connected component in the subsort ordering, whereas a sort **Bool** would clearly belong to a different connected component and could have other sorts, for example a supersort **Prop** of propositions, related to it in the same component. Intuitively, connected components gather together related sorts of data such as numerical data, truth-value data, and so on. Graphically, we can visualize the partial order of subsort inclusions as an acyclic graph (the corresponding *Hasse diagram*), and then the connected components are exactly those of the underlying graph, as in the following example:



3.4 Operator declarations

In a Maude module, an operator is declared with the keyword **op** followed by its *name*, followed by a colon, followed by the list of sorts for its arguments (called the operator's *arity* or *domain sorts*), followed by **->**, followed by the sort of its result (called the operator's *coarity* or *range sort*), optionally followed by an attribute declaration (the discussion of operator attributes is postponed to Section 4.4), followed by white space and a period. Thus the general scheme has the form

```

op <OpName> : <Sort-1> ... <Sort-k> -> <Sort> [<OperatorAttributes>] .

```

Here are some operator declarations for our **NUMBERS** module.

```

op zero : -> Zero .
op s_ : Nat -> NzNat .
op sd : Nat Nat -> Nat .
ops _+_ *_ : Nat Nat -> Nat .

```

If the argument list is empty, the operator is called a *constant*. Thus `zero` is a constant.

The name of the operator is a string of characters that may consist of several identifiers, due to the presence of blanks or other special characters. Underscores (`_`) play a special role in these strings. If no underscore character occurs in the operator string—as in the case of the operator `sd` above—then the operator is declared in *prefix* form. If underscore characters occur in the string, then their number must coincide with the number of sorts declared as arguments of the operator (in particular, constant names cannot include any underscore character). The operator is then in *mixfix* form, with the n -th underscore indicating the place where arguments of the n -th sort must be placed in expressions formed with that operator. In the above example the operators `s_`, `+_`, and `*_` are in mixfix form.

There may or may not be any other characters before or after any of the underbars. If no other characters appear, we say that the operator has been declared with *empty syntax*. For example, we could declare a sort `NatSeq` of sequences of natural numbers formed with empty syntax as follows:

```
sort NatSeq .
subsort Nat < NatSeq .
op __ : NatSeq NatSeq -> NatSeq [assoc] .
```

where `assoc` is an attribute declaring that sequence concatenation is associative (see Section 4.4.1). With this operator declaration we can write number sequences such as

```
zero (s zero) (s s zero)
```

Operators having the same arity and coarity can be declared simultaneously by using the keyword `ops` and giving the non-empty list of their corresponding names after the `ops` keyword and before the `:`, as is done for the declarations of `+_` and `*_` in the example above.

An operator can also be declared using *several identifiers*. This can be due to the presence of special characters, or to blank spaces, or both. Consider for example the operator declaration

```
op [_] and then [_] : Command Command -> Command .
```

that may allow a natural language style in the syntax of a programming language. It uses eight identifiers in the Maude sense, but declares a single binary operator, with the underscores indicating the place of the arguments in the mixfix notation. Internally, Maude also associates to this operator a corresponding *single-identifier form* by using backquotes. We could have equivalently defined the operator using the single-identifier form, namely,

```
op ‘[_]and‘then‘[_] : Command Command -> Command .
```

Of course, both variants are equivalent and have the same mixfix display, but the version without backquotes is obviously more convenient.²

The declaration of an operator requires an extra pair of parentheses if we already use parentheses as part of the syntax of the operator. Suppose we had in a programming language a binary operator (`_ only after _`). Then, we have to declare it as follows:

```
op ((_ only after _)) : Command Command -> Command .
```

Since an operator may be declared using several identifiers, in an `ops` declaration involving several operators each operator declaration can be enclosed in parentheses if necessary, to indicate where the syntax of each operator begins and ends. We could have declared both operators together, as follows:

```
ops ([_] and then [_]) ((_ only after _)) :
  Command Command -> Command .
```

Thus, one or several Maude identifiers can be used in operator declarations. Regarding style, the preferred one, particularly for single-identifier operators with prefix syntax, is to use lower case names. However, for a composed name such as a *meta parse* operator, the subsequent names will be juxtaposed and will typically begin with a capital letter to enhance readability, e.g., `metaParse`.

3.5 Kinds

The equational logic underlying Maude is membership equational logic [248, 31]. In this logic sorts are grouped into equivalence classes called *kinds*. For this purpose, two sorts are grouped together in the same equivalence class if and only if they belong to the same connected component. Maude sorts are user-defined, while kinds are implicitly associated with connected components of sorts and are considered as “error supersorts.” Terms (see Section 3.8) that have a kind but not a sort are understood as *undefined* or *error* terms.

In Maude modules, kinds are not independently and explicitly named. Instead, a kind is identified with its equivalence class of sorts and can be named by enclosing the name of one or more of these sorts in square brackets `[. . .]`; when using more than one sort, they are separated by commas.

For example, suppose we add a partial predecessor function to our `NUMBERS` module,

```
op p : NzNat -> Nat .
```

² In Full Maude, operator names in operator declarations must be given as single identifiers. Multiple-identifier names are also supported, but their equivalent single-identifier form must be used in their declarations.

Then Maude will parse the term `p(zero)` and assign it the kind `[Nat]`, or equivalently `[NatSeq]` or also `[Nat, NatSeq]`, since the sorts `Nat` and `NatSeq` belong to the same connected component. Although any sort, or list of sorts in the connected component, can be enclosed in brackets to denote the corresponding kind, Maude uses a canonical representation for kinds; specifically, Maude prints the kind using a comma-separated list of the *maximal elements* of the connected component.

The Maude system also lifts automatically to kinds all the operators involving sorts of the corresponding connected components to form *error expressions*. Such error expressions allow us to give expressions to be evaluated the *benefit of the doubt*: if, when they are simplified, they have a legal sort, then they are okay; otherwise, the fully simplified error expression is returned, which the user can interpret as an error message. Equational simplification can also occur at the kind level, so that operators can map error terms to defined terms, which may be useful for error recovery.

It is also possible to explicitly declare operators at the kind level. This corresponds to declaring a partial operation, which is defined for those argument values for which Maude can determine that the resulting term has a sort. Note that the operation is considered to be total at the kind level. As an example, consider the following fragment of a graph specification:

```
sorts Node Edge .
ops source target : Edge -> Node .
sort Path .
subsort Edge < Path .
op _;_ : [Path] [Path] -> [Path] .
```

The sorts `Node` and `Edge`, along with the `source` and `target` operators mapping edges to nodes, axiomatize the basic graph concepts. The sort `Path` is intended to be the paths through the graph, sequences of edges with the target of one edge being the source of the next edge. Edges are singleton paths, and `_;_` denotes the *partial* concatenation operation, indicated by giving kinds rather than sorts in the argument list. Later, in Section 4.3, we will see how to specify when a sequence of edges has sort `Path`.

To emphasize the fact that an operator defined at the kind level in general defines only a *partial* function at the sort level, Maude also supports a notational variant in which an (always total) operator at the kind level can equivalently be defined as a partial operator between sorts in the corresponding kinds, with syntax ‘`~>`’ instead of ‘`->`’ to indicate partiality. For example, the above operator declaration can be equivalently specified by

```
op _;_ : Path Path ~> Path .
```

More generally, the partial operator declaration

```
op <OpName> : <Sort-1> ... <Sort-k> ~> <Sort> .
```

is equivalent to the total operator declaration at the kind level

```
op <OpName> : [<Sort-1>] ... [<Sort-k>] -> [<Sort>] .
```

3.6 Operator overloading

Operators in Maude can be *overloaded*, that is, we can have several operator declarations for the same operator with different arities and coarities. Consider extending our number module with a new sort `Nat3` (of natural numbers modulo 3), constants 0, 1, and 2 of sort `Nat3`, and two further operator declarations for `_+_`.

```
op _+_ : NzNat Nat -> NzNat .
sort Nat3 .
ops 0 1 2 : -> Nat3 .
op _+_ : Nat3 Nat3 -> Nat3 .
```

Now `_+_` is overloaded, having three declarations. However, there are two different kinds of overloading present in the example. The additional declaration of `_+_` with first argument `NzNat` is an example of *subsort overloading*. Here the two `_+_` operators on `Nat` and `NzNat` are supposed to have the same behavior on their shared argument values, that is, the operator on the subsort `NzNat` is the *restriction* of the operator on the larger sort `Nat`. The main point of such declarations is to give more sort information, for example that the result of adding a nonzero natural number to any natural number is nonzero. Many more examples of this form of overloading can be found in the predefined data modules for the number hierarchy (Chapter 9) and in other modules throughout the book.

In contrast, the sorts `Nat` and `NzNat` on the one hand, and the sort `Nat3` on the other belong to two different *connected components* in the subsort ordering and therefore natural number addition and addition modulo 3 are semantically unrelated. This form of overloading is called *ad-hoc overloading*. Both subsort and ad-hoc overloading of operators are allowed in Maude. However, to avoid ambiguous expressions we require that if the sorts in the arities of two operators with the same syntactic form are pairwise in the same connected components, then the sorts in the coarities must likewise be in the same connected component.

Strictly speaking, this requirement would rule out ad-hoc overloaded constants. For this reason, we have declared two different constants `zero` and `0` for the corresponding zero elements. However, this requirement can be relaxed, and it is often natural to do so. For example, the constants of a parameterized module (see Chapter 8.3) can appear in many different connected components for different instances of the module, and it may be cumbersome to rename them all. To allow this relaxation, constants—and, more generally, terms (see Section 3.8)—can be *qualified by their sort*, by enclosing them in parentheses followed by a dot and the sort name. In this way, we could have instead

declared 0 as an ad-hoc overloaded constant for natural numbers and for natural numbers modulo 3, and could then disambiguate the expression $0 + 0$ by writing, for example, $0 + (0).\text{Nat}$ and $0 + (0).\text{Nat3}$, or $(0 + 0).\text{Nat}$ and $(0 + 0).\text{Nat3}$.

3.7 Variables

A variable is constrained to range over a particular sort or kind. Variables can be declared on-the-fly in Maude with syntax consisting of an identifier (the variable name), a colon, and another identifier (its sort) or kind expression (its kind). For example, $N:\text{Nat}$ declares a variable named N of sort Nat , and $X:[\text{Nat}]$ declares a variable named X of kind $[\text{Nat}]$.

The scope of an on-the-fly variable declaration is the declaration's occurrence. Thus *each* such variable must be accompanied by its sort or kind.

A variable can also be declared in a module using the keyword `var` followed by an identifier (the variable name), followed by a colon with white space before and after, followed by an identifier (its sort) or kind expression (its kind), followed by white space and a period.

```
var N : Nat .
var X : [Nat] .
```

The scope of such a declaration is the entire module. It has the effect of replacing occurrences of N and X by the on-the-fly versions $N:\text{Nat}$ and $X:[\text{Nat}]$.

Multiple variables of the same sort can be declared using the keyword `vars`.

```
vars M N : Nat .
vars X Y : [Nat] .
```

Both upper and lower case names for variables are possible. However, upper case variable names are more customary in Maude. The syntactic conventions for the acceptable names of variables in variable declarations are the same as those for constant operators, that is, for operators with empty arity. In particular, the underscore ‘`_`’ cannot be used in the name of a variable, but the colon ‘`:`’ can; thus the scanning for ‘`:`’ in order to extract the appropriate sort or kind from an on-the-fly variable declaration is done from right to left.

3.8 Terms and preregularity

A term is either a constant, a variable, or the application of an operator to a list of argument terms. The sort of a constant or variable is its declared sort. In the application of an operator, the argument list must agree with the declared arity of the operator. That is, it must be of the same length, and

each term must have sort (or at least kind) in the connected component of the corresponding declared argument sort. Using prefix form—which can always be used for *any* operator, regardless of having been declared with either prefix or mixfix syntax—the syntax of operator application is the operator’s name followed by ‘(’, followed by a list of argument terms separated by commas, followed by ‘)’. Here are some examples of prefix notation from our numbers module.

```
s_(zero)
s_(sd(N:Nat, M:Nat))
p(s_(zero))
_+_ (N:Nat, M:Nat)
```

The application of an operator declared with mixfix form also has a mixfix syntax: the operator’s mixfix name with each underscore replaced by the corresponding term from the argument list. The mixfix form of the above examples is

```
s zero
s sd(N:Nat, M:Nat)
p(s zero)
N:Nat + M:Nat
```

The *kind* of a term is the result kind of its topmost operator. For example, the kind of `p(s zero)` is `[Nat]`, since `Nat` is the result sort of `p`. If a module’s grammar is unambiguous (see the discussion on parsing in the following section), then each term has a single kind. But we can also associate *sorts* to terms. In general, even if the grammar is unambiguous, a term may have several sorts, due to the subsort ordering. Specifically, constants have the sort they are declared with and any supersort of it. Given a term of the form $f(t_1, \dots, t_n)$, if t_i has sort s_i for $i = 1, \dots, n$ and there is an operator declaration $f : s_1 \dots s_n \rightarrow s$, then the term $f(t_1, \dots, t_n)$ has sort s and any of its supersorts. For example, in our example `NUMBERS` module the term `s s 0` has sorts `NzNat` and `Nat`.

A very desirable property of a module is that each term has a *least* sort that can be assigned to it. Such a least sort gives us the most detailed information on how to classify such a term as a data element. For example, the least sort of the term `s s 0` is `NzNat`, and this gives us the most precise classification of such a term in the sort hierarchy. Given an arbitrary signature Σ , we can have terms that fail to have a least sort. However, if Σ satisfies a simple syntactic property called *preregularity* [173], we can guarantee that any Σ -term will have a least sort. We call Σ *preregular* if for each n , given an n -argument function symbol f and sorts s_1, \dots, s_n such that $f(x_1 : s_1, \dots, x_n : s_n)$ is a well-formed Σ -term, then there is a least sort s among all the sorts s' appearing in (possibly overloaded) operator declarations of the form $f : s'_1, \dots, s'_n \rightarrow s'$ in Σ such that for $1 \leq i \leq n$ we have $s_i \leq s'_i$. For example, the signature

```

sorts A B C D .
subsorts A < B C < D .
op a : -> A .
op f : B -> B .
op f : C -> C .

```

fails to be preregular, because for the sort A the term $f(X:A)$ is a well-formed term, but there is no least sort for the result of f with arguments greater or equal to A , since either B or C can be chosen as result sorts, and they are incomparable in the sort hierarchy. As a consequence, both $f(X:A)$ and $f(a)$ do not have a least sort: they have sorts B , C , and D , and B and C are minimal sorts among those sorts.

As already mentioned in Section 3.4 for the `assoc` attribute and further explained in Section 4.4.1, operators can be declared with equational axioms such as associativity (`assoc`), commutativity (`comm`), and identity (`id:`). This means that, if we denote by A the corresponding associativity and/or commutativity, and/or identity equations, we are not really interested in syntactic terms t , but rather in equivalence classes modulo A , that is, in the equivalence class $[t]_A$ of each term t , since all representatives of the class are viewed as equivalent representations. Preregularity *modulo* A now means that we can assign a least sort not just to any well-formed term t , but also to its equivalence class $[t]_A$. As further explained in Section 24.2.5, Maude assumes that modules are preregular modulo whatever axioms such as `assoc`, `comm`, and `id:` have been declared for operators, checks syntactic conditions ensuring preregularity modulo A , and generates warnings when a module fails to satisfy such preregularity conditions.

A *ground term* is a term containing no variables: only constants and operators. Intuitively, ground terms denote either data in case no equations apply to the term (for example, `s zero` is data) or functional expressions indicating how an equationally defined function is applied to data (for example, `(s zero) + (s zero)`). Ground terms modulo equations constitute the initial algebra associated with a specification, as discussed later in Section 4.3.

3.9 Parsing

As seen in previous sections, the Maude language supports user-definable syntax including mixfix operator declarations. Parsing is done in stages using a bison/flex-based parser for Maude's surface syntax, a grammar generator which generates the context-free grammar for the user-defined mixfix parts of a Maude module over the user's signature, and the MSCP context-free parser (generator) that generates a parser for the module's context-free grammar. MSCP was developed by J. Quesada [310, 309].

With mixfix syntax, the occurrence of ambiguities in the parsing of terms is very common. Of course, we can always provide unambiguous grammars,

which are frequently surprisingly large, or use parentheses for breaking the possible ambiguities. But usually we would like to have a more powerful alternative. Maude reduces such ambiguities by using a mechanism based on *precedence values* and *gathering patterns*.

Let us assume the following declarations for some arithmetic expressions:

```
sort Nat .
ops 1 2 3 : -> Nat .
ops _+_ *_ : Nat Nat -> Nat .
```

An expression like $1 + 2 * 3$ is ambiguous, since both $(1 + 2) * 3$ and $1 + (2 * 3)$ are valid parses. This kind of ambiguity is usually solved by assigning a *precedence* to each of the operators. In Maude, the precedence of an operator is given by a natural number,³ where a lower value indicates a tighter binding.

Operator precedence then defines how an expression should be parsed when several operators are present. We can assign a precedence to an operator with a `precedence` (abbreviated `prec`) attribute, which takes the precedence value as an argument. For example, one would expect multiplication to be evaluated before addition. Thus, we can give precedences, e.g., 33 and 31 to the operators `_+_` and `_*_`, respectively, as follows:

```
op _+_ : Nat Nat -> Nat [prec 33] .
op *_ : Nat Nat -> Nat [prec 31] .
```

The term $1 + 2 * 3$ is now unambiguous: its only possible parse is $1 + (2 * 3)$.

Precedence can be overridden using parentheses; we can always write $(1 + 2) * 3$ in case this is the term we are interested in. For those operators for which the user does not specify a precedence value, a default one is given (see Section 3.9.1 for a discussion on the default precedence values). For example, both operators `_+_` and `_*_` above get 41 as their default precedence, and hence the ambiguity.

The precedence mechanism is not enough, however. For example, the expression $1 + 2 + 3$ is still ambiguous, because both parses $(1 + 2) + 3$ and $1 + (2 + 3)$ are possible. Usually, programming languages define a way of associating operators to solve this kind of problems, so that the *associativity* of the operators determines which is evaluated first. For example, addition usually is left-associative, and therefore we expect to parse it as $(1 + 2) + 3$. In Maude, we can specify not only the associativity of operators, but general *gathering patterns* for each operator.

The gathering pattern of an operator restricts the precedences of terms that are allowed as arguments. We give a (non-empty) sequence of as many `E`, `e`, or `&` values as the number of arguments in the operator, that is, one of these values for each argument position:

³ The maximum allowed precedence value is $2^{31} - 1$.

- **E** indicates that the argument must have a precedence value lower or equal than the precedence value of the operator,
- **e** indicates that the argument must have a precedence value strictly lower than the precedence value of the operator, and
- **&** indicates that the operator allows any precedence value for the corresponding argument.

In fact, the precedence values work because of their combination with the gathering patterns. For example, the precedence values given to `+_` and `*_` work as expected because their *default* gathering pattern is `(E E)` (see Section 3.9.2), which forces them to be applied only to terms of smaller or equal precedence value. Thus, `1 + (2 * 3)` is a valid parse for `1 + 2 * 3`. On the other hand, since the precedence of a term is given by the precedence of its top operator, `(1 + 2) * 3` is not a valid parse for `1 + 2 * 3`, because the term `1 + 2` has precedence value `33`, which is greater than the precedence of `*_`.

Moreover, by default, all constants have precedence 0 (see Section 3.9.1), and therefore they are also valid arguments for both operators.

We can specify `+_` and `*_` as left-associative by giving to them gathering pattern `(E e)`.

```
op _+_ : Nat Nat -> Nat [prec 33 gather (E e)] .
op *__ : Nat Nat -> Nat [prec 31 gather (E e)] .
```

In this way, we force the second argument of these operators to be of a strictly lower precedence. Then, a term with `+_` as top operator (or any other operator with the same precedence) like `2 + 3` is nonvalid as second argument for `+_`. But it would be valid as first argument, since terms with equal precedence are allowed. Now the only possible parse for the expression `1 + 2 + 3` is `(1 + 2) + 3`.

Note that parentheses could be described as an operator `(_)` with precedence 0 and gathering pattern `(&)`. Thus, any term can appear inside parentheses, and any subterm of a term can be enclosed in parentheses.

3.9.1 Default precedence values

Maude associates default precedence values to those operators for which the user does not specify this information as part of the operator declaration. The default precedence values are entirely similar to those used by OBJ3 [174]. The rules for the assignment of default precedence values are:

- Operators with standard form (constants and prefix operators) always have precedence 0, regardless of user settings. The user cannot change the precedence value or gathering pattern for operators in standard form.
- Mixfix operators which begin and end with something different from an underbar have precedence 0. Operators as, for example, `(_)`, `<_:_|_>`, and `if_then_else_fi` follow this rule.

- Mixfix operators which begin or end with an underbar have precedence 15 for a unary operator and 41 for everything else. Note that this ‘or’ is exclusive. Operators like, e.g., `not_`, `_!`, or `to_:_` fall into this category.
- Mixfix operators which begin and end with an underbar have precedence 41. This rule applies, e.g., to the operators `__`, `_+_`, `_*_`, and `_?_:_`.

3.9.2 Default gathering patterns

As for precedence values, Maude assigns default gathering patterns to all those operators for which the user does not specify this information as part of the operator declaration. The default gathering patterns are also entirely similar to those used by OBJ3 [174]. The rules for the assignment of the default gathering patterns are:

- All arguments of prefix operators have a gathering value `&`, regardless of the user specification.
- If the underbar corresponding to an argument is not adjacent to another underbar, and it is neither the leftmost nor the rightmost token in the operator, then the default gathering value for such an argument is `&`. In other words, if an underbar appears between tokens different from the underbar, then its corresponding argument will have this default gathering pattern. For example, the default gathering pattern for the operator `if_then_else_fi` is `(& & &)`, the default gathering pattern for the operator `[_and then_]` is `(& &)`, and the default gathering pattern for the operator `(_)` is `(&)`.
- If the underbar corresponding to an argument is adjacent to another underbar, or if it is the leftmost or the rightmost token in the operator, then the default gathering value for such an argument is `E`. Thus, e.g., the default gathering pattern for the operator `not_` is `(E)`, the default gathering pattern for the operator `_?_:_` is `(E & E)`, the default gathering pattern for the operator `_+_` is `(E E)`, and the default gathering pattern for the operator `__` is `(E E)`.

Those binary operators which start with an underscore, end with an underscore, and have a precedence greater than 0 are handled as special cases:

- The operator will have gathering pattern `(e E)` if it has the `assoc` attribute (see Section 4.4.1). For example, the following operators fall into this category.

```
op _+_ : Nat Nat -> Nat [assoc] .
op *__ : Nat Nat -> Nat [assoc] .
op __ : NatList NatList -> NatList [assoc] .
```

- If the operator does not have the `assoc` attribute, but its first argument, its last argument, and its coarity are in the same connected component of sorts, then:

1. if the subsort relations allow it to right-associate but not left-associate, then the first argument's gathering pattern will change to e , and
2. if the subsort relations allow it to left-associate but not right-associate, then the last argument's gathering pattern will change to e .

Assuming $\text{Int} < \text{IntList}$, then the operators

```
op _<:_ : Int IntList -> IntList .
op _:>_ : IntList Int -> IntList .
```

have, by default, gathering patterns $(e\ E)$ and $(E\ e)$, respectively. According to the general rule, since their argument bars are the leftmost and the rightmost tokens, the gathering pattern should be $(E\ E)$ for both of them. However, both operators fall into the second special case, since they are binary operators which start and end with underscores, have a precedence greater than 0 (by default 41), and are not declared associative. Given the subsort relation, the operator $_<:_$ may right-associate, but not left-associate, that is, $1 <: 2 <: 3$ should be parsed as $1 <: (2 <: 3)$, but $(1 <: 2) <: 3$ should not be a valid parse. Therefore, $_<:_$ gets default gathering pattern $(e\ E)$. And similarly for $_:>_$, although in this case it can left-associate, and therefore it gets default gathering pattern $(E\ e)$.

3.9.3 The extended signature of a module

In addition to the signature defined by the user, parsing of terms takes place in an extended grammar in which information for handling parentheses, sort and equality predicates, `if_then_else_fi`, and qualification operators are included. These structures belong to the so-called *extended signature of a module*. The main structures added in the extended signature of a module are:

- *Sort disambiguation*. For each sort S in the signature of a module, Maude adds to the signature the operator

```
op (_).S : S -> S .
```

This helps in the disambiguation of ad-hoc overloaded constants and terms. As an example, remember from Section 3.6 that if we declare 0 as an ad-hoc overloaded constant for natural numbers and for natural numbers modulo 3, then we can disambiguate the expression $0 + 0$ by writing, for example, $0 + (0).\text{Nat}$ and $0 + (0).\text{Nat3}$, or $(0 + 0).\text{Nat}$ and $(0 + 0).\text{Nat3}$. As another example, in the module `META-MODULE` (see Section 14.3), the term `none` is ambiguous, since the operator `none` is used as the empty set of operator declarations, equations, rules, etc. We can disambiguate it by writing `(none).OpDeclSet`. Of course, these disambiguation operators can be used not only for constants, but for any term.

For example, we can write $(2 + 3).Nat$ as a valid term in the predefined module `NAT`.

- *Parentheses.* The extended signature of a module contains the operator

```
op ( _ ) : S -> S .
```

for each sort `S` in its signature. These operators allow the use of parentheses without having to declare a parentheses operator for each sort. For example, $(2 + 3)$, $(2 + 3) + 5$, $(2 + (3 + 5))$, $((2 + 3) + 5)$, are all valid terms in `NAT`, thanks to these declarations.

- *Equivalent single-identifier form* for all operators. Each declared operator, including those in mixfix form, may also be used in their equivalent single-identifier prefix form. For example, in the `NAT` module, the term `+_ (2, 3)` is equivalent to $2 + 3$, and the terms `if true then 2 + 3 else - 3 fi` and `if_then_else_fi(true, +_(2, 3), -(3))` are equivalent; any combination is possible so `if_then_else_fi(true, 2 + 3, - 3)` is also valid.
- *Flattened associative argument lists.* Operators with the attribute `assoc` may be used in Maude in a nonparenthesized flattened form (see Section 4.8). This is possible thanks to the precedence-gathering values in mixfix notation, but it is also possible in prefix syntax. For example, `gcd(2, 3, 4)` is a valid term in `NAT`, where `gcd` is the greater common divisor operator, which is declared as a binary associative operator. Of course, this term can always be written in the standard format as `gcd(2, gcd(3, 4))` or `gcd(gcd(2, 3), 4)`. Furthermore, we can combine this possibility with the single-identifier form to write things like `+_ (2, 3, 4)` instead of `+_ (_+_ (2, 3), 4)` or `+_ (2, +_ (3, 4))`, but of course, since `+_` is declared with the `assoc` attribute in the predefined module `NAT`, we can just write $2 + 3 + 4$.
- *Polymorphic operators and the `BOOL` module.* All the information contained in the predefined modules `TRUTH-VALUE`, `TRUTH`, `BOOL-OPS`, and `BOOL` is included in the extended signature of each module (unless this inclusion is explicitly disabled). In particular, appropriate instances of the polymorphic operators contained in `TRUTH` (that is, `if_then_else_fi`, `_==_`, and `_=/=_`) are generated for each sort in the module; in addition, for each sort `S`, a sort predicate `_ :: S` is also added. All these modules and operators are fully explained in Section 9.1.

3.9.4 Parsing examples

Maude provides the `parse` command for parsing terms. The command does not do anything other than parsing the given term in the extended signature of the module. This is exactly what is done when a term appears in a command, before executing such a command. For example, when we try to reduce a term $(2 + 3) * 5$, the system first parses it and then reduces it. If the term

is ambiguous, or there is no parse for it, an error message is given and no further action takes place.

```
Maude> reduce in NAT : 2 + true .
Warning: <standard input>, line 1:
  didn't expect token true: 2 + true <---*HERE*
Warning: <standard input>, line 1: no parse for term.
```

For testing the parsing of terms we can use the `parse` command.

```
Maude> parse in NAT : 2 + true .
Warning: <standard input>, line 1:
  didn't expect token true: 2 + true <---*HERE*
Warning: <standard input>, line 1: no parse for term.
```

As other commands, parsing can take place either in the module explicitly mentioned in the command or in the current module.

We illustrate the use of the `parse` command for the examples introduced in the previous sections. Let us first consider a module `PARSING-EX1` with constants 1, 2, and 3, and binary operators `_+_` and `_*_`.

```
fmod PARSING-EX1 is
  sort Nat .
  ops 1 2 3 : -> Nat .
  ops _+_ *__ : Nat Nat -> Nat .
endfm
```

Since `_+_` and `_*_` are declared without precedence values, and therefore both get the default value 41, we obtain the following result.

```
Maude> parse 1 + 2 * 3 .
Warning: <standard input>, line 13: ambiguous term, two parses are:
1 + (2 * 3) -versus- (1 + 2) * 3
```

Arbitrarily taking the first as correct. Nat: 1 + (2 * 3)

As a first solution, we may consider using parentheses.

```
Maude> parse in PARSING-EX1 : 1 + (2 * 3) .
Nat: 1 + (2 * 3)

Maude> parse in PARSING-EX1 : (1 + 2) * 3 .
Nat: (1 + 2) * 3
```

Let us now consider the module `PARSING-EX2`, where `_+_` and `_*_` are declared with precedences 33 and 31, respectively.

```
fmod PARSING-EX2 is
  sort Nat .
  ops 1 2 3 : -> Nat .
  op _+_ : Nat Nat -> Nat [prec 33] .
  op *__ : Nat Nat -> Nat [prec 31] .
endfm
```

Now, parentheses are not necessary for parsing the term $1 + 2 * 3$.

```
Maude> parse in PARSING-EX2 : 1 + 2 * 3 .
Nat: 1 + 2 * 3
```

Of course, we may still use parentheses.

```
Maude> parse in PARSING-EX2 : (1 + 2) * 3 .
Nat: (1 + 2) * 3
```

Since the default gathering patterns for binary operators like `+_` and `*_` is (E E), a term like $1 + 2 + 3$ is ambiguous.

```
Maude> parse in PARSING-EX2 : 1 + 2 + 3 .
Warning: <standard input>, line 30: ambiguous term, two parses are:
1 + (2 + 3) -versus- (1 + 2) + 3
```

Arbitrarily taking the first as correct. Nat: $1 + (2 + 3)$

As above, we may use parentheses to parse such terms.

```
Maude> parse in PARSING-EX2 : (1 + 2) + 3 .
Nat: (1 + 2) + 3
```

```
Maude> parse in PARSING-EX2 : 1 + (2 + 3) .
Nat: 1 + (2 + 3)
```

Let us now consider the module PARSING-EX3, where `+_` and `*_` are declared to be left-associative, that is, with gathering patterns (E e).

```
fmod PARSING-EX3 is
  sort Nat .
  ops 1 2 3 : -> Nat .
  op _+_ : Nat Nat -> Nat [prec 33 gather (E e)] .
  op *_ : Nat Nat -> Nat [prec 31 gather (E e)] .
endfm
```

Now, the terms above have unambiguous parses.

```
Maude> parse in PARSING-EX3 : 1 + 2 * 3 .
Nat: 1 + 2 * 3
```

```
Maude> parse in PARSING-EX3 : 1 + 2 + 3 .
Nat: 1 + 2 + 3
```

Let us now consider the module PARSING-EX4, where `+_` and `*_` are declared to be associative. Note that in this case, by default, they are assigned gathering patterns (E e).

```
fmod PARSING-EX4 is
  sort Nat .
  ops 1 2 3 : -> Nat .
  op _+_ : Nat Nat -> Nat [prec 33 assoc] .
  op *_ : Nat Nat -> Nat [prec 31 assoc] .
endfm
```

```
Maude> parse in PARSING-EX4 : 1 + 2 * 3 .
Nat: 1 + 2 * 3
```

```
Maude> parse in PARSING-EX4 : 1 + 2 + 3 .
Nat: 1 + 2 + 3
```

We illustrate the use of the extended signature in which all terms are parsed with the following examples.

```
Maude> parse in PARSING-EX1 : (2 + 3).Nat .
Nat: 2 + 3
```

```
Maude> parse in PARSING-EX1 : (2).Nat + 3 .
Nat: 2 + 3
```

```
Maude> parse in PARSING-EX1 : (2).Nat + (3).Nat .
Nat: 2 + 3
```

```
Maude> parse in PARSING-EX1 : ((1) + ((2) + (3))) .
Nat: 1 + (2 + 3)
```

```
Maude> parse in PARSING-EX1 : _+_ (1, _+_ (2, 3)) .
Nat: 1 + (2 + 3)
```

```
Maude> parse in PARSING-EX4 : _+_ (1, 2, 3) .
Nat: 1 + 2 + 3
```

```
Maude> parse in PARSING-EX4 : if 1 == 2 then 1 + 2 else _+_ (1, 2) fi .
Nat: if 1 == 2 then 1 + 2 else 1 + 2 fi
```

```
Maude> parse in PARSING-EX4 :
  if _==(1, 2)
  then if_then_else_fi(1 + 2 :: Nat, 1 * 1, 2 * 1)
  else _+_ (1, 2)
  fi .
Nat: if 1 == 2
  then if (1 + 2) :: Nat
  then 1 * 1
  else 2 * 1
  fi
  else 1 + 2
  fi
```

Functional Modules

Functional modules define data types and operations on them by means of equational theories. The data types consist of elements that can be named by ground terms. Two ground terms denote the same element if and only if they belong to the same equivalence class as determined by the equations. That is, the mathematical semantics of a functional module is its *initial algebra*. Maude's functional modules are assumed to have the nice property that equations, considered as simplification rules by using them only in the left to right direction, are Church-Rosser and terminating (see Section 4.7). This means that repeated application of the equations as simplification rules eventually reaches a term to which no further equations apply, and the result, called the *canonical form*, is the same regardless of the order of application of the equations. Thus each equivalence class has a natural representative, its canonical form, that can be computed by equational simplification. As explained in Section 1.2, this ensures that the initial algebra and the canonical term algebra of the functional module are isomorphic, and therefore that the module's mathematical and operational semantics coincide.

The equational logic on which Maude functional modules are based is an extension of order-sorted equational logic [173] called *membership equational logic* [248, 31]. Thus, functional modules support multiple sorts, subsort relations, operator overloading, and assertions of membership in a sort.

As was mentioned in Section 3.2, a functional module is declared in Maude using the keywords

```
fmod <ModuleName> is <DeclarationsAndStatements> endfm
```

For example,

```
fmod NUMBERS is
  ...
endfm
```

declares a module named `NUMBERS`. The dots stand for the actual declarations and statements that may appear in the functional module. Declarations in-

clude the importation of other functional modules (see Chapter 8), and sort, subsort, and operator declarations. Statements include equational and membership axioms. Declarations were discussed in Chapter 3. What remains to be explained are equational and membership statements.

4.1 Unconditional equations

Unconditional equations are declared using the keyword `eq`, followed by a term (its lefthand side), the equality sign `=`, then a term (its righthand side), optionally followed by a list of statement attributes (see Section 4.5 later in this chapter) enclosed in square brackets, and ending with white space and a period. Thus the general scheme is the following:

```
eq <Term-1> = <Term-2> [(StatementAttributes)] .
```

The terms t and t' in an equation $t = t'$ must both have the same kind. In order for the equation to be executable, any variable appearing in t' must also appear in t . Equations not satisfying this requirement can also be declared (for example, to document a lemma holding true in the module) but in such a case they should always be specified with the `nonexec` attribute (see Section 4.5.3). We can add equations axiomatizing the addition operation in our `NUMBERS` module as follows, where we distinguish two cases for the second argument, according to whether it is zero or not:

```
vars N M : Nat .
eq N + zero = N .
eq N + s M = s (N + M) .
```

The following equations define the symmetric difference operation `sd` on natural numbers, which returns the result of subtracting the smaller from the larger of its two arguments.

```
eq sd(N, N) = zero .
eq sd(N, zero) = N .
eq sd(zero, N) = N .
eq sd(s N, s M) = sd(N, M) .
```

In general, in a functional module one can specify equations (and also conditional equations, as explained in Section 4.3) in three different ways:

1. in the style given above, in which case they are assumed to be executable as simplification rules from left to right;
2. in the same style as above, but with the `nonexec` attribute (see Section 4.5.3), in which case Maude does not use them for simplification (except at the metalevel with a user-given strategy, see Section 14.6); and
3. as equational attributes of specific operators (see Section 4.4.1).

For example, a binary operator f can be declared `assoc` and `comm`, telling Maude that it satisfies the associativity and commutativity axioms. Such equational attributes should *not* be written explicitly as equations in the specification. There are two reasons for this. Firstly, this is redundant, since they have already been declared as equational attributes. Secondly, although declaring such equations either only explicitly as equations, or twice—one time as equational attributes and another as explicit equations—does not affect the *mathematical semantics* of the specification, that is, the initial algebra that the specification denotes (see Section 4.3), it does however drastically alter the specification’s *operational semantics*. For example, if the `comm` attribute for f were to be stated as an equation $f(X, Y) = f(Y, X)$, then using the equation as a simplification rule applied to the term, say, $f(a, b)$, would lead to the nonterminating chain of equational simplifications

$$f(a, b) = f(b, a) = f(a, b) = f(b, a) = \dots$$

This is quite bad, since we want the equations specified by method (1) to be used as simplification rules and assume them to be terminating and Church-Rosser, so that they always simplify a term to a unique result that cannot be further simplified. Instead, if `comm` is declared as an equational attribute, the above kind of looping does not happen: Maude then simplifies terms *modulo* the declared equational attributes, so that the terms $f(a, b)$ and $f(b, a)$ would indeed be treated as identical. For more on equational attributes see Section 4.4.1.

4.2 Unconditional memberships

Unconditional membership axioms specify terms as having a given sort. They are declared with the keyword `mb` followed by a term, followed by ‘:’, followed by a sort (that must always be in the same kind as that of the term), followed by a period. As equations, memberships can optionally have statement attributes (see Section 4.5).

```
mb <Term> : <Sort> [StatementAttributes] .
```

To illustrate this, consider the module `3*NAT` with the basic Peano number declarations as in the `NUMBERS` module and a new sort `3*Nat`.

The fact that `3*Nat` consists of multiples of 3 is expressed using the subsort declaration `Zero < 3*Nat < Nat` and the membership statement `mb (s s M3) : 3*Nat` for `M3` a variable of sort `3*Nat`.

```
fmod 3*NAT is
  sort Zero Nat .
  subsort Zero < Nat .
  op zero : -> Zero .
  op s_ : Nat -> Nat .
```

```

sort 3*Nat .
subsorts Zero < 3*Nat < Nat .
var M3 : 3*Nat .
mb (s s s M3) : 3*Nat .
endfm

```

Memberships axioms can interact in undesirable ways with operators that are declared with the `assoc` or `iter` attributes (see later Sections 4.4.1 and 4.4.2, respectively). This is explained and illustrated with examples in Sections 24.2.8 and 24.2.9.

4.3 Conditional equations and memberships

Equational conditions in conditional equations and memberships are made up of individual equations $t = t'$ and memberships $t : s$. A condition can be either a single equation, a single membership, or a conjunction of equations and memberships using the binary conjunction connective \wedge which is assumed to be associative. Thus the general form of conditional equations and memberships is the following:

```

ceq <Term-1> = <Term-2>
  if <EqCondition-1> /\ ... /\ <EqCondition-k>
  [<StatementAttributes>] .

cmb <Term> : <Sort>
  if <EqCondition-1> /\ ... /\ <EqCondition-k>
  [<StatementAttributes>] .

```

Furthermore, the concrete syntax of equations in conditions has three variants, namely:

- ordinary equations $t = t'$,
- *matching equations* $t := t'$, and
- *abbreviated Boolean equations* of the form t , with t a term in the kind `[Bool]`, abbreviating the equation $t = \text{true}$.

Any term t in the kind `[Bool]` can be used as an abbreviated Boolean¹ equation. . The Boolean terms appearing most often in abbreviated Boolean equations are terms using the built-in equality `_==_` and inequality `_/= _` predicates, and the built-in membership predicates `_ : S` with `S` a sort, including Boolean combinations of such terms with `not_`, `_and_`, `_or_` and other Boolean connectives (see Section 9.1 for a detailed description of all these

¹ By default, any Maude module imports the predefined `BOOL` module (see Section 9.1).

operators). For example, the following Boolean terms in the `NUMBERS` module (assuming that a “greater than” operator `_>_` has also been defined in `NUMBERS`),

```
N == zero
M /= s zero
not (K :: NzNat)
(N > zero or M /= s zero)
```

can appear as abbreviated Boolean equations in a condition, abbreviating, respectively, the equations:

```
(N == zero) = true
(M /= s zero) = true
not (K :: NzNat) = true
(N > zero or M /= s zero) = true
```

To illustrate the use of conditional equations and memberships, let us reconsider the path example from Section 3.5. The following conditional statements express the key membership defining path concatenation and the associativity of this operator:

```
var E : Edge .
vars P Q R S : Path .
cmb E ; P : Path if target(E) = source(P) .
ceq (P ; Q) ; R = P ; (Q ; R)
  if target(P) = source(Q) /\ target(Q) = source(R) .
```

The conditional membership axiom (introduced by the keyword `cmb`) states that an edge concatenated with a path is also a path when the target node of the edge coincides with the source node of the path. This has the effect of defining path concatenation as a partial operation on paths, although it is total on the kind `[Path]` of “confused paths.”

Assuming variables `P`, `E`, and `S` declared as above, `source` and `target` operations over paths are defined by means of conditional equations with matching equations in conditions as follows:²

```
ceq source(P) = source(E) if E ; S := P .
ceq target(P) = target(S) if E ; S := P .
```

Matching equations³ are mathematically interpreted as ordinary equations; however, operationally they are treated in a special way and they must satisfy special requirements. Note that the variables `E` and `S` in the above matching equations do not appear in the lefthand sides of the corresponding conditional equations. In the execution of these equations, these new variables

² Note that the `source` and `target` operations can equivalently be declared as

```
eq source(E ; S) = source(E) .
eq target(E ; S) = target(S) .
```

³ Similar constructs are used in languages like ASF+SDF [104] and ELAN [27].

become instantiated by *matching* the term $E ; S$ against the canonical form of the subject term bound to the variable P (see Section 4.7). In order for this match to decide the equality with the ground term bound to P , the term $E ; S$ must be a *pattern*. Given a functional module M , we call a term t an *M-pattern* if for any well-formed substitution σ such that for each variable x in its domain the term $\sigma(x)$ is in canonical form with respect to the equations in M , then $\sigma(t)$ is also in canonical form. A sufficient condition for t to be an *M-pattern* is the absence of unifiers between its nonvariable subterms and lefthand sides of equations in M .

Ordinary equations $t = t'$ in conditions have the usual operational interpretation, that is, for the given substitution σ , $\sigma(t)$ and $\sigma(t')$ are both reduced to canonical form and are compared for equality, modulo the equational attributes specified in the module's operator declarations such as associativity, commutativity, and identity. Finally, abbreviated Boolean equations are just a special case of ordinary equations once they are expanded out.

The satisfaction of the conditions is attempted sequentially from left to right. Since in Maude matching takes place modulo equational attributes, in general many different matches may have to be tried until a match of all the variables satisfying the condition is found.

The above equations for `source` and `target` illustrate the use of matching equations to bind variables locally, in much the same way that `let` is used in some functional programming languages. In this example, since the matching is purely syntactic, the matching substitution is unique and gives a simple way to name parts of a structure or to name a complicated expression which appears multiple times in the main equation.

For *M*-patterns where some operators are matched modulo some equational attributes, matching substitutions need not be unique. This provides another way of using matching equations, namely to perform a search through a structure without any need to explicitly define a function that does this. For example, for sequences of natural numbers we can define a predicate `_occurs-inner_` that determines if a number occurs in a sequence other than at one of the ends. If one only cares about positive results,⁴ the following will work.

```
op _occurs-inner_ : [Nat] [NatSeq] -> [Bool] .
ceq N:Nat occurs-inner NS:NatSeq = true
    if (NS0:NatSeq N:Nat NS1:NatSeq) := NS:NatSeq .
```

Note that this equation could also be written as

```
eq N:Nat occurs-inner NS0:NatSeq N:Nat NS1:NatSeq = true .
```

⁴ Note that, since when the predicate is not true it remains unevaluated, we have defined it at the kind level, that is, as a partial Boolean function; however, using the `owise` attribute (see Section 4.5.4) it is very easy to add an extra equation making `_occurs-inner_` a *total* Boolean function.

In both cases we check whether the sequence contains the natural number `N:Nat`, but making sure that the sequence contains other elements both before and after `N:Nat`.⁵ With the above definition added to the numbers module, the term

```
zero occurs-inner (zero zero zero zero zero)
```

reduces to `true`, while the term

```
zero occurs-inner (zero zero)
```

does not reduce further.

Matching equations in conditions give great expressive power, but some care is needed in using them to define operations. Consider adding the following to the numbers module, in an attempt to define a test for the presence of `s s zero` in a sequence of natural numbers.

```
op hasTwo : [NatSeq] -> [Bool] .
ceq hasTwo(NS:NatSeq) = N:Nat == s s zero
  if NS0:NatSeq N:Nat NS1:NatSeq := NS:NatSeq .
```

With this addition to the numbers module, `hasTwo(zero zero)` does not get reduced, since the condition requires at least three numbers in the sequence. The term `hasTwo(zero (s s zero) zero)` reduces to `true`. The term `hasTwo(zero (s zero) (s s zero) zero)` also gets reduced, although it may return `true` or `false`; probably not what was intended. The problem is that there are several matches, each giving a different answer, so the conditional equation does not define a function. In fact, this conditional equation causes the Church-Rosser property to fail, and semantically identifies `true` and `false`, thus leading to an inconsistent theory. In contrast, as will be seen in Chapter 6, a *rule* with such a matching condition is not a problem, and does have the effect of searching a sequence of natural numbers for `s s zero`.

In summary, all the sort, subsort, and operator declarations and all the statements in a functional module (plus the functional modules imported if any) define an *equational theory* in *membership equational logic* [248, 31]. Such a theory can be described in mathematical notation as a pair $(\Sigma, E \cup A)$, where Σ is the *signature*, that is, the specification of the sorts, subsorts, kinds, and operators in the module, E is the collection of statements (equations and memberships, possibly conditional) and A is the set of equational attributes,

⁵ Note that here we assume the declaration of the `NatSeq` concatenation operator `--` as given in page 47, where it is declared to be associative. If we consider the declaration of this operator given in page 72, which is also declared to have `nil` as identity element, then we should write this equation as

```
op _occurs-inner_: [Nat] [NatSeq] -> [Bool] .
ceq N:Nat occurs-inner NS:NatSeq = true
  if (I:Nat NS0:NatSeq N:Nat NS1:NatSeq M:Nat) := NS:NatSeq .
since the variables NS0:NatSeq and NS1:NatSeq might be instantiated to nil.
```

such as `assoc` and `comm`, declared for some operators (that is, extra equations that are treated in a special way by the Maude interpreter to simplify modulo such attributes, see Section 4.4.1).

The family of ground terms definable in the syntax of Σ defines a model called a Σ -algebra and denoted T_Σ . In T_Σ , terms syntactically different denote different elements, so that T_Σ will *not* satisfy the equations in $E \cup A$, unless they are trivial equations such as $f(X) = f(X)$. The question is, what is the *optimal model* of the theory $(\Sigma, E \cup A)$? Goguen and Burstall's answer is: a model satisfying the axioms $E \cup A$ and such that it has *no junk* (that is, all elements can be denoted by ground Σ -terms), and *no confusion* (that is, only elements that are *forced to be equal* by the axioms $E \cup A$ are identified). Such a model, called the *initial algebra* of the equational theory $(\Sigma, E \cup A)$, exists [248], is denoted $T_{\Sigma/E \cup A}$, and provides the *mathematical semantics* of the Maude functional module specifying $(\Sigma, E \cup A)$.

Mathematically, $T_{\Sigma/E \cup A}$ can be constructed as the quotient of T_Σ in which the equivalence classes are those terms that are *provably equal* using the axioms $E \cup A$. Operationally, assuming that the axioms E are Church-Rosser and terminating modulo A (see Section 4.7), there is a much more intuitive equivalent description of $T_{\Sigma/E \cup A}$, namely as the family of *canonical forms* for the ground Σ -terms modulo A , that is, those terms that cannot be further simplified by the equations in E modulo A . That is, as explained in Section 1.2, we have then an isomorphism

$$T_{\Sigma/E \cup A} \cong \text{Can}_{\Sigma/E \cup A}$$

between the initial algebra $T_{\Sigma/E \cup A}$ and the canonical term algebra $\text{Can}_{\Sigma/E \cup A}$.

The Maude interpreter computes such canonical forms, which can be viewed as the *values* denoted by the corresponding functional expressions, with the `reduce` command (see Section 25.2 for details and Section 4.9 for examples).

4.4 Operator attributes

Operator declarations may include attributes that provide additional information about the operator: semantic, syntactic, pragmatic, etc. All such attributes are declared within a single pair of enclosing square brackets, '[' and ']', after the sort of the result and before the ending period. We discuss each of the categories of operator attributes below.

4.4.1 Equational attributes

Equational attributes are a means of declaring certain kinds of equational axioms in a way that allows Maude to use these equations efficiently in a built-in way. Currently Maude supports the following equational attributes:

- **assoc** (associativity),
- **comm** (commutativity),
- **idem** (idempotency),
- **id:** $\langle Term \rangle$ (identity, with the corresponding term for the identity element),
- **left id:** $\langle Term \rangle$ (left identity, with the corresponding term for the left identity element), and
- **right id:** $\langle Term \rangle$ (right identity, with the corresponding term for the right identity element).

An operator can be declared with several of these attributes, which may appear in any order in the attribute declaration. However, these attributes are only allowed for *binary* operators satisfying the following requirements:

- For **left id:**, it is required that the right domain sort and the range sort belong to the same kind.
- For **right id:**, it is required that the left domain sort and range sort belong to the same kind.
- For **assoc**, **comm**, **id:**, and **idem**, both domain sorts and the range sort must belong to the same kind.

These requirements are checked at parse time, and if the check fails a warning is output and the operator loses its attributes.

Furthermore, we have the following additional requirements:

- The attribute **idem** cannot be used in any combination of attributes that includes **assoc**, because the necessary matching and normalization algorithms have not been implemented yet. This requirement is quietly enforced by ignoring the attribute **idem** where necessary.
- Only one identity attribute (**left id:**, **right id:**, or **id:**) is allowed. This is enforced by a warning and by ignoring all but the first such attribute.
- Combining the attribute **comm** with either **left id:** or **right id:** silently turns the identity attribute into an **id:**.
- All subsort-overloaded instances of an operator must have the same attributes. This is further explained in Section 4.4.6.

Semantically, declaring a set of equational attributes for an operator is equivalent to declaring the corresponding equations for the operator. Operationally, using equational attributes to declare such equations avoids termination problems and leads to much more efficient evaluation of terms containing such an operator. In fact, the effect of declaring equational attributes is to compute with equivalence classes modulo such equations. This, besides being very expressive, avoids what otherwise would be insoluble termination problems. For example, if a commutativity equation like $x + y = y + x$ is declared as an ordinary equation, then it will easily produce looping, nonterminating simplifications. If it is instead declared with an equational attribute **comm**, this looping behavior does not happen.

In our numbers example we can add a constant `nil` for the empty sequence and refine the declaration of sequence concatenation so that concatenation is associative with identity `nil`.

```
op nil : -> NatSeq .
op __ : NatSeq NatSeq -> NatSeq [assoc id: nil] .
```

As another example, we can form lists of Booleans as a supersort `BList` of `Bool` in an extension of the `BOOL` module (see Section 9.1) with a “cons” operator `.._` having `nil` as a right identity:

```
sort BList .
subsort Bool < BList .
op nil : -> BList .
op .._ : Bool BList -> BList [right id: nil] .
```

Note that, when equational attributes are declared, equational simplification using the other equations in the module does not take place at the purely syntactic level of replacing syntactic equals by equals, but is understood *modulo* the equational attributes. Therefore, the proper understanding of the notions of Church-Rosser and terminating equations, and of canonical forms, is now *modulo* the equational attributes that have been declared. We discuss matching and equational simplification modulo axioms in Section 4.8.

For example, by declaring the addition operation on natural numbers modulo 3 as commutative,

```
op _+_ : Nat3 Nat3 -> Nat3 [comm] .
```

it is enough to have the following equations to define its behavior on all possible combinations of arguments:

```
vars N3 : Nat3 .
eq N3 + 0 = N3 .
eq 1 + 1 = 2 .
eq 1 + 2 = 0 .
eq 2 + 2 = 1 .
```

The equations

```
eq 0 + N3 = N3 .
eq 2 + 1 = 0 .
```

are not needed, because they are subsumed by the first and third equations above, due to commutativity of `_+_`.

Notice that membership axioms and matching modulo associativity can interact in undesirable ways, as explained in Section 24.2.8.

4.4.2 The iter attribute

Maude provides a built-in mechanism called the `iter` (short for *iterated* operator) theory whose goal is to permit the efficient input, output, and manipulation of very large stacks of a unary operator.

Unary operators may be declared to belong to the `iter` theory by including `iter` in their attributes. After declaring

```
sort Foo .
op f : Foo -> Foo [iter] .
```

the term `f(f(f(X:Foo)))` can be input as `f^3(X:Foo)` and will be output in that form. A term such as `f^1234567890123456789(X:Foo)` is too large to be input, output or manipulated in regular notation, but can be input and output in this compact notation and certain (built-in) manipulations may thus be made efficient.

The precise form of the compact iter theory notation is the prefix name of the operator followed by `^[1-9][0-9]*` (in Lex regular expression notation) with no intervening white space. Note that `f^0123(X:Foo)` is not acceptable. Of course, regular notation (and mixfix notation if appropriate) can still be used.

Membership axioms may also interact in undesirable ways with operators declared with the `iter` attribute; see Section 24.2.9 for details.

4.4.3 Constructors

Assuming that the equations in a functional module are (ground) Church-Rosser and terminating, then every ground term in the module (that is, every term without variables) will be simplified to a canonical form, perhaps modulo some declared equational attributes. *Constructors* are the operators appearing in such canonical forms. The operators that “disappear” after equational simplification are instead called *defined functions*. For example, typical constructors in a sort `Nat` are `zero` and `s_`, whereas in the sort `Bool`, `true` and `false` are the only constructors.

It is quite useful for different purposes, including both debugging (see Chapter 24) and theorem proving, to specify when a given operator is a constructor. This can be done with the `ctor` attribute. For example, we can refine our operator declarations in Section 3.4 with constructor information as follows:

```
op zero : -> Zero [ctor] .
op s_ : Nat -> NzNat [ctor] .
op nil : -> NatSeq [ctor].
op __ : NatSeq NatSeq -> NatSeq [ctor assoc id: nil] .
```

Three slightly subtle points should be mentioned, namely the relationships of constructors to operator overloading, to kinds, and to equations. The first

key observation is that constructor declarations are *local to given sorts for the arguments and for the result*. Nothing prevents an operator from being a constructor at some level in the subsort ordering but being a defined function at another. For example, we could have declared a successor function for integers,

```
op s_ : Int -> Int .
```

which is *not* a constructor. Indeed, we can define the sort `Int` with a subsort `NzNeg` of nonzero negative numbers built up with a unary minus constructor `-_`, and we can then specify both unary minus `-_` and successor `s_` as *defined functions* on the integers by giving the equations:

```
sorts NzNeg Int .
subsorts Nat NzNeg < Int .
op -_ : NzNat -> NzNeg [ctor] .
op -_ : Int -> Int .
op s_ : Int -> Int .

var N : Nat .

eq - zero = zero .
eq - (- (s N)) = s N .
eq s (- (s N)) = - N .
```

A related observation is that a defined function, which totally disappears at some level in the subsort ordering, might not go away for terms at the kind level. For example, even though addition may be a defined function, we may encounter an arithmetic error expression in a kind of numbers such as

```
(s s zero) + p zero
```

because the predecessor function `p` has been declared on nonzero natural numbers.

```
op p : NzNat -> Nat .
```

The last point is that constructors may obey certain equations; that is, they do not have to be *free* constructors. The equations that they may obey (even as constructors, not just in other overloaded variants such as the integer successor function above) may be either equational attributes (such as the `assoc` attribute in the above concatenation operator for strings of natural numbers), or ordinary equations, or both. For example, we can add a sort `NatSet` of finite sets of natural numbers to our `NUMBERS` module by declaring a set union operation `_;_` using equational attributes to declare that it is associative and commutative with identity the empty set, and using an ordinary equation to express idempotency.⁶

⁶ Remember that the `idem` attribute cannot be specified together with an `assoc` attribute; therefore idempotency must in this case be specified explicitly by an equation.

```

sort NatSet .
subsort Nat < NatSet .
op empty : -> NatSet [ctor] .
op _;_ : NatSet NatSet -> NatSet [ctor assoc comm id: empty] .
eq N ; N = N .

```

Given an equational specification in which several operators have been declared as constructors by means of the `ctor` attribute and such that the equations are terminating, the *sufficient completeness problem* consists in verifying that the canonical forms of all well-typed ground terms are constructor terms. Intuitively, this means that all defined operations (i.e., those that are not declared as constructors) have been fully defined. Maude's Sufficient Completeness Checker (SCC), described in Section 23.1.5, can be used to ensure that constructor declarations are really correct, so that all functions are fully defined relative to those constructors. We can take the `NUMBERS` module, incrementally introduced in Chapter 3 and the previous sections of this chapter, to illustrate how the SCC can be used to help the specifier in this regard.

```

fmod NUMBERS is
  sort Zero .
  sorts Nat NzNat .
  subsort Zero NzNat < Nat .
  op zero : -> Zero [ctor] .
  op s_ : Nat -> NzNat [ctor] .
  op sd : Nat Nat -> Nat .
  ops _+_ *_ : Nat Nat -> Nat .
  op _+_ : NzNat Nat -> NzNat .
  op p : NzNat -> Nat .

  vars I N M : Nat .
  eq N + zero = N .
  eq N + s M = s (N + M) .
  eq sd(N, N) = zero .
  eq sd(N, zero) = N .
  eq sd(zero, N) = N .
  eq sd(s N, s M) = sd(N, M) .

  sort Nat3 .
  ops 0 1 2 : -> Nat3 .
  op _+_ : Nat3 Nat3 -> Nat3 [comm] .
  vars N3 : Nat3 .
  eq N3 + 0 = N3 .
  eq 1 + 1 = 2 .
  eq 1 + 2 = 0 .
  eq 2 + 2 = 1 .

  sort NatSeq .
  subsort Nat < NatSeq .
  op nil : -> NatSeq [ctor].

```

```

op _- : NatSeq NatSeq -> NatSeq [ctor assoc id: nil] .

sort NatSet .
subsort Nat < NatSet .
op empty : -> NatSet [ctor].
op _;- : NatSet NatSet -> NatSet [ctor assoc comm id: empty] .
eq N ; N = N .
endfm

```

For expository reasons, since the `ctor` declaration had not yet been explained, some operators and constants were declared without the `ctor` attribute when they were introduced in Section 3.6. The SCC reports the first term it finds not reducible to a constructor. In this case, the first such report we get is the following:

```

Maude> (scc NUMBERS .)
Checking sufficient completeness of NUMBERS ...
Warning: This module has equations that are not left-linear which
will be ignored when checking.
Failure: The term 0 was found to be a counterexample. Since the
analysis is incomplete, it may not be a real counterexample.

```

We fix this error by adding the `ctor` attribute to the declaration of the constants 0, 1, and 2 of sort `Nat3`:

```
ops 0 1 2 : -> Nat3 [ctor].
```

After this declaration is corrected, a more serious bug is found by the SCC, namely,

```

Maude> (scc NUMBERS .)
Checking sufficient completeness of NUMBERS ...
Warning: This module has equations that are not left-linear which
will be ignored when checking.
Failure: The term zero * zero was found to be a counterexample.
Since the analysis is incomplete, it may not be a real
counterexample.

```

This message shows that the definition of multiplication is incomplete, because we have declared the operator without the `ctor` attribute but we have forgotten the equations defining such operation on natural numbers. For example, we can add the following equations to make up for this omission:

```

eq N * zero = zero .
eq N * s M = (N * M) + N .

```

A further iteration of the SCC on the amended specification shows that the equations for the predecessor operation `p` are missing as well. Since `p` is only defined on nonzero natural numbers, only one equation needs to be added:

```
eq p(s N) = N .
```

The corrected `NUMBERS` module after this analysis (together with some additional declarations introduced in the following sections) is presented in Section 4.9. Here is the tool output on the corrected module:

```
Maude> (scc NUMBERS .)
Checking sufficient completeness of NUMBERS ...
Warning: This module has equations that are not left-linear which
will be ignored when checking.
Success: NUMBERS is sufficiently complete under the assumption
that it is weakly-normalizing, confluent, and sort-decreasing.
```

4.4.4 Polymorphic operators

A number of Maude's built-in operators are *polymorphic* in one or more arguments, in the sense that the operator has meaning when these arguments are of any known sort. Examples include Boolean operators such as the conditional, `if_then_else_fi`, which is polymorphic in its second and third arguments, and the equality test `_==_` which is polymorphic in both arguments (see Section 9.1). The user can also define polymorphic operators using the `polymorphic` attribute (abbreviated `poly`). This attribute takes a set of natural numbers enclosed in parentheses that indicates which arguments are polymorphic, with 0 indicating the range. For polymorphic operators that are not constants, at least one argument should be polymorphic to avoid ambiguities. Since there are no polymorphic equations, polymorphic operators are limited to constructors and built-ins. Polymorphic operators are always instantiated with the polymorphic arguments going to the kind level, which further limits their generality. The sort name in a polymorphic position of an operator declaration is purely a place holder—any legal type name could be used. The recommended convention is to use `Universal`.

One reasonable use for polymorphic operators beyond the existing built-ins is to define heterogeneous lists, as follows, where `CONVERSION` denotes a predefined module described in Section 9.9 having types for different numbers as well as strings; this module is imported by means of a `protecting` declaration, which will be explained in Section 8.1.1.

```
fmod HET-LIST is
  protecting CONVERSION .

  sort List .
  op nil : -> List .
  op __ : Universal List -> List [ctor poly (1)] .
endfm
```

As an example, we can form the following heterogeneous lists:

```
Maude> red 4 "foo" 4.5 1/2 nil .
result List: 4 "foo" 4.5 1/2 nil
```

```
Maude> red (4 "foo" nil) 4.5 1/2 nil .
result List: (4 "foo" nil) 4.5 1/2 nil
```

4.4.5 Format

The `format` attribute is intended to control the white space between tokens as well as color and style when printing terms for programming-language-like specifications. Consider the following mixfix syntax operator:

```
op (op _:_->_[_].) : Qid TypeList Type AttrSet -> OpDecl .
```

There are eleven places where white space can be inserted:

```
op  _  _  :  _  ->  _  [  _  ]  .
  ^   ^   ^   ^   ^   ^   ^   ^   ^   ^   ^
```

A format attribute must have an instruction word for each of these places. For example, the formatting specification for the above operator could be chosen to be:

```
[format (d d d d d s d d s d)]
```

Instruction words are formed from the following alphabet:

```
d default spacing
  (cannot be part of a larger word: must occur on its own)
+ increment global indent counter
- decrement global indent counter
s space
t tab
i number of spaces determined by indent counter
n newline
```

Note that, in general, each place may have an entire *word* combining several of the above symbols. We can illustrate how this feature is used in several operators in (submodules of) the `META-LEVEL` module in the file `prelude.maude` (see Chapter 14).

- Each assignment will be printed in a new line, indented one tab.

```
op _<_ : Variable Term -> Assignment
  [ctor prec 63 format (nt d d d)] .
```

- Each importation after the first one will be printed in a new line, with the current indentation.

```
op __ : ImportList ImportList -> ImportList
  [ctor assoc id: nil format (d ni d)] .
```



```

op cyan : -> Color [format (cu o)] .
op magenta : -> Color [format (mu o)] .
op __ : ColorList ColorList -> ColorList [assoc] .
endfm

```

To see the colors in this module, load the COLOR-TEST module into Maude and execute the command:⁸

```

Maude> reduce red green blue yellow cyan magenta .
reduce in COLOR-TEST : red green blue yellow cyan magenta .
rewrites: 0 in 0ms cpu (0ms real) (~ rews/sec)
result ColorList: red green blue yellow cyan magenta

```

Let us consider the following module FORMAT-DEMO, where a small programming language is defined.

```

fmod FORMAT-DEMO is
  sorts Variable Expression BoolExp Statement .
  subsort Variable < Expression .
  ops a b c d : -> Variable .
  op 1 : -> Expression .
  op +_ : Expression Expression -> Expression [assoc comm] .
  op ;_ : Statement Statement -> Statement [assoc prec 50] .
  op <=_ : Expression Expression -> BoolExp .

  op while_do_od : BoolExp Statement -> Statement
    [format (nir! o r! o++ --nir! o)] .

  op let_:=_ : Variable Expression -> Statement
    [format (nir! o d d d)] .
endfm

```

Note the use of the `format` attribute for operators `while_do_od` and `let_:=_`. Since both represent statements, which should start in a new line, but at the current indentation level, both include `ni` in the instruction words for their first positions; this position also has characters `r!` in both cases, so that they start in boldface red font. Since there is a `o` for the next position, reverting to original color and style, only the first word (`while` and `let`) is shown in red. In the case of `while_do_od`, the condition of the loop starts at the second position. The `do` word is shown in boldface red, and then the indentation counter is incremented, so that the body of the `while_do_od` statement is indented. For the position marking the beginning of `od`, the counter is decremented, so that it appears at the level of `while` in a new line (`n`), in boldface red font (`r!`). The last position reverts the original color and style, although notice that the indentation counter remains the same, so that successive statements will be given the same level of indentation. In the case of `let_:=_`, the three last positions contain only `d` (default spacing), since it

⁸ Try it in your terminal. The colors are not shown here for obvious reasons.

is to be presented as a single-line statement in which `let` is shown in boldface red.

We can illustrate the difference between using the `format` attribute and not using it with the following commands (as before, you should execute the example in your terminal to see the colors).

```
Maude> set print format off .
Maude> parse
  while a <= d do
    let a := a + b ;
    while b <= d do
      let b := b + c ;
      let c := c + 1
    od
  od
.
Statement: while a <= d do let a := a + b ; while b <= d do let b :=
  b + c ; let c := c + 1 od od

Maude> set print format on .
Maude> parse
  while a <= d do
    let a := a + b ;
    while b <= d do
      let b := b + c ;
      let c := c + 1
    od
  od
.
Statement:
while a <= d do
  let a := a + b ;
  while b <= d do
    let b := b + c ;
    let c := c + 1
  od
od
```

For more examples of format attributes, you can see the operator declarations in the module `LTL` (in the file `model-checker.maude`) discussed in Chapter 13, or in the modules `META-TERM` and `META-MODULE` (in the file `prelude.maude`), described in Chapter 14.

4.4.6 Ditto

An operator can have several subsort-overloaded instances. Maude requires that all these instances should have the *same* attributes, *except* for the case of the `ctor` attribute, that may be present in some instances but absent in

others (see Section 4.4.3), and/or the `metadata` attribute (see Section 4.5.2). It is for example forbidden to have a subsort-overloaded instance in which an operator is declared `assoc` only, and another such instance in which it is declared `assoc` and `comm`.

The `ditto` attribute can be given to an operator for which another subsort-overloaded instance *has already appeared*, either in the same module or in a submodule. The `ditto` attribute is just a shorthand stating that this operator, being subsort overloaded, should have the same attributes as those appearing explicitly in a previous subsort-overloaded version, except for the `ctor` and `metadata` attributes, which are outside the scope of `ditto`. In this way we can avoid writing out a possibly long attribute list again and again.

It is not allowed to combine `ditto` with other attributes, except for `ctor` and `metadata`. That is, an operator given the `ditto` attribute either has no other explicitly given attributes, or can only have in addition either the `ctor` attribute if it is a constructor, or a `metadata` attribute, or both the `ctor` and `metadata` attributes. Furthermore, it is forbidden to use `ditto` on the first declared instance of an operator, since this is nonsensical.

In our numbers module we can add equational attributes to the declarations of `+_` and `*_`, and then use `ditto` to declare the same attributes in other subsort-overloaded versions.

```
ops _+_ *__ : Nat Nat -> Nat [assoc comm].
op _+_ : NzNat Nat -> NzNat [ditto] .
op *__ : NzNat NzNat -> NzNat [ditto] .
```

For an example making extensive use of the `ditto` attribute see the LTL-SIMPLIFIER module (in the file `model-checker.maude`), discussed in Chapter 13.

4.4.7 Operator evaluation strategies

If a collection of equations is Church-Rosser and terminating, given an expression, no matter how the equations are used from left to right as simplification rules, we will always reach the same final result. However, even though the final result may be the same, some orders of evaluation can be considerably more efficient than others. More generally, we may be able to achieve the termination property provided we follow a certain order of evaluation, but may lose termination when any evaluation order is allowed. It may therefore be useful to have some way of controlling the way in which equations are applied by means of strategies.

In general, given an expression $f(t_1, \dots, t_n)$ we can try to evaluate it to its reduced form in different ways, such as:

- first obtaining the reduced form of all the t_i and then applying equations for f at the top of the term; this is called a *bottom-up*, or *eager* strategy;

- evaluating only some of the arguments, and then trying to evaluate at the top with equations for f ; for example, an `if_then_else-fi` operator will typically be evaluated by evaluating first the first argument, and then the `if_then_else-fi` operator at the top;
- trying to evaluate the top of the term first, and then, if this fails, either not evaluating the subterms at all, or trying to evaluate only some of them, that is, some kind of *lazy* evaluation strategy.

Typically, a functional language is either eager, or lazy with some strictness analysis added for efficiency, and the user has to live with whatever the language provides. Maude adopts OBJ3's [174] flexible method of user-specified *evaluation strategies* on an operator-by-operator basis, adding some improvements to the OBJ3 approach to ensure a correct implementation [135].

For an n -ary operator f an evaluation strategy is specified as a list of numbers from 0 to n ending with 0. The nonzero numbers denote argument positions, and a 0 indicates evaluation at the top of the given function symbol. The strategy then specifies what argument positions must be simplified (in the order indicated by the list) before attempting simplification at the top with the equations for the top function symbol. In functional programming terminology, the argument positions to be evaluated are usually called *strict* argument positions, so we can view an evaluation strategy as a flexible, user-definable way of specifying strictness requirements on argument positions. In the simplest case, a strategy consists of a list of nonzero numbers followed by a 0, so that some arguments are treated strictly and then the function symbol's equations are applied. For example, in Maude, if no strategy is specified, all argument positions are assumed strict, so that for f with n argument positions its default strategy is $(1\ 2\ \dots\ n\ 0)$; this is the "eager evaluation" case. The opposite extreme is a form of lazy evaluation such as the lazy append operator in the SIEVE example below. This operator has strategy (0) , thus only equations at the top are tried during evaluation.

The syntax to declare an n -ary operator with strategy $(i_1\ \dots\ i_k\ 0)$, where $i_j \in \{0, \dots, n\}$ for $j = 1, \dots, k$, is

```
op <OpName> : <Sort-1> ... <Sort-n> -> <Sort> [strat (i1 ... ik 0)] .
```

As a simple example consider the operators `_and-then_` and `_or-else_` in the module EXT-BOOL, that can be found in the file `prelude.maude` (see Section 9.1).

```
fmod EXT-BOOL is
  protecting BOOL .
  op _and-then_ : Bool Bool -> Bool
    [strat (1 0) gather (e E) prec 55] .
  op _or-else_ : Bool Bool -> Bool
    [strat (1 0) gather (e E) prec 59] .
  var B : [Bool] .
  eq true and-then B = B .
```

```

eq false and-then B = false .
eq true or-else B = true .
eq false or-else B = B .
endfm

```

These operators are computationally more efficient versions of Boolean conjunction and disjunction that avoid evaluating the second of the two Boolean subterms in their arguments when the result of evaluating the first subterm provides enough information to compute the conjunction or the disjunction. For example, letting `B:[Bool]` stand for an arbitrary Boolean expression

```

Maude> red false and-then B:[Bool] .
result Bool: false

```

while if `B:[Bool]` does not evaluate to `true` or `false`, then `false and B:[Bool]` does not evaluate to `false`, and if evaluation of `B:[Bool]` does not terminate then neither will evaluation of `false and B:[Bool]`.

If some of the argument positions are never mentioned in some of the operator strategies, the notion of canonical form becomes now *relative* to the given strategies and may not coincide with the standard notion. Let us consider as a simple example the following two functional modules, which we have displayed side-by-side to emphasize their only difference, namely, the evaluation strategy associated to the operator `g`.

```

fmod STRAT-EX1 is                fmod STRAT-EX2 is
  sort S .                        sort S .
  ops a b : -> S .                ops a b : -> S .
  op g : S -> S .                 op g : S -> S [strat(0)] .
  eq a = b .                       eq a = b .
endfm                              endfm

```

The canonical form of the term `g(a)` in `STRAT-EX1` is `g(b)`, but in `STRAT-EX2` it is `g(a)` itself, because the equation cannot be applied inside the term due to the lazy strategy `strat(0)` of the operator `g`.

This may be just what we want, since we may be able to achieve termination to a canonical form relative to some strategies in cases when the equations may be nonterminating in the standard sense. More generally, operator strategies may allow us to compute with infinite data structures which are evaluated on demand, such as the following formulation of the *sieve of Eratosthenes*, which finds all prime numbers using lazy lists.

The infinite list of primes is obtained from the infinite list of all natural numbers greater than 1 by filtering out all the multiples of numbers previously taken. Thus, first we take 2 and delete all the multiples of numbers greater than 2; then we take 3 and delete all the multiples of 3 greater than 3; and so on. The operation `nats-from_` generates the infinite list of natural numbers starting in the given argument; the operation `filter_with_` is used to delete all the multiples of the number given as second argument in the list provided as

first argument; and the operation `sieve_` is used to iterate this process with successive numbers.

Of course, since we are working with infinite lists, we cannot obtain as a result an infinite list. Such an infinite structure is only shown partially by means of the operation `show_upto_`, which shows only a finite prefix of the whole infinite list. Moreover, the generation and filtering processes have to be done in a lazy way. This is accomplished by giving to the list constructor `.._` a lazy strategy `strat(0)` that avoids evaluating inside the term, and using an operation `force` with an eager strategy `strat(1 2 0)` to “force” the evaluation of elements inside the list. Specifically, in order to apply the first equation, we must evaluate the arguments `L` and `S` before reconstructing the list `L . S` in the righthand side.

`NAT` denotes the predefined module of natural numbers and arithmetic operations on them (see Section 9.2), which is imported by means of a `protecting` declaration, explained in Section 8.1.1. Note the use of the symmetric difference operator `sd` (see Section 9.2) to decrement `I` in the third equation, and the successor operator `s_` to increment `I` in the sixth equation.

```
fmod SIEVE is
  protecting NAT .
  sort NatList .
  subsort Nat < NatList .
  op nil : -> NatList .
  op .._ : NatList NatList -> NatList [assoc id: nil strat (0)] .
  op force : NatList NatList -> NatList [strat (1 2 0)] .
  op show_upto_ : NatList Nat -> NatList .
  op filter_with_ : NatList Nat -> NatList .
  op nats-from_ : Nat -> NatList .
  op sieve_ : NatList -> NatList .
  op primes : -> NatList .

  vars P I E : Nat .
  vars S L : NatList .

  eq force(L, S) = L . S .
  eq show nil upto I = nil .
  eq show E . S upto I
    = if I == 0
      then nil
      else force(E, show S upto sd(I, 1))
      fi .
  eq filter nil with P = nil .
  eq filter I . S with P
    = if (I rem P) == 0
      then filter S with P
      else I . filter S with P
      fi .
  eq nats-from I = I . nats-from (s I) .
```

```

eq sieve nil = nil .
eq sieve (I . S) = I . sieve (filter S with I) .
eq primes = sieve nats-from 2 .
endfm

```

We can then evaluate expressions in this module with the `reduce` command (see Sections 4.9 and 25.2). For example, to compute the list of the first ten prime numbers we evaluate the expression:

```

Maude> reduce show primes upto 10 .
result NatList: 2 . 3 . 5 . 7 . 11 . 13 . 17 . 19 . 23 . 29

```

In the case of associative or commutative binary operators, evaluation strategies might reduce some arguments that the user does not expect to be reduced. The reason is that in such cases terms represent equivalence classes and it might be quite hard to say what is the first or the second argument. The adopted solution is that mentioning either argument implies both.

The paper [135] documents the operational semantics and the implementation techniques for Maude's operator evaluation strategies in much more detail. The mathematical semantics of a Maude functional module having operator evaluation strategies is documented in [188] and is further discussed in Section 4.7.

Of course, operator evaluation strategies, while quite useful, are by design restricted in their scope of applicability to *functional modules*.⁹ As we shall see in Chapter 6, *system modules*, specifying rewrite theories that are not functional, need not be Church-Rosser or terminating, and require much more general notions of strategy. Such general strategies are provided by Maude using reflection by means of *internal strategy languages*, in which strategies are defined by rewrite rules at the metalevel (see Section 14.6). However, as discussed in Section 4.4.9, specifying *frozen* arguments in operators restricts the rewrites allowed with rules in a system module (as opposed to equations) in a way quite similar to how operator evaluation strategies restrict the application of equations in a functional module.

4.4.8 Memo

If an operator is given the `memo` attribute, this instructs Maude to *memoize* the results of equational simplification (that is, the canonical forms) for those subterms having that operator at the top. This means that when the canonical form of a subterm having that operator at the top is obtained, an entry associating to that subterm its canonical form is stored in the memoization table for this operator. Whenever the Maude interpreter encounters a subterm whose top operator has the `memo` attribute, it looks to see if its canonical form is already stored. If so, that result is used; otherwise, equational simplification

⁹ More precisely, the scope of applicability of operator evaluation strategies is restricted to functional modules and to the *equational* part of system modules.

proceeds according to the operator's strategy. Giving to some operators the `memo` attribute allows trading off space for time in equational simplifications: more space is needed, but if subcomputations involving memoized operators have to be repeated many times, then a computation may be substantially sped up, provided that the machine's main memory limits are not exceeded.

An operator's `memo` attribute and its user's specified or default evaluation strategy (see Section 4.4.7) may interact with each other, impacting the size of the memoization table. The issue is how many entries for different subterms, all having the same canonical form, may be possibly stored in the memoization table. If the operator has the default, bottom-up strategy, the answer is: *only one such entry is possible*. For other strategies, different terms having the same canonical form may be stored, making the memoization table bigger. For example, using the default strategy (1 2 0) for a memoized operator `f`, only subterms of the form `f(v, v')` with `v` and `v'` fully reduced to canonical form (up to the strategies given for all operators) will be mapped to their corresponding canonical forms. This is because, with the default strategy, equational simplification at the top of `f` can only happen after all its arguments are in canonical form. For other operator strategies this uniqueness may be lost, even when evaluating just one subterm involving `f`. For example, if `f`'s strategy is (0 1 2 0), then both the starting term `f(t, t')` and the term `f(v, v')` (where `v` and `v'` are, respectively, the canonical forms of `t` and `t'`) will be mapped to the final result, since the strategy specifies rewriting at the top twice. That is, each time the operator's strategy calls for rewriting at the top, Maude will add the current version of the term to the set of terms that will be mapped to the final result. Furthermore, other terms of the form `f(u, u')`, with `u` and `u'` having also `v` and `v'` as their canonical forms may appear in other subcomputations, and will then also be stored in the memoization table.

In general, whenever an application will perform an operation many times, it may be useful to give that operator the `memo` attribute. This may be due to the high frequency with which the operator is called by other operators in a given application, or to the highly recursive nature of the equations defining that operator. For example, the recursive definition of the Fibonacci function is given as follows, where `NAT` denotes the predefined module of natural numbers and arithmetic operations on them (as described in Section 9.2), which is imported by means of a `protecting` declaration (see Section 8.1.1).

```
fmod FIBONACCI is
  protecting NAT .
  op fibo : Nat -> Nat .

  var N : Nat .
  eq fibo(0) = 0 .
  eq fibo(1) = 1 .
  eq fibo(s s N) = fibo(N) + fibo(s N) .
endfm
```

Due to the highly recursive nature of this definition of `fibonacci`, the evaluation of an expression like `fibonacci(50)` will compute many calls to the same instances of the function again and again, and will expand the original term into a whole binary tree of additions before collapsing it to a number. The exponential number of repeated function calls makes the evaluation of `fibonacci` with the above equations very inefficient, requiring over 61 billion rewrite steps for `fibonacci(50)`:

```
Maude> red fibonacci(50) .
reduce in FIBONACCI : fibonacci(50) .
rewrites: 61095033220 in 132081000ms cpu (145961720ms real)
(462557 rews/sec)
result NzNat: 12586269025
```

If we instead give the Fibonacci function the `memo` attribute,

```
op fibonacci : Nat -> Nat [memo] .
```

the change in performance is quite dramatic:

```
Maude> red fibonacci(50) .
reduce in FIBONACCI : fibonacci(50) .
rewrites: 148 in 0ms cpu (0ms real) (~ rews/sec)
result NzNat: 12586269025

Maude> red fibonacci(100) .
reduce in FIBONACCI : fibonacci(100) .
rewrites: 151 in 0ms cpu (1ms real) (~ rews/sec)
result NzNat: 354224848179261915075

Maude> red fibonacci(1000) .
reduce in FIBONACCI : fibonacci(1000) .
rewrites: 2701 in 0ms cpu (11ms real) (~ rews/sec)
result NzNat: 434665576869374564356885276750406258025646605173717804
024817290895365554179490518904038798400792551692959225930803226347
752096896232398733224711616429964409065331879382989696499285160037
04476137795166849228875
```

In some cases we may introduce a *constant operator* as an abbreviation for a possibly complex expression that may require a substantial number of equational simplification steps to be reduced to canonical form; furthermore, the operator may be used repeatedly in different subcomputations. In such cases one can declare a constant operator, give it the `memo` attribute, and give an equation defining it to be equal to the expression of interest. For example, suppose we have defined a search space with initial state `myState` and a function `findAnswer` to search the space for a state satisfying some property. Then we can name the search result and use it again without redoing an expensive computation as follows:

```
op myAns : -> Answer [memo] .
eq myAns = findAnswer(myState) .
```

Maude will then remember the result of rewriting the constant in the memoization table for that operator and will not repeat the work until the memoization tables are cleared. Memoization tables can be cleared explicitly by the command

```
do clear memo .
```

Automatic clearing before each top level rewriting command can be turned on and off with

```
set clear memo on .
set clear memo off .
```

By default, `set clear memo` is off.

4.4.9 Frozen arguments

The `frozen` attribute is only meaningful for system modules (see Chapter 6) that may have both rules and equations. It has no direct effect for functional modules having only equations and memberships: it can only have an *indirect* effect if the functional module is later imported by a system module. For this reason, examples of the use of frozen operators are postponed to Chapter 6.

Given a system module `M`, by declaring a given operator, say `f`, as `frozen`, rewriting with rules is always forbidden in all proper subterms of a term having `f` as its top operator. However, it may still be possible to rewrite that term at the top, provided rules having `f` as the top symbol of their lefthand side exist in `M`. To specify that all the arguments of an operator are frozen, one includes the attribute `frozen` in the operator's list of attributes; for example,

```
op f : S1 ... Sn -> S [frozen] .
```

The freezing idea can be generalized, so that only specific *argument positions* of the operator `f` are frozen. For example, in a system module specifying the semantics of a programming language with rewrite rules, we may want to specify a sequential composition operator `_;_` as frozen in its second argument, but not in the first argument, so as to prevent any execution of the second program fragment of the composition from happening before the first fragment has been fully evaluated. We can specify this by stating

```
op _;_ : Program Program -> Program [frozen (2)] .
```

More generally, if the list of argument positions in an operator `f` is $1 \dots n$, then we can freeze any sublist of argument positions, say $i_1 \dots i_m$, by declaring,

```
op f : S1 ... Sn -> S [frozen (i1 ... im)] .
```

Of course, if the actual list of specified positions is $1 \dots n$ itself, then this is equivalent to the first mode of declaring the `frozen` attribute for `f` without listing any positions.

As for operator evaluation strategies (see Section 4.4.7), in the case of associative or commutative binary operators mentioning either argument in the list of frozen positions implies both.

4.4.10 Special

Many operators in predefined modules (see Chapters 9 and 14) have the `special` attribute in their declarations. This means that they are to be treated as *built-in* operators, so that, instead of having the standard treatment of any user-defined operator, they are associated with appropriate C++ code by “hooks” which are specified following the `special` attribute identifier.

For example, the file `prelude.maude` contains a predefined module `NAT` for natural numbers and usual operations on them (see Section 9.2). Among others, the declarations in the `NAT` module for the operations of addition and of quotient of integer division, and for a less than predicate are the following:

```

op _+_ : NzNat Nat -> NzNat
  [assoc comm prec 33
   special (id-hook ACU_NumberOpSymbol (+)
            op-hook succSymbol (s_ : Nat ~> NzNat))] .
op _+_ : Nat Nat -> Nat [ditto] .

op _quo_ : Nat NzNat -> Nat
  [prec 31 gather (E e)
   special (id-hook NumberOpSymbol (quo)
            op-hook succSymbol (s_ : Nat ~> NzNat))] .

op _<_ : Nat Nat -> Bool
  [prec 37
   special (id-hook NumberOpSymbol (<)
            op-hook succSymbol (s_ : Nat ~> NzNat)
            term-hook trueTerm (true)
            term-hook falseTerm (false))] .

```

Notice that the `special` attribute exists in order to bind Maude syntax to built-in C++ functionality. It is absolutely not for users to mess with and it is absolutely not backwards compatible; this is why Maude will sometimes crash or become unstable if the prelude from a different version is loaded. For the same reason, other operator attributes that appear together with `special` in an operator declaration cannot be modified either.

4.5 Statement attributes

In a functional module, statements are equations and membership axioms, conditional or not. Any such statement may have associated *attributes*. Currently five attributes are available: `label`, `metadata`, `nonexec`, `owise`, and `print`. The attributes `label`, `metadata`, `nonexec`, and `print` can also be used on rules in system modules. Moreover, the attribute `metadata` can also be associated to operator declarations.

4.5.1 Labels

The `label` attribute must be followed by an identifier. Statement labels can be used for tracing and debugging and at the metalevel to name particular axioms. In our numbers example we could label the axiom for idempotency for natural number sets

```
eq N ; N = N [label natset-idem] .
```

Syntactic sugar for labels generalizing the Maude 1 style for rule labels is also supported. Then the above label could have also been written

```
eq [natset-idem] : N ; N = N .
```

4.5.2 Metadata

The `metadata` attribute must be followed by a string (that is, by a data element in the `STRING` module, see Section 9.8). The `metadata` attribute is intended to hold data about the statement in whatever syntax the user cares to create/parse. It is like a comment that is carried around with the statement. Usual string escape conventions apply. For example, we could add the distributive law

```
eq (N + M) * I = (N * I) + (M * I) [metadata "distributive law"] .
```

with the comment documenting that this is the distributive law.

The `metadata` attribute can also be associated to operator declarations. Note that, like `ctor`, `metadata` is attached to a specific operator declaration and not to the (possibly overloaded) operator itself. Thus:

- two subsorted overloaded declarations may have different `metadata` attributes,
- a `metadata` attribute is not copied by the `ditto` attribute (see Section 4.4.6), and
- a declaration may have a `metadata` attribute as well as a `ditto` attribute.

Under these conditions, the following ad-hoc example is therefore legal:

```
fmod METADATA-EX is
  sorts Foo Bar .
  subsort Foo < Bar .
  op f : Foo -> Foo [memo metadata "f on Foos"] .
  op f : Bar -> Bar [ditto metadata "f on Bars"] .
endfm
```

4.5.3 Nonexec

The `nonexec` attribute allows the user to include statements in a module that are ignored by the Maude rewrite engine. For example we could make the distributive law nonexecutable as follows.

```
eq (N + M) * I = (N * I) + (M * I)
  [nonexec metadata "distributive law"] .
```

Similarly, a rule can be declared with the `nonexec` attribute in a system module.

Although nonexecutable from the point of view of Core Maude, such statements are part of the semantics of the module and can for example be used at the metalevel for controlled execution or theorem proving purposes.

4.5.4 Otherwise

Sometimes, in the definition of an operation by equations, there are certain cases that can be easily defined by equations, and then some remaining case or cases that it is more difficult or cumbersome to define. One would in such situations like to say, *otherwise*, that is, in all remaining cases not covered by the above equations, do so and so.¹⁰

Consider, for example, the problem of membership of a natural number in a finite set of numbers.

```
op _in_ : Nat NatSet -> Bool .
```

The easy part is to define when a number belongs to a set:

```
var N : Nat .
var NS : NatSet .
eq N in N ; NS = true .
```

It is somewhat more involved to define when it *does not* belong. A simple way is to use the `otherwise` (abbreviated `owise`) attribute and give the additional equation:

```
eq N in NS = false [owise] .
```

¹⁰ Indeed, several languages have conventions of this kind, including ASF+SDF [104].

The intuitive operational meaning is clear: if the first equation does not match, then the number in fact is not in the set, and the predicate should be false. But what is the *mathematical* meaning? That is, how can we interpret the meaning of the second equation so that it becomes a useful *shorthand* for an ordinary equation? After all, the second equation, as given, is even more general than the first and in direct contradiction with it. We of course should reject any constructs that violate the logical semantics of the language.

Fortunately, there is nothing to worry about, since the **owise** attribute is indeed a shorthand for a corresponding *conditional* equation. We first explain the idea in the context of this example and then discuss the general construction. The idea is that, whether an equation, or a set of equations, defining the meaning of an operation f match a given term, is itself a property defined by a predicate, say $enabled_f$, which is effectively definable by equations. In our example we can introduce a predicate **enabled-in**, telling us when the first equation applies, by just giving its lefthand side arguments as the predicate's arguments:

```
op enabled-in : [Nat] [NatSet] -> [Bool] .
eq enabled-in(N, N ; NS) = true .
```

Note that we do not have to define when the **enabled-in** predicate is *false*. That is, this predicate is really defined on the kind `[Bool]`. Our second **owise** equation is simply a convenient shorthand for the *conditional* equation

```
ceq N in NS = false if enabled-in(N, NS) /= true .
```

This is just a special case of a completely general *theory transformation* that translates a specification containing equations with the **owise** attribute into a semantically equivalent specification with no such attributes at all. A somewhat subtle aspect of this transformation¹¹ is the interaction between **owise** equations and the operator evaluation strategies discussed in Section 4.4.7. Suppose that an **owise** equation was used in defining the semantics of an operator f . If f was (implicitly or explicitly) declared with a strategy, say,

$$f : s_1 \dots s_n \rightarrow s \quad [\text{strat } (i_1 \dots i_k 0)] .$$

then, the $enabled_f$ predicate should be defined with the *same* strategy,

$$enabled_f : [s_1] \dots [s_n] \rightarrow [Bool] \quad [\text{strat } (i_1 \dots i_k 0)] .$$

This will make sure that the reduction of f 's arguments prior to applying equations for f —including the equations that will be introduced in our transformation to replace the **owise** equations—takes place in exactly the same way for f and for $enabled_f$, so that failure of matching the normal equations is correctly captured by the failure of the $enabled_f$ predicate. Furthermore, as we shall see, after the failure of matching the non-**owise** equations, the

¹¹ We thank Joseph Hendrix for pointing out this subtlety.

matching substitution obtained when we apply the desugared version of an **owise** equation will then properly take into account the evaluation of those arguments of f specified by f 's evaluation strategy.

In general, if we are defining the equational semantics of an operation $f : s_1 \dots s_n \rightarrow s$ and we have given a partial definition of that operation by (possibly conditional) equations

$$\begin{aligned} f(u_1^1, \dots, u_n^1) &= t_1 \text{ if } C_1 \\ &\dots \\ f(u_1^m, \dots, u_n^m) &= t_m \text{ if } C_m \end{aligned}$$

then we can give one or more **owise** equations defining the function in the remaining cases by means of equations of the form

$$\begin{aligned} f(v_1^1, \dots, v_n^1) &= t'_1 \text{ if } C'_1 \text{ [owise]} \\ &\dots \\ f(v_1^k, \dots, v_n^k) &= t'_k \text{ if } C'_k \text{ [owise]} \end{aligned}$$

We can view such **owise** equations as shorthand notation for corresponding ordinary conditional equations of the form

$$\begin{aligned} f(y_1, \dots, y_n) &= t'_1 \text{ if } \text{enabled}_f(y_1, \dots, y_n) \neq \text{true} \\ &\quad \wedge \text{enabled}_f(v_1^1, \dots, v_n^1) := \text{enabled}_f(y_1, \dots, y_n) \\ &\quad \wedge C'_1 \\ &\dots \\ f(y_1, \dots, y_n) &= t'_k \text{ if } \text{enabled}_f(y_1, \dots, y_n) \neq \text{true} \\ &\quad \wedge \text{enabled}_f(v_1^k, \dots, v_n^k) := \text{enabled}_f(y_1, \dots, y_n) \\ &\quad \wedge C'_k \end{aligned}$$

where the variables y_1, \dots, y_n are *fresh new variables* not appearing in any of the above **owise** equations, and with y_i of kind $[s_i]$, $1 \leq i \leq n$. All this assumes that in the transformed specification we have declared the predicate $\text{enabled}_f : [s_1] \dots [s_n] \rightarrow [\text{Bool}]$, with the same evaluation strategy as f . Note the somewhat subtle use of the matching equations (see Section 4.3) $\text{enabled}_f(v_1^j, \dots, v_n^j) := \text{enabled}_f(y_1, \dots, y_n)$, $1 \leq j \leq k$, in the conditions. Since f and enabled_f have the same strategy, after the arguments of the matching instance of the expression $\text{enabled}_f(y_1, \dots, y_n)$ become evaluated according to the strategy, we are then able to match $\text{enabled}_f(v_1^j, \dots, v_n^j)$ to that result, obtaining the desired substitution for the variables of the lefthand side of the j^{th} **owise** equation. That is, we obtain the same substitution as the one we would have obtained matching $f(v_1^j, \dots, v_n^j)$ to the same subject term after its subterms under f had been evaluated according to f 's strategy.

Of course, the semantics of the enabled_f predicate is defined in the expected way by the equations

$$\text{enabled}_f(u_1^1, \dots, u_n^1) = \text{true if } C_1 .$$

...

$$\text{enabled}_f(u_1^m, \dots, u_n^m) = \text{true if } C_m .$$

The possibility of using multiple `owise` equations allows us to simplify definitions of functions defined by cases on data with nested structure. Here is a simple, if silly, example in which the sort `R` has elements `a(n)` and `b(n)`, for natural numbers `n`, and the sort `S` has elements `g(r)` and `h(r)`, with `r` of sort `R`. The operation `f` treats constructors `g` and `h` differently, distinguishing only whether the subterm of sort `R` is constructed by `a` or not. Again, the predefined module `NAT` of natural numbers (Section 9.2) is imported by means of a `protecting` declaration (Section 8.1.1).

```
fmod OWISE-TEST1 is
  protecting NAT .

  sorts R S .
  op f : S Nat -> Nat .
  ops g h : R -> S .
  ops a b : Nat -> R .

  var r : R .
  vars m n : Nat .
  eq f(g(a(m)), n) = n .
  eq f(h(a(m)), n) = n + 1 .
  eq f(g(r), n) = 0 [owise] .
  eq f(h(r), n) = 1 [owise] .
endfm
```

The four cases are illustrated by the following reductions.

```
Maude> red f(g(a(0)), 3) .
result NzNat: 3

Maude> red f(g(b(0)), 3) .
result Zero: 0

Maude> red f(h(b(0)), 3) .
result NzNat: 1

Maude> red f(h(a(0)), 3) .
result NzNat: 4
```

The subtle interaction between `owise` equations and operator evaluation strategies can be illustrated by the following example:

```
fmod OWISE-TEST2 is
  sort Foo .
  ops a b c d : -> Foo .
  op f : Foo -> Foo [strat (0 1 0)] .
```

```

op g : Foo -> Foo [strat (0)] .

var X : Foo .
eq b = c .
eq f(a) = d .
eq f(X) = g(X) [owise] .
endfm

```

Now consider the term $f(b)$. Intuitively, one could expect that, given that the first equation for f cannot be applied to this term, the `owise` equation is applied obtaining the term $g(b)$, and this is then expected to be the final result of the reduction, because the strategy (0) for g forbids evaluating its argument. However, as we can see in the following reduction, this is not the case.

```

Maude> red f(b) .
result Foo: g(c)

```

The result is $g(c)$, because the `owise` equation is not considered until after evaluating the final 0 in the strategy for f , and by then $f(b)$ is simplified to $f(c)$ as instructed by the 1 in such strategy; then, the `owise` equation applied to $f(c)$ produces $g(c)$.

It can be interesting to consider the semantically equivalent transformed specification:

```

fmod OWISE-TEST2-TRANSFORMED is
  sort Foo .
  ops a b c d : -> Foo .
  op f : Foo -> Foo [strat (0 1 0)] .
  op enabled-f : Foo -> Bool [strat (0 1 0)] .
  op g : Foo -> Foo [strat (0)] .

  vars X Y : Foo .
  eq b = c .
  eq f(a) = d .
  eq enabled-f(a) = true .
  ceq f(Y) = g(X)
    if enabled-f(Y) /= true /\ enabled-f(X) := enabled-f(Y) .
endfm

Maude> red f(b) .
result Foo: g(c)

```

where, as pointed out in our comments on the general transformation, the fact that `enabled-f` has the same strategy as f and the use of the matching equation

```

enabled-f(X) := enabled-f(Y)

```

are crucial for obtaining a semantically equivalent specification.

4.5.5 Print

The `print` attribute allows the user to specify information to be printed when a statement (equation, membership axiom, or rule) is executed. A `print` attribute declaration looks like

```
eq f(X) = b [print "X = " X] .
```

The keyword `print` is followed by a possibly empty list of items where each item is either a string constant or a variable. Mentioned variables must actually occur in the statement. If a non-occurring variable appears as a `print` item, it will be pruned and Maude will issue a warning.

Here is an example that uses the `print` attribute to track calls to a recursive function that reverses a list.

```
fmod PRINT-ATTRIBUTE-EX is
  sorts Foo FooList .
  ops a b : -> Foo [ctor] .
  subsort Foo < FooList .
  op nil : -> FooList [ctor] .
  op _ : FooList FooList -> FooList [ctor assoc id: nil] .

  op reverse : FooList -> FooList .
  eq reverse(nil) = nil .
  eq reverse(foo:Foo fl:FooList) = reverse(fl:FooList) foo:Foo
    [print "first = " foo:Foo ", rest = " fl:FooList] .
endfm
```

Maude will only use the `print` attribute in `print` attribute mode, which is off by default. Thus to run the above example (after loading it into Maude) it is necessary to execute the following command (see Section 25.7).

```
Maude> set print attribute on .
```

Then reducing the expression `reverse(a b a b)` results in the following output:

```
Maude> red in PRINT-ATTRIBUTE-EX : reverse(a b a b) .
reduce in PRINT-ATTRIBUTE-EX : reverse(a b a b) .
first = a, rest = b a b
first = b, rest = a b
first = a, rest = b
first = b, rest = nil
rewrites: 5 in 0ms cpu (0ms real) (18587 rewrites/second)
result FooList: b a b a
```

The `print` attribute is an alternative to tracing (see Section 24.1.1) to find out which statements Maude is executing. It allows the user control of what information is printed. It is also a nice way to show what is going on in demos.

4.6 Admissible functional modules

The `nonexec` attribute allows us to include arbitrary equations or memberships, conditional or not, in a functional module and likewise in a functional theory (see Section 8.3.1). Any such statement is then disregarded for purposes of execution by the Maude engine: it can only be used in a controlled way at the metalevel. But what about all the other statements? That is, what requirements should be imposed on *executable* equations and memberships so that they can be given an operational interpretation and can be executed by the Maude engine?

The intuitive idea is that we want to use such equations as *simplification rules* from left to right to reach a single final result or canonical form. For this purpose, the executable equations and memberships (that is, all statements not having the `nonexec` attribute) should be Church-Rosser and terminating (*modulo* the equational attributes declared in the module) in the sense explained in Section 4.7 below. This guarantees that, given a term t , all chains of equational simplification using those equations and memberships end in a unique canonical form (again, modulo the equational attributes). Furthermore, under the preregularity assumption (see Section 3.8), such a canonical form has the *smallest sort possible* in the subsort ordering.

The traditional requirement in this context is that, given a conditional equation¹² $t = t' \text{ if } C_1 \wedge \dots \wedge C_n$, the set of variables appearing in t contains those appearing in both t' and in the conditions C_i . In Maude, this requirement is relaxed to support matching equations in conditions (see Section 4.3) which can introduce new variables not present in t . Specifically, all executable conditional equations in a Maude functional module M have to satisfy the following *admissibility requirements*, ensuring that all the extra variables will become instantiated by matching:

1. $\text{vars}(t') \subseteq \text{vars}(t) \cup \bigcup_{j=1}^n \text{vars}(C_j)$.
2. If C_i is an equation $u_i = u'_i$ or a membership $u_i : s$, then

$$\text{vars}(C_i) \subseteq \text{vars}(t) \cup \bigcup_{j=1}^{i-1} \text{vars}(C_j).$$

3. If C_i is a matching equation $u_i := u'_i$, then u_i is an M -pattern and

$$\text{vars}(u'_i) \subseteq \text{vars}(t) \cup \bigcup_{j=1}^{i-1} \text{vars}(C_j).$$

¹² For the purposes of this discussion we can regard unconditional equations as the special case of conditional equations with empty condition, or with the condition $\text{true} = \text{true}$.

In a similar way, all executable conditional memberships $t : s \text{ if } C_1 \wedge \dots \wedge C_n$ must satisfy conditions 2–3 above.

In summary, therefore, we expect all executable equations and memberships in a functional module (and also in a system module) to be Church-Rosser and terminating (see Section 4.7 below, and [31, Section 10.1]) and to satisfy the above admissibility requirements.

4.7 Matching and equational simplification

Although this section and the next are quite technical, and it may be possible to skip them in a first reading, they introduce the concepts of *matching* and *equational simplification* that are essential to understand how Maude works. Therefore, we advise the reader to come back to them as needed to gain a better understanding of those concepts.

Recall from Section 4.3 that a functional module defines an equational theory $(\Sigma, E \cup A)$ in membership equational logic, with A the equations specified as equational attributes in operators (see Section 4.4.1), and E the (possibly conditional) equations and memberships specified as statements.

Ground terms in the signature Σ form a Σ -algebra denoted T_Σ . Similarly, equivalence classes of terms modulo $E \cup A$ define the Σ -algebra denoted $T_{\Sigma/E \cup A}$, which is the *initial model* for the theory $(\Sigma, E \cup A)$ specified by the module [248].

Given a set X of variables, we can add them to the signature Σ as new constants, and get in this way a term algebra $T_\Sigma(X)$ where now the terms may have variables in X .

Given a set X of variables, each having a given kind, a (ground) *substitution* is a kind-preserving function $\sigma : X \rightarrow T_\Sigma$. Such substitutions may be used to represent assignments of terms in T_Σ to the variables in X , and also assignments of elements in $T_{\Sigma/E \cup A}$ to such variables by σ picking up a representative of the corresponding $E \cup A$ -equivalence class. For example, a very natural choice is to assign to each x in X a term $\sigma(x)$ which is in *canonical form* according to $E \cup A$. Furthermore, under the preregularity, Church-Rosser, and termination assumptions (more on this below) this canonical form will have a *least sort*. Therefore, we may allow each variable x in X to have either a kind or a sort assigned to it, and can call the substitution σ *well-sorted* relative to $E \cup A$ if the least sort of $\sigma(x)$ is smaller or equal to that of x . By substituting terms for variables (as indicated by σ) in the usual way, a substitution $\sigma : X \rightarrow T_\Sigma$ is extended to a homomorphic function on terms $\sigma : T_\Sigma(X) \rightarrow T_\Sigma$ that we denote with the same name.

Given a term $t \in T_\Sigma(X)$, corresponding to the lefthand side of an oriented equation, and a subject ground term $u \in T_\Sigma$, we say that t *matches*¹³ u if there is a substitution σ such that $\sigma(t) \equiv u$, that is, $\sigma(t)$ and u are syntactically equal terms.

¹³ Some authors would instead say that u matches t .

For an oriented Σ -equation $l = r$ to be used in equational simplification, it is required that all variables in the righthand side r also appear among the variables of the lefthand side l . In the case of a conditional equation $l = r \text{ if } \textit{cond}$, this requirement is relaxed, so that more variables can appear in the condition \textit{cond} , provided that they are introduced by matching equations according to the admissibility requirements in Section 4.6; then the variables in the righthand side r must be among those in the lefthand side l or in the condition \textit{cond} . Under this assumption, given a theory (Σ, E) a term t *rewrites* to a term t' using such an equation if there is a subterm $t|_p$ of t at a given position¹⁴ p of t such that l matches $t|_p$ via a well-sorted substitution¹⁵ σ and t' is obtained from t by replacing the subterm $t|_p \equiv \sigma(l)$ with the term $\sigma(r)$. In addition, if the equation has a condition \textit{cond} , the substitution σ must make the condition provably true according to the equations and memberships in E , which are assumed to be Church-Rosser and terminating and are used also from left to right to try to simplify the condition. Note that, in general, the variables instantiated by σ must contain both those in the lefthand side, and those in the condition (which are incrementally matched using the matching equations).

We denote this step of *equational simplification* by $t \rightarrow_E t'$, where the possible equations for rewriting are chosen in the set E . The reflexive and transitive closure of the relation \rightarrow_E is denoted \rightarrow_E^* .

In many texts, equational simplification is also called (*equational*) *rewriting* but, since in Maude we have two very different types of rewriting, rewriting with *equations* in functional modules, and rewriting with *rules* in system modules, each with a completely different semantics, to avoid confusion we favor the terminology of equational simplification for the process of rewriting with equations.

A set of equations E is *confluent* when any two rewritings of a term can always be unified by further rewriting: if $t \rightarrow_E^* t_1$ and $t \rightarrow_E^* t_2$, then there exists a term t' such that $t_1 \rightarrow_E^* t'$ and $t_2 \rightarrow_E^* t'$. This is summarized in Figure 4.1. For an example of a nonconfluent specification, and a particular way to turn it into a confluent one, see Section 5.6.

A set of equations E is *terminating* when there is no infinite sequence of rewriting steps

$$t_0 \rightarrow_E t_1 \rightarrow_E t_2 \rightarrow_E \dots$$

If E is both confluent and terminating, a term t can be reduced to a unique *canonical form* $t \downarrow_E$, that is, to a unique term that can no longer be rewritten. Therefore, in order to check semantic equality of two terms $t = t'$ (that is, that they belong to the same equivalence class), it is enough to check that their

¹⁴ We can represent a term t as a tree, and use strings of numbers to identify positions p in the tree, thus identifying subterms $t|_p$. For example, for $t = f(g(a), h(b))$, we have $t|_{nil} = t$, $t|_1 = g(a)$, $t|_{11} = a$, $t|_2 = h(b)$, and $t|_{21} = b$.

¹⁵ Note that if a variable x has a sort s instead of a kind, well sortedness of σ means that $\sigma(x)$ must provably have sort s (or lower) according to the equations E .

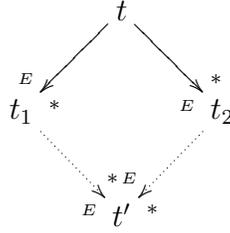


Fig. 4.1. Confluence diagram

respective canonical forms are equal, $t \downarrow_E = t' \downarrow_E$, but, since canonical forms cannot be rewritten anymore, the last equality is just syntactic coincidence: $t \downarrow_E \equiv t' \downarrow_E$.

In membership equational theories a third important property is *sort decreasingness*. Intuitively, this means that, assuming E is confluent and terminating, the canonical form $t \downarrow_E$ of a term t by the equations E should have the *least sort possible* among the sorts of all the terms equivalent to it by the equations E ; and it should be possible to compute this least sort from the canonical form itself, using only the operator declarations and the memberships. By a *Church-Rosser and terminating* theory (Σ, E) we precisely mean one that is confluent, terminating, and sort-decreasing. For a more detailed treatment of these properties, we refer the reader to the paper [31].

Since Maude functional modules have an initial algebra semantics, we are primarily interested in *ground* terms. Therefore, we can relax the above Church-Rosser and termination requirements by requiring that they just hold for ground terms, without losing the desired coincidence between the mathematical and operational semantics. In this way, we obtain notions of *ground* Church-Rosser, terminating, confluent, etc. specifications. In practice, some perfectly reasonable Maude functional modules are ground confluent, but fail to be confluent. This however is not a problem, since ground confluence (together with ground termination) is just what is needed to ensure uniqueness of canonical forms. Indeed, under the ground Church-Rosser and termination assumptions, it is easy to prove that we have the desired isomorphism

$$T_{\Sigma/E} \cong \text{Can}_{\Sigma/E}$$

ensuring the coincidence between the *mathematical semantics* of (Σ, E) provided by the initial algebra $T_{\Sigma/E}$, and its *operational semantics* by equational simplification provided by the algebra $\text{Can}_{\Sigma/E}$ of canonical forms.

Equational specifications (Σ, E) in Maude functional modules (and in the equational part of system modules), are assumed to be ground Church-Rosser and terminating up to the context-sensitive strategy specified by the evaluation strategies declared for the operators in Σ (see Section 4.4.7). More precisely, we can view the information about operator evaluation strategies

as a function μ that assigns to each operator $f \in \Sigma$ with n argument sorts a string of numbers indicating the argument positions to be evaluated and ended with a 0 (that is, the information given in the operator's **strat** attribute, or, if no such information is given, the string $1 \dots n 0$). This then defines a more restricted rewrite relation \rightarrow_E^μ where we can only rewrite in subterms in positions that can be evaluated according to μ . If the relation \rightarrow_E^μ is (ground) confluent, we call the specification (ground) μ -confluent; similarly, if \rightarrow_E^μ is (ground) terminating, we call it (ground) μ -terminating. We define the concepts of (ground) μ -sort-decreasing and (ground) μ -Church-Rosser in the same way. When we talk about the specification being “ground Church-Rosser and terminating up to the context-sensitive strategy specified by the evaluation strategies,” we exactly mean that it is ground μ -Church-Rosser and ground μ -terminating. Of course, when no such strategies are declared, this specializes to the usual notions of ground Church-Rosser and ground terminating. Under the ground μ -Church-Rosser and ground μ -terminating assumptions, the μ -canonical forms define a canonical term algebra $Can_{\Sigma/E}^\mu$ (see [188]), which provides a perfect mathematical model for the module's *operational semantics*, since its elements are the *values* that the user gets when evaluating expressions in such a module. The question then arises: how is this model related to the module's *mathematical semantics*? In general, the quotient map $t \mapsto [t]_E$ sending each μ -canonical form to its E -equivalence class is a *surjective* homomorphism

$$q : Can_{\Sigma/E}^\mu \longrightarrow T_{\Sigma/E},$$

but not necessarily an isomorphism. If q fails to be an isomorphism, this means that μ -rewriting is a *sound* deductive method for proving E -equalities, but it is *incomplete*. Therefore we call the specification μ -semantically complete iff q is an isomorphism. μ -semantic completeness therefore expresses the complete agreement between the mathematical and operational semantics of the module. Specifications where evaluation strategies are used mainly for efficiency and/or termination purposes, that is, those where the execution becomes more efficient by avoiding wasteful computation in unnecessary parts of the term and/or that would not terminate without the strategy restrictions are typically μ -semantically complete. Instead, specifications such as the sieve of Eratosthenes in Section 4.4.7, where the main intent is to compute with infinite data structures in a lazy way, are typically μ -semantically incomplete. Not all is lost in this second case: we still have a good mathematical model associated to our specification, namely, $Can_{\Sigma/E}^\mu$, but this is a more concrete model than $T_{\Sigma/E}$, that is, one in which fewer elements are identified.

What are the appropriate notions when we have a theory of the form $(\Sigma, E \cup A)$? Then matching must be defined *modulo the equational axioms* A , and all the above concepts, including those for μ -rewriting, must be generalized to equational simplification, confluence, and termination *modulo* A . We discuss this in more detail in Section 4.8 below. See also [188] for a detailed treatment of μ -rewriting and μ -semantic completeness modulo axioms A .

As already mentioned, the operational semantics of functional modules is *equational simplification*, that is, equational rewriting of terms until a canonical form is obtained in the sense explained above. Notice that the system does not check the ground confluence and termination properties: they are left to the user's responsibility. However, in some cases it is possible to check these properties with Maude's Church-Rosser checker and termination tools [73, 126, 118, 117]. Similar checkings are also possible for functional modules with evaluation strategies; for example, the Maude's MTT termination tool can check μ -termination (also called context-sensitive termination [223]). Moreover, although the relations between the standard Church-Rosser property and the μ -Church-Rosser property are somewhat subtle [222, 224], the work in [188] shows how one can use standard tools in conjunction with Maude's Sufficient Completeness Checker [190] to check both the μ -Church-Rosser property and μ -semantic completeness. See Sections 23.1.3, 23.1.2, and 23.1.5 for more information on the tools, and Sections 12.4 and 13.4 for some examples of their use.

4.8 More on matching and simplification modulo

In the Maude implementation, rewriting modulo A is accomplished by using a *matching modulo A algorithm*. More precisely, given an equational theory A , a term t (corresponding to the lefthand side of an equation) and a subject term u , we say that t *matches u modulo A* (or that t *A -matches u*) if there is a substitution σ such that $\sigma(t) =_A u$, that is, $\sigma(t)$ and u are equal modulo the equational theory A (compare with the syntactic definition of matching in Section 4.7 above).

Given an equational theory $A = \cup_i A_{f_i}$ corresponding to all the attributes declared in different binary operators, Maude synthesizes a combined matching algorithm for the theory A , and does both equational simplification (with equations) and rewriting (with rules in system modules, see Chapter 6) modulo the axioms A .

Note, however, that for operators f whose equational axioms A include the associativity axiom, to achieve the effect of simplification modulo A using an A -matching algorithm, we have to attempt matching a lefthand side of the form $f(t_1, t_2)$ not only on a subject term u , but also on all its f -subterms, that is, on those "fragments" of the top structure of the term that could be matched by $f(t_1, t_2)$. For example, assuming a binary associative operator \mathbf{f} and constants \mathbf{a} , \mathbf{b} , \mathbf{c} , and \mathbf{d} of the appropriate sort, the term $t = \mathbf{f}(\mathbf{a}, \mathbf{b})$ does not match the term $u = \mathbf{f}(\mathbf{a}, \mathbf{f}(\mathbf{b}, \mathbf{f}(\mathbf{c}, \mathbf{d})))$, that is, there is no substitution making both terms equal modulo associativity; however, because of associativity of \mathbf{f} , u is equivalent to $\mathbf{f}(\mathbf{f}(\mathbf{a}, \mathbf{b}), \mathbf{f}(\mathbf{c}, \mathbf{d}))$ and then t trivially matches the first subterm. This becomes easier to see using mixfix notation; if $f = _._.$, then $t = \mathbf{a} . \mathbf{b}$ and $u = \mathbf{a} . \mathbf{b} . \mathbf{c} . \mathbf{d}$, and we clearly

see that t matches a fragment of u . For the case where the only axiom is associativity, the $_._.$ -subterms of $a . b . c . d$ are

```
a . b
a . b . c
b . c
b . c . d
c . d
```

If the operation $_._.$ had been declared both associative and commutative, then we should add to those the additional subterms

```
a . c
a . d
b . d
a . b . d
a . c . d
```

If the term $f(t_1, t_2)$ matches either u or an f -subterm of u modulo A , then we say that $f(t_1, t_2)$ *matches u with extension modulo A* (or that $f(t_1, t_2)$ *A -matches u with extension*). For example, the lefthand side of the equation $a . b = e$ matches $a . b . c . d$ with extension modulo associativity, and the lefthand side of $a . d = g$ matches $a . b . c . d$ with extension modulo associativity and commutativity.

For f a binary operator with equational attributes A_f including the associativity axiom, we now define how a subject term u is A_f -rewritten with extension using an equation $f(t_1, t_2) = v$. First of all, $f(t_1, t_2)$ must A_f -match with extension a *maximal f -subterm* w of u (that is, an f -subterm of u that is not itself an f -subterm of a bigger f -subterm). This means that there is an f -subterm w_0 of w and a substitution σ such that $\sigma(f(t_1, t_2)) =_{A_f} w_0$. Then, the corresponding A_f -rewriting with extension step rewrites u to the term obtained by replacing the subterm w_0 by $\sigma(v)$.

Note that a term $f(t_1, t_2)$ A_f -matches with extension a maximal f -subterm if and only if it A_f -matches without extension *some* f -subterm. This is of course the important practical advantage of A -matching and A -rewriting with extension, namely, that only maximal f -subterms of a term have to be inspected to get the effect of rewriting equivalence classes modulo A . For more technical details on rewriting modulo a set of axioms, see, e.g., [102].

Matching with extension for an associative operator essentially corresponds to matching without extension for a collection of associated equations. For example, we could have “generalized” the equation $a . b = e$ with $_._.$ associative to the equations

```
eq a . b = e .
eq X . a . b = X . e .
eq a . b . Y = e . Y .
eq X . a . b . Y = X . e . Y .
```

so that we could have achieved the same effect by rewriting only at the top of maximal f -subterms (without extension). Similarly, for $_._$ associative and commutative, we could have generalized the same equation $\mathbf{a} . \mathbf{b} = \mathbf{e}$ to the equations

$$\begin{aligned} \text{eq } \mathbf{a} . \mathbf{b} &= \mathbf{e} . \\ \text{eq } \mathbf{a} . \mathbf{b} . \mathbf{Y} &= \mathbf{e} . \mathbf{Y} . \end{aligned}$$

In Maude this generalization does not have to be performed explicitly as a transformation of the specification. It is instead achieved implicitly in a built-in way by performing A -matching with extension. If the equational axioms declared for a binary operator f include the associativity axiom, then a subject term u with f as its top operator is internally represented (but this representation can also be externally used, see Section 3.9.3) as the *flattened term* $f(u_1, \dots, u_n)$, with the u_1, \dots, u_n having top operators different from f . Furthermore, if a (two-sided) identity element e has been declared for f , then $u_i \neq e$, $1 \leq i \leq n$. That is, we assume in this case that all identities have been simplified away.

Relative to this internal representation, it is then easy to define the notion of an f -subterm. If the axioms of f include associativity but not commutativity, then the f -subterms of the term $f(u_1, \dots, u_n)$ are all terms of the form $f(u_k, \dots, u_{k+h})$ with $1 \leq k \leq n-1$ and $1 \leq h \leq n-k$.

Similarly, if the axioms of f include associativity and commutativity, then the f -subterms of $f(u_1, \dots, u_n)$ are all terms of the form $f(u_{k_1}, \dots, u_{k_h})$ with $1 \leq k_{i_1} < \dots < k_{i_h} \leq n$, and $2 \leq h \leq n$.

The concepts of positions in a term and depth of a term, that are important in many situations, refer to this flattened form. The compact notation for terms constructed with operators having the `iter` attribute (Section 4.4.2) is also considered a form of flattened notation, so that, for the purpose of calculating term depth, if the top level is at level 0, then the occurrence of $\mathbf{X}:\mathbf{Foo}$ in $\mathbf{f}^3(\mathbf{X}:\mathbf{Foo})$ is at level 1, not level 3.

Adding axioms for an identity element e to a possibly associative and/or commutative operation f leads to some subtle cases, where the proper application of the general notions may not always coincide with the user's expectations. To begin with, unexpected cases of *nontermination* may be introduced by an unwary user. For example, the equation

$$\text{eq } \mathbf{a} . \mathbf{X} = \mathbf{b} . \mathbf{a} .$$

will cause nontermination when $_._$ is declared associative with identity 1, since we have, for example,

$$\begin{aligned} \mathbf{d} . \mathbf{a} &\rightarrow \mathbf{d} . \mathbf{b} . \mathbf{a} \\ &\rightarrow \mathbf{d} . \mathbf{b} . \mathbf{b} . \mathbf{a} \\ &\vdots \\ &\rightarrow \mathbf{d} . \mathbf{b} . \mathbf{b} . \dots . \mathbf{b} . \mathbf{a} \end{aligned}$$

$$\vdots$$

by instantiating each time the variable X to the identity element 1 .

A second source of unexpected behavior is the fact that a lefthand side involving an associative operator may, in the presence of an additional identity attribute, match a term not involving at all that operator. Thus, for the above equation, we have also the nonterminating chain of rewriting steps

$$\begin{aligned} a &\rightarrow b . a \\ &\rightarrow b . b . a \\ &\vdots \\ &\rightarrow b . b . \dots . b . a \\ &\vdots \end{aligned}$$

In a similar way, in the presence of an identity element, the user's expectations about when a lefthand side will match with extension a subject term may not fully agree with the proper technical definition. Consider, for example, a binary operation $_.$ that is associative and commutative, and that has an identity element 1 , and let

$$\text{eq } a . X = c .$$

be an equation. Then,

1. The lefthand side $a . X$ matches the subject term a modulo the axioms by instantiating X to 1 , giving rise to the simplification

$$a \rightarrow c.$$

2. The same lefthand side matches the subject term $a . b . c$ with extension in *three* different ways, namely, with substitutions $X \mapsto b . c$, $X \mapsto b$, and $X \mapsto c$, giving rise to the three simplifications

$$a . b . c \rightarrow c$$

$$a . b . c \rightarrow c . c$$

$$a . b . c \rightarrow b . c$$

3. For the same subject term $a . b . c$, the substitution $X \mapsto 1$ *is not* a match with extension of the above lefthand side, because the term $a . 1$ is not a $_.$ -subterm of the term $a . b . c$. However, because of item 1 above, we know that the equation will match that way not at the top, but "one level down," leading to the simplification

$$a . b . c \rightarrow c . b . c$$

It is also important to realize that there is no match with extension between the lefthand side of the equation $a = b$ and the subject term $a . b . c$ (because the lefthand side a is not a $_._$ -term), although again the equation will match that way not at the top, but “one level down,” leading to the simplification

$$a . b . c \rightarrow b . b . c$$

Of course, lefthand sides may contain several operators, each matched modulo a different theory. Maude will then match each fragment of a lefthand side according to its given theory.

Consider, for example, the following specification where $_._$ is associative and $+_$ is associative and commutative:

```
fmod XMATCH-TEST is
  sort Elt .
  ops a b c d e : -> Elt .
  op _._ : Elt Elt -> Elt [assoc] .
  op _+_ : Elt Elt -> Elt [assoc comm] .
  vars X Y Z : Elt .
  eq X . (Y + Z) = (X . Y) + (X . Z) [metadata "distributivity"] .
endfm
```

The lefthand side of the distributivity equation will produce 12 matches with extension for the subject term

$$a . b . (c + d + e)$$

Enumerating these by hand would be tedious and error prone, however Maude provides the `xmatch` command (see also Section 25.3) for just this purpose:

```
xmatch X . (Y + Z) <=? a . b . (c + d + e) .
```

The output given by Maude consists of the substitution for each match with extension together with the portion of the subject actually matched:

```
Maude> xmatch X . (Y + Z) <=? a . b . (c + d + e) .
xmatch in XMATCH-TEST : X . Z + Y <=? a . b . c + d + e .
Decision time: 0ms cpu (0ms real)
```

```
Solution 1
Matched portion = (whole)
X:Elt --> a . b
Y:Elt --> c
Z:Elt --> d + e
```

```
Solution 2
Matched portion = b . (c + d + e)
X:Elt --> b
Y:Elt --> c
```

Z:Elt --> d + e

Solution 3

Matched portion = (whole)

X:Elt --> a . b

Y:Elt --> d

Z:Elt --> c + e

Solution 4

Matched portion = b . (c + d + e)

X:Elt --> b

Y:Elt --> d

Z:Elt --> c + e

Solution 5

Matched portion = (whole)

X:Elt --> a . b

Y:Elt --> e

Z:Elt --> c + d

Solution 6

Matched portion = b . (c + d + e)

X:Elt --> b

Y:Elt --> e

Z:Elt --> c + d

Solution 7

Matched portion = (whole)

X:Elt --> a . b

Y:Elt --> c + d

Z:Elt --> e

Solution 8

Matched portion = b . (c + d + e)

X:Elt --> b

Y:Elt --> c + d

Z:Elt --> e

Solution 9

Matched portion = (whole)

X:Elt --> a . b

Y:Elt --> c + e

Z:Elt --> d

Solution 10

Matched portion = b . (c + d + e)

X:Elt --> b

Y:Elt --> c + e

Z:Elt --> d

```
Solution 11
Matched portion = (whole)
X:Elt --> a . b
Y:Elt --> d + e
Z:Elt --> c
```

```
Solution 12
Matched portion = b . (c + d + e)
X:Elt --> b
Y:Elt --> d + e
Z:Elt --> c
```

Note that extension is only used for matching the top operation, `_._` in this example, but not `+_+`. This is the reason why the subterm `Y + Z` of the lefthand side should match the entire maximal `+_+`-subterm of the subject term, and not just some `+_+`-subterm.

For operators with the `iter` attribute, the situation with matching is analogous to the `assoc` theory, so that proper subterms of say `f^3(X:Foo)`, such as `f^2(X:Foo)` and `f(X:Foo)`, can also be matched by means of extension.

4.9 The reduce, match, trace, and show commands

Here we assemble the whole module for the `NUMBERS` running example to illustrate some of the basic commands for interacting with Maude. See Chapter 25 for full details about these and other Maude commands.

Notice that, since the result of the `_in_` predicate is a Boolean value, we import the predefined module `BOOL` (see Section 9.1) by means of a `protecting` declaration (described in Section 8.1.1).

```
fmod NUMBERS is
  protecting BOOL .

  sort Zero .
  sorts Nat NzNat .
  subsort Zero NzNat < Nat .
  op zero : -> Zero [ctor] .
  op s_ : Nat -> NzNat [ctor] .
  op sd : Nat Nat -> Nat .
  ops _+_ *__ : Nat Nat -> Nat [assoc comm] .
  op _+_ : NzNat Nat -> NzNat [ditto] .
  op *__ : NzNat NzNat -> NzNat [ditto] .
  op p : NzNat -> Nat .

  vars I N M : Nat .
  eq N + zero = N .
  eq N + s M = s (N + M) .
```

```

eq sd(N, N) = zero .
eq sd(N, zero) = N .
eq sd(zero, N) = N .
eq sd(s N, s M) = sd(N, M) .
eq N * zero = zero .
eq N * s M = (N * M) + N .
eq p(s N) = N [label partial-predecessor] .

eq (N + M) * I = (N * I) + (M * I)
  [nonexec metadata "distributive law"] .

sort Nat3 .
ops 0 1 2 : -> Nat3 [ctor] .
op _+_ : Nat3 Nat3 -> Nat3 [comm] .
vars N3 : Nat3 .
eq N3 + 0 = N3 .
eq 1 + 1 = 2 .
eq 1 + 2 = 0 .
eq 2 + 2 = 1 .

sort NatSeq .
subsort Nat < NatSeq .
op nil : -> NatSeq .
op __ : NatSeq NatSeq -> NatSeq [assoc id: nil] .

sort NatSet .
subsort Nat < NatSet .
op empty : -> NatSet .
op _;_ : NatSet NatSet -> NatSet [assoc comm id: empty] .
eq N ; N = N [label natset-idem] .

op _in_ : Nat NatSet -> Bool .
var NS : NatSet .
eq N in N ; NS = true .
eq N in NS = false [owise] .
endfm

```

First, we evaluate some expressions using the `reduce` command. Maude repeats the command filling in any omitted optional information. Then statistics about the execution are printed.¹⁶ Finally, the result is printed, prefaced by its least sort.

The first two examples evaluate the sum of three ones in `Nat` and in `Nat3`.

```

Maude> red s zero + s zero + s zero .
reduce in NUMBERS : s zero + s zero + s zero .
rewrites: 4 in 0ms cpu (0ms real) (~ rews/sec)

```

¹⁶ The cpu and real time information is not printed if the user has made use of the `set show timing off` command (see Section 25.9).

```

result NzNat: s s s zero

Maude> red 1 + (1 + 1) .
reduce in NUMBERS : 1 + (1 + 1) .
rewrites: 2 in 0ms cpu (0ms real) (~ rews/sec)
result Nat3: 0

```

The next example illustrates the effect of the idempotency equation for sets of natural numbers.

```

Maude> red zero ; s zero ; zero ; s zero .
reduce in NUMBERS : zero ; s zero ; zero ; s zero .
rewrites: 2 in 0ms cpu (0ms real) (~ rews/sec)
result NatSet: zero ; s zero

```

Finally we convince ourselves that the `owise` attribute works.

```

Maude> red zero in s zero ; zero ; s s zero .
reduce in NUMBERS : zero in s zero ; zero ; s s zero .
rewrites: 1 in 0ms cpu (0ms real) (~ rews/sec)
result Bool: true

```

```

Maude> red zero in s zero ; s s zero .
reduce in NUMBERS : zero in s zero ; s s zero .
rewrites: 1 in 0ms cpu (0ms real) (~ rews/sec)
result Bool: false

```

The commands `xmatch` and `match` operate in the same way, unless the subject term has an operator that needs extension on top, in which case it can match proper subterms in the same theory layer, as required for rewriting modulo that theory. The `xmatch` command was illustrated in Section 4.8. Here we compare `match` and `xmatch` on a pattern that splits a sequence of natural numbers into two parts. To be safe, we ask for at most five matches, but in fact there are only four.

```

Maude> match [5] NS0:NatSeq NS1:NatSeq <=? zero zero zero .
match [5] in NUMBERS : NS0:NatSeq NS1:NatSeq <=? zero zero zero .
Decision time: 0ms cpu (0ms real)

```

```

Solution 1
NS0:NatSeq --> nil
NS1:NatSeq --> zero zero zero

```

```

Solution 2
NS0:NatSeq --> zero
NS1:NatSeq --> zero zero

```

```

Solution 3
NS0:NatSeq --> zero zero
NS1:NatSeq --> zero

```

```
Solution 4
NSO:NatSeq --> zero zero zero
NS1:NatSeq --> nil
```

Using the `xmatch` command for the same pattern and term, we see that in addition to the whole term matches, Maude also reports matches within the subterm `zero zero`. In fact, there are two occurrences of this subterm. We only show five of the matches.

```
Maude> xmatch [5] NSO:NatSeq NS1:NatSeq <=? zero zero zero .
xmatch [5] in NUMBERS : NSO:NatSeq NS1:NatSeq <=? zero zero zero .
Decision time: 0ms cpu (7ms real)
```

```
Solution 1
Matched portion = zero zero
NSO:NatSeq --> nil
NS1:NatSeq --> zero zero
```

```
Solution 2
Matched portion = zero zero
NSO:NatSeq --> zero
NS1:NatSeq --> zero
```

```
Solution 3
Matched portion = zero zero
NSO:NatSeq --> zero zero
NS1:NatSeq --> nil
```

```
Solution 4
Matched portion = (whole)
NSO:NatSeq --> nil
NS1:NatSeq --> zero zero zero
```

```
Solution 5
Matched portion = (whole)
NSO:NatSeq --> zero
NS1:NatSeq --> zero zero
```

Let us consider now a small example using the `trace` command. We turn on selective tracing and choose to trace only uses of the equation labeled `partial-predecessor`.

```
Maude> set trace on .
Maude> set trace select on .
Maude> trace select partial-predecessor .

Maude> red s s p(s zero) + s p(s zero) .
reduce in NUMBERS : s s p(s zero) + s p(s zero) .
***** equation
```

```

eq p(s N) = N [label partial-predecessor] .
N --> zero
p(s zero)
--->
zero
rewrites: 3 in 0ms cpu (0ms real) (~ rews/sec)
result NzNat: s s s zero

```

Note that Maude only reports one use of this equation, despite the fact that there are two occurrences in the term. This is because, when performing equational simplification, occurrences of the same subterm are internally shared¹⁷ and hence there is only one occurrence of the subterm `p(s zero)` in the internal representation.

We can ask Maude to show the module `FIBONACCI` (assuming it has been loaded).

```

Maude> show module FIBONACCI .
fmod FIBONACCI is
  protecting NAT .
  op fibo : Nat -> Nat [memo] .
  var N : Nat .
  eq fibo(0) = 0 .
  eq fibo(1) = 1 .
  eq fibo(s s N) = fibo(N) + fibo(s N) .
endfm

```

The `show sorts` command shows all the sorts declared and for each sort its sub- and super-sorts.

```

Maude> show sorts NUMBERS .
sort Bool .
sort Zero . subsorts Zero < Nat NatSet NatSeq .
sort Nat . subsorts NzNat Zero < Nat < NatSet NatSeq .
sort NzNat . subsorts NzNat < Nat NatSet NatSeq .
sort Nat3 .
sort NatSeq . subsorts NzNat Zero Nat < NatSeq .
sort NatSet . subsorts NzNat Zero Nat < NatSet .

```

¹⁷ However, this sharing—i.e., treating the term as a dag instead of as a tree—is not done in a *maximal* way, so that all subterms that can be shared are; instead, term sharing is itself introduced incrementally by equational simplification, since Maude analyzes righthand sides of equations to identify its shared subterms. As explained by Eker in [135], in the presence of operator evaluation strategies (Section 4.4.7) term sharing has to be done carefully. Furthermore, when rewriting is performed with a rule in a system module (see Chapter 6), rather than with an equation, Maude will incrementally *unshare* those parts of the subject term needed to ensure that *all* possible rewrite with rules are considered. This is because rules in system modules need not be confluent. As a consequence, two identical subterms could be rewritten in totally different ways; but this of course would be prevented if they were to be shared.

The `show components` command shows the connected components (kinds) in the sort partial order.

```
Maude> show components NUMBERS .
[Bool]:
  1      Bool

[NatSeq, NatSet]:
  1      NatSeq
  2      NatSet
  3      Nat
  4      Zero
  5      NzNat

[Nat3] (error free):
  1      Nat3
```

Note that the name of the kind corresponding to the connected component containing the natural numbers contains the names of two sorts. These are the maximal sorts in the component. The `(error free)` comment about the sort `Nat3` means that all terms of kind `[Nat3]` are in fact of sort `Nat3`.

4.10 A number hierarchy

As further examples of Maude functional modules, we describe below a hierarchy of modules specifying the number hierarchy from the natural to the rational numbers in a somewhat different form than in [174, C.7]. Since `NAT`, `INT`, and `RAT` are predefined modules in the Maude prelude (see Chapter 9), to avoid clashes with the predefined versions we call the modules below `PEANO-NAT`, `PEANO-INT`, and `PEANO-RAT`, because they follow the natural number notation based on zero and successor, as in the Peano axiomatization.

The natural numbers are defined by means of two *constructors*, zero (`0`) and successor (`s_`). Since some operations are undefined for zero, we introduce the subsort `NzNat` of nonzero natural numbers, which becomes the result sort for the successor operator. Then the predecessor `p_` is the inverse of the successor over nonzero natural numbers, so that the term `p 0` is undefined.

We consider several typical arithmetical operations on natural numbers: addition, product, difference (denoted `d`), quotient (whose second argument is of sort `NzNat`, thus avoiding division by zero), greatest common divisor, and a greater than predicate.

Notice that most of the arithmetical operators are declared as *commutative* by means of an equational attribute `comm`; this applies in particular to the difference operator `d`, which is symmetric because its result when applied to two natural numbers is the result of subtracting the least from the greatest of the two.

Moreover, the operators for addition `+_`, for product `*_`, and for greatest common divisor `gcd` are *overloaded*, with one declaration for `Nat` and another for `NzNat`, thus making explicit the fact that when applied to two positive natural numbers, the corresponding result is also positive.

The result of the greater than `>` predicate is a Boolean value. Instead of defining our own Boolean values, we will make use of the predefined module `BOOL` described in Section 9.1, which provides values `true` and `false`, together with typical Boolean operations on them. Here, we import `BOOL` by means of a `protecting` declaration, which will be described in Section 8.1.1.

```
fmod PEANO-NAT is
  protecting BOOL .
  sorts Nat NzNat Zero .
  subsorts Zero NzNat < Nat .
  op 0 : -> Zero [ctor] .
  op s_ : Nat -> NzNat [ctor] .
  op p_ : NzNat -> Nat .
  op +_ : Nat Nat -> Nat [comm] .
  op +_ : NzNat NzNat -> NzNat [comm] .
  op *_ : Nat Nat -> Nat [comm] .
  op *_ : NzNat NzNat -> NzNat [comm] .
  op >_ : Nat Nat -> Bool .
  op d : Nat Nat -> Nat [comm] .
  op quot : Nat NzNat -> Nat .
  op gcd : Nat Nat -> Nat [comm] .
  op gcd : NzNat NzNat -> NzNat [comm] .
```

Except for the constructors, the remaining operations are equationally defined in an inductive way by distinguishing different cases over the constructors.

For addition, product and difference, due to its commutativity attribute, it is enough to have two equations to deal with the case in which one of the arguments is zero and the case in which both are positive and therefore of the form $s\ N$ for some natural number N . The equations for the greater than predicate are similar, but we need three equations because this operator is not commutative; notice that $N' > 0$ is true in the second equation, because the variable N' is assumed to have sort `NzNat`.

On the other hand, the case distinction for quotient and greatest common divisor is not based on constructors but on comparing the values of the two arguments, making good use of conditional equations. In particular, the equations for `gcd` follow the well-known Euclidean algorithm, based on repeatedly subtracting the least argument from the greatest until both coincide.

```
vars N M : Nat .
vars N' M' : NzNat .
eq p s N = N .
eq N + 0 = N .
eq (s N) + (s M) = s s (N + M) .
```

```

eq N * 0 = 0 .
eq (s N) * (s M) = s (N + (M + (N * M))) .
eq 0 > M = false .
eq N' > 0 = true .
eq s N > s M = N > M .
eq d(0, N) = N .
eq d(s N, s M) = d(N, M) .
ceq quot(N, M') = s quot(d(N, M'), M') if N > M' .
eq quot(M', M') = s 0 .
ceq quot(N, M') = 0 if M' > N .
eq gcd(0, N) = 0 .
eq gcd(N', N') = N' .
ceq gcd(N', M') = gcd(d(N', M'), M') if N' > M' .
endfm

```

Now we specify the integer numbers as an extension of the natural numbers. The specification of the latter is imported by means of a `protecting` declaration, meaning (as explained in Section 8.1.1) that the natural numbers are not disturbed by this extension.

We add supersorts `Int` and `NzInt` of `Nat` and `NzNat`, respectively. The new constructor that builds negative integers is the unary minus operator `-`, which is self-inverse, as the first equation asserts. The remaining operations are extensions to the integers of previous operations on natural numbers.

```

fmod PEANO-INT is
protecting PEANO-NAT .
sorts Int NzInt .
subsort Nat < Int .
subsorts NzNat < NzInt < Int .
op -_ : Int -> Int [ctor] .
op -_ : NzInt -> NzInt [ctor] .
op +_ : Int Int -> Int [comm] .
op *_ : Int Int -> Int [comm] .
op *_ : NzInt NzInt -> NzInt [comm] .
op quot : Int NzInt -> Int .
op gcd : Int Int -> Nat [comm] .
op gcd : NzInt NzInt -> NzNat [comm] .

```

The equations for product, quotient, and `gcd` simply treat appropriately the sign. But the new equations for addition of integers have to distinguish cases according to whether the two arguments have the same or different sign; in the latter case, the addition reduces by means of two conditional equations to the (symmetric) difference `d` on natural numbers.

```

vars I J : Int .
vars I' J' : NzInt .
vars N' M' : NzNat .
eq - - I = I .
eq - 0 = 0 .

```

```

eq I + 0 = I .
eq M' + (- M') = 0 .
ceq M' + (- N') = - d(N', M') if N' > M' .
ceq M' + (- N') = d(N', M') if M' > N' .
eq (- I) + (- J) = - (I + J) .
eq I * 0 = 0 .
eq I * (- J) = - (I * J) .
eq quot(0, I') = 0 .
eq quot(- I', J') = - quot(I', J') .
eq quot(I', - J') = - quot(I', J') .
eq gcd(- I', J') = gcd(I', J') .
endfm

```

We follow a similar pattern to specify rational numbers on top of the integers.

First we import the module PEANO-INT in `protecting` mode, because integers are not modified in any way by this extension. Then, we add a new division operator `_/_` that is a constructor building rational numbers from the integers. The operations for unary minus, addition, and product are extended to the larger set of numbers.

```

fmod PEANO-RAT is
  protecting PEANO-INT .
  sorts Rat NzRat .
  subsort Int < Rat .
  subsorts NzInt < NzRat < Rat .
  op _/_ : Rat NzRat -> Rat [ctor] .
  op _/_ : NzRat NzRat -> NzRat [ctor] .
  op -_ : Rat -> Rat .
  op -_ : NzRat -> NzRat .
  op +_ : Rat Rat -> Rat [comm] .
  op *_ : Rat Rat -> Rat [comm] .
  op *_ : NzRat NzRat -> NzRat [comm] .

```

The first two equations allow to simplify the result of iterating the division operator, so that in the end we have a fraction with two integers as numerator and denominator. Moreover, the third equation simplifies such a fraction of integers to its reduced form, by dividing both numerator and denominator by its greatest common divisor. Further simplification is obtained by the next two equations, that treat the cases in which the denominator is one or the numerator is zero.

The last equations define unary minus, addition, and product on rational numbers in an inductive way by means of well-known algebraic properties of such operations.

```

vars I' J' : NzInt .
vars R S : Rat .
vars R' S' : NzRat .
eq R / (R' / S') = (R * S') / R' .

```

```

eq (R / R') / S' = R / (R' * S') .
ceq J' / I' = quot(J', gcd(J', I')) / quot(I', gcd(J', I'))
  if gcd(J', I') > s 0 .
eq R / s 0 = R .
eq 0 / R' = 0 .
eq R / (- R') = (- R) / R' .
eq - (R / R') = (- R) / R' .
eq R + (S / R') = ((R * R') + S) / R' .
eq R * (S / R') = (R * S) / R' .
endfm

```

Figure 8.1 in Section 8.1.5 illustrates in a diagrammatic way the relationship between the modules above. This hierarchy happens to be a linear (or total) order of theory inclusions, with `BOOL` at the bottom.

As mentioned before, in Chapter 9 we will see predefined versions of the three number modules above, with a variety of additional operations.

4.11 Partial operations, subsorting, and errors

In this section we show different techniques to specify a partial function, by means of a simple example.

Let us consider the natural numbers with zero, successor, and an addition operation, as in the `PEANO-NAT` module of the previous section.

```

fmod NAT-PRED-KIND is
  sort Nat .
  op 0 : -> Nat [ctor] .
  op s_ : Nat -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat [comm] .
  vars N M : Nat .
  eq N + 0 = N .
  eq (s N) + (s M) = s s (N + M) .

```

Now we extend this specification with a *predecessor* operation that is *undefined for zero*. This operation can be easily made total by defining that the predecessor of zero is zero itself, but this possibility is not useful for our purposes here, where we want to consider a simple *partial operation*, which is undefined in cases where the operation does not quite make sense. The simplest way to declare such a partial function is to make explicit that the result is in the *kind* `[Nat]` instead of in a sort. Then we have an equation giving the result of predecessor over nonzero natural numbers.

```

  op p_ : Nat -> [Nat] .
  eq p s N = N .
endfm

```

The term `p 0` is syntactically correct (it does parse), but it does not have a sort; it has just a kind. On the other hand, terms like `p s s 0` reduce appropriately and the result has sort `Nat`.

```
Maude> red p 0 .
result [Nat]: p 0
```

```
Maude> red p s s 0 .
result Nat: s 0
```

If we consider bigger terms that contain `p 0` as a subterm, then again they have a kind but not a sort.

```
Maude> red p s s 0 + s p 0 .
result [Nat]: s 0 + s p 0
```

Notice that Maude has reduced the term as much as possible, but in the end we do not get a natural number because the subterm `p 0` is meaningless.

Given that the predecessor operation is only undefined for zero, we can refine the previous specification so that we make explicit in its declaration the fact that the predecessor operation is total on nonzero natural numbers. The way to accomplish this is to introduce a new sort `NzNat` for nonzero natural numbers, which is the result sort for the successor operator, and becomes a subsort of the sort of natural numbers by means of a subsort declaration `NzNat < Nat`. Then the predecessor operator is declared as mapping `NzNat` to `Nat`.

```
fmod NAT-PRED-SUB is
  sorts Nat NzNat .
  subsorts NzNat < Nat .
  op 0 : -> Nat [ctor] .
  op s_ : Nat -> NzNat [ctor] .
  op _+_ : Nat Nat -> Nat [comm] .
  op p_ : NzNat -> Nat .
  vars N M : Nat .
  eq N + 0 = N .
  eq (s N) + (s M) = s s (N + M) .
  eq p s N = N .
endfm
```

With the declaration `p_ : NzNat -> Nat`, one could expect that the term `p 0` does not make sense and should be considered ill-formed. However, more flexibility is necessary; indeed, an approach too strict to parsing in the context of order-sorted specifications would consider a term such as `p (s s 0 + s 0)` ill-formed, because the subterm `s s 0 + s 0` has sort `Nat` according to the declaration of the addition operator while the predecessor operator requires an argument of sort `NzNat`. But after equationally simplifying the subterm `s s 0 + s 0` to `s s s 0`, we see that the requirement for the sort of the argument is then indeed satisfied. Therefore, Maude

gives at parsing time the benefit of the doubt, by automatically lifting all operators to the kind level, parsing terms at the level of kinds, and simplifying them as much as possible.

```
Maude> red p (s s 0 + s 0) .
result NzNat: s s 0

Maude> red p (0 + 0) .
result [Nat]: p 0

Maude> red p s s 0 + s p 0 .
result [Nat]: s 0 + s p 0
```

To sum up, with the subsort approach we have a more refined type system that provides more detailed information about the terms involved in the specification and the defining conditions of the corresponding operations.

A dual approach is to consider that errors go to an explicit *error supersort* rather than to the kind. In this example we can introduce a new sort **ErrorNat** which becomes a supersort of the sort of natural numbers by means of a declaration `Nat < ErrorNat`. Now the predecessor operator is declared again as a total operation, but from `Nat` to `ErrorNat`, so that `p 0 = error`, with `error` a constant in the supersort.

```
fmod NAT-PRED-SUPER is
  sorts Nat ErrorNat .
  subsorts Nat < ErrorNat .
  op 0 : -> Nat [ctor] .
  op s_ : Nat -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat [comm] .
  op p_ : Nat -> ErrorNat .
  op error : -> ErrorNat .
  vars N M : Nat .
  eq N + 0 = N .
  eq (s N) + (s M) = s s (N + M) .
  eq p s N = N .
  eq p 0 = error .
endfm
```

The two previous equations provide a result for any term of the form `p n` where `n` is a natural number; for example,

```
Maude> red p (0 + 0) .
result ErrorNat: error

Maude> red p (s s 0 + s 0) .
result Nat: s s 0
```

However, if a term contains the subterm `p 0`, then `p 0` itself, and perhaps other subterms, can be reduced, but the context above `p 0` cannot be further reduced. A way to *propagate errors* explicitly is to introduce overloaded declarations for all the operators, and additional equations to make explicit the desired error propagation.

```

op s_ : ErrorNat -> ErrorNat .
op _+_ : ErrorNat ErrorNat -> ErrorNat [ditto] .
op p_ : ErrorNat -> ErrorNat .
eq s error = error .
eq N + error = error .
eq p error = error .

```

Then all terms built with the operators in module NAT-PRED-SUPER reduce, either to a natural number, or to the `error` constant.

```

Maude> red p (0 + 0) + p (s s 0 + s 0) .
result ErrorNat: error

```

```

Maude> red p s (0 + 0) + p (s s 0 + s 0) .
result Nat: s s 0

```

This last approach with error propagation can become too complex in a specification with several operators, but it can be very useful when the error is merely an exception and one is interested in providing exception handling or error recovery instead of just error propagation. In the previous example, we could consider a different equation such as $N + \text{error} = N$ that makes the error disappear for the addition:

```

Maude> red p (0 + 0) + p (s s 0 + s 0) .
result Nat: s s 0

```

```

Maude> red p p (0 + 0) .
result ErrorNat: error

```

Error constants can also be declared at the level of kinds instead of sorts; then, the type information is less accurate but this can be enough in many situations.

The user should consider these different possibilities in a given application and use the more appropriate for each case.

At the beginning of Section 14.5, we review the criteria used in a complex module like the META-LEVEL module in the Maude prelude (see file `prelude.maude`) to choose between using a user-defined supersort and having an operator map to a kind to represent partial operations.

In this section we have limited ourselves to illustrating the use of *subsorts* in defining partial operations. More sophisticated partial functions—namely, partial functions whose domain of definition is not characterized by purely *syntactic* means, such as subsort declarations, but does indeed require a *semantic* characterization—can be generally defined by *conditional memberships*. This has already been illustrated by the path concatenation operator `_;` in Section 4.3, and is illustrated with many other examples in Chapter 10.

A Hierarchy of Data Types: From Trees to Sets

In Section 4.4.1 we have introduced *equational attributes* as a means of declaring some equational properties of binary operators that allow Maude to use these properties efficiently in a built-in way in parsing and in matching modulo such equational axioms. We recall that Maude supports the following equational attributes:

- `assoc` (associativity),
- `comm` (commutativity),
- `id`: $\langle Term \rangle$ (identity, with the corresponding term for the identity element), with variations for left identity and right identity, and
- `idem` (idempotency).

An important restriction to bear in mind is that the `assoc` and `idem` attributes cannot be used together in any combination.

In this chapter we will show that equational attributes correspond to *structural axioms* of well-known data types built with a binary constructor operator. In this way we obtain a hierarchy of data types:

- *non-empty binary trees*, with elements only in their leaves, built with a free binary constructor, that is, a constructor with no equational axioms;
- *non-empty lists*, built with an associative constructor;
- *lists*, built with an associative constructor and an identity;
- *multisets* (or bags), built with an associative and commutative constructor and an identity; and
- *sets*, built with an associative, commutative, and idempotent constructor and an identity.

All these data types are *generic*, so that they can be constructed on top of any given data type of basic elements; for example, we can have lists of natural numbers, lists of Booleans, lists of sets of integers, etc. This genericity corresponds to making use of *parameterized modules* in Maude, which will be introduced later in Section 8.3. Therefore, in this chapter we only consider

constructions over natural numbers. In Section 9.12 we will describe the predefined parameterized versions of lists and sets provided in the Maude prelude, and in Chapter 10 we will describe many other parameterized data types, like stacks, queues, sorted lists, multisets, and different versions of trees.

Although the Maude prelude also provides a predefined module of natural numbers (see Section 9.2), in this chapter we will make use of the following basic specification; natural numbers are built from zero and successor in the style described in Section 4.10, but with only two operations for addition and for calculating the maximum of two numbers, that are defined in a simple way by structural induction over the two constructors.

```
fmod BASIC-NAT is
  sort Nat .
  op 0 : -> Nat [ctor] .
  op s_ : Nat -> Nat [ctor] .
  op +_ : Nat Nat -> Nat .
  op max : Nat Nat -> Nat .
  vars N M : Nat .
  eq 0 + N = N .
  eq s M + N = s (M + N) .
  eq max(0, M) = M .
  eq max(N, 0) = N .
  eq max(s N, s M) = s max(N, M) .
endfm
```

This module will be imported in the modules introduced in the next sections by means of a `protecting` declaration, whose full meaning will be explained in Section 8.1.1. For the time being, it is enough to know that the data elements defined in this imported module are used by the importing module without modifying them in any way.

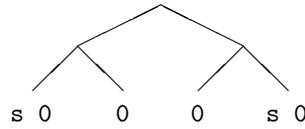
5.1 Nonempty binary trees

Our first module introduces a supersort `Tree` of the sort `Nat` imported from `BASIC-NAT`, so that a natural number corresponds to a tree consisting of a single leaf. Bigger trees, with elements only in their leaves, are constructed from these by means of a binary operator written using empty juxtaposition notation `--`; this operator is a *free constructor* in the sense that it does not satisfy any equational axiom.

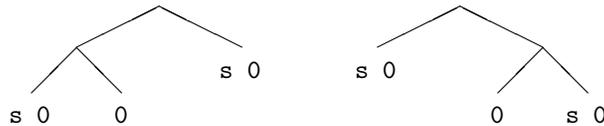
```
fmod BASIC-NAT-TREE is
  protecting BASIC-NAT .

  sorts Tree .
  subsort Nat < Tree .
  op -- : Tree Tree -> Tree [ctor] .
```

For example, the term $((s\ 0)\ 0)\ (0\ (s\ 0))$ corresponds to the tree that we can represent graphically as follows:



Notice that an expression such as $(s\ 0)\ 0\ (s\ 0)$ is ambiguous, because it can be parsed in two different ways, and parentheses are necessary to disambiguate $((s\ 0)\ 0)\ (s\ 0)$ from $(s\ 0)\ (0\ (s\ 0))$. These two different terms correspond to the following two trees:



We can add to this data type a couple of operations to calculate the **width** (which coincides with the number of leaves) and the **depth** of a tree. Each operation is defined by means of two equations, corresponding to the basic trees identified with natural numbers, and to the binary constructor.

```

op depth : Tree -> Nat .
op width : Tree -> Nat .

var N : Nat .
vars T T' : Tree .

eq depth(N) = s 0 .
eq depth(T T') = s(max(depth(T), depth(T'))) .
eq width(N) = s 0 .
eq width(T T') = width(T) + width(T') .
endfm

```

For example, we have the following reductions:

```

Maude> red depth(((s 0) 0) (0 s 0)) .
result Nat: s s s 0

```

```

Maude> red width(((s 0) 0) (0 s 0)) .
result Nat: s s s s 0

```

```

Maude> red depth((s 0) (0 s 0)) .
result Nat: s s s 0

```

```

Maude> red width((s 0) (0 s 0)) .
result Nat: s s s 0

```

5.2 Nonempty lists

A simple way to generate non-empty lists is to begin with singleton lists, and then use the concatenation operation to get lists with two or more elements. However, concatenation cannot be a free constructor, because it is *associative*. This property is not declared directly as an equation, but instead as an equational operator attribute `assoc`.

Singleton lists are identified with natural numbers by means of a subsort declaration `Nat < NeList`.

It is very convenient here to use empty juxtaposition syntax `__` for the concatenation operator; in this way, a list of integers $[x_1, \dots, x_n]$ is written simply as $x_1 \dots x_n$. Notice that now, because of the associativity property, there is no ambiguity, and therefore there is no need for parentheses, as opposed to what happened in the previous section.

```
fmod BASIC-NAT-NE-LIST is
  protecting BASIC-NAT .

  sort NeList .
  subsort Nat < NeList .
  op __ : NeList NeList -> NeList [ctor assoc] .
```

We add one operation to calculate the `length`, that is, the number of elements in a list, and another to `reverse` a list. Again, each operation is defined by means of two equations, corresponding to the singleton lists (identified with natural numbers) and to the binary constructor.

```
op length : NeList -> Nat .
op reverse : NeList -> NeList .

var N : Nat .
vars L L' : NeList .

eq length(N) = s 0 .
eq length(L L') = length(L) + length(L') .
eq reverse(N) = N .
eq reverse(L L') = reverse(L') reverse(L) .
endfm
```

Some reduction examples are the following:

```
Maude> red length(0 (s 0) (s s 0)) .
result Nat: s s s 0
```

```
Maude> red reverse(0 (s 0) (s s 0)) .
result NeList: s s 0 s 0 0
```

5.3 Lists

We consider a variant of the previous example by adding the empty list `nil` as the identity of concatenation, which is again the main constructor for lists, declared this time with both an attribute `assoc` for associativity and an attribute `id: nil` for the empty list as its two-sided identity.

So that the specification becomes more interesting, we will also consider operations `head` and `tail`, which are undefined on the empty list. Therefore, we consider a subsort `NeList` of non-empty lists and then singleton lists are identified with natural numbers by means of a subsort declaration `Nat < NeList < List`. Then, the concatenation operator becomes *subsort overloaded*, having one declaration for non-empty lists and another one for lists. There are two more possibilities of concatenation overloading (`NeList List -> NeList` and `List NeList -> NeList`) but they are unnecessary in this case because of the identity attribute.

Notice that equational attributes in overloaded operators have to coincide, as described in Section 4.4.6, even though, by reading alone the second declaration for concatenation, it may sound a bit strange to say that the empty list is an identity for an operation only defined on non-empty lists. It is also possible to use the `ditto` operator attribute to implicitly repeat the attributes without having to write them all explicitly again.

```
fmod BASIC-NAT-LIST is
  protecting BASIC-NAT .

  sorts NeList List .
  subsorts Nat < NeList < List .
  op nil : -> List [ctor] .
  op _ _ : List List -> List [ctor assoc id: nil] .
  op _ _ : NeList NeList -> NeList [ctor assoc id: nil] .

  op tail : NeList -> List .
  op head : NeList -> Nat .

  var N : Nat .
  var L : List .

  eq tail(N L) = L .
  eq head(N L) = N .
```

We extend the `length` and `reverse` operations to the empty list. The important point to note is that, with identity as an attribute, the identity equations belong to the set A of equational axioms and not to the equations E used for equational simplification, as discussed in Sections 4.4.1 and 4.7. In this way, the congruence classes over which equational simplification takes place are calculated *modulo* associativity and identity. Therefore, we only need two equations to completely specify the behavior of these defined operations:

the singleton case becomes a particular case of the pattern $N L$ by instantiating the variable L with the constant `nil` and applying the identity property.

```

op length : List -> Nat .
op reverse : List -> List .

eq length(nil) = 0 .
eq length(N L) = s length(L) .
eq reverse(nil) = nil .
eq reverse(N L) = reverse(L) N .
endfm

```

Moreover, there is another thing to watch out for in the presence of the identity attribute: the alternative equation `length(L L') = length(L) + length(L')` (with L and L' variables of sort `List`) would cause problems of nontermination. To see this, consider the instantiation with $L' \mapsto \text{nil}$ that gives

```

length(L nil) = length(L) + length(nil)
               = length(L nil) + length(nil)
               = (length(L) + length(nil)) + length(nil)
               = ...

```

because of the identification $L = L \text{ nil}$.

We finish this example with a couple of reductions:

```

Maude> red length(head(0 (s 0) (s s 0))) .
result Nat: s 0

Maude> red reverse(tail(0 (s 0) (s s 0))) .
result NeList: s s 0 s 0

```

A parameterized and much more complete version of lists, specified with the same equational attributes as in this section, will be described in Section 9.12.1. Another specification of lists, based on free constructors, will be explained in Section 10.4.

5.4 Multisets

In the same way as associativity and identity for concatenation provide structural axioms for lists or strings, where the order of the elements matters, we can specify abstractly *multisets* (also known as *bags*) by considering a multiset union constructor (written again with empty juxtaposition syntax) that satisfies associativity, commutativity (because now order between elements does not matter), and identity structural axioms, all of them declared as equational attributes.

```

fmod BASIC-NAT-MSET is
  protecting BASIC-NAT .
  protecting BOOL .

  sort Mset .
  subsorts Nat < Mset .
  op empty-mset : -> Mset [ctor] .
  op _ : Mset Mset -> Mset [ctor assoc comm id: empty-mset] .

```

We consider operations for calculating the `size` of a multiset (that is, the number of elements it has, taking into account the possible repetitions of each one), for calculating the `multiplicity` of an element in a multiset (that is, the number of times it appears), and for checking if an element is `in` a multiset. The result of the last-mentioned operation is a Boolean value, obtained from the `BOOL` module, which is explicitly imported with another `protecting` declaration and also provides the inequality predicate `_=/=_` (see Section 9.1).

```

  op size : Mset -> Nat .
  op mult : Nat Mset -> Nat .
  op _in_ : Nat Mset -> Bool .

  vars N N' : Nat .
  var S : Mset .

  eq size(empty-mset) = 0 .
  eq size(N S) = s size(S) .
  eq mult(N, empty-mset) = 0 .
  eq mult(N, N S) = s mult(N, S) .
  ceq mult(N, N' S) = mult(N, S) if N =/= N' .
  eq N in S = (mult(N, S) =/= 0) .
endfm

```

Notice again the form of the equations for the defined operations; for example, an equation like `size(S S') = size(S) + size(S')` (with `S` and `S'` variables of sort `Mset`) would cause problems of nontermination in the presence of the identity attribute.

The membership operation is defined using the multiplicity function, although it could be defined in a totally independent way. There is an alternative way of defining both of these operations by making use of the `otherwise` attribute explained in Section 4.5.4. To see if a natural number `N` is in a multiset, we can match the multiset against the pattern `N S`; if the matching succeeds, then the result is true, *otherwise*, we know that the element does not occur in the multiset and the result is false.

```

  eq N in N S = true .
  eq N in S = false [owise] .

```

The same technique can be used to count the number of occurrences of a given element (natural number, in this example) in a given multiset: if the multiset matches the pattern $N \mathbf{S}$, then we know that N appears at least once and count recursively the remaining occurrences; *otherwise*, N does not occur and thus its multiplicity is zero.

```
eq mult(N, N S) = s mult(N, S) .
eq mult(N, S) = 0 [owise] .
```

Some reduction examples are the following:

```
Maude> red size(0 (s 0) (s s 0) (s s 0) (s 0) 0) .
result Nat: s s s s s 0
```

```
Maude> red mult(s 0, 0 (s 0) (s s 0) (s s 0) (s 0) 0) .
result Nat: s s 0
```

```
Maude> red s s s 0 in 0 (s 0) (s s 0) (s s 0) (s 0) 0 .
result Bool: false
```

A parameterized and much more complete version of multisets will be described in Section 10.6.

5.5 Sets

Building on the multiset example of the previous section, we can get a specification for sets of natural numbers, where multiplicity of elements does not matter, by adding idempotency as an equation. Since the attributes for associativity and idempotency cannot be combined, we instead use an explicit equation to declare such property.

```
fmod BASIC-NAT-SET is
  protecting BASIC-NAT .
  protecting BOOL .

  sort Set .
  subsorts Nat < Set .
  op empty-set : -> Set [ctor] .
  op __ : Set Set -> Set [ctor assoc comm id: empty-set] .

  vars N N' : Nat .
  vars S S' : Set .

  eq N N = N .
```

The idempotency equation is stated only for singleton sets, because stating it for arbitrary sets in the form $S \mathbf{S} = S$ would cause nontermination due to the identity attribute:

```
empty-set = empty-set empty-set → empty-set ...
```

While a predicate to check membership makes as much sense for sets as for multisets, a multiplicity operation does not. The size operation becomes now the cardinality operation, but for its specification we need to make sure that an element is not counted more than once. In the following equations, this is accomplished with the help of a `delete` operation, that removes a given element from a set. Notice that the equality `_==_` and inequality `_/=/_` predicates come from the `BOOL` predefined module (see Section 9.1), imported by means of a `protecting` declaration.

```
op _in_ : Nat Set -> Bool .
op delete : Nat Set -> Set .
op card : Set -> Nat .

eq N in empty-set = false .
eq N in (N' S) = (N == N') or (N in S) .
eq delete(N, empty-set) = empty-set .
eq delete(N, N S) = delete(N, S) .
ceq delete(N, N' S) = N' delete(N, S) if N /= N' .
eq card(empty-set) = 0 .
eq card(N S) = s card(delete(N, S)) .
endfm
```

Notice that the equations for the above `delete` and `card` operations make sure that further occurrences of `N` in `S` on the righthand side are also deleted or not counted, respectively, because in general we cannot rely on the order in which equations are applied in equational simplification to assume that there are no repeated elements in an expression.

The operations `_in_` and `delete` can also be defined, more succinctly and in a more efficient way, using the `owise` attribute:

```
eq N in N S = true .
eq N in S = false [owise] .
eq delete(N, N S) = delete(N, S) .
eq delete(N, S) = S [owise] .
```

We finish with some reduction examples:

```
Maude> red card(0 (s 0) (s s 0) (s s 0) (s 0) 0) .
result Nat: s s s 0

Maude> red delete(s 0, 0 (s 0) (s s 0) (s s 0) (s 0) 0) .
result Set: 0 s s 0

Maude> red s s s 0 in 0 (s 0) (s s 0) (s s 0) (s 0) 0 .
result Bool: false
```

A parameterized and much more complete version of sets will be described in Section 9.12.2.

5.6 Idempotent semigroups

As a final example in this chapter, we consider one of a different character, but also very related to equational attributes.

Consider the word problem for *idempotent semigroups*: given a binary concatenation operation `--` satisfying the equations for associativity $(xy)z = x(yz)$ and idempotency $xx = x$, when are two words provably equal? For example, do we have $abc = abcbabc$? (Note the absence of parentheses because of associativity.)

The first idea is to build in the associative equation by means of an equational attribute, and use the idempotency equation from left to right as a rewrite rule.

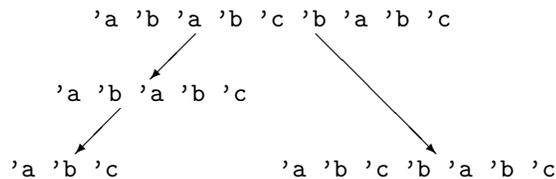
The imported (by means of a `protecting` declaration) module `QID` is a useful predefined module (see Section 9.10) that provides quoted identifiers that in this case represent the generators of the semigroup.

```
fmod NAIVE-IDEM-SEMIGROUP is
  protecting QID .
  sort List .
  subsort Qid < List .
  op -- : List List -> List [ctor assoc] . *** list concatenation
  var L : List .
  eq L L = L .
endfm
```

When we query the Maude system for the equality mentioned above, we obtain

```
Maude> red in NAIVE-IDEM-SEMIGROUP :
      'a 'b 'c == 'a 'b 'c 'b 'a 'b 'c .
result Bool: false
```

However, this result is *wrong*, because these two terms must be identified in the equational theory since both of them can be proved equal to the term `'a 'b 'a 'b 'c 'b 'a 'b 'c` as follows:



The problem is *lack of confluence*. The above specification can be made confluent (while preserving termination, which is obvious, since the length of the word is strictly shorter after each equational simplification) by adding one conditional rule [330], as in the following `IDEM-SEMIGROUP` module. This rule is quite subtle, because its condition involves comparing *sets* of letters in subwords. Note in the following specification that both lists and sets are

non-empty, because there is no use for the empty case, and this at the same time avoids nontermination problems with some equations as we have already mentioned before.

```
fmod IDEM-SEMIGROUP is
  protecting QID .

  sorts List Set .
  subsorts Qid < List Set .
  op _ : List List -> List [ctor assoc] .    *** list concatenation
  op _,_ : Set Set -> Set [ctor assoc comm] . *** set union
  op {_} : List -> Set .                      *** set of a list

  var I : Qid .
  var S : Set .
  vars L P Q : List .

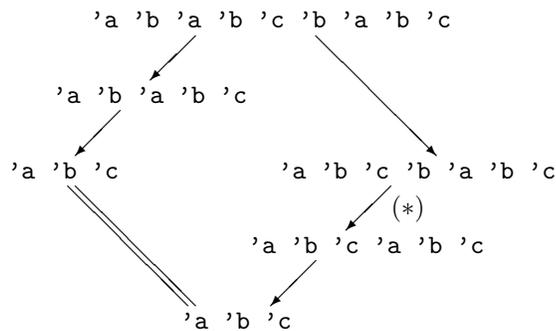
  eq S, S = S .
  eq {I} = I .
  eq {I L} = I, {L} .
  eq L L = L .
  ceq L P Q = L Q if {L} = {Q} /\ {L P} = {L} .
endfm

Maude> red in IDEM-SEMIGROUP : 'a 'b 'c .
result List: 'a 'b 'c

Maude> red 'a 'b 'c 'b 'a 'b 'c .
result List: 'a 'b 'c

Maude> red 'a 'b 'c == 'a 'b 'c 'b 'a 'b 'c .
result Bool: true
```

Now we have obtained the correct result, because the specification is indeed confluent; the example in the diagram above can be completed by using the conditional equation in the step (*).



System Modules

A Maude system module specifies a *rewrite theory*. A rewrite theory has sorts, kinds, and operators (perhaps with frozen arguments), and can have three types of statements: equations, memberships, and rules, all of which can be conditional. Therefore, any rewrite theory has an underlying equational theory, containing the equations and memberships, *plus* the rules. What is the intuitive meaning of such rules? Computationally, they specify *local concurrent transitions* that can take place in a system if the pattern in the rule's lefthand side matches a fragment of the system state and the rule's condition is satisfied. In that case, the transition specified by the rule can take place, and the matched fragment of the state is transformed into the corresponding instance of the righthand side. Logically, that is, when we use rewriting logic as a logical framework to represent other logics as explained in Section 1.4, a rule specifies a *logical inference rule*, and rewriting steps therefore represent inference steps.

As was mentioned in Section 3.2, a system module is declared in Maude using the keywords

```
mod <ModuleName> is <DeclarationsAndStatements> endm
```

As for functional modules the first bit of information in the specification is the module's name, which must be an identifier. For example,

```
mod VENDING-MACHINE is
  ...
endm
```

where the dots stand for all the declarations and statements in the module, which can be:

1. module importations,
2. sort and subsort declarations,
3. operator declarations,
4. variable declarations,

5. equation and membership statements, and
6. rule statements.

Since declarations (1)–(4) and equational statements (5) are exactly as for functional modules, all we have left to explain is how rules (conditional or not) are declared. As for equation and membership statements, rules can be declared with any of the attributes `label`, `metadata`, `nonexec`, and `print` (see Section 4.5). However, the `owise` attribute can only be used with equations.

6.1 Unconditional rules

Mathematically, an unconditional rewrite rule has the form $l : t \rightarrow t'$, where t, t' are terms of the same kind, which may contain variables, and l is the label of the rule. Intuitively, a rule describes a *local concurrent transition* in a system: anywhere in the distributed state where a substitution instance $\sigma(t)$ of the lefthand side t is found, a local transition of that state fragment to the new local state $\sigma(t')$ can take place. And if many instances of the same or of several rules can be matched in different nonoverlapping parts of the distributed state, then all of them can fire concurrently.

An unconditional rule is introduced in Maude with the following syntax:

```
r1 [Label] : Term-1 => Term-2 [StatementAttributes] .
```

As explained in Section 4.5.1, a label can alternatively be declared as a statement attribute; also, Maude allows declaration of *unlabeled rules*. In these two cases, the part “[*Label*] :” is omitted.

As a first example of a system module we consider the following specification of a vending machine which dispenses apples and cakes. The module `VENDING-MACHINE-SIGNATURE` is the underlying functional module. This module is imported by the system module `VENDING-MACHINE`, which then adds the rules for operating the machine. Although not necessary, in addition to making the underlying functional module explicit, such splitting of modules can be useful in organizing a large specification, where a functional part may be shared by several system modules; see Chapter 8 for a discussion on module importation.

The constants `$` and `q` represent coins of one dollar and one quarter, respectively, while the constants `a` and `c` represent apples and cakes, respectively.

```
fmod VENDING-MACHINE-SIGNATURE is
  sorts Coin Item Marking .
  subsorts Coin Item < Marking .
  op _ : Marking Marking -> Marking [assoc comm id: null] .
  op null : -> Marking .
  op $ : -> Coin [format (r! o)] .
  op q : -> Coin [format (r! o)] .
  op a : -> Item [format (b! o)] .
```

```

    op c : -> Item [format (b! o)] .
endfm

```

The `format` declaration for each constant (see Section 4.4.5) is used to print the constants using different colors, so that coins can easily be separated from items in a given marking.

```

mod VENDING-MACHINE is
  including VENDING-MACHINE-SIGNATURE .
  var M : Marking .
  rl [add-q] : M => M q .
  rl [add-$] : M => M $ .
  rl [buy-c] : $ => c .
  rl [buy-a] : $ => a q .
  rl [change] : q q q q => $ .
endm

```

This module specifies a concurrent machine to buy cakes and apples with dollars and quarters. A cake costs a dollar, and an apple three quarters. We can insert dollars and quarters in the machine, although due to an unfortunate design, the machine only accepts buying cakes and apples with dollars. When the user buys an apple the machine takes a dollar and returns a quarter. To alleviate in part this problem, the machine can change four quarters into a dollar.

The machine is *concurrent*, because we can *push several buttons at once* (that is, we can apply several rules at once), provided enough resources exist in the corresponding slots, called *places*. For example, if we have one dollar in the `$` place and four quarters in the `q` place, we can *simultaneously* push the `buy-a` and `change` buttons, and the machine returns, also simultaneously, one dollar in `$`, one apple in `a`, and one quarter in `q`.

Note that, since the Maude interpreter is sequential, the above concurrent transitions in the `VENDING-MACHINE` module are simulated by corresponding *interleavings* of sequential rewriting steps. In a socket-based concurrent implementation, it is possible to execute concurrently many rewriting steps for a wide range of system modules.¹

We might have tried a simpler alternative, namely, using the rule `null => q` instead of the `add-q` rule. However, this would not work. Instead, we have to write `M => M q` with `M` a variable of sort `Marking`. The reason is that the constant `null` is not a `--`-subterm of any marking except itself, and therefore it would be impossible to apply the rule `null => q` with extension (see Section 4.8).

¹ See Chapter 17 for an interesting example of this kind: a concurrent implementation of a mobile language entirely programmed in Maude using sockets as external objects in the way explained in Section 11.4.1.

6.2 Conditional rules

Conditional rewrite rules can have very general conditions involving equations, memberships, and other rewrites; that is, in their mathematical notation they can be of the form

$$l : t \rightarrow t' \text{ if } \left(\bigwedge_i u_i = v_i \right) \wedge \left(\bigwedge_j w_j : s_j \right) \wedge \left(\bigwedge_k p_k \rightarrow q_k \right)$$

with no restriction on which new variables may appear in the righthand side or the condition. There is no need for the condition listing first equations, then memberships, and then rewrites: this is just a notational abbreviation, since they can be listed in any order. However, in Maude, conditions are evaluated from left to right, and therefore the order in which they appear, although mathematically inessential, is very important operationally (see Section 6.3).

In their Maude representation, conditional rules are declared with syntax

```

crl [Label] : <Term-1> => <Term-2>
  if <Condition-1> /\ ... /\ <Condition-k>
  [StatementAttributes] .

```

where the rule's label can instead be declared as a statement attribute, or can be omitted altogether. In either of these two alternatives, the square brackets enclosing the label and the colon are then omitted.

As in conditional equations, the condition can consist of a single statement or can be a conjunction formed with the associative connective \wedge . But now conditions are more general, since in addition to equations and memberships they can also contain rewrite expressions, for which the concrete syntax $t \Rightarrow t'$ is used. Furthermore, equations, memberships, and rewrites can be intermixed in any order. As for functional modules, some of the equations in conditions can be either matching equations or abbreviated Boolean equations.

We can illustrate the usefulness of rewrite expressions in rule conditions by presenting a small fragment of a Maude operational semantics for Milner's CCS language given in [369]:

```

sorts Label Act Process ActProcess .
subsorts Qid < Label < Act .
subsort Process < ActProcess .

op ~_ : Label -> Label .
op tau : -> Act .
op {_}_ : Act ActProcess -> ActProcess [frozen] .
op |_ : Process Process -> Process [frozen assoc comm] .

vars P P' Q Q' : Process .
var L : Label .

crl [par] : P | Q => {tau} (P' | Q')

```

$\text{if } P \Rightarrow \{L\} P' \wedge Q \Rightarrow \{\sim L\} Q' .$

The conditional rule **par** expresses the synchronized transition of two processes composed in parallel. The condition of the rule states that the synchronized transition can take place if one process can perform an action named L and the other can perform the complementary action named $\sim L$. In this representation of CCS, the action performed is remembered by the resulting expression, which is a term of sort **ActProcess**.

Note the use of the **frozen** attribute in some of the operators (see Section 4.4.9).

6.3 Admissible system modules

The same way that equations or memberships expressed in their fullest possible generality cannot be executed by the Maude engine except in a controlled way at the metalevel, conditional rewrite rules in their fullest generality cannot be executed either, except with a strategy at the metalevel. Nonexecutable rules should be identified by giving them the **nonexec** attribute.

As for functional modules, the question now becomes: what are the executability requirements on the executable statements (i.e., those without the **nonexec** attribute) of a system module? It turns out that a quite general class of system modules, called *admissible modules*, are executable by Maude's default interpreter using the **rewrite**, **frewrite**, and **search** commands, that will be introduced and illustrated in Section 6.4 and are further explained in Sections 25.2 and 25.4.

The admissibility requirements for the module's equations and memberships are exactly as for functional modules; they were explained in Section 4.6 and are further discussed below. Two more requirements are needed:

- each executable conditional rule should be *admissible*, and
- the rules should be *coherent* relative to the equations, as has already been mentioned in the introduction.

We explain each of these requirements below.

Given a system module M , a conditional² rule of the form

$$l : t \rightarrow t' \text{ if } C_1 \wedge \dots \wedge C_n$$

such that it does not have the **nonexec** attribute is called *admissible* if it satisfies the exact analogues of the admissibility requirements 1–3 in Section 4.6 for conditional equations, plus the additional requirement

² For the purposes of this discussion, we view unconditional rules as a special case of conditional rules. The general admissibility requirement specializes then to a very easy requirement for an unconditional rule $t \rightarrow t'$, namely, that each variable of t' must appear in t .

4. If C_i is a rewrite $u_i \rightarrow u'_i$, then

$$\text{vars}(u_i) \subseteq \text{vars}(t) \cup \bigcup_{j=1}^{i-1} \text{vars}(C_j),$$

and furthermore u'_i is an $\mathcal{E}(M)$ -pattern (for the notion of pattern see Section 4.3) for $\mathcal{E}(M)$ the equational theory underlying the module M .

Operationally, we try to satisfy such a rewrite condition by reducing the substitution instance $\sigma(u_i)$ to its canonical form v_i with respect to the equations, and then trying to find a rewrite proof $v_i \rightarrow w_i$ (perhaps after many steps of rewriting) with w_i in canonical form with respect to the equations and such that w_i is a substitution instance of u'_i . Search for such a w_i is performed by the Maude engine using a breadth-first strategy.

As for functional modules, when executing an admissible conditional rule in a system module, the satisfaction of all its conditions is attempted sequentially from left to right; but notice that now, besides the fact that many matches for the equational conditions may be possible due to the presence of equational attributes, we also have to deal with the fact that solving rewrite conditions requires *search*, including searching for new solutions when previous ones fail to satisfy subsequent conditions.

We now explain the *coherence* requirement. A rewrite theory has both rules and equations, so that rewriting is performed *modulo* such equations. However, this does not mean that the Maude implementation must have a matching algorithm for each equational theory that a user might specify, which is impossible, since matching modulo an arbitrary equational theory is undecidable.

The equations and memberships specified in a system module M are divided into a set A of axioms corresponding to equational attributes such as associativity, commutativity, idempotency, and (left-, right- or two-sided) identity declared for different operators in the module (see Section 4.4.1), for which matching algorithms exist in the Maude implementation, and a set E of equations and memberships specified in the ordinary way. As already mentioned, for M to be executable, the set of executable statements in E must be Church-Rosser and terminating *modulo* A , or at least ground Church-Rosser and terminating modulo A ; that is, we require that the equational part must be equivalent to an executable functional module.

Moreover, we require that the rules R in the module are *coherent* [377] with respect to the equations E modulo A , or at least *ground coherent*. Coherence means that, given a term t , for each one-step rewrite of it with some rule in R modulo the axioms A to some term t' , which we denote $t \rightarrow_{R/A}^1 t'$, if u is the canonical term we obtain by rewriting t with the equations and memberships in E to canonical form modulo A , denoted $t \rightarrow_{E/A}^1 u$, then there is a one-step rewrite of u with some rule in R modulo A , $u \rightarrow_{R/A}^1 u'$, such that $t' =_{E \cup A} u'$, which by the Church-Rosser and termination properties of E

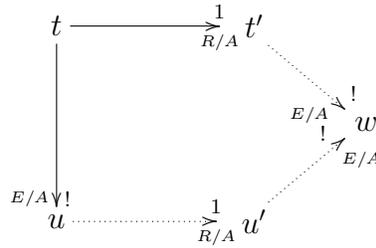


Fig. 6.1. Coherence diagram

modulo A is equivalent to t' and u' having the same canonical form modulo A by E . This requirement is described graphically in Figure 6.1.

Ground coherence is a weaker requirement: we require the exact same diagram to exist only for *ground* terms, and E only needs to be ground Church-Rosser and terminating modulo A .

As explained in [377] (for the free case and for coherence modulo associativity and commutativity), for unconditional rules R , coherence can be checked by checking “critical pairs” between rules R and equations E , and showing that the corresponding instance of the coherence diagram can be filled in for all such pairs. That is, we have to look for appropriate overlaps between lefthand sides of rules and equations using an A -unification algorithm, generate the corresponding critical pairs, and check their coherence. In the case of ground coherence, it is not necessary that the critical pairs can be filled in: it is enough to show that each *ground instance* of such pairs can be filled in. See Section 7.8 for an example of a system module that is not coherent, a discussion of the critical pairs involved, and a method to make the specification coherent. See also Section 13.4 for an example of how coherence can be checked by critical pair analysis. Similarly, for ground coherence and how to check it, see the example in Section 12.4. Section 23.1.4 describes a coherence checker tool that checks coherence of system modules.

Why is coherence so important? What does it mean intuitively? Rewriting modulo an equational theory $E \cup A$, which is what a rewrite theory $\mathcal{R} = (\Sigma, E \cup A, \phi, R)$ really does, is in general undecidable. The undecidability has to do with the fact that we may need to search an entire $E \cup A$ -equivalence class before we can know if a class representative can be rewritten with R , that is, if the $E \cup A$ -equivalence class can be rewritten. Coherence makes the problem decidable: all we need to do is to reduce the term to its canonical form by E modulo A , and then rewrite with R such a canonical form. In a sense, coherence reduces rewriting with R modulo $E \cup A$ to rewriting with E and R modulo A , which is decidable, because we assume we have an A -matching algorithm.

Could we miss any rewrites that way? Coherence guarantees to us that we could not, since any rewrite of a term t with R must also be possible

with t 's canonical form. Maude implicitly assumes this coherence property. For example, the `rewrite` command will at each step first reduce the term to canonical form with E modulo A , and then perform a rewrite step with R in a rule-fair manner. The `frewrite` command uses a somewhat different rewrite strategy to ensure both local fairness and rule fairness, but assumes the same coherence (or ground coherence) property (see Section 25.2 and examples in Section 6.4 below).

A last point about the execution of system modules regards *frozen* argument positions in operators (see Section 4.4.9). This poses a general constraint on any rewriting strategy whatsoever, including those directly supported by Maude for the `rewrite` and `frewrite` commands (see Section 6.4). The general constraint is that *rewriting will never happen below one of the frozen argument positions* in an operator. That is, even though many rewritings may be possible and there can be a large amount of nondeterminism (so that different rewriting strategies may lead to quite different results) rewriting under frozen arguments is *always forbidden*. In fact, this does not only belong to the module's operational semantics, but also to the latest initial model semantics for rewrite theories developed in [36]; we give a brief informal summary of this semantics below.

Mathematically, a system module, when “flattened” with its imported submodules, exactly specifies a (generalized) *rewrite theory* in the sense of [36], that is, a four-tuple

$$\mathcal{R} = (\Sigma, E \cup A, \phi, R),$$

where $(\Sigma, E \cup A)$ is the membership equational theory specified by the signature, equational attributes, and equation and membership statements in the module (just as in the case of functional modules); ϕ is a function, assigning to each operator in Σ the set of natural numbers corresponding to its frozen arguments (the empty set when no argument is frozen); and R is the collection of (possibly conditional) rewrite rules specified in the module and its submodules.

Intuitively, such a rewrite theory specifies a *concurrent system*. The equational theory $(\Sigma, E \cup A)$ specifies the “statics” of the system, that is, the algebraic structure of the set³ of states, which is specified by the initial algebra $T_{\Sigma/E \cup A}$. The rules R and the freezing information ϕ specify the concurrent system's “dynamics,” that is, the possible concurrent transitions that the system can perform. In rewriting logic, such, possibly complex, concurrent transitions exactly correspond to *rewrite proofs*; but since several rewrite proofs can indeed correspond to the *same* concurrent computation (describing, for example, different semantically equivalent interleavings), rewriting logic has an equational theory of *proof equivalence* [243, 36].

The *initial model* $\mathcal{T}_{\mathcal{R}}$ of the rewrite theory \mathcal{R} associates to each kind k a labeled transition system (in fact, a *category*) whose set of states is $T_{\Sigma/E \cup A, k}$,

³ More precisely, each kind k in Σ corresponds to a different choice for a set of states, namely the set $T_{\Sigma/E \cup A, k}$.

and whose labeled transitions have the form $[\alpha] : [t] \rightarrow [t']$, with $[t], [t'] \in T_{\Sigma/E \cup A, k}$, and with $[\alpha]$ an equivalence class of rewrite proofs modulo the equational theory of proof equivalence. Indeed what the different $[\alpha]$ represent are the different “truly concurrent” computations of the system specified by \mathcal{R} .

6.4 The `rewrite`, `frewrite`, and `search` commands

Now we illustrate the use of the Maude commands available for system modules. Recall the vending machine example:

```
mod VENDING-MACHINE is
  including VENDING-MACHINE-SIGNATURE .
  var M : Marking .
  rl [add-q] : M => M q .
  rl [add-$] : M => M $ .
  rl [buy-c] : $ => c .
  rl [buy-a] : $ => a q .
  rl [change]: q q q q => $ .
endm
```

In addition to the `show` commands discussed in Section 4.9, there is an additional `show rls` command for system modules to show the rules of a module. For example, showing the sorts and the rules of the `VENDING-MACHINE` module we get:

```
Maude> show sorts VENDING-MACHINE .
sort Bool .
sort Coin .      subsort Coin < Marking .
sort Item .      subsort Item < Marking .
sort Marking .  subsorts Item Coin < Marking .

Maude> show rls VENDING-MACHINE .
rl M => q M [label add-q] .
rl M => $ M [label add-$] .
rl $ => c [label buy-c] .
rl $ => q a [label buy-a] .
rl q q q q => $ [label change] .
```

6.4.1 The `rewrite` command

We can use the `rewrite` command (abbreviated `rew`) to explore the behavior of different initial markings. The bracketed number between the command and the term to be rewritten provides an upper bound for the number of rule applications that are allowed.


```

mod VENDING-MACHINE-TOP is
  including VENDING-MACHINE-SIGNATURE .
  var M : Marking .
  rl [add-q] : M => M q .
  rl [add-$] : M => M $ .
  rl [buy-c] : $ M => c M .
  rl [buy-a] : $ M => a q M .
  rl [change]: q q q q => $ .
endm

```

Now starting with two dollars and two quarters, and executing increasing numbers of rewrites we see that Maude applies the rules `add-$`, `add-q`, `buy-c`, `buy-a`, and `change`.

```

Maude> rew [2] in VENDING-MACHINE-TOP : $ $ q q .
Advisory: "v.maude", line 18 (mod VENDING-MACHINE-TOP): collapse at
top of $ M may cause it to match more than you expect.
Advisory: "v.maude", line 19 (mod VENDING-MACHINE-TOP): collapse at
top of $ M may cause it to match more than you expect.
rewrite [2] in VENDING-MACHINE-TOP : $ $ q q .
rewrites: 2 in 0ms cpu (0ms real) (~ rews/sec)
result Marking: $ $ $ q q q

Maude> rew [3] $ $ q q .
rewrite [3] in VENDING-MACHINE-TOP : $ $ q q .
rewrites: 3 in 0ms cpu (0ms real) (~ rews/sec)
result Marking: $ $ q q q c

Maude> rew [4] $ $ q q .
rewrite [4] in VENDING-MACHINE-TOP : $ $ q q .
rewrites: 4 in 0ms cpu (0ms real) (~ rews/sec)
result Marking: $ q q q q a c

Maude> rew [5] $ $ q q .
rewrite [5] in VENDING-MACHINE-TOP : $ $ q q .
rewrites: 5 in 0ms cpu (0ms real) (~ rews/sec)
result Marking: $ $ a c

```

The advisory is about the modified rules for buying. Maude is letting us know that the pattern `$ M` will match a term not containing the top-level operator `__`, when `M` is instantiated to `null`. This is exactly what we want in this case, but it may not always be what the user intended, so Maude gives you a heads up; see Section 24.2.6 for more details.

Notice that rewriting in `VENDING-MACHINE` is not terminating. If we remove the rules for adding coins we obtain a terminating system and can explore vending behavior using unbounded rewriting. Let us consider the following module `SIMPLE-VENDING-MACHINE`.

```

mod SIMPLE-VENDING-MACHINE is

```

```

including VENDING-MACHINE-SIGNATURE .
r1 [buy-c] : $ => c .
r1 [buy-a] : $ => a q .
r1 [change]: q q q q => $ .
endm

```

For example, starting with two dollars and rewriting as much as possible we can get an apple, a cake and a quarter in change.

```

Maude> rew in SIMPLE-VENDING-MACHINE : $ $ .
rewrite in SIMPLE-VENDING-MACHINE : $ $ .
rewrites: 2 in 0ms cpu (0ms real) (~ rews/sec)
result Marking: q a c

```

Starting with two dollars and three quarters and using only three rewrite rule applications we get an apple and a cake with a dollar left over.

```

Maude> rew [3] $ $ q q q .
rewrite [3] in SIMPLE-VENDING-MACHINE : $ $ q q q .
rewrites: 3 in 0ms cpu (0ms real) (~ rews/sec)
result Marking: $ a c

```

The command `continue` *<Bound>* (abbreviated `cont`) tells Maude to continue rewriting using at most *<Bound>* additional rule applications. For example, we can continue the last rewrite command (that performed three rewrites) for one more step to get an apple and two cakes:

```

Maude> cont 1 .
rewrites: 1 in 0ms cpu (0ms real) (~ rews/sec)
result Marking: a c c

```

6.4.2 The `frewrite` command

Let us see now what happens when we use another strategy for rewriting in the original `VENDING-MACHINE` module. The `frewrite` command (abbreviated `frew`) rewrites a term using a depth-first position-fair strategy that makes it possible for some rules to be applied that could be “starved” using the left-most, outermost rule fair strategy of the `rewrite` command. The strategies for the `rewrite` and `frewrite` commands are described in detail in Section 25.2.

```

Maude> frew [2] in VENDING-MACHINE : $ $ q q .
frewrite [2] in VENDING-MACHINE : $ $ q q .
rewrites: 2 in 0ms cpu (0ms real) (~ rews/sec)
result (sort not calculated): ($ q) ($ $) q q

Maude> frew [12] $ $ q q .
frewrite [12] in VENDING-MACHINE : $ $ q q .
rewrites: 12 in 0ms cpu (0ms real) (~ rews/sec)
result (sort not calculated):
  c (q a) ($ q) ($ $) (q q) ($ q) (q q) q q

```

With two rewrites, one quarter and one dollar are added. With sufficiently many rewrites (twelve will do), a cake and an apple can be obtained.

In contrast to `rewrite`, that reduces the whole term using equations after each rule rewrite, `frewrite` only reduces the subterm rewritten (to preserve positions not yet visited). Thus, when rewriting stops, the term may not be fully reduced and hence Maude will not know the exact least sort of the term yet. This is the reason for the `(sort not calculated)` comment in place of a sort in the `result` line. In the case of a term with an associative and commutative top operator, the term may not be in its fully flattened form with canonical order of subterms. This accounts for the parentheses in the result term and the fact that the coins and items are not listed in order as they are in the result of a `rewrite`.

The top-down rule-fair strategy of the `rewrite` command can result in nontermination even though there is a terminating sequence of rewrites. As an example consider the following module:

```
mod BB-TEST is
  sort Expression .
  ops a b bingo : -> Expression .
  op f : Expression Expression -> Expression .

  rl a => b .
  rl b => a .
  rl f(b, b) => bingo .
endm
```

Giving the `rewrite` command with input term `f(a, a)` will result in the following looping computation:

```
f(a, a) => f(b, a) => f(a, a) => f(b, a) => f(a, a) => ...
```

This is because using the top-down rule-fair strategy of the `rewrite` command, the third rule always fails to match and never gets a chance to be applied. As already mentioned above, the `frewrite` command uses on the other hand a position-fair bottom-up strategy that makes it possible for other rules to be applied. As a consequence, some rewriting computations that could be non-terminating using the `rewrite` command become terminating with `frewrite`. For example, using the `frewrite` command in place of `rewrite` in the above example we get

```
Maude> frew in BB-TEST : f(a, a) .
frewrite in BB-TEST : f(a, a) .
rewrites: 3 in 0ms cpu (0ms real) (~ rews/sec)
result Expression: bingo
```

6.4.3 The search command

The `rewrite` and `frewrite` commands each explore just one possible behavior (sequence of rewrites) of a system described by a set of rewrite rules and an

initial state. The `search` command allows one to explore (following a breadth-first strategy) the reachable state space in different ways. Its syntax conforms to the following general scheme

```
search [ n, m ] in <ModId> : <Term-1> <SearchArrow> <Term-2>
  such that <Condition> .
```

where

- n is an optional argument providing a bound on the number of desired solutions;
- m is another optional argument stating the maximum depth of the search;
- the module $\langle ModId \rangle$ where the search takes place can be omitted;
- $\langle Term-1 \rangle$ is the starting term;
- $\langle Term-2 \rangle$ is the pattern that has to be reached;
- $\langle SearchArrow \rangle$ is an arrow indicating the form of the rewriting proof from $\langle Term-1 \rangle$ until $\langle Term-2 \rangle$:
 - $\Rightarrow 1$ means a rewriting proof consisting of exactly one step,
 - $\Rightarrow +$ means a rewriting proof consisting of one or more steps,
 - $\Rightarrow *$ means a proof consisting of none, one, or more steps, and
 - $\Rightarrow !$ indicates that only canonical final states are allowed, that is, states that cannot be further rewritten; and

The one step arrow $\Rightarrow 1$ is an abbreviation of the one-or-more steps arrow $\Rightarrow +$ with the depth bound m set to 1.

- $\langle Condition \rangle$ states an optional property that has to be satisfied by the reached state; the syntactic form of the condition is the same as the one of conditions for conditional rules (see Section 6.2).

For example, for our *finite* vending machine, `SIMPLE-VENDING-MACHINE`, we can use the `search` command to answer the question: if I have a dollar and three quarters, can I get a cake and an apple? This is done by searching for states that match a corresponding pattern. In this example, we use the $\Rightarrow !$ symbol, meaning that we are searching for terminal states, that is, for states that cannot be further rewritten. Moreover, no bound in the number of solutions or in the depth of the search is needed.

```
Maude> search in SIMPLE-VENDING-MACHINE :
      $ q q q =>! a c M:Marking .
search in SIMPLE-VENDING-MACHINE : $ q q q =>! a c M:Marking .
```

```
Solution 1 (state 4)
states: 6 rewrites: 5 in 0ms cpu (0ms real) (~ rews/sec)
M:Marking --> null
```

```
No more solutions.
states: 6 rewrites: 5 in 0ms cpu (1ms real) (~ rews/sec)
```

The answer is yes, and the desired state is numbered 4. To see the sequence of rewrites that allowed us to reach this state we can type

```
Maude> show path 4 .
state 0, Marking: $ q q q
===[ rl $ => q a [label buy-a] . ]===>
state 2, Marking: q q q q a
===[ rl q q q q => $ [label change] . ]===>
state 3, Marking: $ a
===[ rl $ => c [label buy-c] . ]===>
state 4, Marking: a c
```

One can get just the sequence of labels of applied rules with a similar command:

```
Maude> show path labels 4 .
buy-a
change
buy-c
```

It is also possible to print out the current search graph generated by a search command using the command `show search graph`. After the above search we get

```
Maude> show search graph .
state 0, Marking: $ q q q
arc 0 ==> state 1 (rl $ => c [label buy-c] .)
arc 1 ==> state 2 (rl $ => q a [label buy-a] .)

state 1, Marking: q q q c

state 2, Marking: q q q q a
arc 0 ==> state 3 (rl q q q q => $ [label change] .)

state 3, Marking: $ a
arc 0 ==> state 4 (rl $ => c [label buy-c] .)
arc 1 ==> state 5 (rl $ => q a [label buy-a] .)

state 4, Marking: a c

state 5, Marking: q a a
```

This search graph is represented graphically in Figure 6.2.

From the same initial state, `$ q q q`, we can see if it is possible to reach a final state with an apple and more things, learning that there are exactly two possibilities:

```
Maude> search $ q q q =>! a M:Marking such that M:Marking /= null .
search in SIMPLE-VENDING-MACHINE : $ q q q =>! a M:Marking
  such that M:Marking /= null = true .
```

```
Solution 1 (state 4)
states: 6 rewrites: 6 in 0ms cpu (0ms real) (~ rews/sec)
```

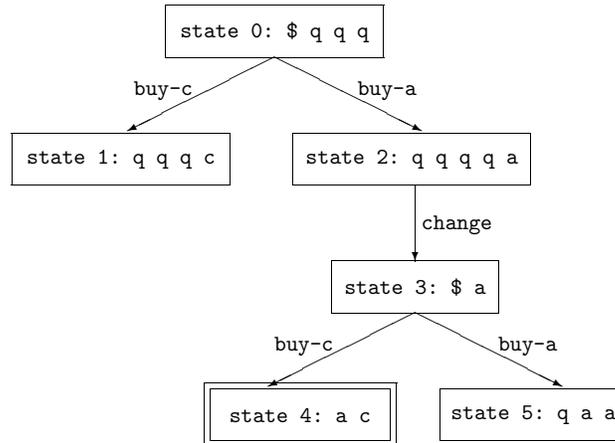


Fig. 6.2. Graphical representation of search graph in example

M:Marking --> c

Solution 2 (state 5)

states: 6 rewrites: 7 in 0ms cpu (0ms real) (~ rews/sec)

M:Marking --> q a

No more solutions.

states: 6 rewrites: 7 in 0ms cpu (0ms real) (~ rews/sec)

In the following example with a different initial state, namely, \$ q q q q, we are looking for intermediate states from which it is possible to get later either two apples (and two quarters left) or two cakes, getting exactly one solution.

```

Maude> search $ q q q q =>+ M:Marking
      such that M:Marking => a a q q /\ M:Marking => c c .
search in SIMPLE-VENDING-MACHINE : $ q q q q =>+ M:Marking
      such that M:Marking => q q a a /\ M:Marking => c c .
  
```

Solution 1 (state 1)

states: 2 rewrites: 10 in 0ms cpu (0ms real) (96153 rewrites/second)

M:Marking --> \$ \$

No more solutions.

states: 9 rewrites: 38 in 0ms cpu (0ms real) (95477 rewrites/second)

Sometimes it is necessary to impose a limit on the number of solutions searched for, since in general the number of such solutions could be infinite. In the previous examples there were only one or two solutions, so imposing a bound would not make any difference. But, returning to the coin generating

(and thus nonterminating) vending machine module `VENDING-MACHINE`, the search space becomes now infinite, so it is important to be able to limit either the number of solutions sought or the depth of the search, or both.

We can look for different ways to use a dollar and three quarters to buy an apple and two cakes. First we ask for one solution, and then use the bounded `continue` command to see another solution. Note that here we use the search mode `=>+`, which means searching for states reachable by at least one rewrite. Searching for terminal states in the `VENDING-MACHINE` module is futile!

```
Maude> search [1] in VENDING-MACHINE : $ q q q =>+ a c c M:Marking .
search in VENDING-MACHINE : $ q q q =>+ a c c M .
```

```
Solution 1 (state 108)
states: 109 rewrites: 1857 in 0ms cpu (41ms real)(~rews/sec)
M --> q q q q
```

```
Maude> cont 1 .
Solution 2 (state 113)
states: 114 rewrites: 185 in 0ms cpu (4ms real) (~ rews/sec)
M --> null
```

We can also use the maximum depth optional argument, but if we put a too small depth, we do not get any solution:

```
Maude> search [, 4] $ q q q =>+ a c c M:Marking .
search [, 4] in VENDING-MACHINE : $ q q q =>+ a c c M .
```

```
No solution.
states: 66 rewrites: 875 in 10ms cpu (3ms real) (87500 rews/sec)
```

By increasing the depth to 10 we will get 98 solutions. If we are interested in only a few of those, we can set both bounds, like in the following example:

```
Maude> search [4, 10] $ q q q =>+ a c c M:Marking .
search [4, 10] in VENDING-MACHINE : $ q q q =>+ a c c M .
```

```
Solution 1 (state 108)
states: 109 rewrites: 1857 in 0ms cpu (7ms real) (~ rews/sec)
M --> q q q q
```

```
Solution 2 (state 113)
states: 114 rewrites: 2042 in 0ms cpu (7ms real) (~ rews/sec)
M --> null
```

```
Solution 3 (state 160)
states: 161 rewrites: 3328 in 0ms cpu (11ms real) (~ rews/sec)
M --> q q q q q
```

```
Solution 4 (state 164)
```

```
states: 165 rewrites: 3524 in 0ms cpu (12ms real) (~ rews/sec)
M --> q
```

If we insist now in the marking `M` being different from `null`, then one of the previous solutions is discarded, but we still get four solutions:

```
Maude> search [4, 10] $ q q q =>+ a c c M:Marking
      such that M:Marking /= null .
search [4, 10] in VENDING-MACHINE : $ q q q =>+ a c c M
      such that M /= null = true .
```

Solution 1 (state 108)

```
states: 109 rewrites: 1858 in 0ms cpu (5ms real) (~ rews/sec)
M --> q q q q
```

Solution 2 (state 160)

```
states: 161 rewrites: 3331 in 10ms cpu (13ms real) (333100 rews/sec)
M --> q q q q q
```

Solution 3 (state 164)

```
states: 165 rewrites: 3528 in 10ms cpu (14ms real) (352800 rews/sec)
M --> q
```

Solution 4 (state 175)

```
states: 176 rewrites: 3904 in 10ms cpu (15ms real) (390400 rews/sec)
M --> $ q q q q
```

In Chapter 12 we will see how the `search` command can be used to model check invariant properties of a concurrent system specified in Maude as a system module.

In case you forget to set a bound on the `search` command or on its continuation, you can also interrupt a search in progress by typing control-C. In this case one of two things will happen, depending on what Maude is doing at the instant you hit control-C. If Maude is not doing a rewrite, the command will exit. If Maude is doing a rewrite, you will end up in the debugger. In this latter case it is probably best to type `abort`, since the debugger has no special support for search at the moment. See Sections 24.1.3 and 25.12 for more information on the debugger.

The full syntax and different options for the `search` command and for all the other commands illustrated in this section are explained in detail in Chapter 25.

6.5 More examples of system modules

In this section we show by means of examples how Petri nets and conjunctive planning problems can be represented as system modules in Maude, thus providing more examples of this kind of modules as well as of the use of some basic commands over them.

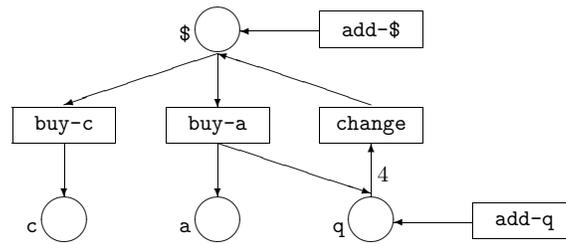


Fig. 6.3. Petri net of the vending machine

6.5.1 Petri nets

Petri nets constitute a model of concurrent computation, used both at the theoretical and practical levels [314]. For example, the `VENDING-MACHINE` module in Section 6.1, which specifies a concurrent machine to buy cakes and apples with dollars and quarters, can be represented graphically as the Petri net concurrent automaton depicted in Figure 6.3.

Here we consider a bigger example to illustrate a general technique that can be used to represent place/transition Petri nets as system modules in Maude.

Let us consider a Petri net modeling a small library, where a token represents a book, that can be in several different states: just bought (j), available (a), borrowed (b), requested (r), and not available (n). The possible transitions are the following:

- *buy*: when there are four accumulated requests, the library places an order to buy two copies of each requested book (although this representation does not distinguish among different books or copies of the same book).
- *file*: a book just bought is filed, making it available.
- *borr*: an available book can be borrowed.
- *ret*: a borrowed book can be returned.
- *lose*: a borrowed book can become lost, and thus no longer available.
- *disc*: an available book is discarded because of its bad condition, and thus it is no longer available either.
- *req1*: a user may place a request to buy a non available book, but only when there are two accumulated requests these are honored.
- *req2*: the library may decide to place a request to buy a new book, thus creating a new token in the system, with no precondition in this representation.

The corresponding graphical representation for this Petri net in the usual style is depicted in Figure 6.4.

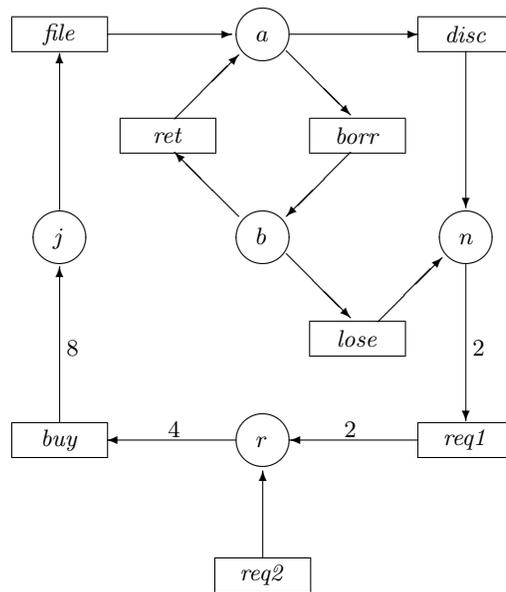


Fig. 6.4. A Petri net model of a library

A *marking*⁴ on a Petri net is a multiset over its set of places, denoting the available resources (or tokens) in each place. A transition goes from the marking representing the resources it consumes (its preset) to the marking representing the resources it produces (its postset). Therefore, in the following system module, first we define a multiset structure for markings by means of the corresponding structural axioms (note that multiset union has empty syntax), using the techniques introduced in Section 5.4). Then, each transition gives rise to a rewrite rule from its preset to its postset. The only exception to this general method is the rule corresponding to the transition *req2*; instead of a rule of the form $1 \Rightarrow r$, we have to write $M \Rightarrow M r$ with M a variable of sort `Marking`. The reason is that the constant `1` is not a `--`-subterm of any marking except itself, according to the definition in Section 4.8, and thus it would be impossible to apply the rule $1 \Rightarrow r$ with extension (see Section 4.8).

```

mod LIBRARY-PETRI-NET is
  sorts Place Marking .
  subsort Place < Marking .
  op 1 : -> Marking [ctor] .
  op -- : Marking Marking -> Marking [ctor assoc comm id: 1] .
  ops a b n r j : -> Place [ctor] .

```

⁴ In the traditional graphical representation, markings are depicted by dots, indicating tokens in the corresponding circular slots for their places, thus the name “marking.”


```

--->
r r
***** rule
rl M => r M [label req2] .
M --> r r
r r
--->
r r r
***** rule
rl M => r M [label req2] .
M --> r r r
r r r
--->
r r r r
***** rule
rl r r r r => j j j j j j j [label buy] .
empty substitution
r r r r
--->
j j j j j j j
***** rule
rl M => r M [label req2] .
M --> j j j j j j j
j j j j j j j
--->
r j j j j j j j
***** rule
rl M => r M [label req2] .
M --> r j j j j j j j
r j j j j j j j
--->
r r j j j j j j j
***** rule
rl M => r M [label req2] .
M --> r r j j j j j j j
r r j j j j j j j
--->
r r r j j j j j j j
***** rule
rl M => r M [label req2] .
M --> r r r j j j j j j j
r r r j j j j j j j
--->
r r r r j j j j j j j
***** rule
rl r r r r => j j j j j j j [label buy] .
empty substitution
r r r r j j j j j j j
--->

```

```
(j j j j j j j) j j j j j j j
rewrites: 10 in 20ms cpu (54ms real) (500 rews/sec)
result Marking: j j j j j j j j j j j j j j
```

By using the `frewrite` command we can explore computations where other rules are applied, like the following traced example shows.

```
Maude> set trace on .
Maude> frewrite [10] 1 .
frewrite [10] in LIBRARY-PETRI-NET : 1 .
***** rule
r1 M => r M [label req2] .
M --> 1
1
--->
r 1
***** rule
r1 M => r M [label req2] .
M --> r
r
--->
r r
***** rule
r1 M => r M [label req2] .
M --> r
r
--->
r r
***** rule
r1 M => r M [label req2] .
M --> r
r
--->
r r
***** rule
r1 r r r r => j j j j j j j [label buy] .
empty substitution
r r r r
--->
j j j j j j j
***** rule
r1 j => a [label file] .
empty substitution
j
--->
a
***** rule
r1 M => r M [label req2] .
M --> j
```

```

j
--->
r j
***** rule
rl j => a [label file] .
empty substitution
j
--->
a
***** rule
rl M => r M [label req2] .
M --> j
j
--->
r j
***** rule
rl j => a [label file] .
empty substitution
j
--->
a
rewrites: 10 in 0ms cpu (19ms real) (~ rews/sec)
result (sort not calculated): a (r j) a (r j) a j j j

```

In the above Petri net, which is also known as a place/transition net, tokens are indistinguishable. Thanks to its underlying membership equational logic, Maude can be used to represent more general classes of Petri nets, especially algebraic and colored Petri nets, where tokens are distinguishable and can represent data objects. Systematic approaches to representing different classes of Petri nets in rewriting logic can be found in [335] and [342].

6.5.2 Blocks world

A generalization of Petri net computations is provided by conjunctive planning problems, where the states are described by means of some kind of conjunction of propositions describing basic facts. A typical example in artificial intelligence is the blocks world of a robot. In the present version there is a table on top of which we have the blocks, which can be moved only by means of a robot arm. We have as basic propositions some predicates whose intuitive meaning is given in the accompanying comments.

The rule `pickup` describes how the robot arm, being empty, takes a block from the table, while `putdown` corresponds to the inverse move. The pair of rules `unstack` and `stack` describe similar moves when the block is on top of another one. As opposed to the Petri net transitions, note that here the states in rules have variables, so that they can be instantiated with identifiers for blocks, which are obtained from the predefined module `QID` of identifiers (described later in Section 9.10), which is imported by means of a `protecting` declaration (explained in Section 8.1.1).

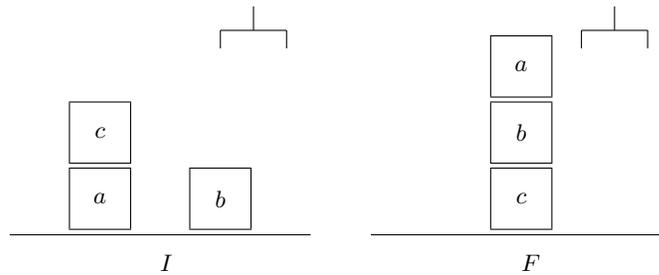


Fig. 6.5. Initial and final states in a world with three blocks

```

mod BLOCKS-WORLD is
  protecting QID .
  sorts BlockId Prop State .
  subsort Qid < BlockId .
  subsort Prop < State .

  op table : BlockId -> Prop [ctor] .      *** block is on the table
  op on : BlockId BlockId -> Prop [ctor] . *** block A is on block B
  op clear : BlockId -> Prop [ctor] .      *** block is clear
  op hold : BlockId -> Prop [ctor] .       *** robot arm holds block
  op empty : -> Prop [ctor] .              *** robot arm is empty
  op 1 : -> State [ctor] .
  op _&_ : State State -> State [ctor assoc comm id: 1] .

  vars X Y : BlockId .

  rl [pickup] : empty & clear(X) & table(X) => hold(X) .
  rl [putdown] : hold(X) => empty & clear(X) & table(X) .
  rl [unstack] : empty & clear(X) & on(X, Y) => hold(X) & clear(Y) .
  rl [stack] : hold(X) & clear(Y) => empty & clear(X) & on(X, Y) .
endm

```

Consider, for example, the states described in Figure 6.5. The initial state I on the left and the final state F on the right are respectively described by the following two terms of sort `State`:

```

empty & clear('c) & clear('b) & table('a) & table('b) & on('c, 'a)

empty & clear('a) & table('c) & on('a, 'b) & on('b, 'c)

```

The fact that the “sequential plan” (in a self-explanatory intuitive notation)

```

unstack(c,a); putdown(c); pickup(b); stack(b,c); pickup(a); stack(a,b)

```

moves the blocks from state I to state F corresponds directly to a sequence of computational rewrite steps applying the corresponding rewrite rules.

If we just ask for a sequence of rewrites starting from the initial state I by using the `rewrite` command, it gets very boring as the robot arm, after picking up the block `b` keeps stacking and unstacking it on `c`; we show a sequence of 5 rewrites:

```
Maude> rew [5] empty & clear('c) & clear('b) & table('a) &
          table('b) & on('c, 'a) .
rewrite [5] in BLOCKS-WORLD : empty & clear('c) & clear('b) &
          table('a) & table('b) & on('c, 'a) .

***** rule
rl empty & table(X) & clear(X) => hold(X) [label pickup] .
X --> 'b
empty & table('a) & table('b) & clear('b) & clear('c) & on('c, 'a)
--->
(table('a) & clear('c) & on('c, 'a)) & hold('b)
***** rule
rl clear(Y) & hold(X) => empty & clear(X) & on(X, Y) [label stack] .
Y --> 'c
X --> 'b
table('a) & clear('c) & hold('b) & on('c, 'a)
--->
(table('a) & on('c, 'a)) & empty & clear('b) & on('b, 'c)
***** rule
rl empty & clear(X) & on(X, Y) => clear(Y) & hold(X) [label unstack] .
X --> 'b
Y --> 'c
empty & table('a) & clear('b) & on('b, 'c) & on('c, 'a)
--->
(table('a) & on('c, 'a)) & clear('c) & hold('b)
***** rule
rl clear(Y) & hold(X) => empty & clear(X) & on(X, Y) [label stack] .
Y --> 'c
X --> 'b
table('a) & clear('c) & hold('b) & on('c, 'a)
--->
(table('a) & on('c, 'a)) & empty & clear('b) & on('b, 'c)
***** rule
rl empty & clear(X) & on(X, Y) => clear(Y) & hold(X) [label unstack] .
X --> 'b
Y --> 'c
empty & table('a) & clear('b) & on('b, 'c) & on('c, 'a)
--->
(table('a) & on('c, 'a)) & clear('c) & hold('b)
rewrites: 5 in 0ms cpu (42ms real) (~ rews/sec)
result State: table('a) & clear('c) & hold('b) & on('c, 'a)
```

To see that it is possible to go from state I to state F we can use the `search` command as follows, where the symbol `=>*` means that we are interested in sequences of zero or more rewrites.

```
Maude> search empty & clear('c) & clear('b) & table('a) &
      table('b) & on('c,'a)
      =>* empty & clear('a) & table('c) & on('a,'b) & on('b,'c) .
```

```
Solution 1 (state 21)
empty substitution
```

```
No more solutions.
```

Now, to show the path leading from the initial to the final state, we use the `show path` command:

```
Maude> show path 21 .
state 0, State: empty & table('a) & table('b) & clear('b) &
      clear('c) & on('c, 'a)
===[ r1 empty & clear(X) & on(X, Y) => clear(Y) & hold(X)
      [label unstack] . ]===>
state 2, State: table('a) & table('b) & clear('a) & clear('b) &
      hold('c)
===[ r1 hold(X) => empty & table(X) & clear(X)
      [label putdown] . ]===>
state 5, State: empty & table('a) & table('b) & table('c) &
      clear('a) & clear('b) & clear('c)
===[ r1 empty & table(X) & clear(X) => hold(X)
      [label pickup] . ]===>
state 8, State: table('a) & table('c) & clear('a) & clear('c) &
      hold('b)
===[ r1 clear(Y) & hold(X) => empty & clear(X) & on(X, Y)
      [label stack] . ]===>
state 13, State: empty & table('a) & table('c) & clear('a) &
      clear('b) & on('b, 'c)
===[ r1 empty & table(X) & clear(X) => hold(X)
      [label pickup] . ]===>
state 17, State: table('c) & clear('b) & hold('a) & on('b, 'c)
===[ r1 clear(Y) & hold(X) => empty & clear(X) & on(X, Y)
      [label stack] . ]===>
state 21, State: empty & table('c) & clear('a) & on('a, 'b) &
      on('b, 'c)
```

Much less information is obtained with the `show path labels` command, that only returns the sequence of names of applied rules:

```
Maude> show path labels 21 .
unstack
putdown
pickup
stack
pickup
stack
```

The reader can check that this sequence of rules coincides indeed with the “sequential plan” that we proposed before.

A simpler version of the blocks world with no robot arm is used in Section 21.2.3 to illustrate an object-oriented approach.

Playing with Maude

with Miguel Palomino
and Alberto Verdejo

Mathematical games and puzzles of all sorts constitute an important subclass of mathematical problems, with a long tradition and an extensive literature [25, 162, 16, 329]. Most of them have in common the fact that they are easy to state and understand, which does not mean that a precise solution is always trivial to find.

In this chapter we make use of Maude system modules to present and illustrate some general techniques that allow solving a diverse enough selection of these problems. We are not concerned with finding neat and concise mathematical solutions, but rather we would like to find out how easy it is to express those problems in the rewriting logic formalism underlying Maude, and how far we can go in their resolution by the use of just brute force and as little ingenuity as possible. In this regard, a clear conclusion is that many of these problems can be represented/specified in Maude in a much simpler way than it would be possible in more conventional languages. Among the main reasons why the rule-based programming paradigm supported by Maude allows so natural a representation of many problems, we would like to mention:

- The syntax is user-definable to a great extent (as described in Chapter 3), which allows the user to choose the most appropriate syntax for each problem. In particular, operators declared by the user can have attributes like associativity and commutativity (see Section 4.4.1), which makes *multiset* rewriting trivial. All the specifications in this chapter make essential use of this feature.
- *Membership equational logic* allows a (first-order) version of functional programming to describe the static aspects of a system.
- The dynamic aspects are described by means of rewrite rules that represent the possible transitions or *changes* in a system. Those rules need only specify the part of the system that actually changes, which makes them quite simple. This corresponds to the fact that *rewriting logic* is a logic very suitable for expressing concurrent action and change [230] in which the so-called “frame problem” [186] is totally avoided.

- The transitive closure of the relation defined by the rules is automatically computed by the Maude system. This, combined with the flexible `search` command, lets the user explore all computations starting at a given state (as described in Section 6.4 and further illustrated here).

On the other hand, it is also true that some of these examples suffer from the state explosion problem, which makes them difficult to solve just by checking all possible combinations. As explained in Section 14.6, Maude’s `META-LEVEL` module can be used to define sophisticated rewriting *strategies*, that can guide and control the rewriting process in a given system module. Defining efficient strategies of this kind is obviously the way of avoiding such combinatorial explosions in many cases.

Most of the problems introduced here are well known and can be found (in some form or another) in a number of sources: see [16] for a classic reference on the subject, [162] for a delightful exposition on how to tackle these problems, [25] for an online presentation, or even [329] for more algebraic ones. In many cases, a clear mathematical solution exists, but not always. In any case, our goal in this chapter is just to show the ease with which Maude lends itself to the specification of these problems, and to try to solve them without much thinking.

This chapter is thus an introduction to rule-based programming in Maude by means of a collection of puzzles showing the language’s expressive power. Even though these examples can be considered quite simple, we hope that readers will find them attractive enough to be encouraged to use Maude for playing with other games of their own, and also for developing more “serious” applications.

7.1 Writing on a blackboard

To begin illustrating the above ideas in Maude, let us consider the following simple example. We have some natural numbers written on a blackboard and we are allowed, at any given time, to replace any two of them by their arithmetic mean. In this case the static part corresponds to the representation of the blackboard and the numbers themselves.

Although we could start by specifying natural numbers in Maude, this is not necessary because we can use the predefined module `NAT` described later in Section 9.2, that provides in particular the addition `_+_` and quotient `_quo_` operations. With these operations the arithmetic mean of natural numbers `N` and `M` can be represented by the term `(N + M) quo 2`.

The blackboard itself can be represented as a *non-empty multiset*, of numbers, using the techniques described in Section 5.4, as follows:

```
sort Blackboard .
subsort Nat < Blackboard .
op __ : Blackboard Blackboard -> Blackboard [ctor assoc comm] .
```

The `subsort` declaration means that a single natural number constitutes a valid representation for the blackboard. Multiset union is represented with empty syntax `...`. Note that this operator has two *attributes*, `assoc` and `comm`, so that terms of sort `Blackboard` are considered *modulo* associativity and commutativity; in this way, for example the terms `2 0` and `0 2` denote the same blackboard, while the terms `2 0` and `0 2 2` denote instead different blackboards, because the number of times a given number appears in the blackboard matters.

Finally, the system's dynamics is specified by the single *rule*

```
rl [replace] : N M => (N + M) quo 2 .
```

It is important to note that it is enough to specify the behavior of the two numbers that are going to be erased, without considering the rest of the numbers in the blackboard.

The `rewrite` command (see Section 6.4) can be used to *execute* the system, by means of the Maude interpreter, which applies the rules and stops when no rule can be applied.

```
Maude> rewrite 6 3 2 .
result NzNat: 4
```

But the numbers chosen to be replaced by their mean can be arbitrarily selected in a nondeterministic way, and this affects the final result. The `search` command can be used to explore the computation graph, as described in Section 6.4. This command receives as arguments the term to be rewritten, the relation used to obtain the desired result states (`=>*` for zero or more rewrites), and the resulting state (a new variable `N:Nat` of sort `Nat` in this case). The computation graph is traversed in a *breadth-first* way.

```
Maude> search 6 3 2 =>* N:Nat .
```

```
Solution 1 (state 4)
N --> 4
Solution 2 (state 5)
N --> 3
No more solutions.
```

In this example, there are exactly two final results corresponding to blackboards with either the natural number 4 or 3, as shown in the two solutions displayed by Maude. Furthermore, since blackboards with just one element cannot be further transformed, in the above `search` command we obtain the same solutions using the relation `=>*` for zero or more rewrites as using the relation `=>!` for rewriting until terminal states, that is, searching for states that cannot be further rewritten. However, this is not the case in most of the examples that follow, and therefore the user must be careful with the exact form of the `search` command.

Notice that there might be different paths leading to the same final state, but the Maude `search` command only enumerates the different substitutions that satisfy the search requirements (with each substitution corresponding to a different state) and not the different ways of obtaining the same substitution. In the previous example, there are two ways of reaching the natural number 3 and one way of reaching 4; however the number of solutions is just two. Maude only looks at the first path it finds to a given state; since the search takes place following a breadth-first strategy, the first path that is found is also the shortest path.

Let us consider now a problem discussed in [162, Chapter 12]. Assume that the numbers $1, 2, \dots, 9$ are written on the blackboard, and that the `replace` rule is modified so that it is allowed to erase any two numbers a and b and to write the new number $ab + a + b$. What numbers can be on the blackboard after applying such operation as many times as possible until there is only one number in the blackboard?

The complete specification as a Maude module is then as follows, where the second line imports the predefined module `NAT` that specifies the natural numbers with the usual notation and arithmetic operations (see Section 9.2 later). This module is imported by means of a `protecting` declaration, whose meaning will be explained in Section 8.1.1. This importation will appear in most of the modules in this chapter.

The constant `initial` is used to give a name to the initial configuration of the blackboard by means of an equation.

```

mod BLACKBOARD is
  protecting NAT .
  sort Blackboard .
  subsort Nat < Blackboard .

  op __ : Blackboard Blackboard -> Blackboard [ctor assoc comm] .
  op initial : -> Blackboard .

  vars M N : Nat .

  eq initial = 1 2 3 4 5 6 7 8 9 .

  rl [replace] : M N => M * N + M + N .
endm

```

We find the solution to the above question by means of the following invocation of the `search` command

```
Maude> search initial =>* N:Nat .
```

```
Solution 1 (state 14989)
N --> 3628799
```

```
No more solutions.
```

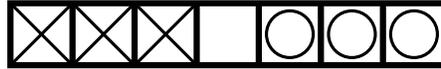


Fig. 7.1. Rabbits ready to jump

which shows that there can be only one possible solution: 3628799.

7.2 The hopping rabbits game

Two teams of n rabbits each, wearing T-shirts marked with a cross and a circle respectively, are placed facing each other on a row with $2n + 1$ positions. The x -team occupies the first n positions and the o -team the last n ; the middle one is left empty. For example, Figure 7.1 shows the initial configuration for $n = 3$.

The goal is to swap the positions of the teams (the players of each team are indistinguishable), with the rabbits moving according to the rules of the game:

1. Rabbits from the x -team can only move rightward, and rabbits from the o -team can only move leftward.
2. A rabbit is allowed to advance one position if that position is empty.
3. A rabbit can jump over a rival if the position behind it is free.

This puzzle is also known as the *toads and frogs puzzle* or *traffic jam puzzle*. It is possible to generalize the puzzle so that the number of elements in each team is different [16].

We represent the state of the game as a *non-empty list* of rabbits, specified by means of an *associative* append operator written with empty syntax `__`; note that associativity is built into the list constructor `__` using the attribute `assoc`, as described in Section 5.2. Each rabbit is represented as a constant `x` or `o`, according to its team, and the constant `free` represents the empty position.

The initial state of the game depends on the number n of rabbits in each team. This is specified by means of an operator `initial` that builds the appropriate initial state, as indicated in the equations below that define inductively the behavior of this operator; more explicitly, in the base case with $n = 0$ there is only the empty position, and in the inductive case we add a rabbit of the appropriate team at each end of the list (the term $s(N)$ represents the successor of N , that is, $N + 1$).

Notice how equations are used to define the initial state, while rules are used to represent the transitions corresponding to the legal moves in the game. As pointed out in the introduction above, we are indeed using two logics, each for a different purpose: equational logic for the static aspects of a system, and rewriting logic for the dynamic aspects.

Since the rules need only specify the parts of the system that change, in this game we only need to consider the positions adjacent to the free position. Thus there are four possible legal moves, and each one is represented by a rule whose label identifies the corresponding move.

The complete specification as a Maude module is then as follows:

```

mod RABBIT-HOP is
  protecting NAT .
  sorts Rabbit RabbitList .
  subsort Rabbit < RabbitList .

  ops x o free : -> Rabbit [ctor] .
  op  _ _ : RabbitList RabbitList -> RabbitList [ctor assoc] .
  op  initial : Nat -> RabbitList .

  var  N : Nat .
  eq  initial(0) = free .
  eq  initial(s(N)) = x initial(N) o .

  rl [xAdvances] : x free => free x .
  rl [xJumps] : x o free => free o x .
  rl [oAdvances] : free o => o free .
  rl [oJumps] : free x o => o x free .
endm

```

Since we are interested in learning how to reach the final position, and in general there are several possible rules that can be applied in a given state, we use the `search` command. The example below is with $n = 3$.

```
Maude> search initial(3) =>* o o o free x x x .
```

```
Solution 1 (state 71)
empty substitution
```

```
No more solutions.
```

The shown solution consists of the empty substitution because the final pattern `o o o free x x x` in the `search` command above is a ground term, that is, it has no variables.

The sequence of 15 steps leading to the final position can be obtained by using the `show path` command (see Section 6.4) as follows, where we only show the beginning of the output.

```

Maude> show path 71 .
state 0, RabbitList: x x x free o o o
===[ rl x free => free x [label xAdvances] . ]===>
state 1, RabbitList: x x free x o o o
===[ rl free x o => o x free [label oJumps] . ]===>
state 4, RabbitList: x x o x free o o

```

```

===[ rl free o => o free [label oAdvances] . ]==>
state 9, RabbitList: x x o x o free o
===[ rl x o free => free o x [label xJumps] . ]==>
state 14, RabbitList: x x o free o x o
...

```

A more compact way of obtaining the sequence of moves, with less information, is provided by the `show path labels` command:

```

Maude> show path labels 71 .
xAdvances
oJumps
oAdvances
xJumps
xJumps
xAdvances
oJumps
oJumps
oJumps
xAdvances
xJumps
xJumps
oAdvances
oJumps
xAdvances

```

Recall that the Maude `search` command does not enumerate the different ways of obtaining the same substitution. In this example, there are at least two ways of reaching the final position, namely, the one shown above and its symmetric; however, the solution itself, in the sense of the final state reached is unique, corresponding to the empty substitution.

7.3 The Josephus problem

As related in [16], Flavius Josephus was a famous Jewish historian who, during the Jewish-Roman war in the first century, was trapped in a cave with a group of 40 Jewish soldiers surrounded by Romans. Legend has it that, preferring death to being captured, the Jews decided to gather in a circle and rotate a dagger around so that every third remaining person would commit suicide. Apparently, Josephus was too keen to live and quickly found out the safe position.

The problem of finding that safe position can be modeled very easily in Maude. The circle representation becomes a (circular) *list*, once the beginning position is chosen. The associative operator `_` is used to build non-empty lists of nonzero natural numbers (which belong to the sort `NzNat` in the predefined module `NAT`, see Section 9.2) representing the original positions of the soldiers

in the circle. Though it is not explicitly represented, we assume that the dagger is initially at position 1.

The idea then consists in continually taking the first two elements in the list and moving them to the end of it while “killing” the third one; when only two are left, the one who initially has the dagger has to commit suicide. Note that in this way the dagger remains always implicitly located at the beginning of the list. Since we need to keep track of both the actual start and end of the list, we enclose the whole list using the operator `{_}`. In this way, we force rewriting to take place only at the top of the term that represents the state.

As in the previous example, the operator `initial` and the corresponding equations are used to build inductively the initial state. Then we specify the rules corresponding to the system transitions; we have three rules, one for each of the cases when there are two, three, or more soldiers in the circle. Notice that there is no rule corresponding to a single soldier list, because this is the situation in which the last remaining soldier decides not to follow the rules of the game.

```

mod JOSEPHUS is
  protecting NAT .
  sorts Morituri Circle .
  subsort NzNat < Morituri .

  op __ : Morituri Morituri -> Morituri [ctor assoc] .
  op {_} : Morituri -> Circle [ctor] .
  op initial : NzNat -> Morituri .

  var M : Morituri .
  vars I1 I2 I3 N : NzNat .

  eq initial(1) = 1 .
  eq initial(s(N)) = initial(N) s(N) .

  rl [kill1>3] : { I1 I2 I3 M } => { M I1 I2 } .
  rl [kill13] : { I1 I2 I3 } => { I1 I2 } .
  --- Rule kill13 is necessary because M cannot be empty
  rl [kill12] : { I1 I2 } => { I2 } .
endm

```

Had we been in the same position as Josephus (and had we had a laptop to run Maude on it), we could have found out the safe spot by executing the command:

```

Maude> rewrite { initial(41) } .
result Circle: {31}

```

Note that at any moment, until the end, only one of the three rules can be applied, thus the final state is reached deterministically.

It is also easy to modify the program so that every i -th person commits suicide, where i is a parameter. The idea is the same, but because of the

parameter it is now necessary to explicitly represent the dagger. For that, we use the constructor `dagger : NzNat NzNat -> Morituri`, whose second argument stores the value of i while the first one acts as a counter: each time an element is moved from the beginning of the list to the end, the first argument is decreased by one; once it reaches 1, the element that is currently the head of the list is “killed,” i.e., removed from the list.

```

mod JOSEPHUS-GENERALIZED is
  protecting NAT .
  sorts Morituri Circle .
  subsort NzNat < Morituri .

  op dagger : NzNat NzNat -> Morituri [ctor] .
  op _ : Morituri Morituri -> Morituri [ctor assoc] .
  op { _ } : Morituri -> Circle [ctor] .
  op initial : NzNat NzNat -> Morituri .

  var M : Morituri .
  vars I I1 I2 N : NzNat .

  eq initial(1, I) = dagger(I, I) 1 .
  eq initial(s(N), I) = initial(N, I) s(N) .

  rl [kill] : { dagger(1, I) I1 M } => { dagger(I, I) M } .
  rl [next] : { dagger(s(N), I) I1 M } => { dagger(N, I) M I1 } .
  rl [last] : { dagger(N, I) I1 } => { I1 } .
  --- The last one throws the dagger away!
endm

Maude> rewrite { initial(41, 3) } .
result Circle: {31}

```

As expected, the safe position obtained in this case coincides with the one obtained previously.

7.4 The three basins puzzle

The following is a classic puzzle with a recent cameo in the 1995 Hollywood hit *Die Hard: With a Vengeance*. In the movie, McClane and Zeus have to deactivate a bomb by placing 4 gallons of water on a scale. The supply of water is unlimited, but they only have three basins with capacities of 3, 5, and 8 gallons, respectively.

The problem can be specified in Maude as follows. A basin is represented by means of the constructor `basin` with two natural numbers as arguments: the first one is the basin’s capacity and the second one indicates how full it is. We can think of a basin as an object with two attributes, namely, its capacity and its current content. This way of thinking leads to an *object-based* style of

programming, where objects change their attributes as result of interacting with other objects; these interactions are represented as rules on *configurations* that are non-empty *multisets* of objects (for more information on this style of programming and how it is supported by Maude, see Chapter 11 later). The multiset constructor is written with empty syntax and declared with attributes `assoc` and `comm`. The constant `initial` defines the initial configuration.

At any given time, we can either empty one of the basins, or fill it completely; the rules `empty` and `fill` below take care of this. When there is enough space in one of the basins to hold the current content of another, we can transfer all the water from this second one by using the rule `transfer1` (note that this is a *conditional rule* introduced with the keyword `cr1`). The case when, after pouring one basin over another, there is still some water left is dealt with by the conditional rule `transfer2` (where the operator `sd` denotes the subtraction operation over natural numbers, specified in the predefined Maude module `NAT`, see Section 9.2). These last two rules could be combined into a single one, but the specification would not be so clear.

```

mod DIE-HARD is
  protecting NAT .
  sorts Basin BasinSet .
  subsort Basin < BasinSet .

  op basin : Nat Nat -> Basin [ctor] . --- Capacity / Content
  op _ _ : BasinSet BasinSet -> BasinSet [ctor assoc comm] .
  op initial : -> BasinSet .

  vars M1 N1 M2 N2 : Nat .

  eq initial = basin(3, 0) basin(5, 0) basin(8, 0) .

  r1 [empty] : basin(M1, N1) => basin(M1, 0) .
  r1 [fill] : basin(M1, N1) => basin(M1, M1) .
  cr1 [transfer1] : basin(M1, N1) basin(M2, N2)
    => basin(M1, 0) basin(M2, N1 + N2)
    if N1 + N2 <= M2 .
  cr1 [transfer2] : basin(M1, N1) basin(M2, N2)
    => basin(M1, sd(N1 + N2, M2)) basin(M2, M2)
    if N1 + N2 > M2 .
endm

```

We can now find out the *shortest* solution with the help of the `search` command, due to the breadth-first way of searching (the argument `[1]` tells Maude to look only for one solution). Notice that the *pattern* used after the arrow `=>*` represents any set of basins with at least one of them containing exactly 4 gallons.

```
Maude> search [1] initial =>* basin(N:Nat, 4) B:BasinSet .
```

```
Solution 1 (state 75)
B:BasinSet --> basin(3, 3) basin(8, 3)
N:Nat --> 5
```

The sequence of actions that leads to the solution can be seen using the command `show path` as follows, where we omit part of the information about the rules used.

```
Maude> show path 75 .
state 0, BasinSeet: basin(3, 0) basin(5, 0) basin(8, 0)
===[ rl ... fill ]===>
state 2, BasinSeet: basin(3, 0) basin(5, 5) basin(8, 0)
===[ crl ... transfer2 ]===>
state 9, BasinSeet: basin(3, 3) basin(5, 2) basin(8, 0)
===[ crl ... transfer1 ]===>
state 20, BasinSeet: basin(3, 0) basin(5, 2) basin(8, 3)
===[ crl ... transfer1 ]===>
state 37, BasinSeet: basin(3, 2) basin(5, 0) basin(8, 3)
===[ rl ... fill ]===>
state 55, BasinSeet: basin(3, 2) basin(5, 5) basin(8, 3)
===[ crl ... transfer2 ]===>
state 75, BasinSeet: basin(3, 3) basin(5, 4) basin(8, 3)
```

7.5 Crossing the bridge

The four members of U2, the famous rock band, are in a tight situation. Their concert starts in 17 minutes, and in order to get to the stage they must first cross an old bridge through which only a maximum of two persons can walk over at the same time. It is already dark and, because of the bad condition of the bridge, to avoid falling into the darkness it is necessary to cross it with the help of a flashlight. Unfortunately, they only have one. Knowing that it takes Bono, Edge, Adam, and Larry, 1, 2, 5, and 10 minutes, respectively, to cross the bridge, is there a way that they can make it to the concert on time?

The current state of the group can be represented in Maude by a *multiset* (a term of sort `Group` below) consisting of performers, the flashlight, and a watch to keep track of the time. The flashlight and the performers have a `Place` associated with them, indicating whether their current position is to the left or to the right of the bridge; each performer, in addition, also carries the time it takes him to cross the bridge. As in the previous example, we can think of this specification as one fitting an object-based style of programming. In order to change the position from `left` to `right` and vice versa, we use an auxiliary operation `changePos`, which is defined by means of two simple equations.

The traversing of the bridge is modeled by two rewrite rules: the first one for the case in which a single person crosses it, and the second for when there are two persons. Note that for some persons to be allowed to cross, their

position relative to the bridge must be the same as that of the flashlight, which is represented by having the same variable `P` twice on the lefthand side of the rules. Moreover, since `--` is commutative, the condition in the second rule involves no loss of generality.

```

mod U2 is
  protecting NAT .
  sorts Performer Object Group Place .
  subsorts Performer Object < Group .

  ops left right : -> Place [ctor] .
  op changePos : Place -> Place .
  op flashlight : Place -> Object [ctor] .
  op watch : Nat -> Object [ctor] .
  op performer : Nat Place -> Performer [ctor] .
  op -- : Group Group -> Group [ctor assoc comm] .
  op initial : -> Group .

  var P : Place .
  vars M N N1 N2 : Nat .

  eq initial
    = watch(0) flashlight(left) performer(1, left)
      performer(2, left) performer(5, left) performer(10, left) .

  eq changePos(left) = right .
  eq changePos(right) = left .

  rl [one-crosses] :
    watch(M) flashlight(P) performer(N, P)
    => watch(M + N) flashlight(changePos(P))
      performer(N, changePos(P)) .
  crl [two-cross] :
    watch(M) flashlight(P) performer(N1, P) performer(N2, P)
    => watch(M + N1) flashlight(changePos(P))
      performer(N1, changePos(P))
      performer(N2, changePos(P))
    if N1 > N2 .
endm

```

A solution can now be found quickly by looking for a state in which all performers (and the flashlight) are to the `right` of the bridge. Notice how the `search` command is invoked with a `such that` clause that imposes a condition that solutions have to fulfill; in our example, that the total time should be less than or equal to 17 minutes (see Section 25.4 for all the possible options of the `search` command).

```

Maude> search [1] initial
=>* flashlight(right) watch(N:Nat)

```

```

    performer(1, right) performer(2, right)
    performer(5, right) performer(10, right)
  such that N:Nat <= 17 .

```

```

Solution 1 (state 402)
N --> 17

```

The solution takes exactly 17 minutes (a happy ending after all!) and the complete sequence of appropriate actions can be shown as follows:

```

Maude> show path 402 .
state 0, Group: flashlight(left) watch(0)
    performer(1, left) performer(2, left)
    performer(5, left) performer(10, left)
===[ crl ... two-cross ]===>
state 5, Group: flashlight(right) watch(2)
    performer(1, right) performer(2, right)
    performer(5, left) performer(10, left)
===[ rl ... one-crosses ]===>
state 15, Group: flashlight(left) watch(3)
    performer(1, left) performer(2, right)
    performer(5, left) performer(10, left)
===[ crl ... two-cross ]===>
state 71, Group: flashlight(right) watch(13)
    performer(1, left) performer(2, right)
    performer(5, right) performer(10, right)
===[ rl ... one-crosses ]===>
state 158, Group: flashlight(left) watch(15)
    performer(1, left) performer(2, left)
    performer(5, right) performer(10, right)
===[ crl ... two-cross ]===>
state 402, Group: flashlight(right) watch(17)
    performer(1, right) performer(2, right)
    performer(5, right) performer(10, right)

```

After sorting out the information, it becomes clear that Bono and Edge should be the first to cross. Then Bono returns with the flashlight, which he gives to Adam and Larry. Finally, Edge takes the flashlight back to Bono and they cross the bridge together for the last time.

Note that, in order for the `search` command to stop, we need to tell Maude to look only for one solution. Otherwise, it will continue exploring all possible combinations, taking increasingly larger amounts of time, and it will never end.

If we invoked the same `search` command but omitting the `such that` clause, then we would get

```

Maude> search [1] initial
=>* flashlight(right) watch(N:Nat)
    performer(1, right) performer(2, right)

```

```
performer(5, right) performer(10, right) .
```

```
Solution 1 (state 396)
N --> 19
```

Unfortunately, this “solution” of the search, although as short as possible in terms of number of rewrite steps, is not a solution of the problem because it does not satisfy our time constraint, taking 19 minutes, two more minutes than allowed.

7.6 The looping chips game

In the next game, taken from [25], four chips of different colors have been placed in consecutive places on a 12×1 board whose ends have been glued together in a circular fashion. Each chip can be moved 5 places from its current location, either clockwise or counterclockwise, assuming the final position is empty. The goal is to arrange the chips in reverse order, over the original four squares.

The state is again represented by a *multiset* of `Places`, with each `Place` determined by its position in the board and the color of the chip on it or `e` if empty.¹ As in previous examples, places can be understood as objects and the state of the game at each moment is given by a configuration of objects. The constants `initial` and `final` represent the initial and final configurations.

There are two possible legal moves in the game, but taking advantage of the circularity of the board, it is possible to represent both of them together in a single rule, as shown below; notice how the first part of the rule’s condition considers the two possible directions of the move by exchanging the variables `I` and `J`, and the second part, `C /= e`, forbids moves of empty positions.

```
mod CHIPS is
  protecting NAT .
  sorts Place Board Chip .
  subsort Place < Board .

  ops r b g y e : -> Chip [ctor] . --- colors and empty
  op place : Nat Chip -> Place [ctor] .
  op _ : Board Board -> Board [ctor assoc comm] .
  ops initial final : -> Board .

  eq initial
    = place(0, r) place(1, b) place(2, g) place(3, y)
      place(4, e) place(5, e) place(6, e) place(7, e)
```

¹ In this example, we could also use a list representation for the state; this would simplify the representation of places, but instead the corresponding rules would be more complex.

```

    place(8, e) place(9, e) place(10,e) place(11,e) .

eq final
  = place(0, y) place(1, g) place(2, b) place(3, r)
    place(4, e) place(5, e) place(6, e) place(7, e)
    place(8, e) place(9, e) place(10,e) place(11,e) .

vars I J : Nat .
var C : Chip .

crl [move] :
  place(I, C) place(J, e) => place(I, e) place(J, C)
  if (((I + 5) rem 12 == J) or ((J + 5) rem 12 == I))
    /\ C /= e .
endm

```

Then, we can use the command

```
Maude> search initial =>* B:Board such that B:Board == final .
```

No solution.

to prove that it is *not* possible, by using the allowed moves, to reverse the original order of the chips. Since the constant `final` is not a pattern (because it can be reduced), it cannot be used after the arrow in the `search` command; however, the clause at the end allows us to say the same thing in the correct way. Of course, we could also write as pattern the whole term in the righthand side of the equation for `final`, but it is simpler to abbreviate such a long term.

7.7 The Khun Phan puzzle

The Khun Phan puzzle is one of those typical puzzles based on a rectangular board over which some pieces can be slid. The goal is to move the pieces so as to reach a certain configuration in which sometimes a picture becomes clear, or other times a piece understood as representing some character is freed from his guards. Figure 7.2 shows the initial configuration that we will consider. The board is a 4×5 rectangle, there is one 2×2 piece, five rectangular pieces of size 2×1 , and four smaller squares with dimension 1×1 ; there are only two empty spaces that must be used to slide the pieces. The goal we consider is to move the pieces so as to put the big square in the position where the small ones are initially. An additional twist would be to reach a completely symmetric position with respect to the original one.

The state of the board is also represented as a *multiset* of pieces with the operator `__`. There is a different constructor for each piece, `bigsq`, `hrect`, `vrect`, and `smallsq`, and another one, `empty`, to indicate an empty space (that is considered to be just a special kind of piece). These constructors take

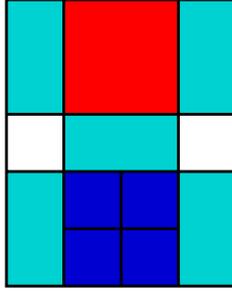


Fig. 7.2. The Khun Phan puzzle

two natural numbers as arguments, corresponding to the coordinates of the upper left corner of the piece; the origin $(1, 1)$ is located at the upper left corner of the board.

The representation of the moves as rewrite rules is then immediate: each move involves a piece and at least one empty space. For each kind of piece there are four rules, corresponding to the four possible directions. For example, moving the big square one position to the right is captured by the rule `Sqr` below. Again, we can think of these pieces as objects and of rules as interactions between them. The complete specification is as follows:

```

mod KHUN-PHAN is
  protecting NAT .
  sorts Piece Board .
  subsort Piece < Board .

  op _ : Board Board -> Board [ctor assoc comm] .
  ops empty bigsq smallsq hrect vrect : Nat Nat -> Piece [ctor] .
  op initial : -> Board .

  vars X Y : Nat .

  eq initial
    = vrect(1, 1)      bigsq(2, 1)      vrect(4, 1)
      empty(1, 3)      hrect(2, 3)      empty(4, 3)
      vrect(1, 4) smallsq(2, 4) smallsq(3, 4) vrect(4, 4)
        smallsq(2, 5) smallsq(3, 5) .

  rl [sqr] : smallsq(X, Y) empty(s(X), Y)
    => empty(X, Y) smallsq(s(X), Y) .
  rl [sql] : smallsq(s(X), Y) empty(X, Y)
    => empty(s(X), Y) smallsq(X, Y) .
  rl [squ] : smallsq(X, s(Y)) empty(X, Y)
    => empty(X, s(Y)) smallsq(X, Y) .
  rl [sqd] : smallsq(X, Y) empty(X, s(Y))

```

```

=> empty(X, Y) smallsq(X, s(Y)) .

r1 [Sqr] : bigsq(X, Y) empty(s(s(X)), Y) empty(s(s(X)), s(Y))
=> empty(X, Y) empty(X, s(Y)) bigsq(s(X), Y) .
r1 [Sql] : bigsq(s(X), Y) empty(X, Y) empty(X, s(Y))
=> empty(s(s(X)), Y) empty(s(s(X)), s(Y)) bigsq(X, Y) .
r1 [Squ] : bigsq(X, s(Y)) empty(X, Y) empty(s(X), Y)
=> empty(X, s(s(Y))) empty(s(X), s(s(Y))) bigsq(X, Y) .
r1 [Sqd] : bigsq(X, Y) empty(X, s(s(Y))) empty(s(X), s(s(Y)))
=> empty(X, Y) empty(s(X), Y) bigsq(X, s(Y)) .

r1 [hrectr] : hrect(X, Y) empty(s(s(X)), Y)
=> empty(X, Y) hrect(s(X), Y) .
r1 [hrectl] : hrect(s(X), Y) empty(X, Y)
=> empty(s(s(X)), Y) hrect(X, Y) .
r1 [hrectu] : hrect(X, s(Y)) empty(X, Y) empty(s(X), Y)
=> empty(X, s(Y)) empty(s(X), s(Y)) hrect(X, Y) .
r1 [hrectd] : hrect(X, Y) empty(X, s(Y)) empty(s(X), s(Y))
=> empty(X, Y) empty(s(X), Y) hrect(X, s(Y)) .

r1 [vrectr] : vrect(X, Y) empty(s(X), Y) empty(s(X), s(Y))
=> empty(X, Y) empty(X, s(Y)) vrect(s(X), Y) .
r1 [vrectl] : vrect(s(X), Y) empty(X, Y) empty(X, s(Y))
=> empty(s(X), Y) empty(s(X), s(Y)) vrect(X, Y) .
r1 [vrectu] : vrect(X, s(Y)) empty(X, Y)
=> empty(X, s(s(Y))) vrect(X, Y) .
r1 [vrectd] : vrect(X, Y) empty(X, s(s(Y)))
=> empty(X, Y) vrect(X, s(Y)) .
endm

```

Then we can use the command

```
Maude> search initial =>* B:Board bigsq(2, 4) .
```

to get all possible 964 final configurations of the game.² The final state used, `B:Board bigsq(2,4)`, represents any final situation such that the upper left corner of the big square is at coordinates (2,4). No wonder it takes some time to find a solution: close examination of the first one, corresponding to the shortest path leading to the final configuration due to the breadth-first search, reveals that it involves 112 moves! We can use the `show path labels` command to see only the sequence of the names of the applied rules, corresponding to such moves (we only show some of them at the beginning and at the end).

```
Maude> show path labels 23721 .
hrectr
```

² Recall that the `search` command does not enumerate the different ways of reaching each configuration.

```

squ
sql
hrectl
vrectu
.....
sqr
hrectu
squ
sql
Sql

```

On the other hand, the command

```

Maude> search initial
=>* vrect(1, 1) smallsq(2, 1) smallsq(3, 1) vrect(4, 1)
      smallsq(2, 2) smallsq(3, 2)
      empty(1, 3)      hrect(2, 3)      empty(4, 3)
      vrect(1, 4)      bigsq(2, 4)      vrect(4, 4) .

```

No solution.

shows that it is *not* possible to reach a position symmetric to the initial one.

7.8 Crossing the river

A shepherd needs to transport to the other side of a river a wolf, a goat, and a cabbage. He has only a boat with room for the shepherd himself and another item. The problem is that in the absence of the shepherd the wolf would eat the goat, and the goat would eat the cabbage.

We represent with constants `left` and `right` the two sides of the river. The shepherd and his belongings are represented as objects with an attribute indicating the side of the river in which each is located and are grouped together with a multiset operator `__`; the constant `initial` denotes the initial situation, where we assume that all the objects are located on the left riverbank.

```

sorts Side Group .
ops left right : -> Side [ctor] .
--- shepherd, wolf, goat, cabbage
ops s w g c : Side -> Group [ctor] .
op __ : Group Group -> Group [ctor assoc comm] .
op initial : -> Group .

```

The rules represent the ways of crossing the river allowed by the capacity of the boat; an auxiliary `change` operation is used to modify the corresponding attributes.

```

op change : Side -> Side .
eq change(left) = right .
eq change(right) = left .

r1 [shepherd-alone] : s(S) => s(change(S)) .
r1 [wolf] : s(S) w(S) => s(change(S)) w(change(S)) .
r1 [goat] : s(S) g(S) => s(change(S)) g(change(S)) .
r1 [cabbage] : s(S) c(S) => s(change(S)) c(change(S)) .

```

It only remains to specify the facts that the wolf eats the goat when they are alone, and that the goat eats the cabbage. One might think about using additional rules to represent these “eating transitions,” but this would not be correct, because it would allow paths in which the shepherd leaves for example the goat and the cabbage alone and later comes back to find that the cabbage is still there. The point is that the eating actions do not constitute additional alternatives to the ones encoded by the rules. If the goat is left alone with the cabbage, no action should be able to preempt its eating it. Instead of rules, we use equations, that represent instant actions and therefore should be executed immediately.

```

ceq w(S) g(S) s(S') = w(S) s(S') if S /= S' .
--- wolf eats goat
ceq c(S) g(S) w(S') s(S') = g(S) w(S') s(S') if S /= S' .
--- goat eats cabbage

```

Note that the statement of the problem is underspecified; it is not clear what exactly should happen if the wolf, the goat, and the cabbage were left alone. In the previous equations we have decided that the goat is not fast enough and gets eaten by the wolf before it can take a bite of the cabbage.

At this point it would seem that our module is ready to be executed but, in fact, a major problem has inadvertently crept in: lack of coherence. Indeed, the following diagram is not commutative, where \rightarrow^1 denotes one step of rewriting by means of the rules.

$$\begin{array}{ccc}
 w(\text{left}) \ g(\text{left}) \ c(\text{left}) \ s(\text{right}) & \xrightarrow{1} & w(\text{left}) \ g(\text{left}) \ c(\text{left}) \ s(\text{left}) \\
 \parallel & & \parallel \\
 w(\text{left}) \ c(\text{left}) \ s(\text{right}) & \xrightarrow{1} & w(\text{left}) \ c(\text{left}) \ s(\text{left})
 \end{array}$$

That is, some rewrites would be missed by first simplifying terms with the equations, so that coherence is lost and the module becomes inadmissible.

Coherence could be recovered by introducing additional rules; for example, a rule like

```

r1 w(left) c(left) s(right) => w(left) g(left) c(left) s(left) .

```

would make the above diagram commutative. However, such a rule would generate goats out of the blue and does not correspond to any realistic aspect of the problem at hand.

An alternative solution, that applies in general [294, 257], is to *encapsulate* the whole state within an auxiliary operator, say $\{_ \}$, so that rules are specified relative to it and can be forced to be triggered only if no equation is enabled. This is the approach followed in the module below, where `toBeEaten` is an operator that is true of a state if either the wolf or the goat can eat.

```

mod RIVER-CROSSING is
  sorts Side Group State .

  ops left right : -> Side [ctor] .
  op change : Side -> Side .

  --- shepherd, wolf, goat, cabbage
  ops s w g c : Side -> Group [ctor] .
  op _ : Group Group -> Group [ctor assoc comm] .

  op {_} : Group -> State [ctor] .
  op toBeEaten : Group -> Bool .

  op initial : -> State .

  vars S S' : Side .
  var G : Group .

  eq change(left) = right .
  eq change(right) = left .

  ceq w(S) g(S) s(S') = w(S) s(S') if S /= S' .
  --- wolf eats goat
  ceq c(S) g(S) w(S') s(S') = g(S) w(S') s(S') if S /= S' .
  --- goat eats cabbage

  ceq toBeEaten(w(S) g(S) s(S') G) = true if S /= S' .
  ceq toBeEaten(c(S) g(S) s(S') G) = true if S /= S' .
  eq toBeEaten(G) = false [owise] .

  eq initial = { s(left) w(left) g(left) c(left) } .

  crl [shepherd-alone] : { s(S) G } => { s(change(S)) G }
    if not(toBeEaten(s(S) G)) .
  crl [wolf] : { s(S) w(S) G } => { s(change(S)) w(change(S)) G }
    if not(toBeEaten(s(S) w(S) G)) .
  rl [goat] : { s(S) g(S) G } => { s(change(S)) g(change(S)) G } .
  crl [cabbage] : { s(S) c(S) G } => { s(change(S)) c(change(S)) G }
    if not(toBeEaten(s(S) c(S) G)) .
endm

```

The resulting module is coherent and, by using the `search` command, we can confirm that there is a way the shepherd can safely take his belongings to the other side of the river.

```
Maude> search initial =>* { w(right) s(right) g(right) c(right) } .
```

```
Solution 1 (state 27)
empty substitution
```

```
No more solutions.
```

Now, by requesting the path leading from the initial state to the final one,

```
Maude> show path 27 .
state 0, State: {s(left) w(left) g(left) c(left)}
===[ rl ... goat ]===>
state 3, State: {s(right) w(left) g(right) c(left)}
===[ crl ... shepherd-alone ]===>
state 8, State: {s(left) w(left) g(right) c(left)}
===[ crl ... wolf ]===>
state 14, State: {s(right) w(right) g(right) c(left)}
===[ rl ... goat ]===>
state 21, State: {s(left) w(right) g(left) c(left)}
===[ crl ... cabbage ]===>
state 25, State: {s(right) w(right) g(left) c(right)}
===[ crl ... shepherd-alone ]===>
state 26, State: {s(left) w(right) g(left) c(right)}
===[ rl ... goat ]===>
state 27, State: {s(right) w(right) g(right) c(right)}
```

we obtain the shortest sequence of moves necessary to safely carry all the belongings from one side to the other. We can see the “skeleton” of such moves as follows:

```
Maude> show path labels 27 .
goat
shepherd-alone
wolf
goat
cabbage
shepherd-alone
goat
```

In Section 13.7 we will see a different approach to solve this problem, based on model checking the system proposed at the beginning of this section, with neither equations nor rules for the eating actions.

7.9 Dominoes on a chessboard

We are given an 8×8 board and 31 dominoes, each of which can be used to cover exactly two squares of the board. Is it possible to arrange the dominoes on the board so as to leave uncovered the upper left and the lower right corners?

The answer is no, and a neat solution is given in [162, Chapter 1] among other places. It is enough to imagine the board painted like a chessboard and realize that each domino necessarily covers both a black and a white square: since the corners to be left uncovered are of the same color, such a covering is not possible. This solution, however, requires some ingenuity but, given our present lazy approach, we are just going to model the problem in Maude and try to solve it by sheer force.

Again, the state of the board is represented as a *multiset* of squares. Each square has three arguments: the first two are its coordinates (column/row) and the third indicates whether it is already covered or still empty. Since the position of the squares in the board is fixed, the attribute `comm` for `--` could be thought to be unnecessary. This, however, allows a more homogeneous and simple presentation of the rules, taking care of positioning the dominoes *both* horizontally and vertically, by focusing only on those two squares involved in placing a domino. Having the board represented as a list by removing the attribute `comm` would force us to represent all the squares in between them in one of the rules.

```

mod CHESS-COVER is
  protecting NAT .
  sorts Status Pos Board State .
  subsort Pos < Board .

  ops e c : -> Status [ctor] . --- empty and covered
  op sq : Nat Nat Status -> Pos [ctor] .
  op -- : Board Board -> Board [ctor assoc comm] .
  ops initial final : -> Board .

  vars I J I1 J1 : Nat .
  var B : Board .

  eq initial
    = sq(1,1,e) sq(2,1,e) sq(3,1,e) sq(4,1,e) sq(5,1,e) ...
      ... sq(4,8,e) sq(5,8,e) sq(6,8,e) sq(7,8,e) sq(8,8,e) .

  eq final
    = sq(1,1,e) sq(2,1,c) sq(3,1,c) sq(4,1,c) sq(5,1,c) ...
      ... sq(4,8,c) sq(5,8,c) sq(6,8,c) sq(7,8,c) sq(8,8,e) .

  rl [hor] : sq(I, J, e) sq(s(I), J, e)
    => sq(I, J, c) sq(s(I), J, c) .

```

```

r1 [ver] : sq(I, J, e) sq(I, s(J), e)
=> sq(I, J, c) sq(I, s(J), c) .
endm

```

In the above equations, defining the constants `initial` and `final`, we have omitted the lengthy but obvious representation of the 64 squares in the chessboard.

Now, the command

```

Maude> search initial =>* B:Board such that B:Board == final .

```

should return the answer. This time, however, a state explosion problem occurs and in our computer the program runs out of memory before producing any result. To solve it, we are forced to use some ingenuity after all.

Note that, instead of placing the dominoes in an arbitrary order, we could do it starting either from the top of the board towards the bottom, or from the left towards the right, or even in a diagonal manner beginning at the upper left corner. The first two approaches still fail to return an answer, but the third does. To implement it, we need an auxiliary operator `cDiag`, that checks whether all positions in the board that come before a given square according to the diagonal order have been already covered. Furthermore, as we did with the Josephus examples in Section 7.3, we need to have full control of all the elements in the board and for that we enclose them inside the constructor `{_}`.

```

op {_} : Board -> State [ctor] .
op cDiag : Nat Nat Board -> Bool .

ceq cDiag(I, J, sq(I1, J1, e) B) = false
  if (I1 + J1 < I + J) /\ (I1 + J1 >= 3) .
eq cDiag(I, J, B) = true [owise] .

crl [hor] : { B sq(I, J, e) sq(s(I), J, e) }
=> { B sq(I, J, c) sq(s(I), J, c) }
  if cDiag(I, J, B) .
crl [ver] : { B sq(I, J, e) sq(I, s(J), e) }
=> { B sq(I, J, c) sq(I, s(J), c) }
  if cDiag(I, J, B) .

```

The “otherwise” attribute, `owise`, is just a convenient way of specifying the behavior of `cDiag` in all remaining cases without having to write equations for them. The result is still an equational theory since the `owise` attribute is just a shorthand for a conditional equation (as explained in Section 4.5.4).

Finally, the result of the command

```

Maude> search { initial } =>* { B:Board }
      such that B:Board == final .

```

No solution.

proves that such a covering is indeed not possible.

7.10 Black or white

Imagine an 8×8 board where the four corners are colored white and all the other squares are black. Is it possible to make all the squares white by recoloring rows and columns? “Recoloring” is the operation of changing the colors of all the squares in a row or a column. The solution to this problem can be found in [162, Chapter 12].

The board is represented as in the previous example, but now the third attribute of a square is the corresponding color, either black (constant `b`) or white (`w`).

Recoloring a row or a column is represented as a rewrite rule that makes use of an auxiliary operation `recolor` that changes the color from black to white and vice versa.

In addition to the constant `initial`, representing the initial coloring of the board (whose representation is again omitted), we have a predicate `allWhite?` to check whether all the squares are white. This predicate has a very simple equational specification thanks to the `owise` attribute; intuitively, the corresponding equations below just mean that all the squares are white when it is not true that at least one square is black.

```

mod RECOLORING is
  protecting NAT .
  sorts Place Board Color .
  subsort Place < Board .

  ops w b : -> Color [ctor] .
  op sq : Nat Nat Color -> Place [ctor] .
  op _ _ : Board Board -> Board [ctor assoc comm] .
  op initial : -> Board .

  op recolor : Color -> Color .
  op allWhite? : Board -> Bool .

  vars I J : Nat .
  vars C1 C2 C3 C4 C5 C6 C7 C8 : Color .
  var B : Board .

  eq recolor(b) = w .
  eq recolor(w) = b .

  eq allWhite?(sq(I,J,b) B) = false .
  eq allWhite?(B) = true [owise] .

  eq initial

```

```

= sq(1,1,w) sq(1,2,b) sq(1,3,b) sq(1,4,b) ...
  ... sq(8,5,b) sq(8,6,b) sq(8,7,b) sq(8,8,w) .

rl [recolor-column] :
  sq(I,1,C1) sq(I,2,C2) sq(I,3,C3) sq(I,4,C4)
  sq(I,5,C5) sq(I,6,C6) sq(I,7,C7) sq(I,8,C8)
=> sq(I,1,recolor(C1)) sq(I,2,recolor(C2))
    sq(I,3,recolor(C3)) sq(I,4,recolor(C4))
    sq(I,5,recolor(C5)) sq(I,6,recolor(C6))
    sq(I,7,recolor(C7)) sq(I,8,recolor(C8)) .

rl [recolor-row] :
  sq(1,J,C1) sq(2,J,C2) sq(3,J,C3) sq(4,J,C4)
  sq(5,J,C5) sq(6,J,C6) sq(7,J,C7) sq(8,J,C8)
=> sq(1,J,recolor(C1)) sq(2,J,recolor(C2))
    sq(3,J,recolor(C3)) sq(4,J,recolor(C4))
    sq(5,J,recolor(C5)) sq(6,J,recolor(C6))
    sq(7,J,recolor(C7)) sq(8,J,recolor(C8)) .

endm

```

The command

```
Maude> search initial =>* B:Board such that allWhite?(B:Board) .
```

No solution.

proves that it is *not* possible to get a configuration with all the squares colored white from our initial configuration.

7.11 The game is not over

We have specified several other games and puzzles, but we think that by now the pattern by which these problems are modeled and solved in Maude should be clear. There are however several advanced features available in Maude that can be useful in some examples, and that we have not considered here with the idea of keeping things at an introductory level.

The first one is the possibility of using membership axioms (see Section 4.2) to refine the representation of the state. For example, the multiset constructor allows repetition of elements, but this should be forbidden in some situations; as another example, in the Khun Phan puzzle a piece cannot be stacked on top of another. In the puzzles above we have not made use of this feature, but memberships are the right technical tool to make sure that all the elements are different.

The breadth-first search implemented by the `search` command has been the most appropriate in our examples. Nonetheless, in other examples a different kind of search might be more suitable. As already mentioned at the beginning of this chapter and further explained in Section 14.6, many different

strategies can be defined with the help of the `META-LEVEL` module described in Chapter 14. Furthermore, we are currently designing a strategy language for Maude in which the user can express complex requirements on the computation, and in particular on the kind of search desired [234]. A Core Maude implementation of this strategy language is ongoing, to make it part of a future release of Maude.

The paper [295], from which this chapter has been adapted, contains a small comparison with other rule-based programming languages, namely, ELAN [27], CHR [163], and ASF+SDF [104].

The following are some additional problems that the reader may find interesting to try to solve using the techniques described in this chapter:

1. Mr. Smith and his wife invited four other couples to have dinner at their house. When they arrived, some people shook hands with some others (of course, nobody shook hands with their spouse or with the same person twice), after which Mr. Smith asked everyone how many times they had shaken hands. The answers, it turned out, were different in all cases. How many people did Mrs. Smith shake hands with?
2. The numbers 25 and 36 are written on a blackboard. At each turn, a player writes on the board the (positive) difference between two numbers already on the blackboard—if this number does not already appear on it. The loser is the player who cannot write a number. Prove that the second player will always win.
3. A 3×3 table is filled with numbers. It is allowed to increase simultaneously all the numbers in any 2×2 square by 1. Is it possible, using these operations, to obtain the table $(4, 9, 5), (10, 18, 12), (6, 13, 7)$ from a table initially filled with zeros?

Module Operations

Specifications and code should be structured in *modules* of relatively small size to facilitate understandability of large systems, increase reusability of components, and localize the effects of system changes. In Maude, these goals are achieved by means of a *module algebra* that supports parameterized programming techniques in the OBJ3 style [174] as well as the definition of *module hierarchies*, i.e., acyclic graphs of module importations; that is, each functional or system module can import other Maude modules as submodules. Since the submodule relation is *transitive*, we can in this way develop module *hierarchies*. Mathematically, we can think of such hierarchies as partial orders of *theory inclusions*, that is, the theory of the importing module contains the theories of its submodules as subtheories.

As in Clear [42], OBJ [174], and other specification languages in that tradition, the abstract syntax for writing specifications in Maude can be seen as given by *module expressions*, where the notion of module expression is understood as an expression that defines a new module out of previously defined modules by combining and/or modifying them according to a specific set of operations. In fact, structuring is essential in all specification languages, not only to facilitate the construction of specifications from already existing ones—with more or less flexible reusability mechanisms—but also for managing the complexity of understanding and analyzing large specifications. Maude supports module operations for summation, renaming, and instantiation of parameterized modules.

Section 8.1 introduces module importations and the different modes in which such importations can take place. Section 8.2 discusses the summation and renaming module expressions. Section 8.3 introduces parameterized programming, including the use of theories and views, the parameterization of functional and system modules, and the instantiation of parameterized modules. We refer to [111, 124, 125] for a deeper discussion on the semantics of the Maude module operations.

8.1 Module importation

Recall that a functional module M specifies a membership equational theory of the form $(\Sigma, E \cup A)$, with Σ its signature, A the equational attributes specified for its operators, and E its set of equations and memberships. A *submodule* M' of M is either a module directly imported by M , or a submodule of one of the modules directly imported by M . Then M' specifies a membership equational subtheory $(\Sigma', E' \cup A') \subseteq (\Sigma, E \cup A)$. Specifically, we have three inclusions: $\Sigma' \subseteq \Sigma$, $E' \subseteq E$, and $A' \subseteq A$. Furthermore, since in Maude subsort-overloaded operators must have the *same* equational attributes, Maude will enforce that the inclusion $A' \subseteq A$ satisfies this property.

In a similar way, a system module Q specifies a rewrite theory $(\Sigma, E \cup A, \phi, R)$. A submodule Q' of Q will likewise specify a rewrite subtheory $(\Sigma', E' \cup A', \phi', R') \subseteq (\Sigma, E \cup A, \phi, R)$. This means that we have inclusions $\Sigma' \subseteq \Sigma$, $E' \subseteq E$, $A' \subseteq A$ (again, with the same equational attributes for subsort-overloaded operators), $\phi' \subseteq \phi$, and $R' \subseteq R$, where $\phi' \subseteq \phi$ is an inclusion of functions and means that the freezing function ϕ *extends* the function ϕ' . Note that Q' could be a functional module, which is then understood as the rewrite theory $(\Sigma', E' \cup A', \phi', \emptyset)$, where ϕ' specifies whatever freezing information has been given to the operators of Σ' in Q' . A system module cannot be imported into a functional module.

In Maude, a module—any module expression giving rise to a module—can be imported as a submodule of another in three different modes: **protecting**, **extending**, or **including**. This is done with the syntax declarations

```
protecting <ModuleExpression> .
extending <ModuleExpression> .
including <ModuleExpression> .
```

which can be abbreviated, respectively, to

```
pr <ModuleExpression> .
ex <ModuleExpression> .
inc <ModuleExpression> .
```

In addition to being allowed as arguments of a **protecting**, **extending**, or **including** importation, module expressions can also appear as the source or target of a view (see Section 8.3.2), or as the parameter of a module, provided the top level is a theory (see Section 8.3.3).

Each of the importation modes places specific semantic constraints on the corresponding inclusion between the theory of the submodule and that of the supermodule. The user must be aware that, as explained later, *the Maude system does not check that these constraints are satisfied*, that is, the different modes of importation can be understood as promises by the user, which would need to be proved by him/herself. Although those importation modes have no effect operationally, they do crucially affect the interpretation given to a module by the theorem proving tools. If a user is doubtful about

the appropriate importation mode the default should be to use the **including** mode, which places weaker requirements on the importation.

Importation statements take a module expression as argument, which may be a module name, the summation of module expressions, the renaming of a module expression, or the instantiation of a parameterized module expression. Modules are constructed for each subexpression of a module expression, and so each submodule signature must be legal. Modules and module expressions are cached both to save time and so that the same module corresponding to a module expression will not be imported twice via a diamond of imports. Mutually or self recursive imports occurring through module expressions are detected and disallowed. Cached modules generated by module expressions that no longer have any users (if the module(s) containing the module expression have been replaced) are deleted. When a module M used in a module expression is modified, any modules generated for module expressions that depend on M are deleted and any modules that depend on M are reevaluated if you attempt to use them. Here the notion of “depends on” is transitive through arbitrary nesting of importation and module expressions.

In addition to being imported by the explicit importation statements we have just introduced, modules can be imported in an implicit way (also in the three possible modes) by means of commands **set protect/extend/include module on/off**; see more details in Section 25.13 and the detailed example in Section 9.1.

8.1.1 Protecting

Importing a module M' into M in **protecting** mode intuitively means that *no junk and no confusion* are added to M' when we include it in M . For example, we may import the module **NAT** of natural numbers into a module **FOO**. “Junk” would be added to **NAT** if in **FOO** we have new ground terms in canonical form in any of the sorts in **NAT**, namely **Nat** and **NzNat**. For example, **FOO** may have declared a constant **infinity** of sort **NzNat** to which no equations apply. “Confusion” would be added if different natural numbers are now identified. For example, if **FOO** contains the equation **s s 0 = 0**, then all even numbers will be identified with 0 and all odd numbers with **s 0**.

Let us explain the semantics of the **protecting** relation in more detail for functional modules M' and M , where M' has been imported as a submodule in **protecting** mode, either by an explicit protecting importation in M , or transitively through one of M 's submodules. Let $(\Sigma', E' \cup A') \subseteq (\Sigma, E \cup A)$ be the theory inclusion defined by the module inclusion $M' \subseteq M$. Notice that the existence of the inclusions $\Sigma' \subseteq \Sigma$, $E' \subseteq E$, and $A' \subseteq A$ means that for each sort s' in Σ' there is a well-defined function

$$q_{s'} : T_{\Sigma'/E' \cup A', s'} \longrightarrow T_{\Sigma/E \cup A, s'}$$

mapping the equivalence class $[t]_{E' \cup A'}$ of a ground term t to the equivalence class $[t]_{E \cup A}$. By definition, the submodule inclusion $M' \subseteq M$ is *protecting* if

and only if for each sort s' in Σ' the above function is *bijective*. This captures mathematically the “no junk” (surjectivity) and “no confusion” (injectivity) ideas. Under our ground Church-Rosser and termination assumptions for M' and M this also means that the canonical form of any ground Σ' -term t in M' that has a sort in Σ' must be the same as its canonical form in M . The requirement that t must have a sort is crucial. We do not require that for k' a kind the map

$$q_{k'} : T_{\Sigma'/E' \cup A', k'} \longrightarrow T_{\Sigma/E \cup A, k'}$$

is bijective. The reason is that the notion of *defined function*—that is, an operator that disappears and leaves just a term with constructors—is only meaningful when the result has a sort. The same operator may not disappear for error terms at the kind level. That is, in the module M extending M' there may easily be *new error terms* of kind k' created by new operators in M . For example, if we import the module **NAT** into the module **RAT** of rational numbers, the sorts **Nat** and **Rat** belong to the same kind, but there are now new error terms in the kind, such as $3 + 7/0$. Therefore, we should not care about “error junk” being added by a supermodule at the kind level, provided that the sorts themselves are protected.

For system modules the **protecting** requirement is interpreted exactly as before as far as their underlying equational theories are concerned. That is, if Q protects Q' and the associated theory inclusion is $(\Sigma', E' \cup A', \phi', R') \subseteq (\Sigma, E \cup A, \phi, R)$, then the equational theory inclusion $(\Sigma', E' \cup A') \subseteq (\Sigma, E \cup A)$ must be *protecting*. We furthermore require that for any two ground Σ' -terms t and t' we can reach t' from t by a sequence of rewrites in the module M' if and only if we can do so in the module M ; that is, for ground terms in M' we require that the *reachability relation* is not altered by the supermodule.

Of course, the **protecting** assertion cannot be checked by Maude at runtime: it requires inductive theorem proving. Using the proof techniques in [31] together with an inductive theorem prover for membership equational logic and a Church-Rosser checker such as those described in [126, 128, 73] (which are available in the Maude formal tool environment together with other useful tools for termination and sufficient completeness; see Section 23.1), this can be done for functional modules. Using the fact that initial models of rewrite theories are also models of equational theories [36], similar proof techniques could be developed to prove the protecting relation between rewrite theories.

8.1.2 Extending

A weaker, yet substantial, requirement about a module importation is expressed by the keyword **extending**. Intuitively, the idea is to allow “junk,” but to rule out confusion. Extending importations may appear naturally in situations in which the data of some sort is extended with new data elements, yet not identifying previously defined data, like adding a new constant **infinity** to the natural numbers in a module importing **NAT**. As another example, when

defining the semantics of a programming language in Maude, we can have from the beginning a sort **Program**, and define incrementally the syntax of programs in several modules, say, **EXPRESSION**, **STATEMENT**, **PROCEDURE**, and so on. This will typically give rise to a family of **extending** module importations as more syntax is added.

For functional modules M' and M , where M' has been imported as a submodule in **extending** mode, either by an explicit extending importation in M , or transitively through one of M 's submodules, if $(\Sigma', E' \cup A') \subseteq (\Sigma, E \cup A)$ is the theory inclusion defined by the module inclusion $M' \subseteq M$, the **extending** requirement means that for each sort s' in Σ' the function

$$q_{s'} : T_{\Sigma'/E' \cup A', s'} \longrightarrow T_{\Sigma/E \cup A, s'}$$

is *injective*. For system modules the **extending** requirement is interpreted exactly as before as far as their underlying equational theories are concerned. That is, if Q extends Q' and the associated theory inclusion is $(\Sigma', E' \cup A', \phi', R') \subseteq (\Sigma, E \cup A, \phi, R)$, then the equational theory inclusion $(\Sigma', E' \cup A') \subseteq (\Sigma, E \cup A)$ must be *extending*. We furthermore require that for any two ground Σ' -terms t and t' we can reach t' from t by a sequence of rewrites in the module M' *if and only if* we can do so in the module M ; that is, for ground terms in M' the *reachability relation* is not altered by the supermodule.

Under the Church-Rosser and termination assumptions, the **extending** $(\Sigma', E' \cup A') \subseteq (\Sigma, E \cup A)$ requirement is a form of *conservative extension* requirement, in the sense that it implies that for any Σ' ground terms t and t' that have a sort in $(\Sigma', E' \cup A')$, $E' \cup A'$ proves $t = t'$ if and only if $E \cup A$ proves $t = t'$. In addition, for system modules it further implies that for any two ground Σ' -terms t and t' the reachability relation is not altered by the extension. In summary, equality and reachability are conservatively preserved for ground terms.

Note that the **extending** relation does *not* destroy protecting importations further down the hierarchy. That is, if M imports M' in **extending** mode, but M' imports M'' in **protecting** mode, then M still imports M'' in **protecting** mode, *not* in **extending** mode. If we do not want M to protect M'' (because this is indeed violated), then we have to say so by explicitly giving an **extending** importation declaration for M'' in M .

8.1.3 Including

The most general form of module importation is provided by the **including** keyword. No requirements are made in an **including** importation about maps of the form $q_{s'}$: there can now be junk (lack of surjectivity) and/or confusion (lack of injectivity). Likewise, for system modules it is not anymore required that the reachability relation between ground terms in the submodule is preserved. The **including** keyword does however impose *some* requirements. First of all, there is the requirement that the equational attributes of subsort-overloaded operators must be the same. Furthermore, the **including** relation

does *not* destroy protecting or extending importations further down the hierarchy. That is, if M imports M' in **including** mode, but M' imports M'' in **protecting** (resp. **extending**) mode, then M still imports M'' in **protecting** (resp. **extending**) mode, *not* in **including** mode. If we do not want M to protect or extend M'' (because this is indeed violated), then we have to say so by explicitly giving an **including** importation declaration for M'' in M .

Given that, as already mentioned, there is no difference at runtime between the different modes of importation because the Maude system does not check the corresponding requirements, from a pragmatic point of view, when a user is doubtful about the appropriate importation mode, the best idea is to use the **including** mode so that at least no false assertion is made.

8.1.4 Default conventions in module importations

We have already explained our default convention when a submodule M_0 is imported indirectly and transitively into M through the direct importation by M of a module M_1 that itself imports M_0 . Then, whatever was the mode (**protecting**, **extending**, or **including**) in which M_0 was imported by M_1 is also, by default, the mode in which M_0 is imported by M , *unless* M contains an explicit declaration importing M_0 in a different mode. We now explain what our default convention is in the case of diamond importations.

We talk of a *diamond importation* of M_0 by M , when M_0 is imported indirectly by M through the direct importation of two or more different modules, say M_1, M_2, \dots, M_k . The problem now is that M_0 can be imported by each of the modules M_1, M_2, \dots, M_k in *different* modes. For example, M_1 could import it in **protecting** mode, M_2 in **extending** mode, M_3 in **including** mode, and so on. What should now be the default convention for the mode in which M imports M_0 ? We adopt a convention that is consistent with a *logical* understanding of such importation declarations. Indeed, such declarations impose semantic constraints of decreasing strength; that is, we have:

$$\text{protecting } M_0 \Rightarrow \text{extending } M_0 \Rightarrow \text{including } M_0.$$

The default convention consistent with this logical reading is that *the strongest mode wins*, i.e., **protecting** prevails over **extending**, which itself prevails over **including**. This is because we view the set of all such importing mode declarations as a *conjunction*, and exploit the logical equivalence between $A \Rightarrow B$ and $(A \wedge B) \Leftrightarrow A$.

Note that this “strongest wins” default mode may not always be the correct or intended mode in which M *should* import M_0 . Sometimes it may not be, and then the user should *override* the default convention by declaring explicitly a different mode in which M imports M_0 . A pragmatic reason why this need for overruling the default mode may arise is that, although a weaker mode of importation (say **extending**) does not logically preclude such an importation also satisfying a stronger one (say **protecting**), in practice, when we declare an importation in a weaker mode it can often be because we know or suspect

that it fails to satisfy a stronger mode. For example, when we say “**extending**” we may often mean “**extending and *not* protecting.**”

8.1.5 Some module hierarchy examples

Prime numbers sieve

Section 4.4.7 included a functional module specifying the sieve of Eratosthenes to calculate prime numbers.

```
fmod SIEVE is
  protecting NAT .
  sort NatList .
  subsort Nat < NatList .
  ...
endfm
```

The predefined module of natural numbers (see Section 9.2) is imported in **protecting** mode. This is justified because the elements of sort **Nat** are used to generate the lists of natural numbers by means of a subsort declaration and also as arguments of other operators. However, no new operator of result sort **Nat** is added in the **SIEVE** module, and all the equations in this module identify elements of sort **NatList** without identifying different natural numbers.

Vending machine

The vending machine example in Section 6.1 was presented in a modular way, by separating the underlying signature defining the states of the machine from the rules defining the corresponding transitions.

```
mod VENDING-MACHINE is
  including VENDING-MACHINE-SIGNATURE .
  var M : Marking .
  r1 [add-q] : M => M q .
  ...
endm
```

It is important to notice that in this example the importation mode cannot be either **protecting** or **extending**, because those modes require preservation of the reachability relation, which clearly is not the case when adding (non-identity) rewrite rules to a functional module (where the reachability relation is the identity).

Bank accounts and object configurations

Later, in Section 11.1, devoted to the definition of configurations of objects and messages for object-based programming, we will present several modules where additional data are introduced in order to run some tests. For example, the following module introduces three new constants to be used as object identifiers, and a new constant to be used as a test configuration.

This configuration constant is identified with a term of sort `Configuration` in the imported module `BANK-ACCOUNT` by means of an equation whose right-hand side is omitted below. However, constants `A-001`, `A-002`, and `A-003` are new data elements, i.e., junk, of sort `Oid`. The sort `Oid` was declared in the module `CONFIGURATION`, but since it was imported in `including` mode in `BANK-ACCOUNT`, it is not necessary to import it in a different mode. Therefore, the appropriate importation mode is `extending`.

```
mod BANK-ACCOUNT-TEST is
  ex BANK-ACCOUNT .
  ops A-001 A-002 A-003 : -> Oid .
  op bankConf : -> Configuration .
  eq bankConf = ... .
endm
```

The following example, from Section 11.3, is more interesting, in that it introduces new sorts `MsgBody` and `Request`, not just new constants for a sort in the imported module. Still, the appropriate importation mode is `extending` because there are no new rewrite rules and no equations, and thus no confusion between elements in imported sorts is introduced.

```
mod DATA-AGENTS-CONF is
  ex CONFIGURATION .
  sort MsgBody .
  op msg : Oid Oid MsgBody -> Msg [ctor message] .
  sort Request .
  op w4 : Oid Oid MsgBody -> Request [ctor] .
endm
```

There are several other modules in Chapter 11 illustrating the use of the `extending` mode in importing modules, like `BANK-MANAGER-TEST`, `TICKER-TEST`, `TICKER-FACTORY-TEST`, and `AGENT-TEST`; see Figures 11.1, 11.2, and 11.3.

Number hierarchy

The acyclic importation graph corresponding to the module hierarchy representing the number hierarchy from the natural to the rational numbers, as presented in Section 4.10, is shown in Figure 8.1, where a module above imports the modules below it, with the predefined `BOOL` module at the bottom. Note that in this example all the importations are `protecting` importations.

Hierarchy of predefined modules

A more complex acyclic importation graph corresponds to the hierarchy of predefined modules for basic data types, described later in Chapter 9 and shown in Figure 9.1, where all the importations are in `protecting` mode.

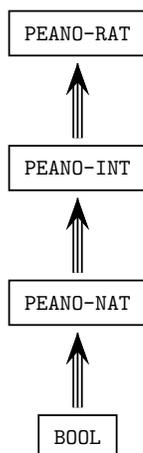


Fig. 8.1. Importation (protecting) graph of number hierarchy modules

8.2 Module summation and renaming

8.2.1 The summation module expression

The summation module operation creates a new module that includes all the information in its summands. The syntax for a summation of module expressions is

ModuleExpression + *ModuleExpression*

with + associative and commutative.

Summation expressions are flattened before being evaluated, so that $A + (B + C)$ and $(C + A) + B$ both create a single new module $A + B + C$. The evaluation of a summation module expression results in the creation of a new module, with such a module expression as its name, which imports the module expressions being combined. The new module will be generated having one type or another, depending on the types of the arguments of the summation module expression. A summation is a functional module if all the summands are functional modules and a system module otherwise.

Although the use of the summation module expression is more interesting in combination with other module expressions, let us consider as an example the following module, in which the union of the predefined `FLOAT` and `STRING` modules (see Chapter 9) are imported together in `protecting` mode to illustrate its use.

```

fmod FLOAT-STRING is
  protecting FLOAT + STRING .
  ...
endfm

```

Notice that a declaration

```
protecting A + B .
```

is *not* equivalent to a sequence of declarations

```
protecting A .
protecting B .
```

because in general the modules A and B may share sorts and operators. The same happens with `extending` declarations, for the same reason. However, a declaration of the form

```
including A + B .
```

is indeed equivalent to a sequence of declarations

```
including A .
including B .
```

8.2.2 Module renaming

The syntax of a renaming module expression is

```
 $\langle ModuleExpression \rangle * ( \langle Renaming \rangle )$ 
```

where $\langle Renaming \rangle$ is a comma-separated sequence of renaming items of the forms:

```
sort  $\langle identifier \rangle$  to  $\langle identifier \rangle$ 
op  $\langle identifier \rangle$  to  $\langle identifier \rangle$ 
op  $\langle identifier \rangle$  to  $\langle identifier \rangle$  [  $\langle attribute-set \rangle$  ]
op  $\langle identifier \rangle$  :  $\langle type-list \rangle$  ->  $\langle type \rangle$  to  $\langle identifier \rangle$ 
op  $\langle identifier \rangle$  :  $\langle type-list \rangle$  ->  $\langle type \rangle$  to  $\langle identifier \rangle$ 
  [  $\langle attribute-set \rangle$  ]
label  $\langle identifier \rangle$  to  $\langle identifier \rangle$ 
```

Renaming $(_*(_))$ binds tighter than summation $(_+_)$, and it groups to the left. Note that, in addition to the typical renamings of sorts and operators, renaming of labels is also supported (which may be useful for metalevel applications). Note also how the renaming of operators allows changing the attributes of the operator being renamed. The only attributes currently allowed in operator maps are `prec`, `gather`, and `format`. The idea is that when you rename an operator, the old syntactic properties may no longer be legal and are reset to defaults, unless you explicitly set them with these attributes; for example, when a change in the syntax of the operator could cause a parsing different from the intended one. Let us see an example in which modifying the grammatical attributes of an operator is useful. Consider the following module defining lists of natural numbers with a `max` operator returning the greatest of the elements in a list of natural numbers.

```
fmod NAT-LIST-MAX is
  pr NAT .
  sort NeNatList .
  subsort Nat < NeNatList .
  op __ : NeNatList NeNatList -> NeNatList [ctor assoc] .
  op max : NeNatList -> Nat .
  var N : Nat .
  var NL : NeNatList .
  eq max(N) = N .
  eq max(N NL) = if N > max(NL) then N else max(NL) fi .
endfm
```

We may obtain the maximum of a list of natural numbers as follows.

```
Maude> red max(4 2 5 3) .
result NzNat: 5
```

Suppose now that we want to change the syntax of the function `max` in the `NAT-LIST-MAX` module above to `maximum_`.

```
fmod NAIVE-NAT-LIST-MIXFIX-MAX is
  pr NAT-LIST-MAX * (op max : NeNatList -> Nat to maximum_) .
endfm
```

We can do the following reduction:

```
Maude> red maximum 2 3 4 1 .
result NeNatList: 2 3 4 1
```

This result may seem strange, but it makes perfect sense. What has happened is that it has been parsed as `(maximum 2) 3 4 1`, the only possible parse given the default precedence values and gathering patterns assigned. Since by default `maximum_` has precedence 15 and gathering E, it cannot take the list `2 3 4 1` as argument because `__` has precedence 41. However, since `__` has gathering e E, `maximum 2` is a valid argument for it (see Section 3.9 for a detailed discussion on the use of precedence values and gathering patterns and their default values). We can of course obtain the intended result by placing parentheses around the set of numbers,

```
Maude> red maximum (2 3 4 1) .
result NzNat: 4
```

but it is more convenient to change the precedence values of the operators. We can, for example, raise the precedence of `maximum_`.

```
fmod NAT-LIST-MIXFIX-MAX is
  pr NAT-LIST-MAX
    * (op max : NeNatList -> Nat to maximum_ [prec 41]) .
endfm
```

having then the following reduction.

```
Maude> red maximum 2 3 4 1 .
result NzNat: 4
```

Notice that if `maximum_` has precedence 41, then `(maximum 2) 3 4 1` is no longer a valid parse.

A renaming can be considered as a function that, given a module M and a list of mappings S , returns a copy of the module, with such a module expression as its name, in which the names of sorts, operators, etc. are changed as indicated by the mappings. However, renaming a module that has imports is a subtle issue. Given a structured specification, the renaming not only causes the creation of a copy of the top module in the structure, but renames also the part of the submodule structure that is affected by the renaming. For any other submodule M' in the structure which is affected by the mappings, a renamed copy of it is generated with name $M' * (S')$, where S' is the subset of mappings in S that affect M' .

A module expression $A * (R)$ evaluates to A if A has no content that is affected by the renaming R . Otherwise $A * (R)$ evaluates to a new module $A * (R')$ where R' is obtained by deleting those renaming items that do not affect A , and canonizing the types in operator renamings with respect to A (see below). If A imports modules B and C , $A * (R')$ will import modules obtained by evaluating $B * (R')$ and $C * (R')$.

There are some subtle cases. Consider for example the following three modules:

```
fmod RENAMING-EX-A is
  sort Foo .
  op a : -> Foo .
  op f : Foo -> Foo .
endfm

fmod RENAMING-EX-B is
  including RENAMING-EX-A .
  sort Bar .
  subsort Foo < Bar .
  op f : Bar -> Bar .
endfm

fmod RENAMING-EX-C is
  inc RENAMING-EX-B * (op f : Bar -> Bar to g) .
endfm
```

Here, the operator `f` in the module `RENAMING-EX-A` looks as though it is not affected by the renaming in the module `RENAMING-EX-C`, but because of the subsort declaration `Foo < Bar` in `RENAMING-EX-B`, it should be renamed for consistency. This is internally handled by the Maude system by canonizing the type `Bar` occurring in the renaming

```
op f : Bar -> Bar to g
```

to the kind expression `[Foo,Bar]`, which includes *all* of the sorts in the kind `[Bar]` occurring in `RENAMING-EX-B`. Thus, the module expression

```
RENAMING-EX-B * (op f : Bar -> Bar to g)
```

evaluates to a new module

```
RENAMING-EX-B * (op f : [Foo,Bar] -> [Foo,Bar] to g)
```

which includes the module expression

```
RENAMING-EX-A * (op f : [Foo,Bar] -> [Foo,Bar] to g)
```

which evaluates to a new module

```
RENAMING-EX-A * (op f : [Foo] -> [Foo] to g)
```

in which `f` has been renamed.

In general, `*` does not distribute over `+`. Consider this other example:

```
fmod RENAMING-EX-D is
  sorts Foo Bar .
endfm
```

```
fmod RENAMING-EX-E is
  inc RENAMING-EX-D .
  op f : Foo -> Foo .
endfm
```

```
fmod RENAMING-EX-F is
  inc RENAMING-EX-D .
  subsort Foo < Bar .
  op f : Bar -> Bar .
endfm
```

It is *not* the case that the module expressions

```
(RENAMING-EX-E + RENAMING-EX-F) * (op f : Bar -> Bar to g)
```

and

```
(RENAMING-EX-E * (op f : Bar -> Bar to g))
+ (RENAMING-EX-F * (op f : Bar -> Bar to g))
```

evaluate to the same module, because in the latter the operator `f` occurring in `RENAMING-EX-E` will not be renamed, since `f : Bar -> Bar` does not occur in `RENAMING-EX-E`.

Operators with the `poly` attribute are only affected by operator renamings that do not specify types. Renaming a module does not change its module type, that is, renamed functional modules (resp. system modules) remain functional (resp. system).

8.3 Parameterized programming

Theories, parameterized modules, and views are the basic building blocks of parameterized programming [42, 174]. As in OBJ, a *theory* defines the interface of a parameterized module, that is, the structure and properties required of an actual parameter.

A *parameterized module* is a module with one or more *parameters*, each of which is expressed by means of one theory, that is, modules can be parameterized by one or more theories. If we want, e.g., to define a list or a set of elements, we may define a module LIST or SET parameterized by a theory expressing the requirements on the type of the elements to store in such data structures. Thus, theories are used to declare the interface requirements for parameterized modules. In the case of lists and sets we do not need any requirement on the data elements, and therefore we may use the trivial theory TRIV, with just a sort `ElT`, as parameter of such modules; but in other cases, say search trees or sorted lists, we may require, e.g., a particular operator, an order relation, or an equivalence relation, in which cases we shall need to use the appropriate theories describing the specific requirements.

The instantiation of the formal parameters of a parameterized module with actual parameter modules or theories requires a *view* mapping entities from the formal interface theory to the corresponding entities in the actual parameter module.

8.3.1 Theories

Theories are used to declare module interfaces, namely the syntactic and semantic properties to be satisfied by the actual parameter modules used in an instantiation. As for modules, Maude supports two different types of theories: *functional theories* and *system theories*, with the same structure of their module counterparts, but with a different semantics. Functional theories are declared with the keywords `fth ... endfth`, and system theories with the keywords `th ... endth`. Both of them can have sorts, subsort relationships, operators, variables, membership axioms, and equations, and can import other theories or modules. System theories can also have rules. Although there is no restriction on the operator attributes that can be used in a theory, there are some subtle restrictions and issues regarding the mapping of such operators (see Section 8.3.2).

Like functional modules, *functional theories* are membership equational logic theories, but they do not need to be Church-Rosser and terminating, and therefore some or all of their statements may be declared with the `nonexec` attribute. Theories have a *loose* semantics, in the sense that any algebra satisfying the equations and membership axioms in the theory is an acceptable model. However, functional theories *can be executed in exactly the same way as functional modules*; that is, the equations and membership axioms not having the `nonexec` attribute should be Church-Rosser and terminating, and can

be executed by equational simplification, whereas the statements declared as `nonexec` can be arbitrary and can only be executed in a controlled way at the metalevel. System theories have a similar loose interpretation, but are treated just as system modules for executability purposes. Theories are then allowed to contain rules and equations which, if declared with the `nonexec` attribute, can be arbitrary, that is, can have variables in their righthand sides or conditions that may not appear in their corresponding lefthand sides and do not obey the admissibility conditions in Sections 4.6 and 6.3. Similarly, conditional membership axioms may have variables in their conditions that do not appear in their head membership assertions. Also, the lefthand side may be a single variable.

Let us begin by introducing the functional theory `TRIV`, which requires just a sort.

```
fth TRIV is
  sort Elt .
endfth
```

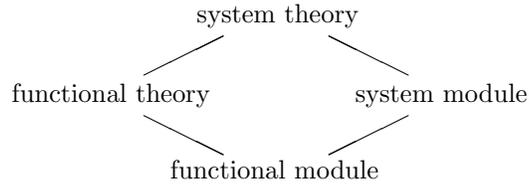
The theory `TRIV` is used very often, for instance in the definition of data structures, such as lists, sets, trees, etc., of elements of some type with no specific requirement; in these cases, it is common to define a module, say `LIST`, `SET`, `TREE`, etc., parameterized by the `TRIV` theory (see Section 8.3.3). The theory `TRIV` is predefined in Maude, together with several useful views from `TRIV` to other predefined modules and theories (see Section 9.11.1).

But we can define more interesting theories. For example, the theory of monoids, with an associative binary operator with identity element `1`, can be specified as follows:

```
fth MONOID is
  including TRIV .
  op 1 : -> Elt .
  op __ : Elt Elt -> Elt [assoc id: 1] .
endfth
```

Notice the importation of the theory `TRIV` into the `MONOID` theory. As for modules, it is possible to structure our theories by importing other theories and modules (and in general module expressions involving theories and modules) into theories. However, a theory cannot be imported into a module: theories can only be used as *parameters* of modules. Also, theories do not have automatic importation as modules do (e.g., `BOOL`, as described in Section 9.1).

Modules and theories can be combined in module expressions (they can be summed, for example), and modules can be imported into theories. Basically, we have a lattice



where summation corresponds to join, and where a module or theory may only import a submodule or subtheory of lesser or equal type.

Although the importation of a module into a theory can be done in any mode, a theory can only be imported in **including** mode into another theory. The **including** importation of a theory into another theory keeps its loose semantics. However, the importation of a theory into another one in **protecting** or **extending** mode would imply additional semantic requirements; such modes of importation are ruled out.¹ On the other hand, although a module keeps its initial interpretation when imported into a theory in **protecting** or **extending** modes, it loses it if imported in **including** mode.

Let us see a few examples illustrating all this.

The theory of commutative monoids can be defined just as the theory of monoids; the binary operator `_+_` is now declared associative, commutative, and has 0 as its identity element.

```

fth +MONOID is
  including TRIV .
  op 0 : -> Elt .
  op _+_ : Elt Elt -> Elt [assoc comm id: 0] .
endfth
  
```

The theory of semirings can be expressed as follows.

```

fth SEMIRING is
  including MONOID .
  including +MONOID .
  vars X Y Z : Elt .
  eq X (Y + Z) = (X Y) + (X Z) [nonexec] .
  eq (X + Y) Z = (X Z) + (Y Z) [nonexec] .
endfth
  
```

Note the use of the **nonexec** attribute, and note also that given the semantics of theory inclusions, there is no difference between having a structured theory or one flat theory including all the declarations.² For example, the theory of commutative rings can be defined directly as follows:

¹ If a theory is imported using a mode other than **including**, the system gives an error message saying that the mode is being treated as if it were **including**. Other illegal importations give an error message saying that they are being ignored.

² The only exception to this semantic equivalence between structured theories and their flattened form is the case in which a theory imports some modules, since any

```
fth RING is
  sort Ring .
  ops z e : -> Ring .
  op _+_ : Ring Ring -> Ring [assoc comm id: z] .
  op *__ : Ring Ring -> Ring [assoc comm id: e] .
  op -_ : Ring -> Ring .
  vars A B C : Ring .
  eq A + (- A) = z [nonexec] .
  eq A * (B + C) = (A * B) + (A * C) [nonexec] .
endfth
```

but could also be defined as a structured theory including the theories of commutative groups and commutative monoids (renamed if necessary), to which the distributivity axiom is added.

As mentioned above, the **including** importation of a theory into another theory keeps its loose semantics. However, if the imported theory contains a module, which therefore must be interpreted with an initial semantics (see Section 6.3), then that initial semantics is maintained by the importation. For example, in the definition of the **TAOSET** theory below, the declaration **protecting** **BOOL** ensures that the initial semantics of the functional module for the Booleans is preserved, which is in fact a crucial requirement.

Let us consider now a hierarchy of theories for partially and totally ordered sets. The most basic theory specifies a transitive and antisymmetric order **_<_** on a set:

```
fth TAOSET is
  protecting BOOL .
  sort Elt .
  op _<_ : Elt Elt -> Bool .
  vars X Y Z : Elt .
  ceq X < Z = true if X < Y /\ Y < Z [nonexec label transitive] .
  ceq X = Y if X < Y /\ Y < X [nonexec label antisymmetric] .
endfth
```

By adding irreflexivity to **TAOSET** we get a theory specifying a strict partial order:

```
fth SPOSET is
  including TAOSET .
  var X : Elt .
  eq X < X = false [nonexec label irreflexive] .
endfth
```

Notice that in this case antisymmetry is implied by irreflexivity and transitivity. Of course, there are different ways of presenting a theory, and in

of the **protecting** or **extending** initiality requirements of the imported module and its submodules *must be preserved*. Those requirements would be lost if the whole structure were to be flattened.

particular one can always write the corresponding flat theory with only the axioms for irreflexivity and transitivity. In the presentation above, the initial semantics of `BOOL` when it is imported in `protecting` mode in `TAOSET` is preserved when the latter is included in `SPOSET`. The same will hold in further importations in this hierarchy of order theories.

On the other hand, by adding reflexivity to `TAOSET` we get a theory specifying a non-strict partial order. Notice the renaming in the importation, so that the name of the order `_<=_` reflects its reflexivity.

```
fth NSPOSET is
  including TAOSET * (op _<_ to _<=_) .
  var X : Elt .
  eq X <= X = true [nonexec label reflexive] .
endfth
```

Having both `_<_` and `_<=_` available together is useful in some applications. There are standard ways of associating a strict partial order with a non-strict partial order and the other way around:

- from $a < b$, one can define $a \leq b$ as equivalent to $a < b$ or $a = b$; and
- from $a \leq b$, one can define $a < b$ as equivalent to $a \leq b$ and $a \neq b$.

These equivalences can be expressed as Maude theories as follows, where we use the same name for both theories because they are equivalent, that is, we have two different presentations of the same theory and in what follows we will not care about which version of `POSET` is used.

```
fth POSET is
  including SPOSET .
  op _<=_ : Elt Elt -> Bool .
  vars X Y : Elt .
  eq X <= X = true [nonexec] .
  ceq X <= Y = true if X < Y [nonexec] .
  ceq X = Y if X <= Y /\ X < Y = false [nonexec] .
endfth

fth POSET is
  including NSPOSET .
  op _<_ : Elt Elt -> Bool .
  vars X Y : Elt .
  eq X < X = false [nonexec] .
  ceq X <= Y = true if X < Y [nonexec] .
  ceq X = Y if X <= Y /\ X < Y = false [nonexec] .
endfth
```

Notice that the axioms are almost the same in both presentations of `POSET`, but, while the first presentation defines the reflexive order `_<=_` in terms of the irreflexive one `_<_`, the second presentation defines the irreflexive order `_<_` in terms of the reflexive one `_<=_`.

To each of the previous theories we can add an axiom requiring the order to be total (or linear), that is, two different elements have to be related one way or the other. In this way, we have the following theories for a strict total order, a non-strict total order, and a total order with both operations.

```
fth STOSET is
  including SPOSET .
  vars X Y : Elt .
  ceq X = Y if X < Y = false /\ Y < X = false [nonexec label total] .
endfth
```

```
fth NSTOSET is
  including NSPOSET .
  vars X Y : Elt .
  ceq X <= Y = true if Y <= X = false [nonexec label total] .
endfth
```

```
fth TOSET is
  including POSET .
  vars X Y : Elt .
  ceq X <= Y = true if Y <= X = false [nonexec label total] .
endfth
```

As already mentioned above, the requirement ensuring the initial semantics of `BOOL` when it is protected in `TAOSET` is then preserved by the remaining theories when `TAOSET` is included in them via a chain of `including` importations. In fact, we are dealing with structures in which part of them, not only the top theory, has a loose semantics, while other parts contain modules with an initial semantics.

This hierarchy of order theories is displayed in Figure 8.2, where we represent by boxes the modules (with initiality constraints), by ovals the theories (with loose semantics), by triple arrows the `protecting` importations, and by single arrows the `including` importations.

Finally, as an example of a system theory, let us consider the theory `CHOICE` of bags of elements with a choice operator defined on the bags by a rewrite rule that nondeterministically picks up one of the elements in the bag. We can specify this theory as follows, where we have a sort `Bag` declared as a supersort of the sort `Elt`.

```
th CHOICE is
  sorts Bag Elt .
  subsort Elt < Bag .
  op empty : -> Bag .
  op __ : Bag Bag -> Bag [assoc comm id: empty] .
  op choice : Bag -> Elt .
  var E : Elt .
  var B : Bag .
```

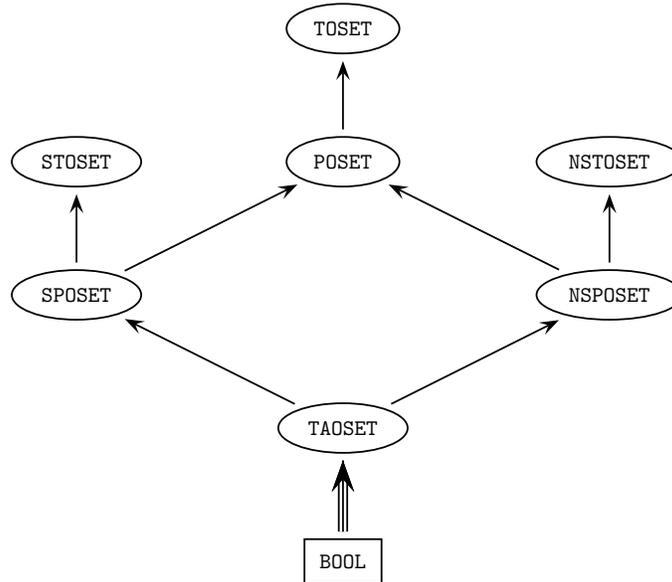


Fig. 8.2. Hierarchy of order theories

```

r1 [choice] : choice(E B) => E .
endth

```

8.3.2 Views

We use *views* to specify how a particular target module or theory is claimed to satisfy a source theory. In general, there may be several ways in which such requirements might be satisfied, if at all, by the target module or theory; that is, there can be many different views, each specifying a particular *interpretation* of the source theory in the target. Each view declaration has an associated set of *proof obligations*, namely, for each axiom in the source theory it should be the case that the axiom's translation by the view holds true in the target. Since the target can be a module interpreted initially, verifying such proof obligations may in general require inductive proof techniques of the style supported for Maude's logic in [73], and for which tools in the Maude formal environment can be used (see Section 23.1). *Such proof obligations are not discharged or checked by the system.*

In the definition of a view we have to indicate its name (which has to be a single identifier, as defined in Section 3.1), the source theory, the target module or theory, and the mapping of each sort and operator in the source theory. The name space of views is separate from the name space of modules and theories, which means that, e.g., a view and a module could have the

same name. In fact, we shall see below how we recommend naming inclusion views as the target theory. The source and target of a view can be any module expression, with the source module expression evaluating to a theory and the target module expression evaluating to a module or a theory.

The syntax for views is as follows:

```
view <ViewName> from <Source> to <Target> is
  <Mappings>
endv
```

The mapping of a sort in the source theory to a sort in the target module or theory is expressed with syntax

```
sort <identifier> to <identifier> .
```

For each sort S in the source theory, there must exist a sort S' in the target module or theory which is its mapping under the view; unmentioned sorts get the identity mapping. Furthermore, if sorts S and T in the source theory are in the same kind, then their mappings S' and T' under the view must be in the same kind in the target module or theory. Finally, if S is a subsort of T , then it must be true that S' is a subsort of T' .

The mapping of operators is expressed with syntax

```
op <identifier> to <identifier> .
op <identifier> : <type-list> -> <type> to <identifier> .
op <op-expr> to term <term> .
```

In the first case, where only an operator identifier is given, the map affects all operators with the same name. Existence of appropriate operators in the target is checked for. In the second case, when explicit arity and coarity are given, the operator map affects not only the operators with such arity and coarity, but also the entire family of subsort-overloaded operators (see Section 3.6) associated with the given operator. The third case is similar to the second one, but instead of mapping the operator to another operator, it is mapped to a given term with variables; *op-expr* is a term consisting of a single operator applied to variables—declared either on-the-fly or with variable declarations in the same view—and the target term is any term with variables, those in the source *op-expr* in the corresponding sorts resulting from the mapping. See below for more details and examples.

Maps must preserve the arities and the types of operators, and sort maps and operator maps must be compatible. For each operator $f : S_1 \dots S_n \rightarrow T$ in the source theory there must exist an operator $f' : S'_1 \dots S'_n \rightarrow T'$ in the target module or theory, where S'_i is the mapping of sort S_i under such a view.

Unmentioned operators also get the identity mapping. Thus, “obvious” parts of a mapping do not need to be explicitly given, namely, any identical

mapping of a sort or operator such that its arity and coarity are mapped to those of an operator with the same name in the target can be omitted.³

As a first example, the following view `StringAsToset` defines a view from the theory `TOSET`, presented in Section 8.3.1, to the predefined functional module `STRING`, described in Section 9.8.

```
view StringAsToset from TOSET to STRING is
  sort Elt to String .
endv
```

Notice that the identity maps `op _<_ to _<_` and `op _<=_ to _<=_` have been omitted.

The maps sending operators to derived operators, that is, terms with variables, allow us to map an operator, not only to another operator, but also to an expression. The view `RingToRat` below is a view from the theory `RING`, presented in Section 8.3.1, to the predefined functional module `RAT`, described in Section 9.6.

```
view RingToRat from RING to RAT is
  sort Ring to Rat .
  op e to term 1 .
  op z to 0 .
endv
```

Notice that we have followed the convention of omitting the “obvious” parts of the map concerning the operators `_+_` and `_*_`. Furthermore, we have used an operator map sending the operator `e` to the term `1`, due to the fact that in `RAT` `1` is not a constant, but the term `s_1(0)` (see Sections 4.4.2, 9.2, and 9.6 for details). Note that the map `op e to term 1` cannot be expressed with the other forms of operator maps, because `1` is not an operator, but just syntactic sugar for the term `s_1(0)`.

As another example, consider the case in which we want to define a view from the theory `NSPOSET` in which we have a sort `Elt` and a non-strict “less or equal” operator `_<=_ : Elt Elt -> Bool`, to a module defining the integers with no such operator but instead with a strict operator “less than” `_<_ : Int Int -> Bool`. Then, we can define a view with maps

```
sort Elt to Int .
op X:Elt <= Y:Elt to term X:Int < Y:Int or X:Int == Y:Int .
```

where we have also used the predefined equality operator `_==_`. The lefthand side of the operator mapping, `X:Elt <= Y:Elt` in this case, which consists of an operator with only variable arguments, must parse to a unique term in the source theory. Each of the variables used in the maps must have a unique base name (e.g., using both `X:Foo` and `X:Bar` in the same argument list is disallowed).

³ In Full Maude (see Chapter 19), maps for all sorts in the source theory have to be explicitly given, even when they are identity maps.

Also, the righthand side, $X:\text{Int} < Y:\text{Int}$ or $X:\text{Int} == Y:\text{Int}$ in this case, must parse to a unique term in the target module or theory. The only variables that may occur in the target term are those appearing in the source term; however, they may occur multiple times or not at all. If the source term parses to a sort S or kind $[S]$, then the target term must parse to sort T or kind $[T]$ such that T and the mapping of S under the view S' belong to the same kind.

Views may also contain variable declarations. The syntax is identical to that in modules and theories. However, its semantics is subtly different. Instead of declaring a single variable, a declaration

```
var X : S .
```

now declares two aliases with the same name; in the lefthand side of an operator mapping, X is an alias for $X:S$ while in the righthand side of an operator mapping, X is an alias for $X:S'$, with S' the mapping of S under the view.

For example, we can define a view from the theory `SPOSET` with a strict order operation `_<_` to the predefined functional module `INT` of integers (see Section 9.4) in such a way that the `_<_` order relation of a poset is mapped to an expression using the “less than or equal” operator `_<=_` on sort `Int` and the predefined inequality operator `_=/=_` in `BOOL` (see Sections 9.4 and 9.1) as follows:

```
view SPosetToInt from SPOSET to INT is
  sort Elt to Int .
  vars X Y : Elt .
  op X < Y to term X <= Y and X /= Y .
endv
```

Alternatively, we can specify this view without a variable declaration as

```
view SPosetToInt from SPOSET to INT is
  sort Elt to Int .
  op X:Elt < Y:Elt to term X:Int <= Y:Int and X:Int /= Y:Int .
endv
```

Note that this view imposes several proof obligations to be checked by the user. In particular, the translations by the view of the axioms in `SPOSET` should hold in the target. Given variables X , Y , and Z of sort `Int`, the following axioms should be true in `INT`:

```
eq X <= X and X /= X = false .
ceq X <= Z and X /= Z = true
  if X <= Y and X /= Y /\ Y <= Z and Y /= Z .
ceq X = Y if X <= Y and X /= Y /\ Y <= X and Y /= X .
```

Of course, since the predefined `INT` module indeed includes both operators `_<_` and `_<=_`, it is not necessary to use the feature described in the previous example. We can instead have simpler view declarations such as the following:

```
view IntAsStoset from STOSSET to INT is
  sort Elt to Int .
endv
```

```
view IntAsToset from TOSET to INT is
  sort Elt to Int .
endv
```

where the identity maps `op _<_ to _<_` and `op _<=_ to _<=_` have been omitted.

We recommend following the convention of naming views from TRIV by the name of the sort to which `Elt` is mapped, when the name of this sort is not structured.⁴ Thus, a view from the theory TRIV to the module INT that sends the sort `Elt` to `Int` should be named `Int` (as we shall see in Section 9.11.1, the view `Int` is predefined in Maude).

```
view Int from TRIV to INT is
  sort Elt to Int .
endv
```

This convention can add understandability to the specifications. As we will see in Section 8.3.4, given a module LIST of lists parameterized by TRIV with a sort `List{X}`, once it is instantiated, e.g., with the view `Int` above, the sort `List{X}` becomes `List{Int}`, defining lists of integers. Using names of views as labels in interfaces of parameterized modules (see Section 8.3.4 below) should be avoided, since this can sometimes generate ambiguities.

We can also have views between theories, which is particularly useful to compose instantiations of views to link the formal parameter of some parameterized module to some actual parameter via some intermediate formal parameter of another parameterized module. We will discuss the uses of these views and give some examples in the coming sections. An example of a view whose target is a theory is the following:

```
view PosetToToset from POSET to TOSET is
  sort Elt to Elt .
endv
```

As said above, identity maps can be omitted. Thus, the following definition is equivalent to the previous one.

```
view PosetToToset from POSET to TOSET is
endv
```

⁴ Notice that a structured sort name, such as `List{Nat}` for example, cannot be used as a view name, because it is not a single identifier; if desired, the user can write the single-identifier form `List' {Nat'}` as view name. The convention is totally general in Full Maude; see Section 19.3.2.

In this example the `PosetToToset` view represents the inclusion of the `POSET` theory into `TOSET`.

In those cases in which a view defines a theory inclusion from `TRIV` into another theory, we recommend following the convention of naming the view with the name of the target theory. An example that will be very useful later is the inclusion of `TRIV` into `TOSET`, which is expressed as a view as follows:

```
view TOSET from TRIV to TOSET is
endv
```

Let us finish this section by commenting on some subtle issues that can arise with operator mappings:

- Operator mappings are not applied to operators that have at least one declaration in a module (as opposed to a theory); if a mapping applies to such an operator, an advisory is generated. Although it does not seem to be useful, Maude does not forbid having subsort-overloaded operators appearing in a theory and in one of its submodules. However, the operator is considered to “belong” to the module, and therefore it cannot be mapped by a view.
- *assoc operators*. Nested occurrences of associative operators may have been flattened, or have been entered in a flattened way such as, for example, `f(a, a, b, b)`. In order to map this to an operator that has different attributes (perhaps including `assoc`) or to a term, flattened occurrences will be temporarily unflattened into a regular term before translation. The precise choice of unflattening is left unspecified.
- *iter operators*. Mapping an `iter` operator (see Section 4.4.2) to a non-`iter` operator causes the efficient representation of towers of symbols to be expanded out, with a potential exponential blow up. Mapping an `iter` operator to a term in which the single argument variable occurs more than once causes a doubly exponential blow up. Maude operates under the principle of “you asked for it, you got it” and if the expansion is too large it will die with a virtual memory exhausted error.
- *Built-in operators*. The built-in operators for holding non-algebraically defined data `StringSymbol`, `FloatSymbol`, and `QuotedIdentifierSymbol` have a special internal representation for their terms, and can only be mapped to operators of identical type.
- *Polymorphic operators*. Polymorphic operators must map to polymorphic operators that are polymorphic on the same arguments. Only generic mappings of the form `f to f'` are considered when mapping polymorphic operators.

8.3.3 Parameterized modules

System modules and functional modules can be parameterized. A parameterized system module has syntax

```
mod M{X1 :: T1 , ... , Xn :: Tn} is ... endm
```

with $n \geq 1$. Parameterized functional modules have completely analogous syntax.

The $\{X_1 :: T_1 , \dots , X_n :: T_n\}$ part is called the *interface*, where each pair $X_i :: T_i$ is a parameter, and each X_i is an identifier—the *parameter name* or *parameter label*—and each T_i is an expression that yields a theory—the *parameter theory*. Each parameter name in an interface must be unique, although there is no uniqueness restriction on the parameter theories of a module—we can have, e.g., two TRIV parameters. The parameter theories of a functional module must be functional theories.

In a parameterized module M , all the sorts and statement labels coming from theories in its interface must be qualified by their names. Thus, given a parameter $X_i :: T_i$, each sort S in T_i must be qualified as $X_i\$S$, and each label l of a statement occurring in T_i must be qualified as $X_i\$l$. In fact, the parameterized module M is flattened as follows. For each parameter $X_i :: T_i$, a renamed copy of the theory T_i , called $X_i :: T_i$ is included. The renaming maps each sort S to $X_i\$S$, and each label l of a statement occurring in T_i to $X_i\$l$. The renaming percolates down through nested inclusions of theories, but has no effect on importations of modules. Thus, if T_i includes a theory T' , when the renamed theory $X_i :: T_i$ is created and included into M , and the renamed theory $X_i :: T'$ will also be created and included into $X_i :: T_i$.⁵ However, the renaming will have no effect on modules imported by either the T_i or T' ; for example, if BOOL is imported by one of these theories, it is not renamed, but imported in the same way into M .

For example, a parameterized module PRELIM-SET with TRIV as interface can be defined as follows:

```
fmod PRELIM-SET{X :: TRIV} is
  protecting BOOL .
  sorts Set NeSet .
  subsorts X$Elt < NeSet < Set .
  op empty : -> Set .
  op _,_ : Set Set -> Set [assoc comm id: empty] .
  op _,_ : NeSet NeSet -> NeSet [ditto] .
  op _in_ : X$Elt Set -> Bool .
  op _-_ : Set Set -> Set .   *** set difference

  var E : X$Elt .
```

⁵ These renamed modules are visible as names when using the `show modules` command (see Section 25.10) and will be shared, but they cannot be referred to directly in module expressions.

```

vars S S' : Set .
eq E, E = E .
eq E in E, S = true .
eq E in S = false [owise] .
eq (E, S) - (E, S') = S - (E, S') .
eq S - S' = S [owise] .
endfm

```

In Maude—unlike OBJ3 and other similar languages—sorts are not systematically qualified by their module name. This convention of not qualifying sorts may be particularly weak when dealing with parameterized modules. However, given that Maude supports ad-hoc overloading and that constants can be qualified in order to be disambiguated (see Section 3.9.3), the problem of ambiguity in a signature is reduced to collisions of sorts. For example, in a module one may very easily need sets of integers and sets of quoted identifiers, in which case, given the specification of the PRELIM-SET module above, we would get two `Set` sorts from different importations which would be confused into one sort. Our solution consists in *qualifying parameterized sorts*, not with the module expression they belong to, but *with the name of the view or views used in the instantiation* of the parameterized module. Since we assume that all views are named, these names are the ones used in the qualification. Specifically, in the body of a parameterized module $M\{X_1 :: T_1, \dots, X_n :: T_n\}$, any sort S can be written in the form $S\{X_1, \dots, X_n\}$. When the module is instantiated with views $V_1 \dots V_n$, then this sort is instantiated to $S\{V_1, \dots, V_n\}$.

Note that, although we strongly recommend it, the parameterization of sorts is optional, and therefore, for example, the above PRELIM-SET specification is a perfectly valid parameterized module.

Sorts declared in the parameterized module $M\{X_1 :: T_1, \dots, X_n :: T_n\}$ may in general be parameterized as $S\{Y_1, \dots, Y_m\}$, with $m \geq 1$, and where each Y_j is an X_i . It is recommended that all sorts declared in a parameterized module be parameterized with $m = n$ and $Y_j = X_j$ for $1 \leq j \leq n$, but this is not enforced—parameterized sorts may duplicate, omit, or reorder parameters and unparameterized sorts are also allowed.

Thus, the previous PRELIM-SET module to define sets could instead have been specified in a better way as follows:

```

fmod BASIC-SET{X :: TRIV} is
protecting BOOL .
sorts Set{X} NeSet{X} .
subsorts X$Elt < NeSet{X} < Set{X} .
op empty : -> Set{X} .
op _,_ : Set{X} Set{X} -> Set{X} [assoc comm id: empty] .
op _,_ : NeSet{X} NeSet{X} -> NeSet{X} [ditto] .
op _in_ : X$Elt Set{X} -> Bool .
op _-_ : Set{X} Set{X} -> Set{X} . *** set difference

```

```

var E : X$Elt .
vars S S' : Set{X} .
eq E, E = E .
eq E in E, S = true .
eq E in S = false [owise] .
eq (E, S) - (E, S') = S - (E, S') .
eq S - S' = S [owise] .
endfm

```

When this module is instantiated with the predefined view `Int`, the sort `Set{X}` becomes `Set{Int}`, and when it is instantiated with the predefined view `Qid` (see Section 9.11.1) it becomes `Set{Qid}`. In the following sections we will see additional examples of how this qualification convention for the sorts of a parameterized module avoids many unintended collisions of sort names, thus making renaming practically unnecessary.⁶

As another simple example of parameterized module, we consider a module `MAYBE{X :: TRIV}` in which we declare a sort `Maybe{X}` as a supersort of the sort `Elt` of the parameter theory and a constant `maybe` of this sort `Maybe{X}`. This technique is useful to declare a partial function as a total function, as explained in Section 4.11, and it will be applied, among other examples, in the `SEARCH-TREE` module of Section 10.9 and in the `PFUN` module of Section 19.3.2.

```

fmod MAYBE{X :: TRIV} is
  sort Maybe{X} .
  subsort X$Elt < Maybe{X} .
  op maybe : -> Maybe{X} [ctor] .
endfm

```

The `PRELIM-SET`, `BASIC-SET`, and `MAYBE` modules above have only one parameter. In general, however, parameterized modules can have several parameters. It can furthermore happen that several parameters are declared with the same parameter theory, that is, we can have, for example, an interface of the form `{X :: TRIV, Y :: TRIV}` involving two copies of the theory `TRIV`. Therefore, parameters cannot be treated as normal submodules, since we do not want them to be shared when their labels are different. We regard the relationship between the body of a parameterized module and the interface of its parameters not as an inclusion, but as a module constructor which is evaluated generating renamed copies of the parameters, which are then included. For the above interface, two copies of the theory `TRIV` are generated, with names `X :: TRIV` and `Y :: TRIV`. As already mentioned, in such copies of parameter theories, sorts are renamed as follows: if Z is the label of a parameter theory T , then each sort S in T is renamed to $Z\$S$ and each statement label l is renamed to $Z\$l$. All occurrences of these sorts and labels in the body of the parameterized module must mention their corresponding renaming.

⁶ In Section 19.3.2, we shall see how this naming convention can be easily extended to the case of Full Maude's *parameterized views*.

Let us consider as an example the following module PAIR, in which we would like to point out the use of the qualifications for the sorts coming from each of the parameters.

```
fmod PAIR{X :: TRIV, Y :: TRIV} is
  sort Pair{X, Y} .
  op <_;> : X$Elt Y$Elt -> Pair{X, Y} .
  op 1st : Pair{X, Y} -> X$Elt .
  op 2nd : Pair{X, Y} -> Y$Elt .

  var A : X$Elt .
  var B : Y$Elt .
  eq 1st(< A ; B >) = A .
  eq 2nd(< A ; B >) = B .
endfm
```

As already mentioned, if a parameter theory is structured, this renaming process for parameter theories is carried out not only at the top level, but for the whole “theory part,” that is, renaming *subtheories* but not renaming submodules. Consider, for example, the following parameterized module defining a lexicographical ordering on pairs of elements of two totally strictly ordered sets.

```
fmod LEX-PAIR{X :: STOSET, Y :: STOSET} is
  sort Pair{X, Y} .
  op <_;> : X$Elt Y$Elt -> Pair{X, Y} .
  op <_ : Pair{X, Y} Pair{X, Y} -> Bool .
  op 1st : Pair{X, Y} -> X$Elt .
  op 2nd : Pair{X, Y} -> Y$Elt .

  vars A A' : X$Elt .
  vars B B' : Y$Elt .
  eq 1st(< A ; B >) = A .
  eq 2nd(< A ; B >) = B .
  eq < A ; B > < A' ; B' > = (A < A') or (A == A' and B < B') .
endfm
```

Representing by boxes the modules (with initiality constraints), by ovals the theories (with loose semantics), by triple arrows the **protecting** and parameter importations, and by single arrows the **including** importations, we can depict the structure of the LEX-PAIR functional module defining a lexicographic order on pairs as in Figure 8.3, where we have two copies not only of STOSET but also of the SPOSET and TAOSET subtheories (see also Figure 8.2 in page 208), but only one copy of the BOOL submodule.

The parameter theory of a module can be any module expression whose result is a theory. The following module defines bags of elements with an **occurrences** operation that returns the number of occurrences of a particular element in a given bag.

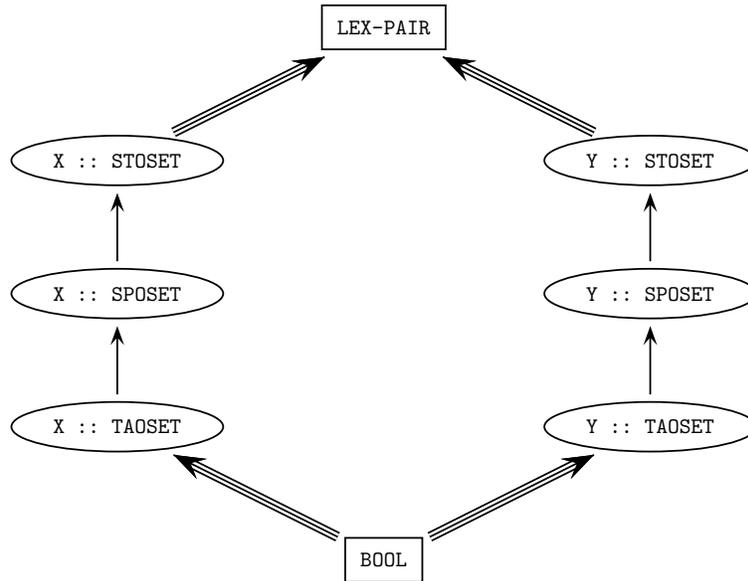


Fig. 8.3. Structure of LEX-PAIR

```

fmod BAG{X :: TRIV * (sort Elt to Element)} is
  protecting NAT .
  sorts Bag{X} NeBag{X} .
  subsorts X$Element < NeBag{X} < Bag{X} .
  op mt : -> Bag{X} .
  op __ : Bag{X} Bag{X} -> Bag{X} [assoc comm id: mt] .
  op __ : Bag{X} NeBag{X} -> NeBag{X} [ditto] .
  op occurrences : X$Element Bag{X} -> Nat .

  vars E E' : X$Element .
  var S : Bag{X} .
  eq occurrences(E, E S) = 1 + occurrences(E, S) .
  eq occurrences(E, S) = 0 [owise] .
endfm

```

Module instantiation will be explained in the next section, and then we shall see some execution examples.

8.3.4 Module instantiation

Instantiation is the process by which actual parameters are bound to the formal parameters of a parameterized module and a new module is created as a result. This can be seen in fact as the evaluation of a module expression. The instantiation requires a view from each formal parameter to its corresponding

actual parameter. Each such view is then used to bind the names of sorts, operators, etc. in the formal parameters to the corresponding sorts, operators (or expressions), etc. in the actual target.

The instantiation of a parameterized module must be made with views explicitly defined previously. Thus, given the views `Int` (from `TRIV` to `INT`) and `IntAsStoSet` (from `STOSET` to `INT`), both introduced in Section 8.3.2, we can define sets of integers with the module expression `BASIC-SET{Int}`, and lexicographically ordered pairs of integers with `LEX-PAIR{IntAsStoSet, IntAsStoSet}`.

As mentioned in Section 8.3.2, there are also views from theories to theories. Using such views we can, for example, instantiate the module `BASIC-SET` with the view `TOSET` (from `TRIV` to `TOSET`) given also in Section 8.3.2. The result is a module `BASIC-SET{TOSET}` which is still parameterized, but now by the theory `TOSET`. We can instantiate it again with a view from `TOSET` to some other theory or module, for example, `IntAsToSet` (from `TOSET` to `INT`), obtaining the module `BASIC-SET{TOSET}{IntAsToSet}`, which defines sets of integers. Note that certain new operations, which would not be meaningful in the original `BASIC-SET` module, could now be defined in a totally parametric way in an extension of `BASIC-SET{TOSET}`. For example, we could define in this way a maximum function

```
op max : NeSet{TOSET}{X} -> X$Elt .
```

as done in the `SET-MAX` module later in this section.

Another interesting use of parameterized modules is the *linking of parameters*. Suppose that we wish to define lists of sets of elements. We may define a module `SET-LIST` parameterized by the theory `TRIV` that imports the module `BASIC-SET` and declares the sort `SetList{X}` with constructors `nil` and `_;`. Note however that `BASIC-SET` is also a parameterized module, which must be instantiated to be imported. In cases like this one, we can use the label of the parameter to *link* the parameter of the module with the parameter of the submodule. Once the module is instantiated, the parameterized submodule gets instantiated with the same view. Thus, if the module `SET-LIST` below is instantiated by, say, the view `Int` to define lists of sets of integers, the submodule `BASIC-SET` also gets instantiated with the same view, providing a definition of sets of integers.⁷

```
fmod SET-LIST{X :: TRIV} is
  protecting BASIC-SET{X} .
  sort SetList{X} .
  subsort Set{X} < SetList{X} .
  op nil : -> SetList{X} [ctor] .
  op _;_ : SetList{X} SetList{X} -> SetList{X}
```

⁷ In Section 19.3.2, we shall introduce the notion of *parameterized views*, a more convenient way of defining this kind of structures. Currently, parameterized views are only available in Full Maude.

```

      [ctor assoc id: nil] .
endfm

```

As another example, let us consider the following modules `MONOMIAL` and `POLYNOMIAL`, defining, respectively, monomials on a set of variables and polynomials on a commutative ring and a set of variables. First, the module `MONOMIAL` defines monomials as terms of the form X^N , with X a variable⁸ and N a nonzero natural number indicating the power to which the variable is raised, and with an *empty syntax* multiplication operation `__` on monomials.

```

fmod MONOMIAL{X :: TRIV} is
  protecting NAT .
  sorts Pow{X} Mon{X} .
  subsorts Pow{X} < Mon{X} .
  *** multiplication
  op __ : Mon{X} Mon{X} -> Mon{X} [assoc comm] .
  op _^_ : X$Elt NzNat -> Pow{X} .
  var X : X$Elt .
  vars N M : NzNat .
  eq (X ^ N) (X ^ M) = X ^ (N + M) .
endfm

```

Once we have the specification of monomials, we can specify polynomials as monomials with coefficients in some commutative ring, and with addition and multiplication operations. Thus, for specifying polynomials on a ring and a set of variables in a module `POLYNOMIAL`, we need to import the above module `MONOMIAL`. But notice that `POLYNOMIAL` is parameterized by *two* theories: `RING`, for the coefficients, and `TRIV`, for the variables. Since we need to import monomials on the *same* set of variables, we need to *bind* or *link* such parameters. This linking is done by means of the label `X` of the parameter theory `X :: TRIV`.

```

fmod POLYNOMIAL{R :: RING, X :: TRIV} is
  protecting MONOMIAL{X} .
  sorts Poly{R, X} .
  subsorts R$Ring < Poly{R, X} .
  *** multiplication
  op __ : Poly{R, X} Poly{R, X} -> Poly{R, X} [assoc comm] .
  *** addition
  op _+_ : Poly{R, X} Poly{R, X} -> Poly{R, X} [assoc comm] .

```

⁸ Note that a variable in a monomial or polynomial is a *constant*, *not* a mathematical variable in the Maude sense. That is, in this example, as later in the lambda calculus example presented in Section 8.3.6, variables are understood as *names*. Of course, in Maude we can also define a variable `X:X$Elt` in the parameter sort to which variables belong as constants, or, more generally, variables such as `P:Poly{R, X}`. In this context, as well as in the lambda calculus examples, such mathematical variables can be distinguished from variables as names by referring to them as *metavariables*.

```

op --_ : Poly{R, X} -> Poly{R, X} .
op _- : R$Ring Mon{X} -> Poly{R, X} .

vars A B : R$Ring .
vars U V : Mon{X} .
vars P Q R : Poly{R, X} .
eq P ++ z = P .
eq P ++ (-- P) = z .
eq P e = P .
eq -- (P ++ Q) = (-- P) ++ (-- Q) .
eq -- (A U) = (- A) U .
eq P (Q ++ R) = (P Q) ++ (P R) .
eq z U = z .
eq z P = z .
eq A (B U) = (A B) U .
eq (A U) ++ (B U) = (A ++ B) U .
eq (A U) (B V) = (A B) (U V) .
eq A B = A * B .
eq A ++ B = A + B .
endfm

```

If the module POLYNOMIAL is instantiated with, say, views `RingToRat` and `Qid`, the submodule MONOMIAL then gets automatically instantiated with `Qid`, thanks to the binding of the parameters.

As an additional example, let us give a more concise definition of the parameterized module LEX-PAIR{X :: STOSET, Y :: STOSET} given in Section 8.3.3 using these ideas as follows:

```

view STOSET from TRIV to STOSET is
endv

fmod LEX-PAIR{X :: STOSET, Y :: STOSET} is
  protecting PAIR{STOSET, STOSET}{X, Y} .
  op _<_ :
    Pair{STOSET, STOSET}{X, Y} Pair{STOSET, STOSET}{X, Y} -> Bool .
  vars A A' : X$Elt .
  vars B B' : Y$Elt .
  eq < A ; B > < < A' ; B' > = (A < A') or (A == A' and B < B') .
endfm

```

In Section 8.2.2, we presented a NAT-LIST-MAX module in which we defined a `max` function that returns the greatest element of a list of natural numbers. However, we can define such a function on lists or sets of any type of elements as long as there is a total order relation available for them. Let us consider the following module SET-MAX, parameterized by the theory TOSET (see Section 8.3.1. Given a non-empty finite set of elements in a totally ordered set, the operation `max` returns the maximum element in the set. Note that we have used the `or-else` operator for short-circuit disjunction from the EXT-BOOL module to improve the efficiency of the calculation.

```

fmod SET-MAX{T :: TOSET} is
  protecting BASIC-SET{TOSET}{T} .
  protecting EXT-BOOL .
  op max : NeSet{TOSET}{T} -> T$Elt .
  var E : T$Elt .
  var S : Set{TOSET}{T} .
  eq max(E, S)
    = if S == empty or-else max(S) < E
      then E
      else max(S)
      fi .
endfm

```

We can now calculate the maximum of a set of integers by instantiating this module with the view `IntAsToset` introduced in Section 8.3.2. Notice that in this example we need an extra set of parentheses to disambiguate between the operator `max` just defined and the associative operator `max` on integers.

```

fmod INT-SET-MAX is
  protecting SET-MAX{IntAsToset} .
endfm

Maude> red max((4, 3, 5, 2, 1)) .
result NzNat: 5

```

Similarly, we can calculate the greatest element in sets of any type with a total order relation; for example, sets of strings, by using the view `StringAsToset` also introduced in Section 8.3.2:

```

fmod STRING-SET-MAX is
  protecting SET-MAX{StringAsToset} .
endfm

Maude> red max("four", "three", "five", "two", "one") .
result String: "two"

```

Notice that, if we have several parameters, we can instantiate the parameterized module or theory with some views going to theories and others going to modules. The result in such case is the expected one, that is, we get a module or theory parameterized by the targets of those views going to theories. For example, the module `RAT-POLY` below gives us a specification of the polynomials with rational coefficients by just importing the module `POLYNOMIAL` introduced above instantiated with the view `RingToRat` from the theory `RING` to the functional module `RAT` (see Section 8.3.2).

```

fmod RAT-POLY{X :: TRIV} is
  protecting POLYNOMIAL{RingToRat, X} .
endfm

```

We can then define the polynomials with rational coefficients and with quoted identifiers as variables by instantiating the module `RAT-POLY` with the following `Qid` view, which is predefined in Maude (see Section 9.11.1).

```
view Qid from TRIV to QID is
  sort Elt to Qid .
endv

fmod QID-RAT-POLY is
  pr RAT-POLY{Qid} .
endfm
```

Let us reduce as an example the following polynomial expression:

```
Maude> red in QID-RAT-POLY :
  (((2 / 3) (('X ^ 2) ('Y ^ 3)))
  ++ ((7 / 5) (('Y ^ 2) ('Z ^ 5))))
  (((1 / 7) ('U ^ 2))
  ++ (1 / 2)) .
result Poly{RingToRat, Qid}:
  (1/3 ('X ^ 2) 'Y ^ 3)
  ++ (1/5 ('U ^ 2) ('Y ^ 2) 'Z ^ 5)
  ++ (2/21 ('U ^ 2) ('X ^ 2) 'Y ^ 3)
  ++ (7/10 ('Y ^ 2) 'Z ^ 5)
```

Summarizing, a parameterized module $M\{X_1 :: T_1, \dots, X_n :: T_n\}$ with n free parameters is instantiated by the module expression $M\{A_1, \dots, A_n\}$, where each A_i is an instance of one of the following three alternatives:

- The name Y_j of a parameter of the correct theory from the module enclosing the module expression. In this case the parameter becomes a *bound parameter* in the module resulting from the instantiation. Each sort $X_i\$S$ is mapped to $Y_j\$S$, and each X_i occurring as a parameter in a parameterized sort becomes Y_j (and similarly for statement labels).
- The name of a view V with a theory as target with the correct source theory. In this case, the parameter becomes a *free parameter* with V 's target theory in the module resulting from the instantiation.
- The name of a view V with a module as target with the correct source theory. In this case, *the parameter disappears*. Each sort $X_i\$S$ is mapped to S' , where S' is the mapping of S under V . Each X_i occurring as a parameter in a parameterized sort becomes V . Each statement label $X_i\$l$ is mapped to l' , where l' is the mapping of l under the view V .

Parameterized modules with free parameters cannot be imported: first all of the free parameters must be instantiated away. Parameterized modules with bound parameters may only be imported, since they were created for module expressions in a context where the parameters are bound by an enclosing parameterized module.

Parameterized functional modules may be instantiated with views that have system modules as their targets; then the instantiated module is promoted to a system module.

Parameterized modules cannot be summed, even if all the parameters are bound. Parameterized modules may be renamed, but the renaming must not affect any sorts or operators coming from a parameter theory. The result of renaming a parameterized module is a parameterized module with the same parameters, and we can use it as any other parameterized module; for example, we can instantiate it with a view, or bind its parameters to the parameters of the module in which the module expression is being imported, as in the following example, where we rename the `SET-LIST` parameterized module above.

```
fmod MY-SET-LIST{Y :: TRIV} is
  pr (SET-LIST
      * (sort Set{X} to MySet{X},
        op __ : SetList{X} SetList{X} -> SetList{X} to __))
      {Y} .
  endfm

fmod MY-QID-SET-LIST is
  protecting MY-SET-LIST{Qid} .
  endfm
```

The `SET-LIST` module has only free parameters and so it can be renamed; however its renaming imports the renaming of `BASIC-SET{X}` which has a bound parameter. Note that the parameter of the sorts appearing in the renaming of the `SET-LIST` module is `X`, since this is the label of the parameter in such module. We have used label `Y` for the parameter of `MY-SET-LIST` to emphasize this fact, although they could be the same.

Allowing renaming of modules with bound parameters requires that renamings be capable of instantiation; that is, parameterized sort names inside a renaming have their parameters instantiated, with an extra pair of curly brackets being added in the case of instantiation by a view with a theory as target.

Let us illustrate these ideas. When, due to instantiation by a view with a theory as target, a bound parameter in a renamed module escapes and needs to be rebound by an extra instantiation, the extra instantiation is inserted *before* rather than after the renaming. Let us consider the following example, where we use the views `TOSET`, from the theory `TRIV` to the theory `TOSET`, and `IntAsToSet`, from the theory `TOSET` to the predefined module `INT`, both described in Section 8.3.2.

```
fmod RENAMING-PAR-MOD-A{X :: TRIV} is
  sort Foo{X} .
  op f : Foo{X} -> Foo{X} .
  endfm
```

```

fmod RENAMING-PAR-MOD-B{X :: TRIV} is
  extending RENAMING-PAR-MOD-A{X} .
  sort Bar{X} .
  op g : Bar{X} -> Foo{X} .
endfm

fmod RENAMING-PAR-MOD-C is
  pr (RENAMING-PAR-MOD-B * (sort Foo{X} to Foo'{X},
    sort Bar{X} to Bar'{X},
    op f : Foo{X} -> Foo{X} to f',
    op g : Bar{X} -> Foo{X} to g')) {TOSET} {IntAsToset} .
endfm

```

In this case, the module `RENAMING-PAR-MOD-A` gets instantiated before it is renamed:

```

RENAMING-PAR-MOD-A{TOSET}{IntAsToset}
  * (sort Foo{TOSET}{IntAsToset} to Foo'{TOSET}{IntAsToset},
    op f : [Foo{TOSET}{IntAsToset}] -> [Foo{TOSET}{IntAsToset}]
      to f')

```

Passing parameters from an enclosing module in nonfinal instantiations is prohibited. This restriction avoids many subtle issues. Thus:

```

fmod ILLEGAL-INST{X :: RING, Y :: POSET} is
  protecting POLYNOMIAL{X, POSET}{Y} .
endfm

```

is *illegal*, because `X` occurs in the nonfinal instantiation `POLYNOMIAL{X, POSET}`. With appropriate views, this example can be correctly written as follows:

```

view RING from RING to RING is
endv

view POSET from TRIV to POSET is
endv

fmod LEGAL-INST{X :: RING, Y :: POSET} is
  protecting POLYNOMIAL{RING, POSET}{X, Y} .
endfm

```

Another way of viewing this restriction is that parameters from an enclosing module and views with theories as targets may not occur in the same instantiation. Note that views with theories as targets may never occur in a final instantiation (otherwise there would be free parameters in an import) and must occur in any nonfinal instantiation (otherwise there would be no free parameters for the next instantiation).

8.3.5 A specification of sorted lists

In this section we present a specification of sorted lists, which are defined as lists in which their elements are sorted, that is, we define sorted lists as a sub-sort of lists. To be able to declare memberships defining such data structure, the list concatenation operator is declared at the kind level, because membership axioms should only be given on associative operators defined at the kind level (see Section 24.2.8). We also specify on lists an operation `length` to compute the length of a list.

```
fmod LIST-KIND{X :: TRIV} is
  protecting NAT .
  sorts NeKList{X} KList{X} .
  subsort X$Elt < NeKList{X} < KList{X} .
  op nil : -> KList{X} .
  op __ : KList{X} KList{X} ~> KList{X} [assoc id: nil] .
  mb NL:NeKList{X} NL':NeKList{X} : NeKList{X} .

  op length : KList{X} -> Nat .
  eq length(N:X$Elt L:KList{X}) = 1 + length(L:KList{X}) .
  eq length(nil) = 0 .
endfm
```

The module `SORTED-LIST-KIND` below has as parameter the theory `NSTOSET`, defined in Section 8.3.1; this theory requires a total non-strict order `_<=_` on a set of elements, thus providing a total ordering on the elements of lists. This module imports in `protecting` mode the module for lists just defined, but first we need to instantiate the latter, parameterized over `TRIV`, using the following view between theories,

```
view NSTOSET from TRIV to NSTOSET is
endv
```

obtaining the module `LIST-KIND{NSTOSET}`, parameterized over `NSTOSET`, so that we are importing lists over a totally ordered set instead of lists over any set. Then, we also add operations `min` and `max` to compute, respectively, the smallest and greatest element of a non-empty sorted list.

```
fmod SORTED-LIST-KIND{X :: NSTOSET} is
  protecting LIST-KIND{NSTOSET}{X} .
  sorts NeSortedList{X} SortedList{X} .
  subsort X$Elt < NeSortedList{X}
           < NeKList{NSTOSET}{X} SortedList{X}
           < KList{NSTOSET}{X} .

  vars N M : X$Elt .
  var SL : SortedList{X} .
  var L : KList{NSTOSET}{X} .
```

```

op nil : -> SortedList{X} .
cmb N M L : NeSortedList{X} if N <= M /\ M L : SortedList{X} .

op min : NeSortedList{X} -> X$Elt .
ceq min(N L) = N if N L : SortedList{X} .

op max : NeSortedList{X} -> X$Elt .
ceq max(L N) = N if L N : SortedList{X} .
endfm

```

The following reductions illustrate the behavior of the specification, using an instantiation with natural numbers.

```

view Nat<= from NSTOSET to NAT is
  sort Elt to Nat .
endv

fmod NAT-SORTED-LIST-KIND is
  protecting SORTED-LIST-KIND{Nat<=} .
endfm

Maude> red in NAT-SORTED-LIST-KIND : length(2 3 4 5 6 7) .
result NzNat : 6

```

The operations `min` and `max` over a sorted list work as expected.

```

Maude> red min(2 3 4 5 6 7) .
result NzNat : 2

Maude> red max(2 3 4 5 6 7) .
result NzNat : 7

```

The same operations cannot be applied to a non-sorted list, returning a non-reduced term in the corresponding kind.

```

Maude> red min(2 4 5 6 3 7) .
result [KList{NSTOSET}{Nat<=}]: min(2 4 5 6 3 7)

Maude> red max(2 4 5 6 3 7) .
result [KList{NSTOSET}{Nat<=}]: max(2 4 5 6 3 7)

```

We can also have sorted lists of other data elements, for instance of strings:

```

view String<= from NSTOSET to STRING is
  sort Elt to String .
endv

```

To avoid the confusion between the `length` operator on strings and the one on lists, we rename the module `SORTED-LIST-KIND` before instantiating it.

```

fmod STRING-SORTED-LIST-KIND is
  pr (SORTED-LIST-KIND
      * (op length : KList{NSTOSET}{X} -> Nat to klength))
      {String<=} .
endfm

Maude> red in STRING-SORTED-LIST-KIND : "one" "two" "three" .
result NeKList{NSTOSET}{String<=}: "one" "two" "three"

Maude> red "one" "three" "two" .
result NeSortedList{String<=}: "one" "three" "two"

Maude> red min("one" "two" "three") .
result [KList{NSTOSET}{String<=}]: min("one" "two" "three")

Maude> red max("one" "three" "two") .
result String: "two"

```

8.3.6 The lambda calculus

This section shows that higher-order (in the sense of lambda calculi) computational models can also be naturally represented inside rewriting logic. We want to represent the reduction over (untyped) λ -calculus terms as rewriting inside rewriting logic. The simple idea is to specify the syntax as a functional module, and then in a system module add the usual β and η reduction rules. The problem is that β has in its righthand side a substitution operator that is usually defined at the mathematical metalevel, outside the syntax of the calculus. To solve this, we follow the approach of so called *explicit* substitution calculi, in which substitution is just another term constructor defined by means of the usual equations.

Our presentation makes an interesting use of parameterization. The calculus is defined with respect to an arbitrary set of variables; the only requirement is ensuring the possibility of always obtaining *new* variables, which are used in renaming bound variables to avoid the capture of free variables. The operation that calculates the free variables of a term is also in the representation, inside the calculus, instead of being implicit at the mathematical metalevel.

We begin our specification with a functional theory describing the requirements on sets of variables. For that, we use the techniques based on equational attributes described in Section 5.5 to specify that the elements of the sort `VarSet` are indeed sets of elements of the sort `Var`. We also require equationally defined operations `_in_`, to check if a variable belongs to a set, and `_\<_` for set difference (this operation is necessary for defining later the free variables operation). Finally, the last equation specifies the requirement for the `new` operation that generates *fresh* variables, but without defining it, so that many different instantiations are possible (notice the `nonexec` attribute). The `VAR` theory imports in `protecting` mode the predefined module `BOOL` (see Section 9.1), which is used in defining the set membership predicate and in the requirement for `new`.

```

fth VAR is
  protecting BOOL .
  sorts Var VarSet .
  subsort Var < VarSet .          *** singleton sets
  op empty-set : -> VarSet .      *** empty set
  op _U_ : VarSet VarSet -> VarSet [assoc comm id: empty-set] .
                                   *** set union
  op _in_ : Var VarSet -> Bool .  *** membership test
  op _\_ : VarSet VarSet -> VarSet . *** set difference
  op new : VarSet -> Var .        *** new variable

  vars E E' : Var .
  vars S S' : VarSet .

  eq E U E = E .
  eq E in empty-set = false .
  eq E in E' U S = (E == E') or (E in S) .
  eq empty-set \ S = empty-set .
  eq (E U S) \ S' = if E in S' then S \ S' else E U (S \ S') fi .
  eq new(S) in S = false [nonexec] .
endfth

```

Now we specify the syntax of the lambda calculus on top of the assumed variables: lambda abstraction, application (written with empty syntax, as usual), explicit substitution of a term for a variable, and the free-variables extracting operation. Extraction of free variables and substitution are defined as usual by “structural” induction on terms, but the second needs to take into account the necessary renaming of bound variables in order to avoid the capture of free variables; here it is where we make use of the `new` operation generating fresh variables.

```

fmod LAMBDA{X :: VAR} is
  sort Lambda{X} .
  subsort X$Var < Lambda{X} .          *** variables
  op \_._ : X$Var Lambda{X} -> Lambda{X} [ctor] .
                                   *** lambda abstraction
  op __ : Lambda{X} Lambda{X} -> Lambda{X} [ctor] .
                                   *** application
  op _[_/_] : Lambda{X} Lambda{X} X$Var -> Lambda{X} .
                                   *** substitution
  op fv : Lambda{X} -> X$VarSet .      *** free variables

  vars X Y : X$Var .
  vars M N P : Lambda{X} .

  *** Free variables
  eq fv(X) = X .
  eq fv(\ X . M) = fv(M) \ X .
  eq fv(M N) = fv(M) U fv(N) .

```

```

eq fv(M [N / X]) = (fv(M) \ X) U fv(N) .

*** Substitution equations
eq X [N / X] = N .
ceq Y [N / X] = Y if X /= Y .
eq (M N)[P / X] = (M [P / X])(N [P / X]) .
eq (\ X . M)[N / X] = \ X . M .
ceq (\ Y . M)[N / X] = \ Y . (M [N / X])
  if X /= Y and (not(Y in fv(N)) or not(X in fv(M))) .
ceq (\ Y . M)[N / X]
  = \ (new(fv(M N))) . ((M [new(fv(M N)) / Y])[N / X])
  if X /= Y /\ (Y in fv(N)) /\ (X in fv(M)) .

*** Alpha conversion
ceq \ X . M = \ Y . (M [Y / X]) if not(Y in fv(M)) [nonexec] .
endfm

```

The following system module simply adds the rewrite rules corresponding to the β and η reduction rules.

```

mod BETA-ETA{X :: VAR} is
  including LAMBDA{X} .
  var X : X$Var .
  vars M N : Lambda{X} .
  r1 [beta] : (\ X . M) N => M [N / X] .
  cr1 [eta] : \ X . (M X) => M if not(X in fv(M)) .
endm

```

We can instantiate this parameterized module by using natural numbers for variables, where the new variable with respect to a given finite set is obtained as the maximum plus one. In order to be able to have the usual number notation, we will use the predefined module `NAT` of natural numbers, described in Section 9.2. Moreover, we also use the parameterized module `SET-MAX` defined in Section 8.3.4 to build sets of natural numbers with a maximum operation, but renaming the set union and difference operations from `_ , _` to `_U_` and from `_ - _` to `_ \ _`, respectively, so that we can use the same notation as in the `VAR` theory above. Since `SET-MAX` is parameterized with respect to the theory `TOSET` requiring a total order, we need the following view:

```

view NatAsToset from TOSET to NAT is
  sort Elt to Nat .
endv

```

With all the previous ingredients in place, we are now able to define a view `VarNat` from the theory `VAR` to the renamed version of the parameterized module `SET-MAX` instantiated with the above view to the natural numbers. We make essential use of the feature that operations can be mapped to terms for the `new` operation to be defined as the maximum plus one; notice that the

term inside the operator map for `new` distinguishes the empty case for the set `S`, because the imported `map` operation from the `SET-MAX` module is only defined on non-empty sets. Identity mappings for the remaining operations can be omitted.

```
view VarNat from VAR to (SET-MAX * (op _,_ to _U_,
                                   op _-_ to _\_) {NatAsToset} is
  sort Var to Nat .
  sort VarSet to Set{TOSET}{NatAsToset} .
  var S : VarSet .
  op empty-set to empty .
  op new(S) to term if S == empty then 1 else s max(S) fi .
endv
```

This view is then used to instantiate the parameterized module `BETA-ETA`.

```
mod UNTYPED-LAMBDA-CALCULUS is
  protecting BETA-ETA{VarNat} .
endm
```

Reduction for untyped λ -calculus is confluent (the well-known Church-Rosser theorem), but not terminating. For example, the term

$$(\lambda 1 . (1 1))(\lambda 1 . (1 1))$$

reduces to itself, as the following trace illustrates, where we ask Maude for a sequence of only two rewrites, tracing rules but not equations. (See Section 25.6 for more information on tracing commands.)

```
Maude> set trace on .
Maude> set trace eqs off .
Maude> rew [2] (\ 1 . (1 1))(\ 1 . (1 1)) .

rewrite [2] in UNTYPED-LAMBDA-CALCULUS : (\ 1 . (1 1)) \ 1 . (1 1) .
***** rule
r1 (\ X:Nat . M:Lambda{VarNat}) N:Lambda{VarNat}
  => M:Lambda{VarNat}[N:Lambda{VarNat} / X:Nat] [label beta] .
X:Nat --> 1
M:Lambda{VarNat} --> 1 1
N:Lambda{VarNat} --> \ 1 . (1 1)
(\ 1 . (1 1)) \ 1 . (1 1)
-->
(1 1)[\ 1 . (1 1) / 1]
***** rule
r1 (\ X:Nat . M:Lambda{VarNat}) N:Lambda{VarNat}
  => M:Lambda{VarNat}[N:Lambda{VarNat} / X:Nat] [label beta] .
X:Nat --> 1
M:Lambda{VarNat} --> 1 1
N:Lambda{VarNat} --> \ 1 . (1 1)
(\ 1 . (1 1)) \ 1 . (1 1)
```

```

--->
(1 1)[\ 1 . (1 1) / 1]
result Lambda{VarNat}: (\ 1 . (1 1)) \ 1 . (1 1)

```

```
Maude> set trace off .
```

The following trace corresponds to reduction to normal form, where no limit on the number of rewrites has been imposed on the `rew` command.

```

Maude> set trace on .
Maude> set trace eqs off .
Maude> rew (\ 1 . ((1 (\ 2 . ((1 2) 2))) 1)) (\ 3 . (\ 4 . 3)) .

rewrite in UNTYPED-LAMBDA-CALCULUS :
  (\ 1 . ((1 \ 2 . ((1 2) 2)) 1)) \ 3 . \ 4 . 3 .
***** rule
rl (\ X:Nat . M:Lambda{VarNat}) N:Lambda{VarNat}
  => M:Lambda{VarNat}[N:Lambda{VarNat} / X:Nat] [label beta] .
X:Nat --> 1
M:Lambda{VarNat} --> (1 \ 2 . ((1 2) 2)) 1
N:Lambda{VarNat} --> \ 3 . \ 4 . 3
(\ 1 . ((1 \ 2 . ((1 2) 2)) 1)) \ 3 . \ 4 . 3
--->
((1 \ 2 . ((1 2) 2)) 1)[\ 3 . \ 4 . 3 / 1]
***** rule
rl (\ X:Nat . M:Lambda{VarNat}) N:Lambda{VarNat}
  => M:Lambda{VarNat}[N:Lambda{VarNat} / X:Nat] [label beta] .
X:Nat --> 3
M:Lambda{VarNat} --> \ 4 . 3
N:Lambda{VarNat} --> \ 2 . (((\ 3 . \ 4 . 3) 2) 2)
(\ 3 . \ 4 . 3) \ 2 . (((\ 3 . \ 4 . 3) 2) 2)
--->
(\ 4 . 3)[\ 2 . (((\ 3 . \ 4 . 3) 2) 2) / 3]
***** rule
rl (\ X:Nat . M:Lambda{VarNat}) N:Lambda{VarNat}
  => M:Lambda{VarNat}[N:Lambda{VarNat} / X:Nat] [label beta] .
X:Nat --> 4
M:Lambda{VarNat} --> \ 2 . (((\ 3 . \ 4 . 3) 2) 2)
N:Lambda{VarNat} --> \ 3 . \ 4 . 3
(\ 4 . \ 2 . (((\ 3 . \ 4 . 3) 2) 2)) \ 3 . \ 4 . 3
--->
(\ 2 . (((\ 3 . \ 4 . 3) 2) 2))[\ 3 . \ 4 . 3 / 4]
***** trial #1
crl \ X:Nat . (M:Lambda{VarNat} X:Nat)
  => M:Lambda{VarNat}
  if not X:Nat in fv(M:Lambda{VarNat}) = true [label eta] .
X:Nat --> 2
M:Lambda{VarNat} --> (\ 3 . \ 4 . 3) 2
***** solving condition fragment

```

```

not X:Nat in fv(M:Lambda{VarNat}) = true
***** failure for condition fragment
not X:Nat in fv(M:Lambda{VarNat}) = true
***** failure #1
***** rule
rl (\ X:Nat . M:Lambda{VarNat}) N:Lambda{VarNat}
  => M:Lambda{VarNat}[N:Lambda{VarNat} / X:Nat] [label beta] .
X:Nat --> 3
M:Lambda{VarNat} --> \ 4 . 3
N:Lambda{VarNat} --> 2
(\ 3 . \ 4 . 3) 2
--->
(\ 4 . 3)[2 / 3]
***** trial #2
crl \ X:Nat . (M:Lambda{VarNat} X:Nat)
  => M:Lambda{VarNat}
  if not X:Nat in fv(M:Lambda{VarNat}) = true [label eta] .
X:Nat --> 2
M:Lambda{VarNat} --> \ 4 . 2
***** solving condition fragment
not X:Nat in fv(M:Lambda{VarNat}) = true
***** failure for condition fragment
not X:Nat in fv(M:Lambda{VarNat}) = true
***** failure #2
***** rule
rl (\ X:Nat . M:Lambda{VarNat}) N:Lambda{VarNat}
  => M:Lambda{VarNat}[N:Lambda{VarNat} / X:Nat] [label beta] .
X:Nat --> 4
M:Lambda{VarNat} --> 2
N:Lambda{VarNat} --> 2
(\ 4 . 2) 2
--->
2[2 / 4]
result Lambda{VarNat}: \ 2 . 2

Maude> set trace off .

```

Finally, consider the term

$$(\lambda 1 . (\lambda 2 . 2))((\lambda 1 . (1\ 1))(\lambda 1 . (1\ 1)))$$

where the constant function $\lambda 1 . (\lambda 2 . 2)$ that discards its argument is applied to the self-reducing term that we have seen above. Under eager evaluation, the reduction of this term does not terminate, since the argument is always reduced before applying the function. Under normal-order evaluation, the function is applied without reducing the argument, and thus reduction reaches in this case a normal form. Maude's top-down default strategy for evaluation of rules with the `rew` command in system modules agrees with the latter, as the following reduction shows:

```

Maude> set trace on .
Maude> set trace eqs off .
Maude> rew (\ 1 . (\ 2 . 2))((\ 1 . (1 1))(\ 1 .(1 1))) .
rewrite in UNTYPED-LAMBDA-CALCULUS :
  (\ 1 . \ 2 . 2) ((\ 1 . (1 1)) \ 1 . (1 1)) .
***** rule
rl (\ X:Nat . M:Lambda{VarNat}) N:Lambda{VarNat}
  => M:Lambda{VarNat}[N:Lambda{VarNat} / X:Nat] [label beta] .
X:Nat --> 1
M:Lambda{VarNat} --> \ 2 . 2
N:Lambda{VarNat} --> (\ 1 . (1 1)) \ 1 . (1 1)
(\ 1 . \ 2 . 2) ((\ 1 . (1 1)) \ 1 . (1 1))
-->
(\ 2 . 2)[(\ 1 . (1 1)) \ 1 . (1 1) / 1]
result Lambda{VarNat}: \ 2 . 2

Maude> set trace off .

```

To reduce the undesirable gap between the textbook presentations of languages with binding constructs and their formalization, a calculus of names and explicit substitutions called CINNI has been developed in [334, 336]. It makes use of a term representation with explicit names proposed by Berklings in the context of the λ -calculus [22] and generalizes an existing substitution calculus for the λ -calculus by Lescanne based on de Bruijn indices [219]. In contrast to most explicit substitution calculi studied in the literature, CINNI is a completely generic calculus of explicit substitutions, in the sense that it can be instantiated not just to the lambda calculus, but, more generally, to the syntax of nearly arbitrary languages with name binding operators. Two applications of CINNI will be presented in Sections 23.2.1 and 23.2.4.

Predefined Data Modules

Maude has a standard library of predefined modules that, by default, are entered into the system at the beginning of each session, so that any of these predefined modules can be imported by any other module defined by the user. Also, by default, the predefined functional module `BOOL` is automatically imported (in `including` mode) as a submodule of any user-defined module, unless such importation is explicitly disabled. These modules can be found in the file `prelude.maude` that is part of the Maude distribution.

We describe below those predefined modules that provide commonly used data types, including Booleans, numbers, strings, and quoted identifiers. The relationships among these modules are shown in the importation graph in Figure 9.1, where all the importations are in `protecting` mode.

We also describe typical *parameterized* collections of data types such as lists and sets, and associations such as maps and arrays. The chapter ends introducing the built-in linear Diophantine equation solver, defined in the file `linear.maude` that is also part of the Maude distribution.

Other predefined modules, also in the `prelude.maude` file, are discussed later; more specifically, the `META-LEVEL` module is discussed in Chapter 14, the `LOOP-MODE` module in Section 18.1, and the `CONFIGURATION` module in Sections 11.1 and 11.4.

Furthermore, this chapter also describes a predefined module `MACHINE-INT` for machine integers, which is obtained from the module `INT` of (arbitrary size) integers, but is distributed in a separate file `machine-int.maude`.

As explained in Section 4.4.10, many operators in predefined modules are declared with the `special` attribute, so that they are to be treated as *built-in* operators associated with appropriate C++ code by “hooks” specified after the `special` attribute. In what follows, to lighten the exposition, we will omit the details about such hooks in special operators, writing `special (...)` instead. The full definitions can be found in the file `prelude.maude`.

Most built-in data types are algebraically constructed, that is, they are built out of constants and constructor operators; however, floating point numbers (floats), strings, and quoted identifiers (qids) are treated as countable sets

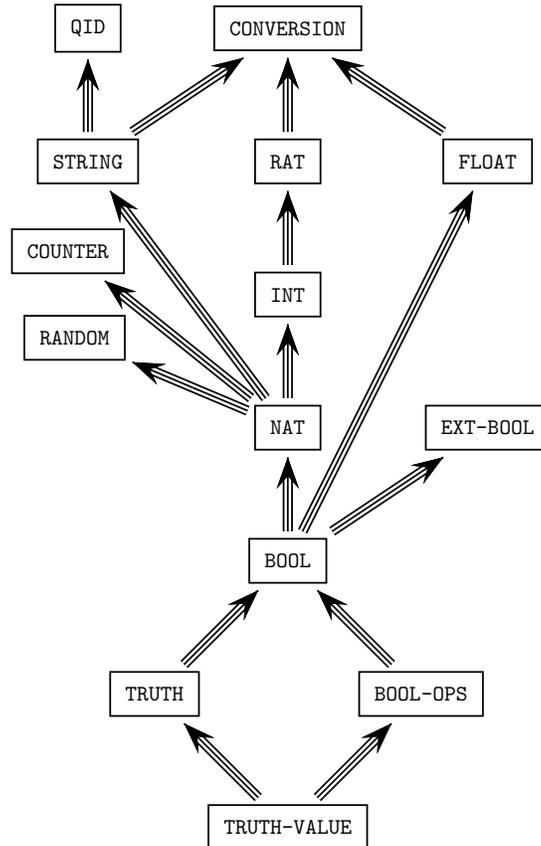


Fig. 9.1. Importation (protecting) graph of predefined modules

of constants and are represented by “special” operators `<Floats>`, `<Strings>`, and `<Qids>`, respectively. These operators are used in specifying the hooks mentioned above, but they cannot be used explicitly in terms.

9.1 Boolean values

There are five modules involving Boolean values, namely, `TRUTH-VALUE`, `TRUTH`, `BOOL-OPS`, `BOOL`, and `EXT-BOOL`. The most basic one is `TRUTH-VALUE`, which has the following definition.

```

fmod TRUTH-VALUE is
  sort Bool .
  op true : -> Bool [ctor special (...)] .

```

```

  op false : -> Bool [ctor special (...)] .
endfm

```

This module just declares the two Boolean values `true` and `false` as constants of sort `Bool`. The key thing to note is the `special` attribute associated with each of the operator declarations for these constants. In the case of Boolean values this is especially important, because certain basic constructs of the language such as conditions in a conditional equation, membership axiom, or rule, and also sort predicates associated with membership assertions evaluate to these built-in truth values.

The module `TRUTH` adds three important operators to `TRUTH-VALUE`.

```

fmod TRUTH is
  protecting TRUTH-VALUE .
  op if_then_else_fi : Bool Universal Universal -> Universal
    [poly (2 3 0) special (...)] .
  op _==_ : Universal Universal -> Bool
    [poly (1 2) prec 51 special (...)] .
  op _/= _ : Universal Universal -> Bool
    [poly (1 2) prec 51 special (...)] .
endfm

```

The operators are, respectively, `if_then_else_fi`, and the built-in operators for equality and inequality predicates.¹ These operators are special in a number of ways. Firstly, they are, by default, automatically added to every module (see Section 3.9.3). Secondly, they are *polymorphic*, so that, for each module, they can be considered to be normal operators that are ad-hoc overloaded for each connected component in the module. This is done by means of the `polymorphic` (or `poly`) attribute, as discussed in Section 4.4.4, and the symbol `Universal`, that should not be considered a common sort, as explained at the end of this section. For example, in the declaration of the `if_then_else_fi` operator, the attribute `poly (2 3 0)` means that `if_then_else_fi` is polymorphic in its second and third arguments as well as in its result.

The `if_then_else_fi` operator first rewrites its first argument, the test. If the result is of sort `Bool`, the *then* or *else* argument is selected, according to whether the test evaluated to `true` or `false`, and rewritten. If the test result is not of sort `Bool` the *then* and *else* arguments are rewritten. For example, working in the `INT` module (see Section 9.4) we get the following reductions:

```

Maude> red in INT : if 4 - 2 == 2 then 0 else 1 fi .
result Zero: 0

Maude> red if 4 - 2 /= 2 then 0 else 1 fi .
result NzNat: 1

```

¹ The `prec` attribute in the last two operators assigns each of them an appropriate precedence value for parsing purposes (see Section 3.9).

The built-in Boolean predicates `_==_` and `_=/=_` have a straightforward operational meaning: given an expression `u == v` with `u` and `v` ground terms (i.e., terms without variables), then both `u` and `v` are simplified by the equations in the module (which are assumed to be Church-Rosser and terminating) to their canonical forms (perhaps modulo some axioms such as `assoc`, etc., see Section 4.4.1) and these canonical forms are compared for equality. If they are equal, the value of `u == v` is `true`; if they are different, it is `false`. The predicate `u /= v` is just the negation of `u == v`.

Similar in spirit to the built-in operators for equality predicates, there are built-in operators for membership predicates: `_ :: S` for each sort `S`. These do not need to be explicitly declared, because they are part of the extended signature associated with each module, as described in Section 3.9.3. The operational meaning for membership operators is analogous to that of the equality operators. Namely, given a term `u` and a sort `S` in its kind, the built-in predicate `u :: S` is evaluated by reducing `u` to its canonical form, computing its *least sort* (under the preregularity, Church-Rosser, and termination assumptions), and checking that it is smaller than or equal to `S`.

But what about the *mathematical* meaning of these built-in predicates? That is, do they really correspond to ordinary equations (and not to negations or other Boolean combinations of such equations)? The point is that these built-in and efficiently implemented equality and inequality predicates could in principle have been defined in a more cumbersome and inefficient way by the user. In fact, assuming that the equations and membership axioms in the user's module are Church-Rosser and terminating modulo the equational axioms in the operator attributes (see Section 4.4.1) and that the operators satisfy the preregularity requirement, the corresponding initial algebra is a *computable* algebraic data type, for which equality, inequality, and membership in a sort are also computable functions. Therefore, by a well-known theorem of Bergstra and Tucker [21], such predicates can themselves be equationally defined by Church-Rosser and terminating equations. It is of course very convenient, and much more efficient, to unburden the user from having to give those explicit equational definitions of the equality, inequality, and membership predicates by providing them in a built-in way.

Note also that, by the above meta-argument, the use of inequality predicates, negations of membership predicates, or any Boolean combination of such predicates in abbreviated Boolean conditions does not involve any real introduction of *negation* (or other Boolean connectives) in the underlying membership equational logic, which remains a Horn logic. What we are really doing is adding more Boolean-valued functions to the module, but such functions, although provided in a built-in way for convenience and efficiency, could have been equationally defined by the user without any use of negation.

The module `BOOL-OPS` imports `TRUTH-VALUE` and adds the usual conjunction, disjunction, exclusive or, negation, and implication operators.² These operators are defined entirely equationally.

```
fmod BOOL-OPS is
  protecting TRUTH-VALUE .
  op _and_ : Bool Bool -> Bool [assoc comm prec 55] .
  op _or_  : Bool Bool -> Bool [assoc comm prec 59] .
  op _xor_ : Bool Bool -> Bool [assoc comm prec 57] .
  op not_  : Bool -> Bool [prec 53] .
  op _implies_ : Bool Bool -> Bool [gather (e E) prec 61] .
  vars A B C : Bool .
  eq true and A = A .
  eq false and A = false .
  eq A and A = A .
  eq false xor A = A .
  eq A xor A = false .
  eq A and (B xor C) = A and B xor A and C .
  eq not A = A xor true .
  eq A or B = A and B xor A xor B .
  eq A implies B = not(A xor A and B) .
endfm
```

Finally, the module `BOOL` puts together all the operators in `TRUTH` and in `BOOL-OPS`.

```
fmod BOOL is
  protecting BOOL-OPS .
  protecting TRUTH .
endfm
```

As noted above, the `BOOL` module is imported (in `including` mode) by default as a submodule of any other module defined by the user. This is accomplished by the command

```
set include BOOL on .
```

that appears in the standard library file `prelude.maude`. The `set include` command can mention any module we wish to import—in this case `BOOL`. However, this default importation can be disabled. For example, if the user wished to have the polymorphic equality, inequality and `if_then_else_fi` operators automatically added to modules, but wanted to exclude the usual Boolean connectives for the built-in truth values, this could be accomplished by writing

```
set include BOOL off .
set include TRUTH on .
```

² See Section 3.9 for information on precedence values and gathering patterns.

Similar commands are available for enabling and disabling implicit importation in `extending` and `protecting` modes (see Section 25.13). For example, the `BOOL` module can be protected by default instead of included by writing

```
set include BOOL off .
set protect BOOL on .
```

The module `EXT-BOOL` declares short-circuit versions of the conjunction and disjunction operators such as those found in LISP and other programming languages. Like the operators declared in `BOOL`, these operators are defined entirely equationally. The short-circuit behavior is the result of the strategy attributes declared for the operators as discussed in Section 4.4.7.

```
fmod EXT-BOOL is
  protecting BOOL .
  op _and-then_ : Bool Bool -> Bool
    [strat (1 0) gather (e E) prec 55] .
  op _or-else_ : Bool Bool -> Bool
    [strat (1 0) gather (e E) prec 59] .
  var B : [Bool] .
  eq true and-then B = B .
  eq false and-then B = false .
  eq true or-else B = true .
  eq false or-else B = B .
endfm
```

When the module `BOOL` is imported, the system automatically adds to the module an operator to test for membership, `_:: S`, for each sort `S`, as mentioned above (see also Section 3.9.3). Again, working in the `NUMBERS` module we have the following examples:

```
Maude> red in NUMBERS : sd(zero, zero) :: Zero .
result Bool: true

Maude> red sd(zero, zero) :: NzNat .
result Bool: false

Maude> red sd(zero, zero) :: Nat .
result Bool: true

Maude> red (zero nil) :: Zero .
result Bool: true
```

The term `sd(zero, zero)` reduces to `zero`, which has least sort `Zero` but also its supersort `Nat`. The term `zero nil` has also sort `Zero` because, using the equational axioms for the `assoc` and `id: nil` attributes, `zero nil` is the same as `zero`, which has sort `Zero`.

Note that these membership predicates are polymorphic on sorts, not on kinds. This is because to be syntactically well-formed the argument term must be of the right kind, namely the connected component containing the

sort being tested. Thus a membership at the kind level is either trivially true or a syntactic error. Also, the presence of the system truth values is required for the predicates to be meaningful, so they are only added to modules that import the module `TRUTH-VALUE` (which is included by default, as part of `BOOL`, unless the user specifies otherwise).

Advisory. In fact, the symbol `Universal` does not denote a real sort: it is instead a place holder for parsing purposes that is given an interpretation by the `polymorphic` attribute (see Section 4.4.4). The concrete effect of the interpretation of `Universal` is the instantiation in each connected component of the operators with one or more `Universal` arguments.

9.2 Natural numbers

The natural numbers module `NAT` provides a Peano-like specification of the natural numbers with an explicit successor function, while at the same time providing efficient built-in operators thanks to the `iter` theory (see Section 4.4.2) and an efficient binary representation of unbounded natural numbers arithmetic using the GNU GMP library.

The natural numbers sort hierarchy has top sort `Nat` and (disjoint) subsorts `Zero` and `NzNat`. The sort `Nat` is generated from the constant `0` (of sort `Zero`) and the successor operator `s_`.

```
fmod NAT is
  protecting BOOL .
  sorts Zero NzNat Nat .
  subsort Zero NzNat < Nat .
  *** constructors
  op 0 : -> Zero [ctor] .
  op s_ : Nat -> NzNat [ctor iter special (...)] .
```

Having `0` and successor as constructors means that you can define functions on the natural numbers by matching into the successor notation; for example:

```
fmod FACTORIAL is
  protecting NAT .
  op _! : Nat -> NzNat .
  var N : Nat .
  eq 0 ! = 1 .
  eq (s N) ! = (s N) * N ! .
endfm
```

Try entering this module into Maude and then entering the commands

```
Maude> red 100 ! .
Maude> red 1000 ! .
```

(The results are omitted; the first has 158 digits and the second 2568 digits.)

Natural numbers can be input, and by default will be output, in normal decimal notation; however 42 is just syntactic sugar for `s_42(0)`. The command `set print number on/off` controls whether or not decimal notation is used by the pretty printer. Thus executing the command `set print number off` will cause numbers to be printed using iteration notation.

Most of the usual arithmetic operators are provided in NAT. They are not defined algebraically but could be given an algebraic definition by the user if desired, for example for theorem proving purposes.

```

*** ARITHMETIC OPERATIONS
*** addition
op _+_ : NzNat Nat -> NzNat [assoc comm prec 33 special (...)] .
op _+_ : Nat Nat -> Nat [ditto] .
*** symmetric difference
op sd : Nat Nat -> Nat [comm special (...)] .
*** multiplication
op *__ : NzNat NzNat -> NzNat [assoc comm prec 31 special (...)] .
op *__ : Nat Nat -> Nat [ditto] .
*** quotient
op _quo_ : Nat NzNat -> Nat [prec 31 gather (E e) special (...)] .
*** remainder
op _rem_ : Nat NzNat -> Nat [prec 31 gather (E e) special (...)] .
*** exponential n^m = n * ... * n (m times)
op _^_ : Nat Nat -> Nat [prec 29 gather (E e) special (...)] .
op _^_ : NzNat Nat -> NzNat [ditto] .
*** exponential modulo modExp(n,m,p) = n^m mod p
op modExp : Nat Nat NzNat ~> Nat [special (...)] .
*** greatest common divisor
op gcd : NzNat NzNat -> NzNat [assoc comm special (...)] .
op gcd : Nat Nat -> Nat [ditto] .
*** least common multiple
op lcm : NzNat NzNat -> NzNat [assoc comm special (...)] .
op lcm : Nat Nat -> Nat [ditto] .
*** minimum
op min : NzNat NzNat -> NzNat [assoc comm special (...)] .
op min : Nat Nat -> Nat [ditto] .
*** maximum
op max : NzNat Nat -> NzNat [assoc comm special (...)] .
op max : Nat Nat -> Nat [ditto] .

```

The operators `_+_` and `*_*_` compute the usual addition and multiplication operations and `_^_` is exponentiation.

The *symmetric difference* operator, `sd`, subtracts the smaller of its arguments from the larger. Thus, for example,

```

Maude> red in NAT : sd(4, 9) .
result NzNat: 5

```

```
Maude> red sd(9, 4) .
result NzNat: 5
```

The quotient and remainder operators, denoted `_quo_` and `_rem_`, satisfy the equation

$$((i \text{ quo } j) * j) + (i \text{ rem } j) = i,$$

for natural numbers i and j . For example,

```
Maude> red in NAT : 11 quo 4 .
result NzNat: 2
```

```
Maude> red 11 rem 4 .
result NzNat: 3
```

The operator `modExp` computes modular exponentiation, with the third argument being the modulus. For example,

```
Maude> red in NAT : modExp(7, 1234, 2) .
result NzNat: 1
```

```
Maude> red modExp(8, 1234, 2) .
result Zero: 0
```

The operators `gcd`, `lcm`, `min`, and `max` compute the greatest common divisor, the least common multiple, the minimum and the maximum, respectively. Since these operators are associative and commutative, they can be used with any number (at least two) of arguments. For example,

```
Maude> red in NAT : gcd(6, 15, 21) .
result NzNat: 3
```

```
Maude> red lcm(6, 15, 21) .
result NzNat: 210
```

```
Maude> red min(6, 15, 21) .
result NzNat: 6
```

```
Maude> red max(6, 15, 21) .
result NzNat: 21
```

```
Maude> red gcd(0, 0) .
result Zero: 0
```

Operators that act on the binary representation of natural numbers interpreted as bit strings are:

- bitwise exclusive or (`_xor_`);
- bitwise and (`_&_`);
- bitwise or (`_|_`);
- rightshift—quotient by a power of 2 (`_>>_`); and
- leftshift—multiplication by a power of 2 (`_<<_`).

```

*** BITSTRING MANIPULATION
*** bitwise exclusive or
op _xor_ : Nat Nat -> Nat [assoc comm prec 55 special (...)] .
*** bitwise and
op &_amp;_ : Nat Nat -> Nat [assoc comm prec 53 special (...)] .
*** bitwise or
op _|_ : NzNat Nat -> NzNat [assoc comm prec 57 special (...)] .
op _|_ : Nat Nat -> Nat [ditto] .
*** right shift -- quotient by power of 2
op _>>_ : Nat Nat -> Nat [prec 35 gather (E e) special (...)] .
*** left shift -- multiplication by power of 2
op _<<_ : Nat Nat -> Nat [prec 35 gather (E e) special (...)] .

```

Here are some examples using the bitwise operators.

```

Maude> red in NAT : 5 xor 7 .
result NzNat: 2

```

```

Maude> red 5 xor 2 .
result NzNat: 7

```

```

Maude> red 5 xor 5 .
result Zero: 0

```

```

Maude> red 5 & 7 .
result NzNat: 5

```

```

Maude> red 5 & 2 .
result Zero: 0

```

```

Maude> red 5 | 7 .
result NzNat: 7

```

```

Maude> red 5 | 2 .
result NzNat: 7

```

```

Maude> red 5 >> 2 .
result NzNat: 1

```

```

Maude> red 5 << 2 .
result NzNat: 20

```

The operators `<_`, `<=_`, `>_`, and `>=_` denote the usual operations for comparing numbers: less than, less than or equal, greater than, and greater than or equal, respectively. The operator `divides` returns true if and only if its second argument is a multiple of the first one.

```

*** TESTS
*** n less than m
op <_ : Nat Nat -> Bool [prec 37 special (...)] .
*** n less than or equal to m
op <=_ : Nat Nat -> Bool [prec 37 special (...)] .

```

```

*** n greater than m
op >_ : Nat Nat -> Bool [prec 37 special (...)] .
*** n greater than or equal to m
op >=_ : Nat Nat -> Bool [prec 37 special (...)] .
*** n divides m
op _divides_ : NzNat Nat -> Bool [prec 51 special (...)] .
endfm

```

Note that, to avoid producing negative numbers, subtraction and bitwise not are not provided. The symmetric difference can be used in place of subtraction.

The operational semantics for most of the built-in operators is such that you only get built-in behavior when all the arguments are actually natural numbers. The exception is associative and commutative built-in operators which will compute as much as possible on natural number arguments and leave the remaining arguments unchanged; for example,

```

Maude> red in NAT : gcd(gcd(12, X:Nat), 15) .
result Nat: gcd(X:Nat, 3)

```

If the built-in operator does not disappear using the built-in semantics, then user equations are tried.

Advisory. It is easy to overload your machine's memory by generating huge natural numbers. There is a limit on exponentiation in that built-in behavior will not occur if the first argument is greater than 1 and the second argument is too large. Similarly, leftshift does not work if the first argument is greater than or equal to 1 and the second argument is too large. Currently "too large" means greater than 1000000 but this may change. Modular exponentiation has no such limits as its built-in semantics takes advantage of the fact that the result cannot be larger than the modulus. This is likely to be useful for cryptographic algorithms.

9.3 Random numbers and counters

The functional module `RANDOM` adds to `NAT` a pseudo-random number generator:

```

fmod RANDOM is
  protecting NAT .
  op random : Nat -> Nat [special (...)] .
endfm

```

The function `random` is the mapping from `Nat` into the range of natural numbers $[0, 2^{32} - 1]$ computed by successive calls to the Mersenne Twister Random Number Generator.³ For example,

³ For information on the Mersenne Twister Random Number Generator, see <http://www-personal.engin.umich.edu/~wagnerr/MersenneTwister.html>.

```
Maude> red in RANDOM : random(17) .
result NzNat: 1171049868
```

Although `random` is purely functional, it caches the state of the random number generator so that evaluating `random(0)` is always fast, as is evaluating `random(n+1)` if `random(n)` was the previous call to the operator `random`. In general, after generating `random(n)`, both `random(n)` and `random(n+1)` are computed efficiently because `random(n)` is a look up, while `random(n+k)` takes k steps of the twister or $O(k)$ time.

By default the seed 0 is used, but a different seed, giving rise to a different function, may be specified by the command line option `-random-seed=n`, where n is a natural number in the range $[0, 2^{32} - 1]$. For example, if we invoke the Maude interpreter with the option `-random-seed=42` and run the previous example again we get

```
Maude> red in RANDOM : random(17) .
result NzNat: 613608295
```

The predefined system module `COUNTER` adds a “counter” that can be used to generate new names and new random numbers in Maude programs that do not want to explicitly maintain this state.

```
mod COUNTER is
  protecting NAT .
  op counter : -> [Nat] [special (...)] .
endm
```

For the `rewrite` and `frewrite` commands (see Sections 6.4 and 25.2), as well as the `erewrite` command (see later Section 11.4), the built-in constant `counter` has special rule rewriting semantics: each time it has the opportunity to do a rule rewrite, it rewrites to the next natural number, starting at 0. In this way the predefined system module `COUNTER` provides a built-in strategy for the application of the implicit nondeterministic rewrite rule

```
r1 counter => N:Nat .
```

that rewrites the constant `counter` to a natural number. The built-in strategy applies this rule so that the natural number obtained after applying the rule is exactly the successor of the value obtained in the preceding rule application.

We can use the `COUNTER` module together with the predefined `RANDOM` module described above to sample various probability distributions. This topic is further discussed in Section 22.6, but we can illustrate the general idea with the following `SAMPLER` module, which can be used to sample a Bernoulli distribution corresponding to tossing a biased coin. This module also imports the predefined module `CONVERSION`, described later in Section 9.9, which includes conversion functions between different types of numbers.

```

mod SAMPLER is
  pr RANDOM .
  pr COUNTER .
  pr CONVERSION .
  op rand : -> [Float] .
  op sampleBernoulli : Float -> [Bool] .
  rl rand => float(random(counter) / 4294967295) .
  rl sampleBernoulli(P:Float) => rand < P:Float .
endm

```

The first rule rewrites the constant `rand` to a floating point number between 0 and 1 pseudo-randomly chosen according to the uniform distribution. This floating point number is obtained by converting the rational number `random(counter) / 4294967295` into a floating point number, where $4294967295 = 2^{32} - 1$ is the maximum value that the `random` function can attain. We can then use the uniform sampling of a number between 0 and 1 to sample the tossing of a coin with a given bias `P:Float` between 0 and 1. This is accomplished by the second rewrite rule in `SAMPLER`.

Sampling capabilities defined in this style for different probability distributions can then be used to specify *probabilistic models* in Maude. This topic is treated more extensively in Section 22.6, where the key notion of *probabilistic rewrite theory* is explained. We can give a flavor for how such models can be simulated in Maude by means of a simple battery-operated clock example. We may represent the state of such a clock as a term `clock(T,C)`, with `T` a natural number denoting the time, and `C` a positive real denoting the amount of battery charge. Each time the clock ticks, the time is increased by one unit, and the battery charge slightly decreases; however, the lower the battery charge, the greater the chance that the clock will stop, going into a state of the form `broken(T,C)`. We can model this system by means of the following Maude specification:

```

mod CLOCK is
  pr SAMPLER .
  sort Clock .
  op clock : Nat Float -> Clock [ctor] .
  op broken : Nat Float -> Clock [ctor] .
  var T : Nat .
  var C : Float .
  rl clock(T,C)
    => if sampleBernoulli(C / 1000.0)
      then clock(s(T), C - (C / 1000.0))
      else broken(T, C)
    fi .
endm

```

This rule models the fact that each time the clock is going to tick a coin is tossed; if the result is `true`, then the clock ticks normally, but if the result is `false`, then the clock breaks down. If the battery charge is high enough,

the bias of the coin will be highly towards normal ticking, but as the battery charge decreases, the bias gradually decreases, so that a breakdown becomes increasingly likely.

One can use a module such as `CLOCK` above to perform *Monte Carlo simulations* of the probabilistic system we are interested in. Of course, we want different arguments for the random number generator to be used each time from the same initial state so that we obtain different behaviors. In Maude this can be easily achieved within the same Maude session by typing the command

```
set clear rules off .
```

which turns off the automatic clearing of rule state information, including counter values (see Section 25.2). This means that when we run several times the same computation, a different counter value will be initially used each time, therefore getting different behaviors in the expected Monte Carlo way. For example, we get the following simulations for the behavior of a clock until it breaks:

```
Maude> rewrite in CLOCK : clock(0, 1.0e+3) .
result Clock: broken(40, 9.607702107358117e+2)

Maude> rewrite in CLOCK : clock(0, 1.0e+3) .
result Clock: broken(46, 9.5501998182355942e+2)

Maude> rewrite in CLOCK : clock(0, 1.0e+3) .
result Clock: broken(16, 9.8411944181564002e+2)

Maude> rewrite in CLOCK : clock(0, 1.0e+3) .
result Clock: broken(6, 9.9401498001499397e+2)

Maude> rewrite in CLOCK : clock(0, 1.0e+3) .
result Clock: broken(28, 9.7237474437709557e+2)
```

Since it is reasonable for a program to use multiple counters, the safe way to do this is to import renamed copies of `COUNTER`; for example

```
protecting COUNTER * (op counter to counter2) .
```

Counters are inactive with respect to search, model checking, and equational rewriting. Notice that there are potentially bad interactions with the debugger (see Section 24.1.3) since another `rewrite/frewrite/erewrite` executed in the debugger will lose the counter state of the interrupted `rewrite/frewrite/erewrite`.

9.4 Integer numbers

The module `INT` extends `NAT` with a unary minus `-_` on nonzero natural numbers to construct the negative integers. Integers can be input, and by default

are output, in normal decimal notation; however, -42 is just an alternative concrete syntax for $- 42$, which itself is just an alternative concrete syntax for $- s_{-}^{42}(0)$.

```
fmod INT is
  protecting NAT .
  sorts NzInt Int .
  subsorts NzNat < NzInt Nat < Int .

  op _- : NzNat -> NzInt [ctor special (...)] .
```

Unary minus is then extended to `Int` so that

```
- - I:Int = I:Int
- 0 = 0
```

The arithmetic operations of `NAT` are extended to integers. In addition, there are operators for subtraction, `_-`, and absolute value, `abs`.

```
*** ARITHMETIC OPERATIONS
*** unary minus
op _- : NzInt -> NzInt [ditto] .
op _- : Int -> Int [ditto] .
*** addition
op _+ : Int Int -> Int [assoc comm prec 33 special (...)] .
*** subtraction
op _- : Int Int -> Int [prec 33 gather (E e) special (...)] .
*** multiplication
op *_ : NzInt NzInt -> NzInt [assoc comm prec 31 special (...)] .
op *_ : Int Int -> Int [ditto] .
*** quotient
op _quo_ : Int NzInt -> Int [prec 31 gather (E e) special (...)] .
*** remainder
op _rem_ : Int NzInt -> Int [prec 31 gather (E e) special (...)] .
*** exponentiation
op _^_ : Int Nat -> Int [prec 29 gather (E e) special (...)] .
op _^_ : NzInt Nat -> NzInt [ditto] .
*** absolute value
op abs : NzInt -> NzNat [special (...)] .
op abs : Int -> Nat [ditto] .
*** greatest common divisor
op gcd : NzInt NzInt -> NzNat [assoc comm special (...)] .
op gcd : Int Int -> Nat [ditto] .
*** least common multiple
op lcm : NzInt NzInt -> NzNat [assoc comm special (...)] .
op lcm : Int Int -> Nat [ditto] .
*** minimum
op min : NzInt NzInt -> NzInt [assoc comm special (...)] .
op min : Int Int -> Int [ditto] .
*** maximum
```

```

op max : NzInt NzInt -> NzInt [assoc comm special (...)] .
op max : Int Int -> Int [ditto] .
op max : NzNat Int -> NzNat [ditto] .
op max : Nat Int -> Nat [ditto] .

```

The operators `_quo_` and `_rem_` satisfy the same equation for integer arguments as for natural numbers. The sign of the quotient is the product of the signs of the arguments.

```

Maude> red in INT : -11 quo 4 .
result NzInt: -2

```

```

Maude> red 11 quo -4 .
result NzInt: -2

```

```

Maude> red -11 quo -4 .
result NzNat: 2

```

```

Maude> red 11 rem -4 .
result NzNat: 3

```

```

Maude> red -11 rem 4 .
result NzInt: -3

```

```

Maude> red -11 rem -4 .
result NzInt: -3

```

Bitwise operations on negative integers use the 2's complement representation and the operator `~_`, computing the bitwise not operation, is added.

```

*** BITSTRING MANIPULATION (TWO'S COMPLEMENT)
*** bitwise not
op ~_ : Int -> Int [special (...)] .
*** bitwise exclusive or
op _xor_ : Int Int -> Int [assoc comm prec 55 special (...)] .
*** bitwise and
op &_amp;_ : Nat Int -> Nat [assoc comm prec 53 special (...)] .
op &_amp;_ : Int Int -> Int [ditto] .
*** bitwise or
op _|_ : NzInt Int -> NzInt [assoc comm prec 57 special (...)] .
op _|_ : Int Int -> Int [ditto] .
*** rightshift
op _>>_ : Int Nat -> Int [prec 35 gather (E e) special (...)] .
*** leftshift
op _<<_ : Int Nat -> Int [prec 35 gather (E e) special (...)] .

```

Tests on integers extend those on the natural numbers.

```

*** TESTS
*** less than
op <_ : Int Int -> Bool [prec 37 special (...)] .
*** less than or equal

```

```

op _<=_ : Int Int -> Bool [prec 37 special (...)] .
*** greater than
op _>_ : Int Int -> Bool [prec 37 special (...)] .
*** greater than or equal
op _>=_ : Int Int -> Bool [prec 37 special (...)] .

op _divides_ : NzInt Int -> Bool [prec 51 special (...)] .
endfm

```

Let us show with an example how a predefined module can be reused to define new subsorts that refine the sort structure of the data type. In the following example, we introduce additional subsorts and overload the successor operator `s_` (originally coming from the module `NAT` imported in `protecting` mode into `INT`) in order to specify the sort of integers greater than three.

```

fmod INT-GT-3 is
  protecting INT .
  sorts One Two Three IntGt3 .
  subsorts One Two Three IntGt3 < NzNat .
  op s_ : Zero -> One [ctor ditto] .
  op s_ : One -> Two [ctor ditto] .
  op s_ : Two -> Three [ctor ditto] .
  op s_ : Three -> IntGt3 [ctor ditto] .
  op s_ : IntGt3 -> IntGt3 [ctor ditto] .
endfm

```

We can check the sort of a number by “reducing” the corresponding constant, as follows:

```

Maude> red -1 .
result NzInt: -1

Maude> red 0 .
result Zero: 0

Maude> red 1 .
result One: 1

Maude> red 2 .
result Two: 2

Maude> red 3 .
result Three: 3

Maude> red 4 .
result IntGt3: 4

Maude> red 12345678901234567890 .
result IntGt3: 12345678901234567890

```

In theory, the sort of integers greater than three could also be specified by means of membership axioms (see Sections 4.2 and 4.3). However, memberships are not guaranteed to work correctly with the number hierarchy, because of the special internal representation for iterated towers of `s_` symbols.

9.5 Machine integers

Versions of Maude prior to 2.0 supported machine integers in place of arbitrary size integers. Initially they were 32-bit in Maude 1.0 but were increased to 64-bit in Maude 1.0.5.

For certain applications, such as specifying programming languages that support machine integers as a built-in data type, it is convenient to have a predefined specification for machine integers. Fortunately, it is straightforward to efficiently emulate machine integers in terms of arbitrary size integers.

First we rename a copy of the regular integers, giving the sorts new names consistent with the new semantics and renaming those operators that either will not be defined on machine integers or else will have new semantics. Note that the operators `~_`, `_&_`, `_|_`, `_<_`, `_<=_`, `_>_`, and `_=>_` are not modified by the renaming.

```
fmod RENAMED-INT is
  protecting INT * (sort Zero to MachineZero,
    sort NzNat to NzMachineNat,
    sort Nat to MachineNat,
    sort NzInt to NzMachineInt,
    sort Int to MachineInt,
    op s_ : Nat -> NzNat to $succ,
    op sd : Nat Nat -> Nat to $sd,
    op -_ : Int -> Int to $neg,
    op +_ : Int Int -> Int to $add,
    op -_ : Int Int -> Int to $sub,
    op *_ : NzInt NzInt -> NzInt to $mult,
    op _quo_ : Int NzInt -> Int to $quo,
    op _rem_ : Int NzInt -> Int to $rem,
    op ^_ : Int Nat -> Int to $pow,
    op abs : NzInt -> NzNat to $abs,
    op gcd : NzInt Int -> NzNat to $gcd,
    op lcm : NzInt NzInt -> NzNat to $lcm,
    op min : NzInt NzInt -> NzInt to $min,
    op max : NzInt NzInt -> NzInt to $max,
    op _xor_ : Int Int -> Int to $xor,
    op _>>_ : Int Nat -> Int to $shr,
    op _<<_ : Int Nat -> Int to $shl,
    op _divides_ : NzInt Int -> Bool to $divides) .
endfm
```

We then give a parameter theory that specifies the number of bits in a machine integer, which must be a power of 2, greater or equal to 2. Notice that this theory is based on the previous module, which is imported in `protecting` mode. Therefore, `$nrBits` is a *parameter constant* ranging over the `NzMachineNat` sort in the `RENAMED-INT` module, which is imported with an initial algebra semantics.

```

fth BIT-WIDTH is
  protecting RENAMED-INT .
  op $nrBits : -> NzMachineNat .

  var N : NzMachineNat .
  eq $divides(2, $nrBits) = true [nonexec] .
  ceq $divides(2, N) = true
    if $divides(N, $nrBits) /\ N > 1 [nonexec] .
endfth

```

Also provided are two predefined views that set the number of bits value `$nrBits` respectively to 32 and 64, the two most common sizes.

```

view 32-BIT from BIT-WIDTH to RENAMED-INT is
  op $nrBits to term 32 .
endv

```

```

view 64-BIT from BIT-WIDTH to RENAMED-INT is
  op $nrBits to term 64 .
endv

```

The module `MACHINE-INT` takes a bit width parameter and defines those operations that have a new semantics when applied to machine integers. In many cases this means applying the operation `$wrap` to the results to correctly simulate the wrap-around effect over an overflow on signed fixed bit width integers by, in effect, extending the sign bit infinitely to the left. In the case of `_^_` the meaning of the operation changes to exclusive or (from exponentiation on arbitrary size integers).

```

fmod MACHINE-INT{X :: BIT-WIDTH} is

  vars I J : MachineInt .
  var K : NzMachineInt .

  op $mask : -> NzMachineInt [memo] .
  eq $mask = $sub($nrBits, 1) .

  op $sign : -> NzMachineInt [memo] .
  eq $sign = $pow(2, $mask) .

  op maxMachineInt : -> NzMachineInt [memo] .
  eq maxMachineInt = $sub($sign, 1) .

  op minMachineInt : -> NzMachineInt [memo] .
  eq minMachineInt = $neg($sign) .

  op $wrap : MachineInt -> MachineInt .
  eq $wrap(I) = (I & maxMachineInt) | $neg(I & $sign) .

```

```

op _+_ : MachineInt MachineInt -> MachineInt
  [assoc comm prec 33] .
eq I + J = $wrap($add(I, J)) .

op -_ : MachineInt -> MachineInt .
eq - I = $wrap($neg(I)) .

op _-_ : MachineInt MachineInt -> MachineInt
  [prec 33 gather (E e)] .
eq I - J = $wrap($sub(I, J)) .

op *__ : MachineInt MachineInt -> MachineInt
  [assoc comm prec 31] .
eq I * J = $wrap($mult(I, J)) .

op _/_ : MachineInt NzMachineInt -> MachineInt
  [prec 31 gather (E e)] .
eq I / K = $wrap($quo(I, K)) .

op _%_ : MachineInt NzMachineInt -> MachineInt
  [prec 31 gather (E e)] .
eq I % K = $rem(I, K) .

op ^_ : MachineInt MachineInt -> MachineInt
  [prec 55 gather (E e)] .
eq I ^ J = $xor(I, J) .

op _>>_ : MachineInt MachineInt -> MachineInt
  [prec 35 gather (E e)] .
eq I >> J = $shr(I, ($mask & J)) .

op _<<_ : MachineInt MachineInt -> MachineInt
  [prec 35 gather (E e)] .
eq I << J = $wrap($shl(I, ($mask & J))) .
endfm

```

Notice that using out of range integer constants may cause incorrect results.

We consider now the instantiation with the predefined view 32-BIT, and show the wrap-around effect in several examples.

```

fmod MACHINE-INT-TEST is
  protecting MACHINE-INT{32-BIT} .
endfm

```

In the first examples, we can see the wrap-around from negative to positive and vice versa:

```

Maude> red -2147483648 - 1 .
result NzMachineNat: 2147483647

```

```
Maude> red 2147483647 + 1 .
result NzMachineInt: -2147483648
```

In the following product, the negative case does not wrap-around but the positive case does:

```
Maude> red -1073741824 * 2 .
result NzMachineInt: -2147483648
```

```
Maude> red 1073741824 * 2 .
result NzMachineInt: -2147483648
```

Division can only cause a wrap-around in this one case:

```
Maude> red -2147483648 / -1 .
result NzMachineInt: -2147483648
```

Remainder never wraps around:

```
Maude> red -2147483648 % -1 .
result MachineZero: 0
```

Finally, we see that the sign bit “falls off the left end” in a left shift:

```
Maude> red -2147483648 << 1 .
result MachineZero: 0
```

The parameterized `MACHINE-INT` module is an interesting example of Maude’s support for what in type theory are called *dependent types* (see, for example, [236]). These are types like the power type $X^{[n]}$ or the `ARRAY{X, [n]}` type depending on a data parameter `n`, for example a natural number. We can view `MACHINE-INT` as the Maude analogue of a dependent type definition; however, note that the data parameter is not just any nonzero natural number, but must also satisfy *additional axioms*, specified in the `BIT-WIDTH` theory. For two other interesting examples of a Maude parameterized module defining the analogue of a dependent type, see the `POWER[n]` module in Section 19.3.1 (the exact analogue of the power type $X^{[n]}$) and the `NAT/{N}` module of natural numbers modulo `N` in Section 21.8. Similarly, the `TUPLE[n]` module in Section 19.3.1 provides a form of dependent type that is not even available in some type theories with dependent types.

9.6 Rational numbers

The module `RAT` extends `INT` with a binary division operator `_/_` to construct the rationals from integers and nonzero naturals. Rationals can be input, and by default are output, in normal decimal notation; however `-5/42` is equivalent to `-5 / 42`, which is equivalent to `- 5 / 42`, which really denotes `- s_5(0) / s_42(0)`. The command

```
set print rat off .
```

switches off the special printing for `_/_` so that rational numbers will be printed with spaces around the foreslash sign. Note that `set print number off` also affects the printing of rational numbers, so with both number and rational pretty-printing switches turned off `-5/42` is printed using the final notation given above.

The numerator and denominator of a rational may contain common factors but these are removed by a single built-in rewrite whenever the rational is reduced (thus `_/_` is *not* a free constructor).

Notice that, in addition to the subsort `NzRat` of nonzero rational numbers, there is a subsort `PosRat` of positive rational numbers.

```
fmod RAT is
  protecting INT .
  sorts PosRat NzRat Rat .
  subsorts NzInt < NzRat Int < Rat .
  subsorts NzNat < PosRat < NzRat .

  op _/_ : NzInt NzNat -> NzRat
    [ctor prec 31 gather (E e) special (...)] .
  vars I J : NzInt .
  vars N M : NzNat .
  var K : Int .
  var Z : Nat .
  var Q : NzRat .
  var R : Rat .
```

The basic arithmetic operations on integers are extended to rational numbers as usual. The operator `_/_` is declared special for the case when the first argument is of sort `NzInt` to enhance performance. The remaining operators are defined in Maude by equations and may do some rewriting even when their arguments are not properly constructed rationals. Note that the choice of equations for defining operators on the rationals is motivated by performance: simpler equations are possible in many cases but they turn out to incur a big performance penalty.

```
*** ARITHMETIC OPERATIONS
op _/_ : NzNat NzNat -> PosRat [ctor ditto] .
op _/_ : PosRat PosRat -> PosRat [ditto] .
op _/_ : NzRat NzRat -> NzRat [ditto] .
op _/_ : Rat NzRat -> Rat [ditto] .
eq 0 / Q = 0 .
eq I / - N = - I / N .
eq (I / N) / (J / M) = (I * M) / (J * N) .
eq (I / N) / J = I / (J * N) .
eq I / (J / M) = (I * M) / J .

op -_ : NzRat -> NzRat [ditto] .
```

```

op _- : Rat -> Rat [ditto] .
eq - (I / N) = - I / N .

op _+_ : PosRat PosRat -> PosRat [ditto] .
op _+_ : PosRat Nat -> PosRat [ditto] .
op _+_ : Rat Rat -> Rat [ditto] .
eq I / N + J / M = (I * M + J * N) / (N * M) .
eq I / N + K = (I + K * N) / N .

op _-_ : Rat Rat -> Rat [ditto] .
eq I / N - J / M = (I * M - J * N) / (N * M) .
eq I / N - K = (I - K * N) / N .
eq K - J / M = (K * M - J) / M .

op *__ : PosRat PosRat -> PosRat [ditto] .
op *__ : NzRat NzRat -> NzRat [ditto] .
op *__ : Rat Rat -> Rat [ditto] .
eq Q * 0 = 0 .
eq (I / N) * (J / M) = (I * J) / (N * M) .
eq (I / N) * K = (I * K) / N .

op ^_ : PosRat Nat -> PosRat [ditto] .
op ^_ : NzRat Nat -> NzRat [ditto] .
op ^_ : Rat Nat -> Rat [ditto] .
eq (I / N) ^ Z = (I ^ Z) / (N ^ Z) .

op abs : NzRat -> PosRat [ditto] .
op abs : Rat -> Rat [ditto] .
eq abs(I / N) = abs(I) / N .

```

The integer operations `quo`, `rem`, `gcd`, `lcm`, `min`, and `max` are also extended to the rational numbers. The operator `quo` gives the number of whole times a rational can be divided by another, `rem` gives the rational remainder. The operator `gcd` returns the largest rational that divides into each of its arguments a whole number of times, while `lcm` returns the smallest rational that is an integer multiple of its arguments.

```

op _quo_ : PosRat PosRat -> Nat [ditto] .
op _quo_ : Rat NzRat -> Int [ditto] .
eq (I / N) quo Q = I quo (N * Q) .
eq K quo (J / M) = (K * M) quo J .

op _rem_ : Rat NzRat -> Rat [ditto] .
eq (I / N) rem (J / M) = ((I * M) rem (J * N)) / (N * M) .
eq K rem (J / M) = ((K * M) rem J) / M .
eq (I / N) rem J = (I rem (J * N)) / N .

op gcd : NzRat Rat -> PosRat [ditto] .
op gcd : Rat Rat -> Rat [ditto] .

```

```

eq gcd(I / N, R) = gcd(I, N * R) / N .

op lcm : NzRat NzRat -> PosRat [ditto] .
op lcm : Rat Rat -> Rat [ditto] .
eq lcm(I / N, R) = lcm(I, N * R) / N .

op min : PosRat PosRat -> PosRat [ditto] .
op min : NzRat NzRat -> NzRat [ditto] .
op min : Rat Rat -> Rat [ditto] .
eq min(I / N, R) = min(I, N * R) / N .

op max : PosRat Rat -> PosRat [ditto] .
op max : NzRat NzRat -> NzRat [ditto] .
op max : Rat Rat -> Rat [ditto] .
eq max(I / N, R) = max(I, N * R) / N .

```

Some examples involving these operations are the following:

```

Maude> red in RAT : 1/2 quo 1/3 .
result NzNat: 1

Maude> red 1/2 rem 1/3 .
result PosRat: 1/6

Maude> red gcd(1/2, 1/3) .
result PosRat: 1/6

Maude> red lcm(1/2, 1/3) .
result NzNat: 1

```

Tests on integers are extended to rational numbers. The test `divides` returns true if a rational number divides another rational number a whole number of times.

```

*** tests
op _<_ : Rat Rat -> Bool [ditto] .
eq (I / N) < (J / M) = (I * M) < (J * N) .
eq (I / N) < K = I < (K * N) .
eq K < (J / M) = (K * M) < J .

op _<=_ : Rat Rat -> Bool [ditto] .
eq (I / N) <= (J / M) = (I * M) <= (J * N) .
eq (I / N) <= K = I <= (K * N) .
eq K <= (J / M) = (K * M) <= J .

op _>_ : Rat Rat -> Bool [ditto] .
eq (I / N) > (J / M) = (I * M) > (J * N) .
eq (I / N) > K = I > (K * N) .
eq K > (J / M) = (K * M) > J .

```

```

op _>=_ : Rat Rat -> Bool [ditto] .
eq (I / N) >= (J / M) = (I * M) >= (J * N) .
eq (I / N) >= K = I >= (K * N) .
eq K >= (J / M) = (K * M) >= J .

op _divides_ : NzRat Rat -> Bool [ditto] .
eq (I / N) divides K = I divides N * K .
eq Q divides (J / M) = Q * M divides J .

```

There are four new operators: `trunc`, `frac`, `floor`, and `ceiling`. The operator `floor` converts a rational number to an integer by rounding down to the nearest integer, `ceiling` rounds up, and `trunc` rounds towards 0. The operator `frac` gives the fraction part of its argument and this always has the same sign as its argument.

```

*** ROUNDING
op trunc : PosRat -> Nat .
op trunc : Rat -> Int .
eq trunc(K) = K .
eq trunc(I / N) = I quo N .

op frac : Rat -> Rat .
eq frac(K) = 0 .
eq frac(I / N) = (I rem N) / N .

op floor : PosRat -> Nat .
op floor : Rat -> Int .
eq floor(K) = K .
eq floor(N / M) = N quo M .
eq floor(- N / M) = - ceiling(N / M) .

op ceiling : PosRat -> NzNat .
op ceiling : Rat -> Int .
eq ceiling(K) = K .
eq ceiling(N / M) = ((N + M) - 1) quo M .
eq ceiling(- N / M) = - floor(N / M) .
endfm

```

Here are some examples of reductions involving the rounding operators:

```

Maude> red in RAT : trunc(9/7) .
result NzNat: 1

Maude> red floor(9/7) .
result NzNat: 1

Maude> red ceiling(9/7) .
result NzNat: 2

Maude> red frac(9/7) .
result PosRat: 2/7

```

```

Maude> red trunc(-9/7) .
result NzInt: -1

Maude> red floor(-9/7) .
result NzInt: -2

Maude> red ceiling(-9/7) .
result NzInt: -1

Maude> red frac(-9/7) .
result NzRat: -2/7

```

9.7 Floating-point numbers

The module `FLOAT` declares sorts and operators for manipulating floating-point numbers, which are implemented using double precision floating-point arithmetic of the underlying hardware platform, conforming to the IEEE-754 standard when supported by the hardware platform. Floating-point numbers are treated as a large set of constants, that is, a floating-point number has no algebraic structure (this is the reason for the special operator declaration `<Floats>`, as explained in the introduction of this chapter).

The sort `FiniteFloat` consists of the floating-point numbers that have a 64-bit representation. Finite floating-point numbers can be input, and by default are output, in scientific notation; they can also be input using decimal point notation. Thus `100.0` is equivalent to `1.0e+2`. The constants `Infinity` and `-Infinity` represent floating-point numbers that are outside the 64-bit representable range. Thus `Infinity` and `-Infinity` are of sort `Float` but not of sort `FiniteFloat`. Note that there are some surprises when using decimal notation to input floating-point numbers. For example, in the `FLOAT` module we have the reduction

```

Maude> red in FLOAT : 1.1 .
result FiniteFloat: 1.1000000000000001

```

This is because floating-point numbers are represented internally using a binary expansion rather than a decimal expansion and `1.1` does not have a finite length binary expansion.

```

fmod FLOAT is
  protecting BOOL .
  sorts FiniteFloat Float .
  subsort FiniteFloat < Float .
  op <Floats> : -> FiniteFloat [special (...)] .
  op <Floats> : -> Float [ditto] .

```

The arithmetic operators `_-`, `_-`, `+_`, `*_`, `/_`, `^_`, and `abs` have the usual interpretation, as in the module `INT`. Note that `1.2 / 0.0` is just an expression of kind `[Float]` and reducing it does not cause your system to crash!

```
*** ARITHMETIC OPERATIONS
op _- : Float -> Float [prec 15 special (...)] .
op _- : FiniteFloat -> FiniteFloat [ditto] .

op +_ : Float Float -> Float
    [prec 33 gather (E e) special (...)] .
op -_ : Float Float -> Float
    [prec 33 gather (E e) special (...)] .
op *_ : Float Float -> Float
    [prec 31 gather (E e) special (...)] .
op /_ : Float Float ~> Float
    [prec 31 gather (E e) special (...)] .
op ^_ : Float Float ~> Float
    [prec 29 gather (E e) special (...)] .

op abs : Float -> Float [special (...)] .
op abs : FiniteFloat -> FiniteFloat [ditto] .
```

The operator `_rem_` computes the remainder of a division, `floor` rounds down to the nearest integer, `ceiling` rounds up, and `sqrt` computes the square root.

```
op _rem_ : Float Float ~> Float
    [prec 31 gather (E e) special (...)] .
op floor : Float -> Float [special (...)] .
op ceiling : Float -> Float [special (...)] .
op sqrt : Float ~> Float [special (...)] .
```

For terms `f1` and `f2` of sort `FiniteFloat`, `f1 rem f2` computes the remainder of dividing `f1` by `f2`. Specifically, `f1 rem f2` is equal to `f1 - n * f2`, where `n` is `f1 / f2` rounded towards zero to the nearest integer. For example,

```
Maude> red in FLOAT : 5.0 rem 2.0 .
result FiniteFloat: 1.0

Maude> red -5.0 rem 2.0 .
result FiniteFloat: -1.0

Maude> red 5.0 rem 2.5 .
result FiniteFloat: 0.0
```

Some examples of reductions using the `floor` and `ceiling` operations are the following:

```
Maude> red in FLOAT : ceiling(2.5) .
result FiniteFloat: 3.0
```

```

Maude> red floor(2.5) .
result FiniteFloat: 2.0

Maude> red ceiling(- 2.5) .
result FiniteFloat: -2.0

Maude> red floor(- 2.5) .
result FiniteFloat: -3.0

Maude> red ceiling(Infinity) .
result Float: Infinity

Maude> red floor(-Infinity) .
result Float: -Infinity

```

The operators `max` and `min` for computing the maximum and the minimum, respectively, work as expected,

```

op min : Float Float -> Float [special (...)] .
op max : Float Float -> Float [special (...)] .

```

as we can see in the following examples:

```

Maude> red in FLOAT : min(2.0, -2.0) .
result FiniteFloat: -2.0

Maude> red max(2.0, -2.0) .
result FiniteFloat: 2.0

Maude> red max(2.0, Infinity) .
result Float: Infinity

Maude> red in FLOAT : min(Infinity, -Infinity) .
result Float: -Infinity

```

The operators `exp` and `log` compute the natural exponent and logarithm, respectively.

```

*** TRANSCENDENTAL OPERATIONS
op exp : Float -> Float [special (...)] .
op log : Float ~> Float [special (...)] .

```

Here are some examples:

```

Maude> red in FLOAT : exp(1.0) .
result FiniteFloat: 2.7182818284590451

Maude> red log(exp(1.0)) .
result FiniteFloat: 1.0

Maude> red log(0.0) .
result Float: -Infinity

```

The constant `pi` approximates the value of π . The number of digits is chosen to be the largest that can accurately be represented as a floating-point number. The trigonometric operators `sin`, `cos`, and `tan` expect arguments in radians. The operators `asin`, `acos`, `atan` are the corresponding inverses.

```
*** TRIGONOMETRIC OPERATIONS
op sin : Float -> Float [special (...)] .
op cos : Float -> Float [special (...)] .
op tan : Float -> Float [special (...)] .
op asin : Float ~> Float [special (...)] .
op acos : Float ~> Float [special (...)] .
op atan : Float -> Float [special (...)] .
op atan : Float Float -> Float [special (...)] .

op pi : -> FiniteFloat .
eq pi = 3.1415926535897931 .
```

Here are some examples of reductions of trigonometric expressions.

```
Maude> red in FLOAT : sin(0.0) .
result FiniteFloat: 0.0

Maude> red sin(pi) .
result FiniteFloat: 1.2246467991473532e-16

Maude> red cos(pi) .
result FiniteFloat: -1.0

Maude> red acos(cos(pi)) .
result FiniteFloat: 3.1415926535897931

Maude> red tan(pi) .
result FiniteFloat: -1.2246467991473532e-16

Maude> red sin(pi / 2.0) .
result FiniteFloat: 1.0

Maude> red cos(pi / 2.0) .
result FiniteFloat: 6.123233995736766e-17

Maude> red tan(pi / 2.0) .
result FiniteFloat: 1.633123935319537e+16

Maude> red atan(tan(pi / 2.0)) .
result FiniteFloat: 1.5707963267948966

Maude> red pi / 2.0 .
result FiniteFloat: 1.5707963267948966
```

Using the binary form of the arc tangent operator, `atan(f1, f2)`, is similar to computing `atan(f1 / f2)`, except that the signs of both arguments are used to control the quadrant of the result.

```

Maude> red in FLOAT : atan(tan(pi / 3.0)) .
result FiniteFloat: 1.0471975511965976

Maude> red atan(tan(pi / 3.0), 1.0) .
result FiniteFloat: 1.0471975511965976

Maude> red atan(tan(pi / 3.0), -1.0) .
result FiniteFloat: 2.0943951023931957

Maude> red atan(- tan(pi / 3.0), -1.0) .
result FiniteFloat: -2.0943951023931957

Maude> red atan(- tan(pi / 3.0), 1.0) .
result FiniteFloat: -1.0471975511965976

```

Numerical comparisons have the usual meaning on floating-point numbers.

```

*** TESTS
op <_ : Float Float -> Bool [prec 51 special (...)] .
op <=_ : Float Float -> Bool [prec 51 special (...)] .
op >_ : Float Float -> Bool [prec 51 special (...)] .
op >=_ : Float Float -> Bool [prec 51 special (...)] .

*** approximate equality
op _=[_]_ : Float FiniteFloat Float -> Bool [prec 51] .
vars X Y : Float .
var Z : FiniteFloat .
eq X =[Z] Y = abs(X - Y) < Z .
endfm

```

The operator `_=[_]_` tests for approximate equality, where the second argument bounds the allowed error. For example:

```

Maude> red in FLOAT : 1.111111111 =[1.0e-9] 1.111111112 .
result Bool: true

Maude> red 1.111111111 =[1.0e-10] 1.111111112 .
result Bool: false

```

9.8 Strings

The module `STRING` declares sorts and operators for manipulating strings of characters. Strings of length one form a subsort `Char` of `String`. Operations on strings are based on the SGI rope package [24], which has been optimized for functional programming, where copying with modification is supported efficiently, whereas arbitrary in-place updates are not.

Strings are input and output using the usual convention of enclosing the string characters in a pair of matching quotes `"..."`. When a string is parsed,

it is interpreted using a subset of ANSI C backslash escape conventions [206, Section A2.5.2].

To define the results of searching a string for an occurrence of another substring the sort `FindResult` is introduced. This sort consists of the natural numbers, returned as the index in the string where a found substring begins (string indexing begins with 0), and a special constant `notFound`, returned if no occurrence is found.

```
fmod STRING is
  protecting NAT .
  sorts String Char FindResult .
  subsort Char < String .
  subsort Nat < FindResult .
  op <Strings> : -> Char [special (...)] .
  op <Strings> : -> String [ditto] .
  op notFound : -> FindResult [ctor] .
```

The operators `ascii` and `char` convert between characters and ASCII codes.

```
*** conversion between ascii code and character
  op ascii : Char -> Nat [special (...)] .
  op char : Nat ~> Char [special (...)] .
```

For a natural number `n` less than 256 and a character `c`, we have `ascii(char(n)) = n` and `char(ascii(c)) = c`. For a natural number `n` greater than 255, `char(n)` is an error term of kind `[String]`. For example,

```
Maude> red in STRING : ascii("#") .
result NzNat: 35

Maude> red char(35) .
result Char: "#"

Maude> red ascii("a") .
result NzNat: 97

Maude> red char(97) .
result Char: "a"

Maude> red char(255) .
result Char: "\377"
```

On strings, `_+_` denotes the concatenation operation, with identity the empty string, `"`. String length is computed by the `length` operator.

```
*** string concatenation
  op _+_ : String String -> String
    [prec 33 gather (E e) special (...)] .

*** string length
  op length : String -> Nat [special (...)] .
```

Here are some examples.

```
Maude> red in STRING : "abc" + "def" .
result String: "abcdef"

Maude> red "ab" + "cd" + "ef" .
result String: "abcdef"

Maude> red "abc" + "" .
result String: "abc"

Maude> red length("abcdef") .
result NzNat: 6

Maude> red length("") .
result Zero: 0
```

The operators `substr`, `find`, and `rfind` deal with finding and extracting substrings. Remember that string indexing begins with 0.

```
*** substring
*** second argument is starting position, third is length
op substr : String Nat Nat -> String [special (...)] .

*** starting position of substring (second argument)
*** least one >= third argument (find)
*** greatest one <= third argument (rfind)
op find : String String Nat -> FindResult [special (...)] .
op rfind : String String Nat -> FindResult [special (...)] .
```

The expression `substr(S:String, Start:Nat, Len:Nat)` returns the substring of `S:String` of length `Len:Nat` beginning at position `Start:Nat`. If the value of the term `Start:Nat + Len:Nat` is greater than `length(S:String)` then the returned substring is the tail of `S:String` starting from position `Start:Nat`. This will be empty if the starting position is past the end of the string.

```
Maude> red in STRING : substr("abc", 0, 2) .
result String: "ab"

Maude> red substr("abc", 1, 2) .
result String: "bc"

Maude> red substr("abc", 1, 3) .
result String: "bc"

Maude> red substr("abc", 3, 2) .
result String: ""
```

`find` searches for the first match from the beginning of the string, while `rfind` searches from the end of the string backwards.

`find(S:String, Pat:String, Start:Nat)` returns the least index of an occurrence of `Pat:String` in `S:String` that is greater than or equal to `Start:Nat`. If no such index exists the constant `notFound` is returned.

`rfind(S:String, Pat:String, Start:Nat)` returns the greatest index of an occurrence of `Pat:String` in `S:String` that is less than or equal to `Start:Nat`. If no such index exists the constant `notFound` is returned.

```
Maude> red in STRING : find("abc", "b", 0) .
result NzNat: 1

Maude> red find("abc", "b", 1) .
result NzNat: 1

Maude> red find("abc", "b", 2) .
result FindResult: notFound

Maude> red find("abc", "d", 2) .
result FindResult: notFound

Maude> red rfind("abc", "b", 2) .
result NzNat: 1

Maude> red rfind("abc", "b", 1) .
result NzNat: 1

Maude> red rfind("abc", "b", 0) .
result FindResult: notFound

Maude> red rfind("abc", "d", 2) .
result FindResult: notFound
```

Some properties relating `substr`, `find`, and `rfind` are the following, where S and P are variables of sort `String`, and I , J , and K are variables of sort `Nat` such that $\text{length}(S) = K$ and $\text{length}(P) = J$.

$$I \leq \text{find}(S, P, I) \leq K - J$$

$$0 \leq \text{rfind}(S, P, I) \leq \min(I, K - J)$$

$$\text{find}(S, S, 0) = 0 = \text{rfind}(S, S, I)$$

$$\text{find}(S, "", I) = \text{if } I \leq K \text{ then } I \text{ else notFound}$$

$$\text{rfind}(S, "", I) = \text{if } I \geq K \text{ then } K \text{ else } I$$

$$\text{find}(S, P, I) \neq \text{notFound}$$

$$\implies \text{substr}(S, 0, \text{find}(S, P, I)) + P + \text{substr}(S, \text{find}(S, P, I) + J, K) = S$$

$$\text{rfind}(S, P, I) \neq \text{notFound}$$

$$\implies \text{substr}(S, 0, \text{rfind}(S, P, I)) + P + \text{substr}(S, \text{rfind}(S, P, I) + J, K) = S$$

The operators `_<_`, `_<=_`, `_>_`, and `_>=_` denote string comparison operations using the lexicographic order, where characters are compared going through their ASCII codes.

```

*** lexicographic string comparison
op <_ : String String -> Bool [prec 37 special (...)] .
op <=_ : String String -> Bool [prec 37 special (...)] .

op >_ : String String -> Bool [prec 37 special (...)] .
op >=_ : String String -> Bool [prec 37 special (...)] .
endfm

```

Here are some examples.

```

Maude> red in STRING : "abc" < "abd" .
result Bool: true

Maude> red "abc" < "abb" .
result Bool: false

Maude> red "abc" < "abcd" .
result Bool: true

```

9.9 String and number conversions

The module `CONVERSION` consolidates all the conversion functions between the three major built-in data types: `Nat/Int/Rat`, `Float`, and `String`.

```

fmod CONVERSION is
  protecting RAT .
  protecting FLOAT .
  protecting STRING .

  *** number type conversions
  op float : Rat -> Float [special (...)] .
  op rat : FiniteFloat -> Rat [special (...)] .

```

The operation `float` computes the floating-point number nearest to a given rational number. If the value of the rational number falls outside the range representable by IEEE-754 double precision finite floating-point numbers, `Infinity` or `-Infinity` is returned as appropriate. This is in accord with the convention that `Infinity` and `-Infinity` are used to handle out-of-range situations in the floating-point world.

The operator `rat` converts finite floating-point numbers to rational numbers exactly (since every IEEE-754 finite floating-point number is a rational number). Of course, if the result happens to be a natural number or an integer, that is what you get. `rat(Infinity)` and `rat(-Infinity)` do not reduce, since they have no reasonable representation in the world of rational numbers. It is intended that the equation

$$\text{float}(\text{rat}(F:\text{FiniteFloat})) = F:\text{FiniteFloat}$$

is satisfied, although this holds only if the third party library (GNU GMP) being used in the implementation meets its related requirements.

```
*** string <-> number conversions
op string : Rat NzNat ~> String [special (...)] .
op rat : String NzNat ~> Rat [special (...)] .
op string : Float -> String [special (...)] .
op float : String ~> Float [special (...)] .
```

The operator `string` converts a rational number to a string using a given base, which must lie in the range 2..36. Rational numbers that are really natural numbers or integers are converted to string representations of natural numbers or integers, so we have for example

```
Maude> red in CONVERSION : string(-1, 10) .
result String: "-1"
```

The operator `rat` converts a string to a rational number using a given base, which must lie in the range 2..36. Of course, if the result happens to be a natural number or an integer, that is what you get. Currently the function is very strict about which strings are converted: the string must be something that the Maude parser would recognize as a natural number, an integer or a rational number. This could be changed to a more generous interpretation in the future.

The operators `string` and `float` for conversion between floating-point numbers and strings satisfy the equation

```
float(string(F:Float)) = F:Float
```

A new sort, `DecFloat`, is introduced to provide the means for arbitrary formatting of floating-point numbers.

```
sort DecFloat .
op <_,_,_> : Int String Int -> DecFloat [ctor] .
op decFloat : Float Nat -> DecFloat [special (...)] .
endfm
```

A `DecFloat` consists of a sign (1, 0 or -1), a string of digits, and a decimal point position (0 is just in front of first digit, $-n$ is n positions to the left, and $+n$ is n positions to the right). Thus, `< -1, "123", 11 >` represents $-1.23e10$. `decFloat(F, N)` converts `F` to a `DecFloat`, rounding to `N` significant digits using the IEEE-754 “round to nearest” rule with trailing zeros if needed. If `N` is 0, an *exact* `DecFloat` representation of `F` is produced—this may require hundreds of digits. For any natural number `N`, `decFloat(Infinity, N)` reduces to `< 1, "Infinity", 0 >`. Here are some examples.

```
Maude> red in CONVERSION : decFloat(Infinity, 9) .
result DecFloat: < 1,"Infinity",0 >
```

```

Maude> red decFloat(-Infinity, 9) .
result DecFloat: < -1,"Infinity",0 >

Maude> red decFloat(123.0, 5) .
result DecFloat: < 1,"12300",3 >

Maude> red decFloat(-123.0, 5) .
result DecFloat: < -1,"12300",3 >

Maude> red decFloat(.123, 5) .
result DecFloat: < 1,"12300",0 >

Maude> red decFloat(.00123, 5) .
result DecFloat: < 1,"12300",-2 >

Maude> red decFloat(0.0, 5) .
result DecFloat: < 0,"00000",0 >

```

Advisory. Counterintuitive results are possible when converting from the approximate world of floating-point numbers to the exact world of rational numbers. For example,

```

Maude> red in CONVERSION : rat(1.1) .
result PosRat: 2476979795053773/2251799813685248

```

This is because, as mentioned above, 1.1 cannot be represented exactly as a floating-point number, and the nearest floating-point number is

```
1.1000000000000000088817841970012523233890533447265625
```

which is the above rational number. (Note that Maude prints the number 1.1 as 1.1000000000000001, using 17 significant digits. The above representation is obtained by reducing `decFloat(1.1, 52)`.)

9.10 Quoted identifiers

The module `QID` is a wrapper for strings in order to provide a Maude representation for tokens of Maude syntax. Quoted identifiers are input and output by preceding a Maude identifier⁴ with a (fore) quote sign. Thus `'abc` is a quoted identifier whose underlying string is `"abc"`. A quoted identifier is also an identifier, as are strings. Thus `''abc` and `'"abc"` are both quoted identifiers.

```

fmod QID is
  protecting STRING .
  sort Qid .
  op <Qids> : -> Qid [special (...)] .

  *** qid <-> string conversions

```

⁴ The syntax of Maude identifiers is discussed in Section 3.1.

```

op string : Qid -> String [special (...)] .
op qid : String ~> Qid [special (...)] .
endfm

```

The operators `qid` and `string` do the wrapping and unwrapping. `string` is injective, since every quoted identifier has a unique corresponding string.

```

Maude> red in QID : string('abc) .
result String: "abc"

Maude> red qid("abc") .
result Qid: 'abc

Maude> red string('a\b) .
result String: "a\\b"

Maude> red qid("a\\b") .
result Qid: 'a\b

Maude> red string('a'[b) .
result String: "a'[b"

Maude> red qid("a[b") .
result Qid: 'a'[b

```

The operator `qid` is only injective on strings without white space, control characters, and certain other characters which are converted to backquote. Thus the equation $\text{qid}(\text{string}(q)) = q$ holds for quoted identifiers q .

```

Maude> red in QID : qid("a b c") .
result Qid: 'a'b'c

Maude> red string('a'b'c) .
result String: "a'b'c"

Maude> red qid("a\t b") .
result Qid: 'a'b

Maude> red string('a'b) .
result String: "a'b"

```

An example of a string that cannot be converted to a quoted identifier is `"a\"b"` since identifiers are not allowed to have unpaired double quotes. Thus `qid("a\"b")` has kind `[Qid]` but does not reduce to something of sort `Qid`.

9.11 Basic theories and standard views

The library of predefined modules provided by Maude in the `prelude.maude` file includes some well-known parameterized data types that will be described in the following sections. Here we will introduce the standard theories that provide the requirements for those parameterized modules.

9.11.1 TRIV

As already described in Section 8.3.1, the simplest non-empty theory is called TRIV and consists of a single sort. A model of this theory is just a set of any cardinality (finite or infinite). The intuition behind this simple theory is that the minimum requirement possible on a parameterized data type construction is having a data type as a set of basic elements to build more data on top of it. For example, in the `LIST{X :: TRIV}` parameterized data type construction we need a data type (set) of basic elements satisfying TRIV to then build lists of such elements.

```
fth TRIV is
  sort Elt .
endfth
```

The file `prelude.maude` includes many views out of TRIV that select the main sort of the built-in modules that we have already described in the previous sections. All these views are named in the same way: by the sort they select; for example, the standard view from TRIV into RAT selecting the sort Rat is also named Rat.

```
view Bool from TRIV to BOOL is
  sort Elt to Bool .
endv
```

```
view Nat from TRIV to NAT is
  sort Elt to Nat .
endv
```

```
view Int from TRIV to INT is
  sort Elt to Int .
endv
```

```
view Rat from TRIV to RAT is
  sort Elt to Rat .
endv
```

```
view Float from TRIV to FLOAT is
  sort Elt to Float .
endv
```

```
view String from TRIV to STRING is
  sort Elt to String .
endv
```

```
view Qid from TRIV to QID is
  sort Elt to Qid .
endv
```

9.11.2 DEFAULT

The theory `DEFAULT` is slightly more complex than `TRIV`, in that in addition to a sort it also requires that there be a distinguished “default” element in such a sort. Notice that `DEFAULT` imports `TRIV` in the following presentation:

```
fth DEFAULT is
  including TRIV .
  op 0 : -> Elt .
endfth
```

The inclusion of the theory `TRIV` into the theory `DEFAULT` is made explicit by the following view, whose name coincides with the name of the target theory.

```
view DEFAULT from TRIV to DEFAULT is
endv
```

The Maude library also includes several views that map from `DEFAULT` to the various built-in data type modules by selecting the main sort and a distinguished element in it. In the case of the number sorts, this element is the zero, while for strings it is the empty string and for quoted identifiers is just the quote. Notice that operator mappings that are the identity (i.e., of the form `op 0 to 0`) do not appear explicitly in the following views but are left implicit. These views are named by appending “0” to the name of the selected sort; for example, the standard view from `DEFAULT` into `RAT` selecting the sort `Rat` and 0 as the default element is named `Rat0`.

```
view Nat0 from DEFAULT to NAT is
  sort Elt to Nat .
endv
```

```
view Int0 from DEFAULT to INT is
  sort Elt to Int .
endv
```

```
view Rat0 from DEFAULT to RAT is
  sort Elt to Rat .
endv
```

```
view Float0 from DEFAULT to FLOAT is
  sort Elt to Float .
  op 0 to term 0.0 .
endv
```

```
view String0 from DEFAULT to STRING is
  sort Elt to String .
  op 0 to term "" .
endv
```

```

view Qid0 from DEFAULT to QID is
  sort Elt to Qid .
  op 0 to term ' .
endv

```

9.11.3 STRICT-WEAK-ORDER and STRICT-TOTAL-ORDER

Although in Section 8.3.5 we have defined the notion of sorted list as based on a totally ordered set of elements, we will see in Section 9.12.6 how to relax this requirement in two different ways. The first possibility is to consider a *partially* strictly ordered set where the incomparability relation is transitive, that is, if a is not comparable with b and b is not comparable with c with respect to the given order, then a and c are not comparable either. The predefined **STRICT-WEAK-ORDER** theory below specifies a strict partial order with this additional requirement, a concept known as *strict weak order*. The second possibility is to consider a *total preorder*, as specified in Section 9.11.4 below.

Given a strict partial order $<$, that is, an irreflexive and transitive binary relation, we define the *incomparability* relation by $x \sim y$ iff both $x \not< y$ and $y \not< x$. Incomparability is symmetric by definition, and its reflexivity follows from the irreflexivity of $<$. Therefore, when we impose the additional requirement of transitivity of incomparability, we get that the relation \sim for a strict weak order is an equivalence relation.

Notice that **STRICT-WEAK-ORDER**, as presented below, imports the theory **TRIV** and also (in **protecting** mode) the module **BOOL**. The three equations express the required properties (antisymmetry is derivable from irreflexivity and transitivity) of the binary relation $_<_$ on the sort **Elt**, as is made explicit in the corresponding labels.

```

fth STRICT-WEAK-ORDER is
  protecting BOOL .
  including TRIV .
  op \_<\_ : Elt Elt -> Bool .
  vars X Y Z : Elt .
  ceq X < Z = true if X < Y /\ Y < Z [nonexec label transitive] .
  eq X < X = false [nonexec label irreflexive] .
  ceq X < Y or Y < X or Y < Z or Z < Y = true if X < Z or Z < X
    [nonexec label incomparability-transitive] .
endfth

```

The following theory extends the previous one with a totality requirement, thus specifying a strict total order. Under these conditions, the incomparability relation reduces to the identity (because any pair of different elements is comparable) and the transitivity of incomparability holds trivially.

```

fth STRICT-TOTAL-ORDER is
  including STRICT-WEAK-ORDER .
  vars X Y : Elt .

```

```

    ceq X = Y if X < Y = false /\ Y < X = false [nonexec label total] .
endfth

```

The theory `STRICT-TOTAL-ORDER` is a different presentation of the equivalent theory `STOSET` for strict total orders introduced in Section 8.3.1.

There is a view from `TRIV` to `STRICT-WEAK-ORDER` that forgets the order and its properties. The name of this view coincides with the name of the target theory.

```

view STRICT-WEAK-ORDER from TRIV to STRICT-WEAK-ORDER is
endv

```

The inclusion from the theory `STRICT-WEAK-ORDER` into `STRICT-TOTAL-ORDER` gives rise to another view, which is also called as the target theory.

```

view STRICT-TOTAL-ORDER from STRICT-WEAK-ORDER
  to STRICT-TOTAL-ORDER is
endv

```

The Maude library includes views that map from `STRICT-TOTAL-ORDER` to built-in data type modules by selecting the main sort and the standard strict total order between the corresponding elements, namely, the “less than” comparison between numbers and the lexicographic ordering between strings, as described in previous sections. Again, operator mappings that are the identity (in this case of the form `op _<_ to _<_`) do not appear explicitly in the following views, but are left implicit. These views are named by appending “<” to the name of the selected sort; for example, the standard view from `STRICT-TOTAL-ORDER` into `RAT` is named `Rat<`.

```

view Nat< from STRICT-TOTAL-ORDER to NAT is
  sort Elt to Nat .
endv

```

```

view Int< from STRICT-TOTAL-ORDER to INT is
  sort Elt to Int .
endv

```

```

view Rat< from STRICT-TOTAL-ORDER to RAT is
  sort Elt to Rat .
endv

```

```

view Float< from STRICT-TOTAL-ORDER to FLOAT is
  sort Elt to Float .
endv

```

```

view String< from STRICT-TOTAL-ORDER to STRING is
  sort Elt to String .
endv

```

As explained in Section 8.3.2, these views impose some proof obligations corresponding in this case to the properties that are stated about the binary relation selected in the target module; recall that such proof obligations are not discharged or checked by the system.

9.11.4 TOTAL-PREORDER and TOTAL-ORDER

The predefined `TOTAL-PREORDER` theory specifies, as its name clearly suggests, a *total preorder*, that is, a total binary relation which is reflexive and transitive. This theory will also be used as requirement for sorting lists in Section 9.12.6.

The notions of strict weak order (see Section 9.11.3) and of total preorder are complementary: the set-theoretic complement of a strict weak order is a total preorder and vice versa. They can also be related in a way that preserves the direction of the order. Given a strict weak order $<$, a total preorder \leq is obtained by defining $x \leq y$ whenever $y \not< x$. In the other direction, a strict weak order $<$ is obtained from a total preorder \leq by defining $x < y$ whenever $y \not\leq x$.

Given a total preorder \leq , we say that two elements x and y are *equivalent* iff both $x \leq y$ and $y \leq x$. Then, it follows from the properties of a total preorder that this is an equivalence relation and, furthermore, two elements are equivalent in a total preorder if and only if they are incomparable in the associated strict weak order (we have seen in Section 9.11.3 that the incomparability relation \sim associated to a strict weak order is an equivalence relation).

Both kinds of relations capture the notion that the set of elements is split into partitions which are linearly ordered. This situation naturally arises when records are compared on a given field.

The theory `TOTAL-PREORDER`, as presented below, imports the theory `TRIV` and the module `BOOL`. The three equations express the required properties of the binary relation `_<=_` on the sort `Elt`.

```
fth TOTAL-PREORDER is
  protecting BOOL .
  including TRIV .
  op _<=_ : Elt Elt -> Bool .
  vars X Y Z : Elt .
  eq X <= X = true [nonexec label reflexive] .
  ceq X <= Z = true if X <= Y /\ Y <= Z [nonexec label transitive] .
  eq X <= Y or Y <= X = true [nonexec label total] .
endfth
```

A total *order* is a total preorder that, in addition, is antisymmetric.

```
fth TOTAL-ORDER is
  inc TOTAL-PREORDER .
  vars X Y : Elt .
  ceq X = Y if X <= Y /\ Y <= X [nonexec label antisymmetric] .
endfth
```

The theory `TOTAL-ORDER` is a different presentation of the equivalent theory `NSTOSET` for *non-strict* total orders introduced in Section 8.3.1. Its name follows the usual convention according to which, when nothing is said, a total order is assumed to be reflexive, that is, non-strict.

There is a view from `TRIV` to `TOTAL-PREORDER`, named like the target theory, that forgets the binary relation and its preorder properties.

```
view TOTAL-PREORDER from TRIV to TOTAL-PREORDER is
endv
```

The following view represents the inclusion from the `TOTAL-PREORDER` theory into `TOTAL-ORDER`.

```
view TOTAL-ORDER from TOTAL-PREORDER to TOTAL-ORDER is
endv
```

In the Maude prelude we can also find views that map from `TOTAL-ORDER` to several built-in data type modules by selecting the main sort and the standard non-strict total order between the corresponding elements, namely, the “less than or equal to” comparison between numbers and the lexicographic ordering between strings. These views are named by appending “<=” to the name of the selected sort; for example, the standard view from `TOTAL-ORDER` into `FLOAT` is named `Float<=`.

```
view Nat<= from TOTAL-ORDER to NAT is
  sort Elt to Nat .
endv
```

```
view Int<= from TOTAL-ORDER to INT is
  sort Elt to Int .
endv
```

```
view Rat<= from TOTAL-ORDER to RAT is
  sort Elt to Rat .
endv
```

```
view Float<= from TOTAL-ORDER to FLOAT is
  sort Elt to Float .
endv
```

```
view String<= from TOTAL-ORDER to STRING is
  sort Elt to String .
endv
```

Again, these views impose some proof obligations that are not discharged or checked by the system.

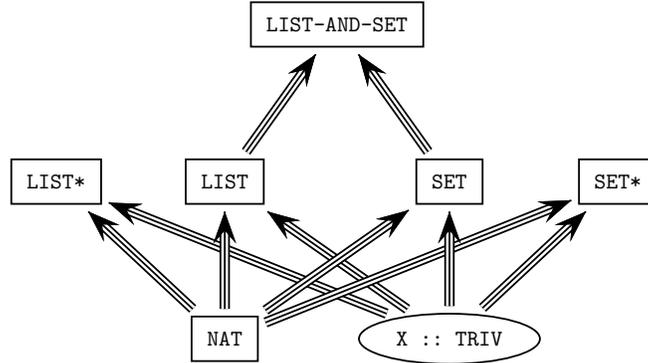


Fig. 9.2. Importation graph of parameterized list and set modules

9.12 Containers: lists and sets

The current Maude prelude includes two parameterized containers: *lists* and *sets*.

Figure 9.2 shows the relationships between the modules described in this section specifying parameterized lists and sets, including the theory `TRIV`. The module specifying sortable lists is not included in this figure, because its relationship is more complex than `protecting` importations (see later Figure 9.4).

Other container data types may be added to the Maude prelude in the future.

9.12.1 Lists

Lists over a given sort of elements (provided by the theory `TRIV`) are constructed from the constant `nil` (representing the empty list) and singleton lists (identified with the corresponding elements by means of a subsort declaration) by means of an *associative* concatenation operator written as juxtaposition with empty syntax `__`.

Since there are several operations that are not well defined over the empty list, it is most useful to define the subsort of non-empty lists.

```

fmod LIST{X :: TRIV} is
  protecting NAT .
  sorts NeList{X} List{X} .
  subsort X$Elt < NeList{X} < List{X} .

  op nil : -> List{X} [ctor] .
  op __ : List{X} List{X} -> List{X} [ctor assoc id: nil prec 25] .
  op __ : NeList{X} List{X} -> NeList{X} [ctor ditto] .

```

```

op _ : List{X} NeList{X} -> NeList{X} [ctor ditto] .

vars E E' : X$Elt .
vars A L : List{X} .
var C : Nat .

```

The operator `append` is just another name for concatenation.

```

op append : List{X} List{X} -> List{X} .
op append : NeList{X} List{X} -> NeList{X} .
op append : List{X} NeList{X} -> NeList{X} .
eq append(A, L) = A L .

```

The operations `head` and `tail` take and discard, respectively, the first (leftmost) element in a list. Analogously, the operations `last` and `front` take and discard, respectively, the last (rightmost) element in a list. It is enough to have one equation for each operation, because the case of a singleton list is obtained by matching modulo identity with `L = nil`.

```

op head : NeList{X} -> X$Elt .
eq head(E L) = E .

op tail : NeList{X} -> List{X} .
eq tail(E L) = L .

op last : NeList{X} -> X$Elt .
eq last(L E) = E .

op front : NeList{X} -> List{X} .
eq front(L E) = L .

```

The predicate `occurs` checks whether an element appears in any position in a list. The two equations in its specification correspond to the typical case analysis (or structural induction) over lists: either the list is empty or we consider the corresponding first element (in the latter case, again one equation is enough).

```

op occurs : X$Elt List{X} -> Bool .
eq occurs(E, nil) = false .
eq occurs(E, E' L) = if E == E' then true else occurs(E, L) fi .

```

Reversing a list is accomplished by means of the operator `reverse`, which is efficiently defined through an auxiliary operator `$reverse` that has an additional *accumulator* argument. With this argument, `$reverse` has a simple *tail-recursive* and thus efficient definition.

```

op reverse : List{X} -> List{X} .
op reverse : NeList{X} -> NeList{X} .
eq reverse(L) = $reverse(L, nil) .

```

```

op $reverse : List{X} List{X} -> List{X} .
eq $reverse(nil, A) = A .
eq $reverse(E L, A) = $reverse(L, E A) .

```

The tail-recursive method of definition just described will be used in the specification of several other operators, including the `size` operator on lists, which computes the number of elements in a list.

```

op size : List{X} -> Nat .
op size : NeList{X} -> NzNat .
eq size(L) = $size(L, 0) .

op $size : List{X} Nat -> Nat .
eq $size(nil, C) = C .
eq $size(E L, C) = $size(L, C + 1) .
endfm

```

In the Maude prelude there are two list instantiations on built-in data types (natural numbers and quoted identifiers) that are needed by the metalevel (see Chapter 14).

```

fmod NAT-LIST is
  protecting LIST{Nat} * (sort NeList{Nat} to NeNatList,
                        sort List{Nat} to NatList) .
endfm

fmod QID-LIST is
  protecting LIST{Qid} * (sort NeList{Qid} to NeQidList,
                       sort List{Qid} to QidList) .
endfm

```

Other instantiations can be built as desired. For example, we can use the view `Int` from `TRIV` to `INT`, and then test some reductions, as follows.

```

fmod INT-LIST is
  pr LIST{Int} .
endfm

Maude> red in INT-LIST : reverse(0 -1 2 -3 4 -5 6) .
result NeList{Int}: 6 -5 4 -3 2 -1 0

Maude> red occurs(7, 0 -1 2 -3 4 -5 6) .
result Bool: false

Maude> red size(0 -1 2 -3 4 -5 6) .
result NzNat: 7

```

9.12.2 Sets

Sets over a given sort of elements (provided by the theory `TRIV`) are built from the constant `empty` and singleton sets (identified with the corresponding elements by means of a subsort declaration) with an *associative, commutative, and idempotent* union operator written `_,_`. The first two such properties are declared as attributes, while the third is written as an equation; remember that the attributes `idem` and `assoc` cannot be used together (see Section 4.4.1).

```
fmod SET{X :: TRIV} is
  protecting EXT-BOOL .
  protecting NAT .
  sorts NeSet{X} Set{X} .
  subsort X$Elt < NeSet{X} < Set{X} .

  op empty : -> Set{X} [ctor] .
  op _,_ : Set{X} Set{X} -> Set{X}
    [ctor assoc comm id: empty prec 121 format (d r os d)] .
  op _,_ : NeSet{X} Set{X} -> NeSet{X} [ctor ditto] .

  var E : X$Elt .
  var N : NeSet{X} .
  vars A S S' : Set{X} .
  var C : Nat .

  eq N, N = N .
```

The prefix operator `union` is just another name for the infix operator `_,_`. Moreover, given the identification between elements and singleton sets, inserting an element is a particular case of union.

```
op union : Set{X} Set{X} -> Set{X} .
op union : NeSet{X} Set{X} -> NeSet{X} .
op union : Set{X} NeSet{X} -> NeSet{X} .
eq union(S, S') = S, S' .

op insert : X$Elt Set{X} -> Set{X} .
eq insert(E, S) = E, S .
```

The definitions of the operators `delete`, that deletes an element from a set, and `_in_`, that checks if an element belongs to a set, are based on the statement attribute `otherwise` (see Section 4.5.4):

1. When a given term representing a set matches the pattern `(E, S)` (modulo the equational attributes of the `_,_` operator), then we can delete the element `E` (and continue deleting, since there may be repetitions of such element in the given term), and state that indeed the element `E` belongs to the set.

2. *Otherwise*, the element E does not belong to the set and deleting such element does not change the set.

```

op delete : X$Elt Set{X} -> Set{X} .
eq delete(E, (E, S)) = delete(E, S) .
eq delete(E, S) = S [owise] .

```

```

op _in_ : X$Elt Set{X} -> Bool .
eq E in (E, S) = true .
eq E in S = false [owise] .

```

The operator $|_|$ computes the cardinality of a set. Its definition goes through an auxiliary operator $\$card$ with an additional accumulator argument that allows a tail-recursive definition. In turn, the specification of $\$card$ is based on an equation that eliminates repetitions of elements in a term representing a set; when such equation can no longer be applied (hence the *owise* attribute in the last equation), the accumulator argument does its job by counting once each different element.

```

op |_| : Set{X} -> Nat .
op |_| : NeSet{X} -> NzNat .
eq | S | = $card(S, 0) .

```

```

op $card : Set{X} Nat -> Nat .
eq $card(empty, C) = C .
eq $card((N, N, S), C) = $card((N, S), C) .
eq $card((E, S), C) = $card(S, C + 1) [owise] .

```

Both the intersection and set difference operations also use an auxiliary operation with a tail-recursive efficient definition. The accumulator argument keeps the elements that belong to both sets (for intersection) or to the first but not to the second set (for difference).

```

op intersection : Set{X} Set{X} -> Set{X} .
eq intersection(S, empty) = empty .
eq intersection(S, N) = $intersect(S, N, empty) .

```

```

op $intersect : Set{X} Set{X} Set{X} -> Set{X} .
eq $intersect(empty, S', A) = A .
eq $intersect((E, S), S', A)
  = $intersect(S, S', if E in S' then E, A else A fi) .

```

```

op _\_ : Set{X} Set{X} -> Set{X} [gather (E e)].
eq S \ empty = S .
eq S \ N = $diff(S, N, empty) .

```

```

op $diff : Set{X} Set{X} Set{X} -> Set{X} .
eq $diff(empty, S', A) = A .
eq $diff((E, S), S', A)
  = $diff(S, S', if E in S' then A else E, A fi) .

```

The following two predicates check whether their first argument is a (proper) subset of the second argument. The second one is defined in terms of the first, and in both cases the corresponding equations use the short-circuit version `_and-then_` of conjunction imported from the `EXT-BOOL` module.

```

op _subset_ : Set{X} Set{X} -> Bool .
eq empty subset S' = true .
eq (E, S) subset S' = E in S' and-then S subset S' .

op _psubset_ : Set{X} Set{X} -> Bool .
eq S psubset S' = S /= S' and-then S subset S' .
endfm

```

The Maude metalevel (see Chapter 14) imports a set instantiation on the built-in data type of quoted identifiers.

```

fmod QID-SET is
  protecting SET{Qid} * (sort NeSet{Qid} to NeQidSet,
                        sort Set{Qid} to QidSet) .
endfm

```

Another example of instantiation with some reductions is the following:

```

fmod INT-SET is
  pr SET{Int} .
endfm

Maude> red in INT-SET : | -1, 2, -3, 3, 2, -1 | .
result NzNat: 4

Maude> red 4 in (-1, 2, -3, 3, 2, -1) .
result Bool: false

Maude> red insert(4, (-1, 2, -3, 3, 2, -1)) .
result NeSet{Int}: 2, 3, 4, -1, -3

Maude> red union((2, 3, 4, -1, -3, 0), (-1, 2, -3, 3, 2, -1)) .
result NeSet{Int}: 0, 2, 3, 4, -1, -3

Maude> red intersection((2, 3, 4, -1, -3, 0),
                        (-1, 2, -3, 3, 2, -1)) .
result NeSet{Int}: 2, 3, -1, -3

Maude> red (2, 3, 4, -1, -3, 0) \ (-1, 2, -3, 3, 2, -1) .
result NeSet{Int}: 0, 4

```

9.12.3 Relating lists and sets

The following module provides some operations that involve both lists and sets; since these data types are not affected by the new operations, both of them are imported in `protecting` mode.

```
fmod LIST-AND-SET{X} :: TRIV} is
  protecting LIST{X} .
  protecting SET{X} .

  var E : X$Elt .
  vars A L : List{X} .
  var S : Set{X} .
```

The operation `makeSet` transforms a list into a set, that is, it forgets the order between the elements and its repetitions; operationally, it simply transforms the constructors `nil` and `__` for lists into the constructors `empty` and `_,_` for sets, but this is done in an efficient way by using an auxiliary operator `$makeSet` with an accumulator argument that allows a tail-recursive definition by structural induction on the list given as first argument. Notice that both operators are overloaded to take into account in their declarations whether their arguments are empty or not.

```
op makeSet : List{X} -> Set{X} .
op makeSet : NeList{X} -> NeSet{X} .
eq makeSet(L) = $makeSet(L, empty) .

op $makeSet : List{X} Set{X} -> Set{X} .
op $makeSet : NeList{X} Set{X} -> NeSet{X} .
op $makeSet : List{X} NeSet{X} -> NeSet{X} .
eq $makeSet(nil, S) = S .
eq $makeSet(E L, S) = $makeSet(L, (E, S)) .
```

An inverse operation `makeList` that transforms a set into a list will be seen in Section 9.12.7, because it only makes sense when we have additional information to put the elements of the set in a sequence in a univocally defined way.

The operations `filter` and `filterOut` take a list and a set as arguments, and return the list formed by those elements of the given list that belong and that do not belong, respectively, to the given set, in their original order. Again, both are defined by means of auxiliary operations with accumulator arguments allowing efficient tail-recursive definitions.

```
op filter : List{X} Set{X} -> List{X} .
eq filter(L, S) = $filter(L, S, nil) .

op $filter : List{X} Set{X} List{X} -> List{X} .
eq $filter(nil, S, A) = A .
eq $filter(E L, S, A)
```

```

= $filter(L, S, if E in S then A E else A fi) .

op filterOut : List{X} Set{X} -> List{X} .
eq filterOut(L, S) = $filterOut(L, S, nil) .

op $filterOut : List{X} Set{X} List{X} -> List{X} .
eq $filterOut(nil, S, A) = A .
eq $filterOut(E L, S, A)
  = $filterOut(L, S, if E in S then A else A E fi) .
endfm

```

For illustration, we consider the following instantiation and some reductions.

```

fmod INT-LIST-AND-SET is
  pr LIST-AND-SET{Int} .
endfm

Maude> red in INT-LIST-AND-SET : filter((1 -1 1 -2 1), (1, 2)) .
result NeList{Int}: 1 1 1

Maude> red filterOut((1 -1 1 -2 1), (1, 2)) .
result NeList{Int}: -1 -2

Maude> red makeSet(1 -1 1 -2 1) .
result NeSet{Int}: 1, -1, -2

```

9.12.4 Generalized lists

With the construction of parameterized lists described in Section 9.12.1, we can build, for example, lists of integers, or lists of lists of integers, but we cannot build lists in which we have as elements both integers and lists of integers; for this, we specify in this section the container of *generalized* or *nestable lists*.

In this specification we cannot use empty syntax in the same way as in Section 9.12.1, because we need something to distinguish the different levels of nesting of lists inside lists. We use an auxiliary sort `Item`, whose data are both elements and generalized lists (see the subsort declarations below); then we put such items next to each other by juxtaposition, getting in this way data of another auxiliary sort `PreList`, and finally we put square brackets around a “prelist” in order to get a generalized list. Notice that there is no empty “prelist” and that the empty generalized list `[]` is declared separately.

```

fmod LIST*{X :: TRIV} is
  protecting NAT .
  sorts Item{X} PreList{X} NeList{X} List{X} .
  subsort X$Elt List{X} < Item{X} < PreList{X} .
  subsort NeList{X} < List{X} .

```

```

op __ : PreList{X} PreList{X} -> PreList{X} [ctor assoc prec 25] .
op [] : PreList{X} -> NeList{X} [ctor] .
op [] : -> List{X} [ctor] .

vars A P : PreList{X} .
var L : List{X} .
vars E E' : Item{X} .
var C : Nat .

```

The operator `append` now corresponds to concatenation of generalized lists and its definition is based on the juxtaposition of the “prelists” inside the generalized lists.

```

op append : List{X} List{X} -> List{X} .
op append : NeList{X} List{X} -> NeList{X} .
op append : List{X} NeList{X} -> NeList{X} .
eq append([], L) = L .
eq append(L, []) = L .
eq append([P], [A]) = [P A] .

```

The operations `head`, `tail`, `last`, and `front` work as for “standard” lists, but now they refer to the first or last *item* in the list, which can be either an element or a nested list. Now we need two equations for each operation, because the singleton case needs to be treated separately (recall that there is no empty “prelist”).

```

op head : NeList{X} -> Item{X} .
eq head([E]) = E .
eq head([E P]) = E .

op tail : NeList{X} -> List{X} .
eq tail([E]) = [] .
eq tail([E P]) = [P] .

op last : NeList{X} -> Item{X} .
eq last([E]) = E .
eq last([P E]) = E .

op front : NeList{X} -> List{X} .
eq front([E]) = [] .
eq front([P E]) = [P] .

```

The predicate `occurs` checks whether an item (either an element or a list) appears in any position of the first level of a generalized list (but it does not go into deeper levels, that is, into nested lists). The three equations in its specification correspond to the typical case analysis (or structural induction) over these lists: either the list is empty, or it is a list with a single item, or it is a list with two or more items.

```

op occurs : Item{X} List{X} -> Bool .
eq occurs(E, []) = false .
eq occurs(E, [E']) = (E == E') .
eq occurs(E, [E' P])
  = if E == E' then true else occurs(E, [P]) fi .

```

The operators `reverse` and `size` for generalized lists work in a similar way to the operators with the same names in Section 9.12.1, and they are also defined by means of auxiliary operators `$reverse` and `$size`, respectively, with a tail-recursive definition. Notice, however, that these auxiliary operators work on “prelists” instead of lists. Moreover, `size` counts the number of items in the first level of a generalized list, but it does not count the items inside nested lists at deeper levels.

```

op reverse : List{X} -> List{X} .
op reverse : NeList{X} -> NeList{X} .
eq reverse([]) = [] .
eq reverse([E]) = [E] .
eq reverse([E P]) = [$reverse(P, E)] .

op $reverse : PreList{X} PreList{X} -> PreList{X} .
eq $reverse(E, A) = E A .
eq $reverse(E P, A) = $reverse(P, E A) .

op size : List{X} -> Nat .
op size : NeList{X} -> NzNat .
eq size([]) = 0 .
eq size([P]) = $size(P, 0) .

op $size : PreList{X} Nat -> NzNat .
eq $size(E, C) = C + 1 .
eq $size(E P, C) = $size(P, C + 1) .
endfm

```

We consider the following instantiation and sample reductions:

```

fmod INT-LIST* is
  pr LIST*{Int} .
endfm

Maude> red in INT-LIST* : append([1 []], [[] 2]) .
result NeList{Int}: [1 [] [] 2]

Maude> red reverse([[1 []] [[] 2]]) .
result NeList{Int}: [[[] 2] [1 []]]

Maude> red occurs(1, [[[] 2] [1 []]]) .
result Bool: false

Maude> red size([[[] 2] [1 []]]) .
result NzNat: 2

```

9.12.5 Generalized sets

The construction of generalized or nestable sets follows exactly the same pattern as the one we have seen for generalized lists in the previous section, but now we use braces instead of square brackets to make explicit the level of nesting. In particular, there is no empty “preset.” Note that braces `{_}` and comma `_,_` exactly reflect standard set theory notation.

Notice that the sort named `Element` plays here the same role as `Item` played for nestable lists; do not confuse this sort with the sort `Elt` coming from the parameter theory `TRIV` in the form `X$Elt`.

The module `SET*` provides for generalized sets the same operations we have seen in Section 9.12.2 for “standard” sets, and, in addition, it specifies a powerset operator that was not possible in the previous setting.

```
fmod SET*{X :: TRIV} is
  protecting EXT-BOOL .
  protecting NAT .
  sorts Element{X} PreSet{X} NeSet{X} Set{X} .
  subsort X$Elt Set{X} < Element{X} < PreSet{X} .
  subsort NeSet{X} < Set{X} .

  op _,_ : PreSet{X} PreSet{X} -> PreSet{X}
    [ctor assoc comm prec 121 format (d r o s d)] .
  op {_} : PreSet{X} -> NeSet{X} [ctor] .
  op {} : -> Set{X} [ctor] .

  vars P Q : PreSet{X} .
  vars A S : Set{X} .
  var E : Element{X} .
  var N : NeSet{X} .
  var C : Nat .

  eq {P, P} = {P} .
  eq {P, P, Q} = {P, Q} .
```

The operations for insertion, deletion, and membership testing now work for items that can be either basic elements or nested sets, but always at the first level of nesting. For example, the membership predicate `_in_` cannot be used to test if a basic element belongs to a set inside another set, but on the other hand can check if a set is a member of another set. In other words, the operation `_in_` exactly corresponds to the set theory membership predicate \in . As in Section 9.12.2, the operators `delete` and `_in_` are defined by means of the `otherwise` attribute. Moreover, each one has an additional equation for the singleton case, which is treated separately because there is no empty “preset.”

```
op insert : Element{X} Set{X} -> Set{X} .
eq insert(E, {}) = {E} .
```

```

eq insert(E, {P}) = {E, P} .

op delete : Element{X} Set{X} -> Set{X} .
eq delete(E, {E}) = {} .
eq delete(E, {E, P}) = delete(E, {P}) .
eq delete(E, S) = S [owise] .

op _in_ : Element{X} Set{X} -> Bool .
eq E in {E} = true .
eq E in {E, P} = true .
eq E in S = false [owise] .

```

The cardinality operator `|_|` computes the number of items (either basic elements or other sets, at the first level of nesting) in a given set. It is defined with the help of an auxiliary tail-recursive operator `$card` on “presets.”

```

op |_| : Set{X} -> Nat .
op |_| : NeSet{X} -> NzNat .
eq | {} | = 0 .
eq | {P} | = $card(P, 0) .

op $card : PreSet{X} Nat -> Nat .
eq $card(E, C) = C + 1 .
eq $card((N, N, P), C) = $card((N, P), C) .
eq $card((E, P), C) = $card(P, C + 1) [owise] .

```

The union operator `union` on generalized sets is based on the “union” operator `_,_` on the “presets” inside the generalized sets.

```

op union : Set{X} Set{X} -> Set{X} .
op union : NeSet{X} Set{X} -> NeSet{X} .
op union : Set{X} NeSet{X} -> NeSet{X} .
eq union({}, S) = S .
eq union(S, {}) = S .
eq union({P}, {Q}) = {P, Q} .

```

The intersection and set difference operations for generalized sets have a specification very similar to the one seen in Section 9.12.2, including the use of tail-recursive auxiliary operations on “presets”.

```

op intersection : Set{X} Set{X} -> Set{X} .
eq intersection({}, S) = {} .
eq intersection(S, {}) = {} .
eq intersection({P}, N) = $intersect(P, N, {}) .

op $intersect : PreSet{X} Set{X} Set{X} -> Set{X} .
eq $intersect(E, S, A) = if E in S then insert(E, A) else A fi .
eq $intersect((E, P), S, A)
  = $intersect(P, S, $intersect(E, S, A)) .

```

```

op _\_ : Set{X} Set{X} -> Set{X} [gather (E e)] .
eq {} \ S = {} .
eq S \ {} = S .
eq {P} \ N = $diff(P, N, {}) .

op $diff : PreSet{X} Set{X} Set{X} -> Set{X} .
eq $diff(E, S, A) = if E in S then A else insert(E, A) fi .
eq $diff((E, P), S, A) = $diff(P, S, $diff(E, S, A)) .

```

The powerset 2^X of a set X is computed by case analysis on the set X : it is either the empty set $\{\}$ or a singleton set $\{E\}$, or it has two or more items $\{E, P\}$. In the last case we compute the total powerset 2^X by computing first the powerset $2^{\{P\}}$ of the set without item E and then the union of this powerset $2^{\{P\}}$ with the result of inserting the distinguished item E into all the items in the same powerset $2^{\{P\}}$. The last process is done by means of an auxiliary operation $\$augment$.

```

op 2^_ : Set{X} -> Set{X} .
eq 2^{} = {{}} .
eq 2^{\{E\}} = {{}, {\{E\}}} .
eq 2^{\{E, P\}} = union(2^{\{P\}}, $augment(2^{\{P\}}, E, {})) .

op $augment : NeSet{X} Element{X} Set{X} -> Set{X} .
eq $augment(\{S\}, E, A) = insert(insert(E, S), A) .
eq $augment(\{S, P\}, E, A)
  = $augment(\{P\}, E, $augment(\{S\}, E, A)) .

```

The specification of the subset predicates that check whether a set is included in another is completely analogous to the specification of the corresponding operations in Section 9.12.2.

```

op _subset_ : Set{X} Set{X} -> Bool .
eq {} subset S = true .
eq {E} subset S = E in S .
eq {E, P} subset S = E in S and-then {P} subset S .

op _psubset_ : Set{X} Set{X} -> Bool .
eq A psubset S = A /= S and-then A subset S .
endfm

```

We consider the following instantiation and sample reductions:

```

fmod QID-SET* is
  pr SET*\{Qid} .
endfm

Maude> red in QID-SET* : {\a} in {{\a}, {\b}, {\a, \b}} .
result Bool: true

Maude> red | {{\a}, {\b}, {\a, \b}} | .
result NzNat: 3

```

```

Maude> red union({'a}, {'b}), {'a, 'b}) .
result NeSet{Qid}: {'a}, {'b}, {'a, 'b}}

Maude> red intersection({'a}, {'b}), {'a, 'b}) .
result Set{Qid}: {}

Maude> red 2^ {'a, 'b, 'c, 'd} .
result NeSet{Qid}:
  {}, {'a}, {'b}, {'c}, {'d}, {'a, 'b}, {'a, 'c}, {'a, 'd},
  {'b, 'c}, {'b, 'd}, {'c, 'd}, {'a, 'b, 'c}, {'a, 'b, 'd},
  {'a, 'c, 'd}, {'b, 'c, 'd}, {'a, 'b, 'c, 'd}}

```

9.12.6 Sortable lists

In Section 8.3.5 we defined the notion of sorted list requiring a totally ordered set of elements, but this requirement can be relaxed. In principle, it is enough to have a transitive and antisymmetric order $<$ on a set E of elements (which are the requirements in the theory `TAOSET` from Section 8.3.1) to be able to define a *sorted list* L over E as a list such that for every pair (u, v) of members in L with u occurring before v and with $u \neq v$, it is the case that $v < u$ is false. However, in what follows we are not interested in defining sorted lists, but in specifying a sorting algorithm (more specifically, the *mergesort* algorithm) in a deterministic way. We require the sorting algorithm to be *stable*, so that incomparable elements remain in the same relative order as in the list provided as argument. For this notion to be well defined, we need to require either a strict weak order or a total preorder.

Sorting lists with respect to a strict weak order

Assume first that $<$ is a *strict weak order* over a set E , that is, a strict partial order with a transitive incomparability relation, which are precisely the requirements in the predefined theory `STRICT-WEAK-ORDER` of Section 9.11.3.

In order to define a *stable sorting* of a list L of elements over E , we consider each element of the list L as a pair (x, i) , where x is the value of the element in E and i is the number indicating the position of x in L . We define an ordering \ll on such pairs as follows: $(x, i) \ll (y, j)$ iff either $x < y$ or $(x \sim y$ and $i < j)$. Then, it follows from the properties of $<$ and \sim that \ll is a strict total order, i.e., it is irreflexive, transitive, and total.

We can now define the stable sorting under $<$ of a list e_1, e_2, \dots, e_n of elements from E as follows: Take the list $(e_1, 1), (e_2, 2), \dots, (e_n, n)$, find its unique ordering $(e_{s_1}, s_1), (e_{s_2}, s_2), \dots, (e_{s_n}, s_n)$ under \ll , and output $e_{s_1}, e_{s_2}, \dots, e_{s_n}$.

The parameterized module `WEAKLY-SORTABLE-LIST`, that specifies a stable version of mergesort on lists, imports “standard” lists (from Section 9.12.1), but first it is necessary to match the parameter theory `TRIV` of lists with the parameter theory `STRICT-WEAK-ORDER`. This is accomplished by means of the predefined view `STRICT-WEAK-ORDER` from `TRIV` to `STRICT-WEAK-ORDER` that forgets the order and its properties (see Section 9.11.3). A renaming is also

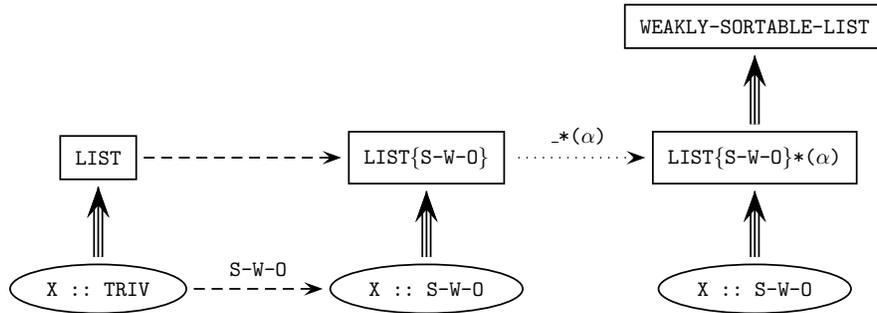


Fig. 9.3. From lists to weakly sortable lists

applied to this instantiation in order to have more convenient sort names. This process is illustrated in the diagram of Figure 9.3, where `STRICT-WEAK-ORDER` has been abbreviated to `S-W-O`, the sort renaming has been abbreviated to α , and where the different types of arrows represent the different relationships between modules: importation (triple arrow), views between theories (dashed arrow named `S-W-O`), instantiation (dashed arrow), and renaming (dotted arrow named $_*(\alpha)$, meaning the renaming whose second argument is α and whose first argument is still unknown).

```
fmod WEAKLY-SORTABLE-LIST{X :: STRICT-WEAK-ORDER} is
  pr LIST{STRICT-WEAK-ORDER}{X}
    * (sort NeList{STRICT-WEAK-ORDER}{X} to NeList{X},
      sort List{STRICT-WEAK-ORDER}{X} to List{X}) .
  sort $Split{X} .

  vars E E' : X$Elt .
  vars A A' L L' : List{X} .
  var N : NeList{X} .
```

The main operation in this module is `sort`, that sorts a given list.⁵ It is defined by case analysis on the list: if it is either the empty list or a singleton list, then it is already sorted; otherwise, we split the given list into two sublists, recursively sort both of them, and then merge the sorted results in order to obtain the final sorted list. This process is accomplished by means of three auxiliary operations, whose names are self-explanatory: `$split` (for the splitting, with an auxiliary result sort `$Split`), `$sort` (for the recursive sorting calls), and `$merge` (for the final merging).

```
op sort : List{X} -> List{X} .
op sort : NeList{X} -> NeList{X} .
```

⁵ We realize that terminology here can be a bit confusing, because in Maude `sort` is also a keyword for types.

```

eq sort(nil) = nil .
eq sort(E) = E .
eq sort(E N) = $sort($split(E N, nil, nil)) .

op $sort : $Split{X} -> List{X} .
eq $sort($split(nil, L, L')) = $merge(sort(L), sort(L'), nil) .

```

The auxiliary operation `$split` has three arguments: the first one is the list to be split and the other two are accumulators (initially both empty) that keep the elements as they are moved from the main list into the appropriate sublists. In this way, we have an efficient tail-recursive definition.

```

op $split : List{X} List{X} List{X} -> $Split{X} [ctor] .
eq $split(E, A, A') = $split(nil, A E, A') .
eq $split(E L E', A, A') = $split(L, A E, E' A') .

```

The auxiliary operation `$merge` also has three arguments, but now the first two are the lists to be merged and the third one is the accumulator where the result is incrementally computed by means of another efficient tail-recursive definition.

The module also provides an operation `merge` that simply calls the previous operation with the empty accumulator. Notice that if both lists are sorted then the result of calling `merge` on them is a sorted list, but in general `merge` is a total function that can be called on any two lists whatsoever.

```

op merge : List{X} List{X} -> List{X} .
op merge : NeList{X} List{X} -> NeList{X} .
op merge : List{X} NeList{X} -> NeList{X} .
eq merge(L, L') = $merge(L, L', nil) .

op $merge : List{X} List{X} List{X} -> List{X} .
eq $merge(L, nil, A) = A L .
eq $merge(nil, L, A) = A L .
eq $merge(E L, E' L', A)
  = if E' < E
    then $merge(E L, L', A E')
    else $merge(L, E' L', A E)
  fi .
endfm

```

The Maude prelude also provides another predefined module for sorting lists, namely, `SORTABLE-LIST`, where the required order is strict and total, as specified in the predefined theory `STRICT-TOTAL-ORDER` of Section 9.11.3. Since the theory `STRICT-TOTAL-ORDER` is a strengthening of `STRICT-WEAK-ORDER` with the additional requirement of totality, we can use it as a parameter theory to specialize our `WEAKLY-SORTABLE-LIST` module to strict total orders, thus getting the `SORTABLE-LIST` module. For

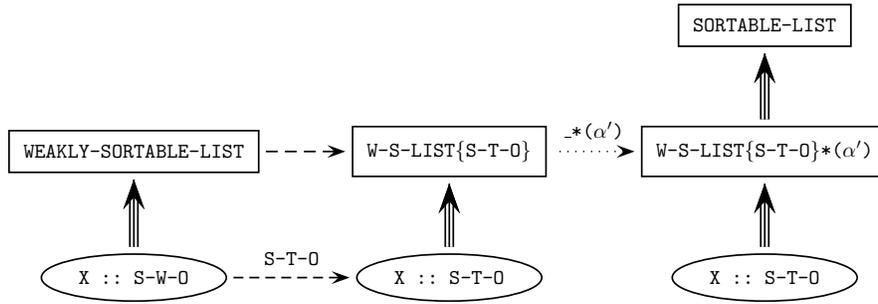


Fig. 9.4. From weakly sortable lists to sortable lists

this we need a view from the theory `STRICT-WEAK-ORDER` into the theory `STRICT-TOTAL-ORDER`, which is precisely the predefined inclusion view `STRICT-TOTAL-ORDER` in Section 9.11.3.

Moreover, since we also use another renaming to have more convenient sort names, the construction of the parameterized module `SORTABLE-LIST` on top of `WEAKLY-SORTABLE-LIST` mirrors the process of constructing `WEAKLY-SORTABLE-LIST` on top of `LIST`, as described in Figure 9.4, where the sort renaming has been abbreviated to α' , `WEAKLY-SORTABLE-LIST` to `W-S-LIST`, `STRICT-WEAK-ORDER` to `S-W-O`, and `STRICT-TOTAL-ORDER` to `S-T-O`. The reader should compare this figure with Figure 9.3 to appreciate the similarity between both.

```
fmod SORTABLE-LIST{X :: STRICT-TOTAL-ORDER} is
  pr WEAKLY-SORTABLE-LIST{STRICT-TOTAL-ORDER}{X}
    * (sort NeList{STRICT-TOTAL-ORDER}{X} to NeList{X},
      sort List{STRICT-TOTAL-ORDER}{X} to List{X}) .
endfm
```

We can use the predefined view `String<` from `STRICT-TOTAL-ORDER` to `String` (where `<` is the lexicographic order on strings) to instantiate the previous module before doing some sample reductions.

```
fmod STRING-SORTABLE-LIST is
  pr SORTABLE-LIST{String<} .
endfm
```

```
Maude> red in STRING-SORTABLE-LIST :
  $split("a" "quick" "brown" "fox" "jumps"
        "over" "the" "lazy" "dog", nil, nil) .
result $Split{STRICT-TOTAL-ORDER}{String<}:
  $split(nil,
    "a" "quick" "brown" "fox" "jumps",
    "over" "the" "lazy" "dog")
```

```
Maude> red merge("a" "quick" "brown" "fox" "jumps",
```

```

    "over" "the" "lazy" "dog") .
result NeList{String<}:
  "a" "over" "quick" "brown" "fox" "jumps" "the" "lazy" "dog"
Maude> red sort("a" "quick" "brown" "fox" "jumps"
               "over" "the" "lazy" "dog") .
result NeList{String<}:
  "a" "brown" "dog" "fox" "jumps" "lazy" "over" "quick" "the"
Maude> red sort("a" "quick" "brown" "fox" "jumps" "over" "the"
               "lazy" "dog" "a" "quick" "brown" "fox" "jumps"
               "over" "the" "lazy" "dog") .
result NeList{String<}: "a" "a" "brown" "brown" "dog" "dog" "fox"
  "fox" "jumps" "jumps" "lazy" "lazy" "over" "over" "quick" "quick"
  "the" "the"

```

Sorting lists with respect to a total preorder

Assume now that \leq is a *total preorder* over a set E , that is, a binary relation satisfying the requirements in the predefined theory TOTAL-PREORDER of Section 9.11.4.

To define a stable sorting of a list L of elements over E , we consider again each element of the list L as a pair (x, i) , where x is the value of the element in E and i is the number indicating the position of x in L . We define an ordering \ll on such pairs as follows, where now the definition of \ll is slightly different given the non-strict nature of total preorders: $(x, i) \ll (y, j)$ iff either $y \not\leq x$ or $(x \leq y$ and $i \leq j)$. Then, the properties of \leq imply that \ll is a (non-strict) total order, i.e., it is reflexive, anti-symmetric, transitive, and total. From this, the definition of a stable sorting under \leq of a list e_1, e_2, \dots, e_n of elements from E follows exactly the same steps as before: Take the list $(e_1, 1), (e_2, 2), \dots, (e_n, n)$, find its unique ordering $(e_{s_1}, s_1), (e_{s_2}, s_2), \dots, (e_{s_n}, s_n)$ under \ll , and output $e_{s_1}, e_{s_2}, \dots, e_{s_n}$.

The following modules WEAKLY-SORTABLE-LIST' and SORTABLE-LIST' specify the mergesort algorithm with respect to a total preorder and a (non-strict) total order, respectively. Their structure is completely analogous to the structure of WEAKLY-SORTABLE-LIST and SORTABLE-LIST already explained above. It is described in the two diagrams of Figure 9.5, where the sort remainings have been abbreviated to γ and γ' , TOTAL-PREORDER to T-PREORDER, TOTAL-ORDER to T-ORDER, and WEAKLY-SORTABLE-LIST' W-S-LIST'.

```

fmod WEAKLY-SORTABLE-LIST'{X :: TOTAL-PREORDER} is
  pr LIST{TOTAL-PREORDER}{X}
    * (sort NeList{TOTAL-PREORDER}{X} to NeList{X},
       sort List{TOTAL-PREORDER}{X} to List{X}) .
  sort $Split{X} .

vars E E' : X$Elt .
vars A A' L L' : List{X} .
var N : NeList{X} .

```

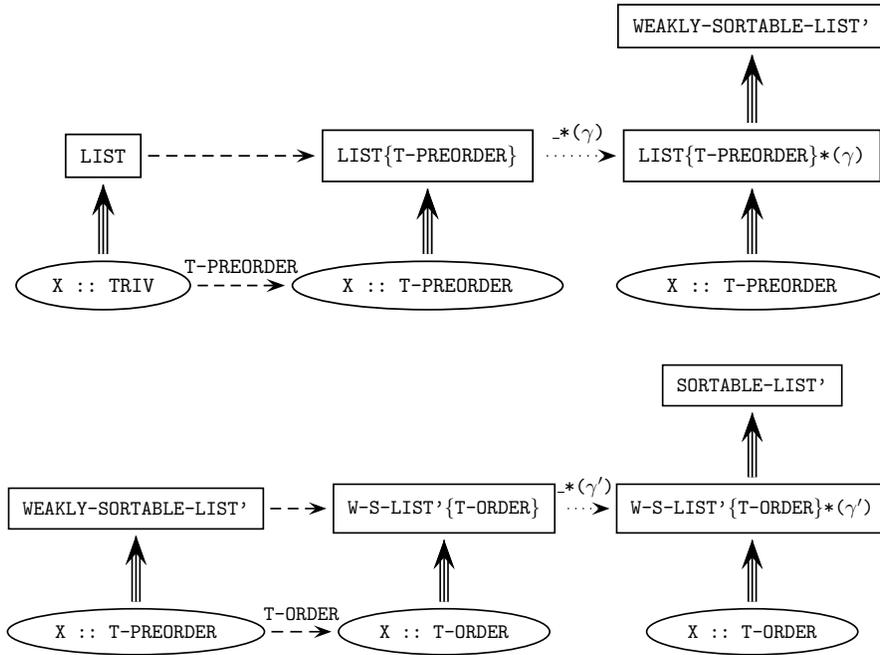


Fig. 9.5. Another version of sortable lists

```

op sort : List{X} -> List{X} .
op sort : NeList{X} -> NeList{X} .
eq sort(nil) = nil .
eq sort(E) = E .
eq sort(E N) = $sort($split(E N, nil, nil)) .

op $sort : $Split{X} -> List{X} .
eq $sort($split(nil, L, L')) = $merge(sort(L), sort(L'), nil) .

op $split : List{X} List{X} List{X} -> $Split{X} [ctor] .
eq $split(E, A, A') = $split(nil, A E, A') .
eq $split(E L E', A, A') = $split(L, A E, E' A') .

op merge : List{X} List{X} -> List{X} .
op merge : NeList{X} List{X} -> NeList{X} .
op merge : List{X} NeList{X} -> NeList{X} .
eq merge(L, L') = $merge(L, L', nil) .

op $merge : List{X} List{X} List{X} -> List{X} .
eq $merge(L, nil, A) = A L .

```

```

eq $merge(nil, L, A) = A L .
eq $merge(E L, E' L', A)
  = if E <= E'
    then $merge(L, E' L', A E)
    else $merge(E L, L', A E')
  fi .
endfm

fmod SORTABLE-LIST' {X :: TOTAL-ORDER} is
pr WEAKLY-SORTABLE-LIST' {TOTAL-ORDER} {X}
  * (sort NeList {TOTAL-ORDER} {X} to NeList {X},
     sort List {TOTAL-ORDER} {X} to List {X}) .
endfm

```

Apart from the changes in the requirement theories and the module names, the main difference between both approaches appears in the third `$merge` equation. In the `WEAKLY-SORTABLE-LIST` module we have

```

eq $merge(E L, E' L', A)
  = if E' < E
    then $merge(E L, L', A E')
    else $merge(L, E' L', A E)
  fi .

```

Here we are dealing with a strict weak order. We test $E' < E$. If it is true, then by irreflexivity we know that $E < E'$ is false, and the element E' from the second list is appended to the merged list. Whereas if $E' < E$ is false, we know that either $E < E'$ holds or E and E' are incomparable. Either way, the element E from the first list is appended to the merged list, either because it is smaller or because it is incomparable and we are preserving the original relative positions in the list (stability).

On the other hand, in the `WEAKLY-SORTABLE-LIST'` module we have

```

eq $merge(E L, E' L', A)
  = if E <= E'
    then $merge(L, E' L', A E)
    else $merge(E L, L', A E')
  fi .

```

In this case we are dealing with a total preorder. We test $E \leq E'$. If it is true, then either $E' \leq E$ is false or E and E' are equivalent. Either way, the element E from the first list is appended to the merged list, either because it is smaller or because it is equivalent and we are preserving the original relative positions in the list (stability). If $E \leq E'$ is false, then $E' \leq E$ holds by totality and therefore E' is appended to the merged list.

We can redo with these modules the same instantiation we considered above, but using now the predefined view `String<=` from `TOTAL-ORDER` to `String`, where `<=` is the non-strict lexicographic order on strings.

```

fmod STRING-SORTABLE-LIST' is
  pr SORTABLE-LIST'{String<=} .
endfm

Maude> red in STRING-SORTABLE-LIST' :
      sort("a" "quick" "brown" "fox" "jumps"
           "over" "the" "lazy" "dog") .
result NeList{String<=}:
  "a" "brown" "dog" "fox" "jumps" "lazy" "over" "quick" "the"

Maude> red sort("a" "quick" "brown" "fox" "jumps" "over" "the"
              "lazy" "dog" "a" "quick" "brown" "fox" "jumps"
              "over" "the" "lazy" "dog") .
result NeList{String<=}: "a" "a" "brown" "brown" "dog" "dog" "fox"
  "fox" "jumps" "jumps" "lazy" "lazy" "over" "over" "quick" "quick"
  "the" "the"

```

9.12.7 Making lists out of sets

In Section 9.12.3 we have seen an operation `makeSet` that transforms a list into a set with the same elements. On the other hand, transforming a set into a list imposes some order on the given elements, which can be done in many different ways, and therefore only makes sense as a function when we have additional information over those elements that allows us to choose a unique sequence. The solution adopted here is to require either a strict or a non-strict total order on the elements, so that the resulting list is the corresponding sorted list. For this we use the `sort` operation defined either in the `SORTABLE-LIST` module or in the `SORTABLE-LIST'` module described in the previous section. In both versions the main operation `makeList` is defined in terms of an auxiliary operation `$makeList` with an accumulator in order to have a more efficient definition.

In both versions the `LIST-AND-SET` module is imported with a double renaming (different in each case), which is needed for correct sharing of a renamed copy of the `LIST` module, because Core Maude does not evaluate the composition of renamings but applies them sequentially. If we computed manually and used this simpler renaming, we would get a different renaming of `LIST` imported by each `protecting` declaration; then, while these renamings would have the same effect, we would import two renamed copies of `LIST` rather than a shared copy.

This is the first version, using a strict total order.

```

fmod SORTABLE-LIST-AND-SET{X :: STRICT-TOTAL-ORDER} is
  pr SORTABLE-LIST{X} .
  pr LIST-AND-SET{STRICT-WEAK-ORDER}{STRICT-TOTAL-ORDER}{X}
    * (sort NeList{STRICT-WEAK-ORDER}{STRICT-TOTAL-ORDER}{X}
       to NeList{STRICT-TOTAL-ORDER}{X},
       sort List{STRICT-WEAK-ORDER}{STRICT-TOTAL-ORDER}{X}

```

```

    to List{STRICT-TOTAL-ORDER}{X})
* (sort NeList{STRICT-TOTAL-ORDER}{X} to NeList{X},
   sort List{STRICT-TOTAL-ORDER}{X} to List{X},
   sort NeSet{STRICT-WEAK-ORDER}{STRICT-TOTAL-ORDER}{X}
   to NeSet{X},
   sort Set{STRICT-WEAK-ORDER}{STRICT-TOTAL-ORDER}{X}
   to Set{X}) .

var E : X$Elt .
var L : List{X} .
var S : Set{X} .

op makeList : Set{X} -> List{X} .
op makeList : NeSet{X} -> NeList{X} .
eq makeList(S) = $makeList(S, nil) .

op $makeList : Set{X} List{X} -> List{X} .
op $makeList : NeSet{X} List{X} -> NeList{X} .
op $makeList : Set{X} NeList{X} -> NeList{X} .
eq $makeList((E, E, S), L) = $makeList((E, S), L) .
eq $makeList((E, S), L) = $makeList(S, E L) [owise] .
endfm

```

Notice that `makeList` is only a partial inverse to `makeSet`, not only because of sorting the elements, but also because in a set repetitions do not matter. In general, for a set `S` and a list `L` we have `makeSet(makeList(S)) = S`, but in general `makeList(makeSet(L)) ≠ L`.

We consider an instantiation with the predefined view `Int<` and some sample reductions.

```

fmod INT-SORTABLE-LIST-AND-SET is
  pr SORTABLE-LIST-AND-SET{Int<} .
endfm

```

Notice that in the following first reduction we get a list different from the original one, while in the second reduction we get a different representation (where repetitions have been eliminated) of the same set. Those possible repetitions are already eliminated before producing the corresponding list, as shown in the third reduction.

```

Maude> red in INT-SORTABLE-LIST-AND-SET :
      makeList(makeSet(1 -1 1 -2 1 0)) .
result NeList{Int<}: -2 -1 0 1

Maude> red makeSet(makeList((5, 4, 3, 4, 5))) .
result NeSet{Int<}: 3, 4, 5

Maude> red makeList((5, 4, 3, 4, 5)) .
result NeList{Int<}: 3 4 5

```

This is the second version, using a non-strict total order.

```
fmod SORTABLE-LIST-AND-SET'{X :: TOTAL-ORDER} is
pr SORTABLE-LIST'{X} .
pr LIST-AND-SET{TOTAL-PREORDER}{TOTAL-ORDER}{X}
  * (sort NeList{TOTAL-PREORDER}{TOTAL-ORDER}{X}
     to NeList{TOTAL-ORDER}{X},
     sort List{TOTAL-PREORDER}{TOTAL-ORDER}{X}
     to List{TOTAL-ORDER}{X})
  * (sort NeList{TOTAL-ORDER}{X} to NeList{X},
     sort List{TOTAL-ORDER}{X} to List{X},
     sort NeSet{TOTAL-PREORDER}{TOTAL-ORDER}{X} to NeSet{X},
     sort Set{TOTAL-PREORDER}{TOTAL-ORDER}{X} to Set{X}) .

var E : X$Elt .
var L : List{X} .
var S : Set{X} .

op makeList : Set{X} -> List{X} .
op makeList : NeSet{X} -> NeList{X} .
eq makeList(S) = $makeList(S, nil) .

op $makeList : Set{X} List{X} -> List{X} .
op $makeList : NeSet{X} List{X} -> NeList{X} .
op $makeList : Set{X} NeList{X} -> NeList{X} .
eq $makeList(empty, L) = sort(L) .
eq $makeList((E, E, S), L) = $makeList((E, S), L) .
eq $makeList((E, S), L) = $makeList(S, E L) [owise] .
endfm
```

We redo the same instantiation, now with the non-strict total order on integers.

```
fmod INT-SORTABLE-LIST-AND-SET' is
pr SORTABLE-LIST-AND-SET'{Int<=} .
endfm

Maude> red in INT-SORTABLE-LIST-AND-SET' :
  makeList(makeSet(1 -1 1 -2 1 0)) .
result NeList{Int<=}: -2 -1 0 1

Maude> red makeSet(makeList((5, 4, 3, 4, 5))) .
result NeSet{Int<=}: 3, 4, 5

Maude> red makeList((5, 4, 3, 4, 5)) .
result NeList{Int<=}: 3 4 5
```

9.13 Maps and arrays

Both *maps* and *arrays* represent a function f between two sets as a set of pairs of the form

$$\{(a_1, f(a_1)), (a_2, f(a_2)), \dots, (a_n, f(a_n))\}$$

in the graph of the function; each pair $(a_i, f(a_i))$ is called an *entry* in both cases.

The difference between maps and arrays is that the former leave undefined the result of f over those values not present in the set above, while the latter assign a “default” value result in that case.

However, notice that the modules below *do not check*, for efficiency reasons, that all values a_i in the first components of a set of pairs like the previous one are *different* (although the operations for insertion and look up make sure that the corresponding result is well defined). The situation of having a set of entries with repeated first components never arises if such a map or array is initially the empty one and then it is only modified by means of the `insert` operation. See Section 19.3.2 for a more careful specification of (finite) partial functions checking these requirements.

9.13.1 Maps

As explained above, a map is defined as a “set” (built with the associative and commutative operator `_ , _`) of entries. Notice that `Entry`, whose only constructor is the operator `_|->_`, is a subsort of `Map`.

The domain and codomain values of the map come from the parameters of the parameterized data type, both of them satisfying the theory `TRIV` and thus providing a set of elements.

The module `MAP` provides a constant `undefined` of the *kind* `[Y$Elt]` corresponding to the sort `Y$Elt` and representing the undefined result.

```
fmod MAP{X :: TRIV, Y :: TRIV} is
  protecting BOOL .
  sorts Entry{X,Y} Map{X,Y} .
  subsort Entry{X,Y} < Map{X,Y} .

  op _|->_ : X$Elt Y$Elt -> Entry{X,Y} [ctor] .
  op empty : -> Map{X,Y} [ctor] .
  op _ , _ : Map{X,Y} Map{X,Y} -> Map{X,Y}
    [ctor assoc comm id: empty prec 121 format (d r os d)] .
  op undefined : -> [Y$Elt] [ctor] .

  var D : X$Elt .
  vars R R' : Y$Elt .
  var M : Map{X,Y} .
```

The operator `insert` adds a new entry to a map, but when the first argument already appears in the domain of definition of the map, the second argument is used to *update* the map. Notice the use of matching and of the `otherwise` attribute to distinguish these two cases in a simple way. Furthermore, in the first case, an auxiliary operation `$hasMapping` is used to make

sure that in the resulting map only one entry is associated with the given value. The operation `$hasMapping` checks whether a domain value actually has an associated entry in a map.

```

op insert : X$Elt Y$Elt Map{X,Y} -> Map{X,Y} .
eq insert(D, R, (M, D |-> R'))
  = if $hasMapping(M, D)
    then insert(D, R, M)
    else (M, D |-> R)
    fi .
eq insert(D, R, M) = (M, D |-> R) [owise] .

op $hasMapping : Map{X,Y} X$Elt -> Bool .
eq $hasMapping((M, D |-> R), D) = true .
eq $hasMapping(M, D) = false [owise] .

```

The lookup operator is represented with the notation `_[_]`. Again, `match` and `owise` are used to distinguish whether or not the second argument appears in the domain of definition of the map provided as first argument. When the answer is affirmative and the map contains exactly one entry associated with such argument (as checked with the auxiliary operation `$hasMapping`), the result is the value provided in that entry. When the answer is negative or the map is not well defined because there is more than one entry associated with the same argument, the result is the constant `undefined` in the kind, with the self-explanatory meaning that in those cases the map is undefined on the given argument.

```

op _[_] : Map{X,Y} X$Elt -> [Y$Elt] [prec 23] .
eq (M, D |-> R)[D]
  = if $hasMapping(M, D) then undefined else R fi .
eq M[D] = undefined [owise] .
endfm

```

We use the predefined views `String` and `Nat` (see Section 9.11.1) to define maps from strings to natural numbers, and do some sample reductions.

```

fmod STRING-NAT-MAP is
  pr MAP{String, Nat} .
endfm

Maude> red in STRING-NAT-MAP :
      insert("one", 1,
            insert("two", 2, insert("three", 3, empty))) .
result Map{String,Nat}: "one" |-> 1, "three" |-> 3, "two" |-> 2

Maude> red insert("one", 1,
                 insert("two", 2,
                       insert("three", 3, empty)))[ "two" ] .
result NzNat: 2

```

```
Maude> red insert("one", 1,
                insert("two", 2,
                    insert("three", 3, empty)))["four"] .
result [FindResult]: undefined

Maude> red ("a" |-> 3, "a" |-> 2)["a"] .
result [FindResult]: undefined
```

The last reduction shows that the undesired repetition of a domain value in two entries of the same map also produces the `undefined` constant as result.

9.13.2 Arrays

As explained above, arrays work like maps, with the difference that an attempt to look up an unmapped value always returns the default value, i.e., arrays have a *sparse array* behavior (hence the name). In the same spirit, mappings to the default value are never inserted.

The main difference between maps and arrays is already made explicit in the parameters of the parameterized data type: while the first one satisfies the theory `TRIV`, the second one satisfies the theory `DEFAULT` that in addition to a set of data provides a default value 0 (see Section 9.11.2).

The constructor for entries is named `_|->_`, as for maps, while the set constructor is denoted here `_;_`.

```
fmod ARRAY{X :: TRIV, Y :: DEFAULT} is
  protecting BOOL .
  sorts Entry{X,Y} Array{X,Y} .
  subsort Entry{X,Y} < Array{X,Y} .

  op _|->_ : X$Elt Y$Elt -> Entry{X,Y} [ctor] .
  op empty : -> Array{X,Y} [ctor] .
  op _;_ : Array{X,Y} Array{X,Y} -> Array{X,Y}
    [ctor assoc comm id: empty prec 71 format (d r os d)] .

  var D : X$Elt .
  vars R R' : Y$Elt .
  var A : Array{X,Y} .
```

The definition of the operator `insert` for arrays adds a check to the definition of the same operator for maps so that, as mentioned above, entries whose second value is the default value 0 are never inserted. Note, however, that mappings to the default value 0 that are created with the constructors `_|->_` and `_;_`, rather than the `insert` operator, are not removed as doing this check each time a new array is formed would be excessively inefficient. Furthermore, as we have already seen for maps, in the first case, an auxiliary operation `$hasMapping` is used to make sure that in the resulting array only one entry is associated with the given value.

```

op insert : X$Elt Y$Elt Array{X,Y} -> Array{X,Y} .
eq insert(D, R, (A ; D |-> R'))
  = if $hasMapping(A, D)
    then insert(D, R, A)
    else if R == 0 then A else (A ; D |-> R) fi
  fi .
eq insert(D, R, A)
  = if R == 0 then A else (A ; D |-> R) fi [owise] .

op $hasMapping : Array{X,Y} X$Elt -> Bool .
eq $hasMapping((A ; D |-> R), D) = true .
eq $hasMapping(A, D) = false [owise] .

```

The definition of the lookup operator for arrays only differs from the one for maps in the occurrence of the default value 0 instead of the constant `undefined`. Now, if an argument has more than one associated entry (as checked with the auxiliary operation `$hasMapping`), it is considered to be “unmapped” and the result is also the default value.

```

op _[_] : Array{X,Y} X$Elt -> Y$Elt [prec 23] .
eq (A ; D |-> R)[D]
  = if $hasMapping(A, D) then 0 else R fi .
eq A[D] = 0 [owise] .
endfm

```

We do the same instantiation for arrays as for maps, with the predefined views `String` from Section 9.11.1 and `Nat0` from Section 9.11.2).

```

fmod STRING-NAT-ARRAY is
  pr ARRAY{String, Nat0} .
endfm

Maude> red in STRING-NAT-ARRAY :
  insert("one", 1,
    insert("two", 2, insert("three", 3, empty))) .
result Array{String,Nat0}: "one" |-> 1 ; "three" |-> 3 ; "two" |-> 2

Maude> red insert("one", 0,
  insert("two", 2, insert("three", 3, empty))) .
result Array{String,Nat0}: "three" |-> 3 ; "two" |-> 2

Maude> red insert("one", 1,
  insert("two", 2,
    insert("three", 3, empty)))[ "two" ] .
result NzNat: 2

Maude> red insert("one", 1,
  insert("two", 2,
    insert("three", 3, empty)))[ "four" ] .
result Zero: 0

```

9.14 A linear Diophantine equation solver

The Maude system includes a built-in linear Diophantine equation solver. The interface to the solver is defined in the file `linear.maude` which contains the functional module `DIOPHANTINE`. The current solver finds non-negative solutions of a system S of n simultaneous linear equations in m variables having the form $Mv = c$, where M is an $n \times m$ integer coefficient matrix, v is a column vector of m variables and c is a column vector of n integer constants.

Both matrices and vectors are represented as sparse arrays with their dimensions implicit and their indices starting from 0. For this we make heavy use of the parameterized module `ARRAY`, described in Section 9.13.2.

First, a data type of pairs of natural numbers to be used as indices for matrices is created.

```
fmod INDEX-PAIR is
  pr NAT .
  sort IndexPair .
  op _,_ : Nat Nat -> IndexPair [ctor] .
endfm
```

Then, we instantiate (and rename as desired) the parameterized module `ARRAY` to obtain matrices of integers. Notice that `Int0` is the view from `DEFAULT` to `INT` given in Section 9.11.2

```
view IndexPair from TRIV to INDEX-PAIR is
  sort Elt to IndexPair .
endv
```

```
fmod MATRIX{X :: DEFAULT} is
  pr (ARRAY * (sort Entry{X,Y} to Entry{Y},
              sort Array{X,Y} to Matrix{Y}))
      {IndexPair, X} .
endfm
```

```
fmod INT-MATRIX is
  pr MATRIX{Int0} * (sort Entry{Int0} to IntMatrixEntry,
                    sort Matrix{Int0} to IntMatrix,
                    op empty to zeroMatrix) .
endfm
```

For example, the matrices

$$\begin{pmatrix} 1 & 2 \\ 0 & -1 \end{pmatrix} \quad \begin{pmatrix} 1 & 2 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

are both represented by the same term

$$(0,0) \mapsto 1 ; (0,1) \mapsto 2 ; (1,1) \mapsto -1$$

Vectors are represented in a similar way as sparse arrays with natural numbers as indices. We use here the view `Int0` already mentioned above and also the view `Nat` from `TRIV` to `NAT` given in Section 9.11.1. The view `IntVector` defined below will be used to construct sets of vectors later on.

```
fmod VECTOR{X :: DEFAULT} is
  pr (ARRAY * (sort Entry{X,Y} to Entry{Y},
              sort Array{X,Y} to Vector{Y}))
      {Nat, X} .
endfm

fmod INT-VECTOR is
  pr VECTOR{Int0} * (sort Entry{Int0} to IntVectorEntry,
                   sort Vector{Int0} to IntVector,
                   op empty to zeroVector) .
endfm

view IntVector from TRIV to INT-VECTOR is
  sort Elt to IntVector .
endv
```

No distinction is made between row and column vectors, so, for example, both the row vector $(-2\ 0\ 0\ 3)$ and its transpose $(-2\ 0\ 0\ 3)^t$ are represented by the same term

```
0 |-> -2 ; 3 |-> 3
```

The constants `zeroMatrix` and `zeroVector` denote the all zero matrix and vector, respectively.

The main module `DIOPHANTINE` begins defining pairs of sets of integer vectors, as follows:

```
fmod DIOPHANTINE is
  pr STRING .
  pr INT-MATRIX .
  pr SET{IntVector}
      * (sort NeSet{IntVector} to NeIntVectorSet,
         sort Set{IntVector} to IntVectorSet,
         op _,_ : Set{IntVector} Set{IntVector} -> Set{IntVector}
           to (_,_) [prec 121 format (d d ni d)]) .

  sort IntVectorSetPair .
  op [_|_] : IntVectorSet IntVectorSet -> IntVectorSetPair
           [format (d n++i n ni n-- d)] .
```

Then, the solver is invoked with the built-in operator

```
op natSystemSolve : IntMatrix IntVector String -> IntVectorSetPair
  [special (...)] .
```

which takes as arguments the coefficient matrix, the constant vector, and a string naming the algorithm to be used (see below), and returns the complete set of solutions encoded as a pair of sets of vectors $[A \mid B]$. The non-negative solutions of the linear Diophantine system correspond exactly to those vectors that can be formed as the sum of a vector from A and a non-negative linear combination of vectors from B .

In particular, if the system S is homogeneous (i.e., $c = \mathbf{zeroVector}$) then A contains just the constant $\mathbf{zeroVector}$ and B is the Diophantine basis of S (which will be empty if S only admits the trivial solution). A homogeneous system either has just the trivial solution or infinitely many solutions.

If S is inhomogeneous (i.e., $c \neq \mathbf{zeroVector}$) then, if S has no solution, both A and B will be empty; otherwise, B will consist of the Diophantine basis of S' , the system formed by setting $c = \mathbf{zeroVector}$, while A contains all solutions of S that are not strictly larger than any element of B . An inhomogeneous system may have no solution (in this case A and B are both empty), a finite number of solutions (in this case A is non-empty and B is empty), or infinitely many solutions (in this case A and B are both non-empty).

In either case, the solution encoding $[A \mid B]$ is unique.

Deciding whether a linear Diophantine system admits a non-negative, non-trivial solution is NP-complete (stated as known in [360]). Furthermore the size of the Diophantine basis of a homogeneous system can be very large. For example the equation: $x + y - kz = 0$, for constant $k > 0$, has a Diophantine basis (i.e., set of minimal, nontrivial solutions) of size $k + 1$.

There are currently two algorithms implemented.

The string "cd" specifies a version of the classical Contejean-Devie algorithm [83] with various improvements. The algorithm is based on incrementing a vector of counters, one for each variable, and so it can only solve systems where the answers involve fairly small numbers. It is fairly insensitive to the number of degrees of freedom in the problem. The improvements in this implementation take effect when an equation has zero or one unfrozen variables with nonzero coefficients and result in either forced assignments or early pruning of a branch of the search. It performs well on the following homogeneous system from [105],

$$\begin{pmatrix} 1 & 2 & -1 & 0 & -2 & -1 \\ 0 & -1 & -2 & 2 & 0 & 1 \\ 2 & 0 & 1 & -1 & -2 & 0 \end{pmatrix} \begin{pmatrix} n_1 \\ n_2 \\ n_3 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

which has a basis of size 13.

```
Maude> red in DIOPHANTINE :
      natSystemSolve(
        (0,0) |-> 1 ; (0,1) |-> 2 ; (0,2) |-> -1 ;
        (0,3) |-> 0 ; (0,4) |-> -2 ; (0,5) |-> -1 ;
        (1,0) |-> 0 ; (1,1) |-> -1 ; (1,2) |-> -2 ;
        (1,3) |-> 2 ; (1,4) |-> 0 ; (1,5) |-> 1 ;
```

```

      (2,0) |-> 2 ; (2,1) |-> 0 ; (2,2) |-> 1 ;
      (2,3) |-> -1 ; (2,4) |-> -2 ; (2,5) |-> 0,
      zeroVector,
      "gcd") .
rewrites: 1 in 10ms cpu (46ms real) (100 rews/sec)
result IntVectorSetPair:
[
  zeroVector
|
  0 |-> 1 ; 1 |-> 1 ; 4 |-> 1 ; 5 |-> 1,
  0 |-> 1 ; 1 |-> 4 ; 2 |-> 9 ; 3 |-> 11,
  0 |-> 10 ; 1 |-> 4 ; 3 |-> 2 ; 4 |-> 9,
  1 |-> 1 ; 2 |-> 1 ; 3 |-> 1 ; 5 |-> 1,
  1 |-> 8 ; 2 |-> 2 ; 4 |-> 1 ; 5 |-> 12,
  0 |-> 2 ; 1 |-> 4 ; 2 |-> 8 ; 3 |-> 10 ; 4 |-> 1,
  0 |-> 3 ; 1 |-> 4 ; 2 |-> 7 ; 3 |-> 9 ; 4 |-> 2,
  0 |-> 4 ; 1 |-> 4 ; 2 |-> 6 ; 3 |-> 8 ; 4 |-> 3,
  0 |-> 5 ; 1 |-> 4 ; 2 |-> 5 ; 3 |-> 7 ; 4 |-> 4,
  0 |-> 6 ; 1 |-> 4 ; 2 |-> 4 ; 3 |-> 6 ; 4 |-> 5,
  0 |-> 7 ; 1 |-> 4 ; 2 |-> 3 ; 3 |-> 5 ; 4 |-> 6,
  0 |-> 8 ; 1 |-> 4 ; 2 |-> 2 ; 3 |-> 4 ; 4 |-> 7,
  0 |-> 9 ; 1 |-> 4 ; 2 |-> 1 ; 3 |-> 3 ; 4 |-> 8
]

```

The string "gcd" specifies an original algorithm based on integer Gaussian elimination followed by a sequence of extended greatest common divisor (gcd) computations. It can "home in" quickly on solutions involving large numbers but it is very sensitive to the number of degrees of freedom and can easily degenerate into a brute force search. Furthermore, termination depends on the bound on the sum of minimal solutions established in [307], which can cause a huge amount of fruitless search after the last minimal solution has been found. It performs well on the "sailors and monkey" problem from [83]:

```

red in DIOPHANTINE :
  natSystemSolve(
    (0,0) |-> 1 ; (0,1) |-> -5 ; (1,1) |-> 4 ; (1,2) |-> -5 ;
    (2,2) |-> 4 ; (2,3) |-> -5 ; (3,3) |-> 4 ; (3,4) |-> -5 ;
    (4,4) |-> 4 ; (4,5) |-> -5 ; (5,5) |-> 4 ; (5,6) |-> -5,
    0 |-> 1 ; 1 |-> 1 ; 2 |-> 1 ; 3 |-> 1 ; 4 |-> 1 ; 5 |-> 1,
    "gcd") .
result IntVectorSetPair:
[
  0 |-> 15621 ; 1 |-> 3124 ; 2 |-> 2499 ; 3 |-> 1999 ;
  4 |-> 1599 ; 5 |-> 1279 ; 6 |-> 1023
|
  0 |-> 15625 ; 1 |-> 3125 ; 2 |-> 2500 ; 3 |-> 2000 ;
  4 |-> 1600 ; 5 |-> 1280 ; 6 |-> 1024
]

```

Finally, the string "" can be passed as third argument of `natSystemSolve`, thus allowing the system to choose which algorithm to use. For convenience, the operator

```
op natSystemSolve : IntMatrix IntVector -> IntVectorSetPair .
```

is equationally defined to invoke the built-in operator with ""

```
eq natSystemSolve(M:IntMatrix, V:IntVector)
  = natSystemSolve(M:IntMatrix, V:IntVector, "") .
endfm
```


Specifying Parameterized Data Structures in Maude

with Miguel Palomino
and Alberto Verdejo

This chapter describes the equational specification in Maude of a series of typical data structures, complementing in this way the list and set data structures provided as predefined modules in Maude and described in Section 9.12.

We start with the well-known stacks, queues, lists, and multisets to continue with binary and search trees; not only are the simple versions considered, but also advanced ones such as AVL and 2-3-4 trees. The operator attributes available in Maude allow the specification of data based on constructors that satisfy some equational properties, like concatenation of lists which is associative and has the empty list as identity, as opposed to the free constructors available in other functional programming languages. Moreover, the expressive version of equational logic on which Maude is based, namely membership equational logic, allows the faithful specification of types whose data are defined not only by means of constructors, but also by the satisfaction of additional properties, like sorted lists, search trees, balanced trees, etc. We will see along this chapter how this is accomplished by means of membership assertions that equationally characterize the properties satisfied by the corresponding data.

As already mentioned, for all the data structures specifications in this chapter we do not need to consider rewriting logic in its full generality, but just membership equational logic; that is, from the Maude language point of view, all of these specifications are *functional modules*. In addition, all the data types that we consider are generic, that is, they are constructions on top of other data types that appear as parameters in the construction. Therefore, our specifications are *parameterized*, making use of the powerful mechanisms for parameterization based on *theories* that describe the requirements that a data type passed as parameter must satisfy for the construction to make sense, as explained in Section 8.3.

The simplest theory we use is **TRIV**, which is predefined in Maude (see Section 9.11.1) and is used as requirement for the parameter of stacks, queues, lists, multisets, binary trees, and general trees.

A more complex theory is the **STOSET** theory below, requiring a *strict* total order over elements of a given sort, that we recall in flattened form

from Section 8.3.1 (as mentioned there, the equation for antisymmetry can be discarded in this theory, because it is implied by the equations for irreflexivity and transitivity). This theory will be useful in the specification of search trees and its refinements.

```
fth STOSET is
  protecting BOOL .
  sort Elt .
  op <_ : Elt Elt -> Bool .
  vars X Y Z : Elt .
  eq X < X = false [nonexec label irreflexive] .
  ceq X < Z = true if X < Y /\ Y < Z [nonexec label transitive] .
  ceq X = Y if X < Y /\ Y < X [nonexec label antisymmetric] .
  ceq X = Y if X < Y = false /\ Y < X = false [nonexec label total] .
endfth
```

As explained in Section 8.3.4, *views* are used to instantiate parameterized modules. A view shows how a particular module satisfies a theory, by mapping sorts and operations in the theory to sorts and operations (or, more generally, terms) in the target module, in such a way that the induced translations on equations and membership axioms are provable in the module. In general, this requires theorem proving, which is not done by the system but can instead be delegated to a theorem prover like the ITP tool (see Section 23.1.1). However, in many simple cases the proof of obligations associated with views is completely obvious, as for example in the following view from the theory `TRIV` to the predefined module `INT` of integers, where, since `TRIV` has no equations, no proof obligations are generated.

```
view Int from TRIV to INT is
  sort Elt to Int .
endv
```

This view is predefined in Maude, together with many others from `TRIV` to other predefined modules, as described in Section 9.11.1.

We assume some knowledge about the data structures that are specified. There are many textbooks that describe well-known imperative and object-oriented implementations [195, 46, 381]. Less known, but closer to our approach, are implementations in functional programming languages such as ML or Haskell [282, 311, 296]; in some cases, our equations are very similar to the ones given in such textbooks.

10.1 Stacks

We begin our collection of data type specifications with stacks. Since stacks can be built over any data type, the requirement theory is `TRIV`. The main sort is `Stack{X}`; notice that its structured name makes explicit the label of

the parameter. In this way, when the module is instantiated with a view, like for example `Nat` from `TRIV` to `NAT` (which is predefined in Maude, see Section 9.11.1), the sort name is also instantiated becoming `Stack{Nat}`, which makes clear that the data are stacks of natural numbers.

The only subtle point in a stack specification is that the `top` and `pop` operations are partial, because they are not defined on the empty stack. In our specification, we use a subsort `NeStack{X}` of non-empty stacks to handle this situation. Then, `push` becomes a constructor of non-empty stacks, while both `empty` and `push` (the latter via subsorting) are constructors of stacks. Then, the `top` and `pop` operations can be defined as total with domain `NeStack{X}`.

Finally, all modules import implicitly (in `including` mode) the predefined `BOOL` module, so that we can use the sort `Bool` and the Boolean values `true` and `false` when necessary. However, since Boolean values are not modified in any way by this importation, we make explicit that it is in `protecting` mode.

```
fmod STACK{X :: TRIV} is
  protecting BOOL .
  sorts NeStack{X} Stack{X} .
  subsort NeStack{X} < Stack{X} .
  op empty : -> Stack{X} [ctor] .
  op push : X$Elt Stack{X} -> NeStack{X} [ctor] .
  op pop : NeStack{X} -> Stack{X} .
  op top : NeStack{X} -> X$Elt .
  op isEmpty : Stack{X} -> Bool .

  var S : Stack{X} .
  var E : X$Elt .

  eq pop(push(E, S)) = S .
  eq top(push(E, S)) = E .
  eq isEmpty(empty) = true .
  eq isEmpty(push(E, S)) = false .
endfm
```

We instantiate this parameterized module with the predefined view `Int` from the theory `TRIV` to the predefined module `INT` of integers, so that we can have an example of term reduction, invoked with the Maude command `red`.

```
fmod STACK-TEST is
  protecting STACK{Int} .
endfm

Maude> red top(push(4, push(5, empty))) .
result NzNat : 4
```

For two alternative, object-oriented specifications of stacks, see Section 21.4.3.

10.2 Queues

The specification for queues is very similar to the one for stacks:

- The requirement theory is TRIV.
- The main sort is `Queue{X}`.
- The `first` and `dequeue` operations are partial, because they are not defined on the empty queue.
- We use a subsort `NeQueue{X}` of non-empty queues.
- `enqueue` becomes a *constructor* of non-empty queues, while both `empty` and `enqueue` (via subsorting) are *constructors* of queues.
- `dequeue` and `first` are defined as total with domain `NeQueue{X}`.
- The operation `isEmpty` returns a Boolean value, taking into account the importation of the predefined `BOOL` module, which we make explicit in `protecting` mode.

```
fmod QUEUE{X :: TRIV} is
  protecting BOOL .
  sort NeQueue{X} Queue{X} .
  subsort NeQueue{X} < Queue{X} .
  op empty : -> Queue{X} [ctor] .
  op enqueue : Queue{X} X$Elt -> NeQueue{X} [ctor] .
  op dequeue : NeQueue{X} -> Queue{X} .
  op first : NeQueue{X} -> X$Elt .
  op isEmpty : Queue{X} -> Bool .

  var Q : Queue{X} .
  var E : X$Elt .

  eq dequeue(enqueue(empty, E)) = empty .
  ceq dequeue(enqueue(Q, E)) = enqueue(dequeue(Q), E)
    if Q /= empty .
  eq first(enqueue(empty, E)) = E .
  ceq first(enqueue(Q, E)) = first(Q) if Q /= empty .
  eq isEmpty(empty) = true .
  eq isEmpty(enqueue(Q, E)) = false .
endfm
```

We consider queues of identifiers by means of the predefined view `Qid` and show a sample reduction.

```
fmod QUEUE-TEST is
  protecting QUEUE{Qid} .
endfm

Maude> red first(enqueue(enqueue(empty, 'a), 'b)) .
result Qid: 'a
```

10.3 Priority queues

For the specification of priority queues, a stronger requirement than TRIV is needed. We are going to identify priorities with elements in the queue, which can be compared by means of a total order. Since priorities can be repeated, the order should be non-strict, like in the NSTOSET theory introduced in Section 8.3.1. However, in the specification of some operations it is more convenient to have available also the strict version of the order. Therefore, we are going to use as requirement the theory TOSET that specifies together both the strict `<` and the non-strict `<=` order relations. This theory was also introduced in Section 8.3.1, but we recall it here as an extension of the STOSET theory, also recalled earlier in this chapter.

```
fth TOSET is
  including STOSET .
  op <=_ : Elt Elt -> Bool .
  vars X Y : Elt .
  eq X <= X = true [nonexec] .
  ceq X <= Y = true if X < Y [nonexec] .
  ceq X = Y if X <= Y /\ X < Y = false [nonexec] .
endfth
```

We have as constructors of priority queues the constant `empty` and the `insert` operator, that adds a new element to the priority queue. However, these constructors are *not free*, because the order of insertion does not matter, the priority being the information that determines the actual order in the queue. This is made explicit in a “commutativity” equation for the `insert` operator, but this is not a standard commutativity equation for a binary operator with both arguments of the same sort, and thus it cannot be expressed as a `comm` attribute; in any case, it is not terminating, and therefore it has been stated as *nonexecutable* by means of the `nonexec` attribute.

We consider the version of priority queues in which the first element is the minimum. Both `findMin` and `deleteMin` are easily specified as total operations on non-empty priority queues by structural induction and, when there are two or more elements in the queue, by comparing the priorities of two elements.

```
fmod PRIORITY-QUEUE{X :: TOSET} is
  sort NePQueue{X} PQueue{X} .
  subsort NePQueue{X} < PQueue{X} .
  op empty : -> PQueue{X} [ctor] .
  op insert : PQueue{X} X$Elt -> NePQueue{X} [ctor] .
  op deleteMin : NePQueue{X} -> PQueue{X} .
  op findMin : NePQueue{X} -> X$Elt .
  op isEmpty : PQueue{X} -> Bool .

  var PQ : PQueue{X} .
  vars E F : X$Elt .
```

```

eq insert(insert(PQ, E), F) = insert(insert(PQ, F), E) [nonexec] .
eq deleteMin(insert(empty, E)) = empty .
ceq deleteMin(insert(insert(PQ, E), F))
  = insert(deleteMin(insert(PQ, E)), F)
  if findMin(insert(PQ, E)) <= F .
ceq deleteMin(insert(insert(PQ, E), F)) = insert(PQ, E)
  if F < findMin(insert(PQ, E)) .
eq findMin(insert(empty, E)) = E .
ceq findMin(insert(insert(PQ, E), F)) = findMin(insert(PQ, E))
  if findMin(insert(PQ, E)) <= F .
ceq findMin(insert(insert(PQ, E), F)) = F
  if F < findMin(insert(PQ, E)) .
eq isEmpty(empty) = true .
eq isEmpty(insert(PQ, E)) = false .
endfm

```

Even though this is a very abstract specification, it is directly executable after instantiating the parameter with an appropriate view, for example to integers, as follows:

```

view IntAsToset from TOSET to INT is
  sort Elt to Int .
endv

fmod PRIORITY-QUEUE-TEST is
  protecting PRIORITY-QUEUE{IntAsToset} .
endfm

Maude> red findMin(insert(insert(empty, 4), 5)) .
result NzNat: 4

```

We consider a slightly more complex instantiation with elements that are pairs, so that the first component of the pair can be thought of as the priority and the second one as related information. For this we define first a parameterized module with a pair constructor and the corresponding order operations that simply reflect the order in the first component.

```

fmod PRIORITY-PAIR{X :: TOSET, Y :: TRIV} is
  sort Priority-Pair{X, Y} .
  op <_,_> : X$Elt Y$Elt -> Priority-Pair{X, Y} .
  ops _<_ _<=_ : Priority-Pair{X, Y} Priority-Pair{X, Y} -> Bool .

  vars A A' : X$Elt .
  vars B B' : Y$Elt .
  eq < A, B > < < A', B' > = A < A' .
  eq < A, B > <= < A', B' > = A <= A' .
endfm

```

Using the `IntAsToset` view above and the predefined view `String` we instantiate this parameterized module and then define the following view:

```
view IntStringAsToset from TOSET to
  PRIORITY-PAIR{IntAsToset, String} is
  sort Elt to Priority-Pair{IntAsToset, String} .
endv
```

Now we instantiate again the `PRIORITY-QUEUE` module and perform a simple reduction.

```
fmod PRIORITY-QUEUE-TEST-PAIR is
  protecting PRIORITY-QUEUE{IntStringAsToset} .
endfm

Maude> red findMin(insert(insert(insert(empty, < 4, "d" >),
                                < 8, "h" >),
                                < 1, "a" >)) .
result Priority-Pair{IntAsToset, String}: < 1, "a" >
```

10.4 Lists

There are different ways of building lists. One possibility is to begin with the empty list and the singleton lists, and then use the concatenation operation to get bigger lists. However, concatenation cannot be a free list constructor, because it satisfies an associativity equation and has the empty list as identity. This approach was applied to concrete lists in Section 5.3, is also used in the predefined module for generic lists described in Section 9.12.1, and appears in many similar examples throughout this book. Given the support for equational attributes (associativity, commutativity, etc.) in Maude, as explained in Section 4.4.1, one can argue that this is indeed the most natural specification for lists in Maude.

Here we use instead the two standard free constructors for lists that can be found in many functional programming languages: the empty list *nil*, here denoted `[]`, and the *cons* operation that adds an element to the beginning of a list, here denoted with the mixfix syntax `_:_`. This approach facilitates proving list properties by structural induction in the ITP (see Section 23.1.1), or using tools like the Church-Rosser Checker (see Section 23.1.3) or the Maude Termination Tool (see Section 23.1.2), and provides a simple basis for specifying sorted lists and sorting operations on them in Section 10.5.

As usual, `head` and `tail` are the selectors associated with the `_:_` constructor. Since they are not defined on the empty list, we avoid their partiality in the same way as we have done for stacks and queues in the previous sections by means of a subsort `NeList{X}` of non-empty lists.

```
fmod LIST-CONS{X :: TRIV} is
  protecting NAT .
```

```

sorts NeList{X} List{X} .
subsort NeList{X} < List{X} .

op [] : -> List{X} [ctor] .
op _:_ : X$Elt List{X} -> NeList{X} [ctor] .
op tail : NeList{X} -> List{X} .
op head : NeList{X} -> X$Elt .

var E : X$Elt .
var N : Nat .
vars L L' : List{X} .

eq tail(E : L) = L .
eq head(E : L) = E .

```

Three interesting operations on lists are list concatenation (here denoted with infix syntax `_+_`), the length of a list, and reversing a list. The `length` operator has a result of sort `Nat`, that comes from the predefined module `NAT`, imported in `protecting` mode. These three operations are defined as usual by structural induction on the two constructors, with an equation for the empty base case and another for the `cons` case `E : L`.

Here, like in most specifications in this chapter, we are not concerned with efficiency and therefore we just specify the operations in a simple way, without using, for example, tail-recursive auxiliary operations in the style of Section 9.12.1.

```

op _+_ : List{X} List{X} -> List{X} .
op length : List{X} -> Nat .
op reverse : List{X} -> List{X} .

eq [] ++ L = L .
eq (E : L) ++ L' = E : (L ++ L') .
eq length([]) = 0 .
eq length(E : L) = 1 + length(L) .
eq reverse([]) = [] .
eq reverse(E : L) = reverse(L) ++ (E : []) .

```

In this specification of generic lists we also add two operations that will be useful later, in Section 10.5, when sorting lists: `take_from_` and `throw_from_`. The first one builds a list by taking the first n elements of the given list, while the second one deletes the first n elements of the given list. Both of them are defined by structural induction on both arguments, the base case being when either the first is 0 or the second is empty. As usual, `s_` denotes the successor operator on natural numbers.

```

op take_from_ : Nat List{X} -> List{X} .
op throw_from_ : Nat List{X} -> List{X} .

```

```

eq take 0 from L = [] .
eq take N from [] = [] .
eq take s N from (E : L) = E : take N from L .

eq throw 0 from L = L .
eq throw N from [] = [] .
eq throw s N from (E : L) = throw N from L .
endfm

```

The following sample reduction shows the result of reversing a list of character strings.

```

fmod LIST-CONS-TEST is
  protecting LIST-CONS{String} .
endfm

Maude> red reverse("one" : "two" : "three" : []) .
result NeList{String}: "three" : "two" : "one" : []

```

10.5 Sorted lists

In order-sorted equational specifications, subsorts must be defined by means of constructors, but it is not possible to have a subsort of sorted lists, for example, defined by a property over lists; a more expressive formalism is needed.¹ *Membership equational logic* allows subsort definition by means of conditions involving equations and/or sort predicates. In this example we use this technique to define a subsort of *sorted lists*, included in the sort of lists imported from the module `LIST-CONS` in Section 10.4. A similar technique is used in Sections 8.3.5 and 24.2.8 to define sorted lists as a subsort of lists based on associative concatenation with identity. Furthermore, we will also specify here different well-known sorting algorithms.

Parameterized sorted lists need a stronger requirement than `TRIV`, because a total order over the elements to be sorted is needed. Since repetitions pose no problems for sorting a list, the order relation should be non-strict, like in the `NSTOSET` theory introduced in Section 8.3.1 and also used in the specification of sorted lists in Section 8.3.5. However, for the specification of the sorting algorithms, it is more convenient to use also the strict version of the order. For these reasons, we will use as requirement for parameterized sorted lists the theory `TOSET`, also introduced in Section 8.3.1 and recalled in Section 10.3.

The parameterized module for sorted lists imports the parameterized list module. However, note that we want lists over a totally ordered set, instead

¹ This lack of expressiveness was understood from within order-sorted algebra, leading to the proposal of *sort constraints* in [175], a precursor of membership axioms. However, the logical basis of sort constraints was never fully developed within order-sorted algebra.

of lists over any set; therefore, first we *partially instantiate* LIST-CONS with an inclusion view from the theory TRIV to the theory TOSET.

```
view TOSET from TRIV to TOSET is
endv
```

We are still left with a parameterized module and corresponding dependent sorts, but now with respect to the TOSET requirement. This is the reason justifying the notation LIST-CONS{TOSET}{X} in the `protecting` importation below, as well as NeList{TOSET}{X} and List{TOSET}{X} as names of the imported sorts.

Notice the three membership axioms defining the subsort SortedList{X}: the empty and singleton lists are always sorted, and a longer list is sorted when the first element is less than or equal to the second, and the list without the first element is also sorted.

```
fmod SORTED-LIST{X :: TOSET} is
  protecting LIST-CONS{TOSET}{X} .

  sorts SortedList{X} NeSortedList{X} .
  subsorts NeSortedList{X} < SortedList{X} < List{TOSET}{X} .
  subsort NeSortedList{X} < NeList{TOSET}{X} .

  vars N M : X$Elt .
  vars L L' : List{TOSET}{X} .
  vars OL OL' : SortedList{X} .
  var NEOL : NeSortedList{X} .

  mb [] : SortedList{X} .
  mb (N : []) : NeSortedList{X} .
  cmb (N : NEOL) : NeSortedList{X} if N <= head(NEOL) .
```

As part of this module, we also define several well-known sorting operations: `insertion-sort`, `quicksort`, and `mergesort`, based on appropriate auxiliary operations. The important point is that we are able to give finer typing to all these sorting operations than the usual typing in other algebraic specification frameworks or functional programming languages. For example, `insertion-sort` is declared as an operation from List{TOSET}{X} to SortedList{X}, instead of the much less informative typing from List{TOSET}{X} to List{TOSET}{X}. The same applies to each of the auxiliary operations. Furthermore, a function that requires its input argument to be a sorted list can now be defined as a *total* function, whereas in less expressive typing formalisms it would have to be either *partial*, or to be defined with exceptional behavior on the erroneous arguments.

The operation `insert-list` inserts an element in the appropriate position of an already sorted list, so that the resulting list is also sorted. The sorting operation `insertion-sort` recursively sorts the list without the first element

and then calls `insert-list`, which inserts the missing element in the correct position.

```

op insertion-sort : List{TOSET}{X} -> SortedList{X} .
op insert-list : SortedList{X} X$Elt -> SortedList{X} .

eq insertion-sort([]) = [] .
eq insertion-sort(N : L) = insert-list(insertion-sort(L), N) .

eq insert-list([], M) = M : [] .
ceq insert-list(N : OL, M) = M : N : OL if M <= N .
ceq insert-list(N : OL, M) = N : insert-list(OL, M) if N < M .

```

The sorting operation `mergesort` splits a given list in half by means of the operations `take_from_` and `throw_from_` described in Section 10.4 above, recursively sorts each sublist, and then calls the commutative `merge` operation on the sorted sublists to obtain the final sorted result. In Section 9.12.6 on sortable lists there is a more efficient (albeit more complex) definition of the `mergesort` algorithm on lists.

```

op mergesort : List{TOSET}{X} -> SortedList{X} .
op merge : SortedList{X} SortedList{X} -> SortedList{X} [comm] .

eq mergesort([]) = [] .
eq mergesort(N : []) = N : [] .
ceq mergesort(L)
  = merge(mergesort(take (length(L) quo 2) from L),
          mergesort(throw (length(L) quo 2) from L))
  if length(L) > 1 .

eq merge(OL, []) = OL .
ceq merge(N : OL, M : OL') = N : merge(OL, M : OL') if N <= M .

```

Finally, `quicksort` works on a list by separating its elements into those smaller than the first element (taken as the pivot) and those bigger than the first, recursively sorts each of the resulting lists, and simply puts them together by concatenating them with the pivot in the middle.

```

op quicksort : List{TOSET}{X} -> SortedList{X} .
op leq-elems : List{TOSET}{X} X$Elt -> List{TOSET}{X} .
op gr-elems : List{TOSET}{X} X$Elt -> List{TOSET}{X} .

eq quicksort([]) = [] .
eq quicksort(N : L)
  = quicksort(leq-elems(L,N)) ++ (N : quicksort(gr-elems(L,N))) .

eq leq-elems([], M) = [] .
ceq leq-elems(N : L, M) = N : leq-elems(L, M) if N <= M .
ceq leq-elems(N : L, M) = leq-elems(L, M) if M < N .

```

```

eq gr-elems([], M) = [] .
ceq gr-elems(N : L, M) = gr-elems(L, M) if N <= M .
ceq gr-elems(N : L, M) = N : gr-elems(L, M) if M < N .
endfm

```

We now apply the sorting operations to lists of natural numbers.

```

view NatAsToset from TOSET to NAT is
  sort Elt to Nat .
endv

fmod SORTED-LIST-TEST is
  protecting SORTED-LIST{NatAsToset} .
endfm

Maude> red insertion-sort(5 : 4 : 3 : 2 : 1 : 0 : []) .
result NeSortedList{NatAsToset}: 0 : 1 : 2 : 3 : 4 : 5 : []

Maude> red mergesort(5 : 3 : 1 : 0 : 2 : 4 : []) .
result NeSortedList{NatAsToset}: 0 : 1 : 2 : 3 : 4 : 5 : []

Maude> red quicksort(0 : 1 : 2 : 5 : 4 : 3 : []) .
result NeSortedList{NatAsToset}: 0 : 1 : 2 : 3 : 4 : 5 : []

```

We study the efficiency of these sorting algorithms using Maude's profiler in Section 24.1.4.

10.6 Multisets

In this section we specify multisets by means of a union constructor that satisfies associativity, commutativity, and identity structural axioms, all declared as attributes. This approach was introduced for concrete multisets in Section 5.4 and is here generalized to the case of generic multisets parameterized over TRIV.

Moreover, in order to have a more complete multiset specification we add several additional interesting operations, again defined in a simple way without taking into account efficiency concerns.

Notice that singleton multisets are identified with elements by declaring $X\$\text{Elt}$ as a subsort of $\text{Mset}\{X\}$.

```

fmod MULTISSET{X :: TRIV} is
  protecting NAT .

  sort Mset{X} .
  subsort X$Elt < Mset{X} .

  op empty : -> Mset{X} [ctor] .
  op __ : Mset{X} Mset{X} -> Mset{X} [ctor assoc comm id: empty] .

```

The operation `isEmpty` checks if a multiset is empty, while `size` calculates the size of a multiset, i.e., the total number of elements taking into account the multiplicity of each one.

```

op isEmpty : Mset{X} -> Bool .
op size : Mset{X} -> Nat .

vars E E' : X$Elt .
vars S S' : Mset{X} .

eq isEmpty(empty) = true .
eq isEmpty(E S) = false .

eq size(empty) = 0 .
eq size(E S) = 1 + size(S) .

```

The `isIn` operation checks if an element appears in a multiset and `mult` calculates the multiplicity of a given element. Both are specified by means of the `otherwise` attribute (explained in Section 4.5.4).

To see if a natural number `E` is in a multiset, we can match the multiset against the pattern `E S`; if the matching succeeds, then the result is true, *otherwise*, we know that the element does not occur in the multiset and the result is false.

The same technique can be used to count the number of occurrences of a given element in a given multiset: if the multiset matches the pattern `E S`, then we know that `E` appears at least once and count recursively the remaining occurrences; *otherwise*, `E` does not occur and thus its multiplicity is zero.

```

op isIn : X$Elt Mset{X} -> Bool .
op mult : X$Elt Mset{X} -> Nat .

eq isIn(E, E S) = true .
eq isIn(E, S) = false [otherwise] .

eq mult(E, E S) = 1 + mult(E, S) .
eq mult(E, S) = 0 [otherwise] .

```

The `delete` operation removes *all* occurrences of an element from a multiset, while `delete1` removes just *one* occurrence. In both cases, the multiset does not change when the element does not appear in it. The equational definition of both operations is also based on the `otherwise` attribute.

```

op delete : X$Elt Mset{X} -> Mset{X} .
op delete1 : X$Elt Mset{X} -> Mset{X} .

eq delete(E, E S) = delete(E, S) .
eq delete(E, S) = S [otherwise] .

eq delete1(E, E S) = S .
eq delete1(E, S) = S [otherwise] .

```

The intersection and difference operations over multisets are similar to the operations with the same name on sets, but they need to take into account the multiplicity of elements, as expected. We define them in a very simple way using again the `otherwise` attribute: if two multisets have an element in common, which is checked by means of matching with a common variable `E`, either we take such element in the case of the intersection or discard it in the case of the difference; *otherwise*, we know the intersection is empty and the difference coincides with the first argument.

```

op intersection : Mset{X} Mset{X} -> Mset{X} .
op difference   : Mset{X} Mset{X} -> Mset{X} .

eq intersection(E S, E S') = E intersection(S, S') .
eq intersection(S, S') = empty [otherwise] .

eq difference(E S, E S') = difference(S, S') .
eq difference(S, S') = S [otherwise] .
endfm

```

We show several sample reductions on multisets of integers.

```

fmod MULTISSET-TEST is
  protecting MULTISSET{Int} .
endfm

Maude> red mult(2, 2 2 1 2 3 2) .
result NzNat: 4

Maude> red intersection(2 2 3 4, 2 2 1 3 3 2) .
result Mset{Int}: 2 2 3

Maude> red difference(2 2 3 4, 2 2 1 3 3 2) .
result NzNat: 4

```

Notice that Maude computes the least sort of the result. While in the first reduction above we expect a natural number as the result of the multiplicity operation, in the last reduction we expect a multiset as the result of the difference operation; indeed, in this example we get a *singleton* multiset, which is identified with the corresponding number and given the least sort according to the subsort relationships `NzNat < Int < Mset{Int}`.

10.7 Binary trees

Section 5.1 introduces non-empty binary trees over natural numbers, where elements can only appear in the leaves. Here we consider a parameterized version of the more standard binary trees with elements in all nodes, including the empty tree. Such binary trees (parameterized by the theory `TRIV`) are built with two free constructors: the empty tree, denoted `empty`, and an operation

`_[_]_` that puts a data element in the root above two given trees, its left and right children. The three selectors associated to this constructor (`root`, `left`, and `right`) only make sense for non-empty trees, that belong to the corresponding subsort.

```
fmod BIN-TREE{X :: TRIV} is
  protecting LIST-CONS{X} .

  sorts NeBinTree{X} BinTree{X} .
  subsort NeBinTree{X} < BinTree{X} .

  op empty : -> BinTree{X} [ctor] .
  op _[_]_ : BinTree{X} X$Elt BinTree{X} -> NeBinTree{X} [ctor] .
  ops left right : NeBinTree{X} -> BinTree{X} .
  op root : NeBinTree{X} -> X$Elt .

  var E : X$Elt .
  vars L R : BinTree{X} .
  vars NEL NER : NeBinTree{X} .

  eq left(L [E] R) = L .
  eq right(L [E] R) = R .
  eq root(L [E] R) = E .
```

The operation that calculates the depth (or height) of a binary tree calls the `max` operation on natural numbers (see Section 9.2) in the recursive non-empty case to obtain the maximum of two such numbers. Notice that the module `NAT` is indirectly imported through the explicit importation (in `protecting` mode) of the module `LIST-CONS`.

```
op depth : BinTree{X} -> Nat .

eq depth(empty) = 0 .
eq depth(L [E] R) = 1 + max(depth(L), depth(R)) .
```

Finally, we also have three operations that calculate the standard binary tree traversals and another operation that returns the list of leaves of a binary tree, all of them with `List{X}` as value sort; this is the reason why this module imports the `LIST-CONS` module (of course, we could instead import the predefined module `LIST` of Section 9.12.1). Since these four operations have the same rank, they are all declared together by means of the keyword `ops`.

```
ops leaves preorder inorder postorder : BinTree{X} -> List{X} .

eq leaves(empty) = [] .
eq leaves(empty [E] empty) = E : [] .
eq leaves(NEL [E] R) = leaves(NEL) ++ leaves(R) .
eq leaves(L [E] NER) = leaves(L) ++ leaves(NER) .
```

```

eq preorder(empty) = [] .
eq preorder(L [E] R) = E : (preorder(L) ++ preorder(R)) .
eq inorder(empty) = [] .
eq inorder(L [E] R) = inorder(L) ++ (E : inorder(R)) .
eq postorder(empty) = [] .
eq postorder(L [E] R)
  = postorder(L) ++ (postorder(R) ++ (E : [])) .
endfm

```

We calculate the postorder traversal of a binary tree over integers as follows:

```

fmod BIN-TREE-TEST is
  protecting BIN-TREE{Int} .
endfm

Maude> red postorder((empty [1] empty) [2] (empty [3] empty)) .
result NeList{Int}: 1 : 3 : 2 : []

```

10.8 General trees

General trees can have a variable number of children for each node. One can specify them by using an auxiliary data type of *forests* that behave like lists of trees, with constructors `empty-forest` and `_:_`. Then the only constructor `_[_]` for general trees has as first argument the element stored in the root node of the tree, and as second argument the forest formed by the children. The operations `root` and `children` are the selectors associated with this constructor.

The operation `#children` calculates the number of children of a tree by calculating the length of the corresponding forest. A tree is a leaf when it has no children, that is, the corresponding forest is empty. In the equation below, the operation `leaf?` is defined by means of `#children` but other equivalent definitions are possible.

```

fmod GEN-TREE{X :: TRIV} is
  protecting LIST-CONS{X} .

  sorts Tree{X} Forest{X} .

  op _[_] : X$Elt Forest{X} -> Tree{X} [ctor] .
  op empty-forest : -> Forest{X} [ctor] .
  op _:_ : Tree{X} Forest{X} -> Forest{X} [ctor] .

  op root : Tree{X} -> X$Elt .
  op children : Tree{X} -> Forest{X} .
  op #children : Tree{X} -> Nat .

```

```

op length : Forest{X} -> Nat .
op leaf? : Tree{X} -> Bool .

var E : X$Elt .
var T : Tree{X} .
var F : Forest{X} .

eq root(E [F]) = E .
eq children(E [F]) = F .

eq length(empty-forest) = 0 .
eq length(T : F) = 1 + length(F) .
eq #children(E [F]) = length(F) .
eq leaf?(T) = #children(T) == 0 .

```

Most operations on general trees are defined by means of corresponding operations on forests (i.e., lists of trees), so that each pair of such operations is defined in a mutually recursive fashion. For example, the depth of a tree is one plus the depth of its forest, while the depth of a forest is the maximum of the depths of all its trees.

The degree of a general tree is the maximum number of children that a node has, and it is defined in a similar way by means of an auxiliary operation on forests that calculates the maximum number of children that a node in any one of the trees in a given forest has.

```

ops depth degree : Tree{X} -> Nat .
ops depth-forest degree-forest : Forest{X} -> Nat .

eq depth(E [F]) = 1 + depth-forest(F) .
eq depth-forest(empty-forest) = 0 .
eq depth-forest(T : F) = max(depth(T), depth-forest(F)) .

eq degree(E [F]) = max(length(F), degree-forest(F)) .
eq degree-forest(empty-forest) = 0 .
eq degree-forest(T : F) = max(degree(T), degree-forest(F)) .

```

On general trees we consider preorder and postorder traversals, which, like the previous operations, are defined by means of corresponding “traversals” on forests. A traversal of a forest is obtained by concatenating the traversals of all the trees in such forest.

```

ops preorder postorder : Tree{X} -> List{X} .
ops preorder-forest postorder-forest : Forest{X} -> List{X} .

eq preorder(E [F]) = E : preorder-forest(F) .
eq preorder-forest(empty-forest) = [] .
eq preorder-forest(T : F) = preorder(T) ++ preorder-forest(F) .

eq postorder(E [F]) = postorder-forest(F) ++ (E : []) .

```

```

eq postorder-forest(empty-forest) = [] .
eq postorder-forest(T : F) = postorder(T) ++ postorder-forest(F) .
endfm

```

The following example shows the result of traversing in postorder a general tree over the integers.

```

fmod GEN-TREE-TEST is
  protecting GEN-TREE{Int} .
endfm

Maude> red postorder(
  1 [ 3 [ 4 [ empty-forest ] : empty-forest ]
    : (2 [ empty-forest ] : empty-forest) ]) .
result NeList{Int}: 4 : 3 : 2 : 1 : []

```

10.9 Binary search trees

This example is similar in style to the one for sorted lists in Section 10.5, but it is a bit more complex. We specify a subsort of *(binary) search trees* by using several membership axioms over terms of the sort of binary trees defined in Section 10.7.

Even though we allowed repeated elements in a sorted list, this should not be the case in a search tree, where all nodes must contain different values. A binary search tree is either the empty binary tree or a non-empty binary tree such that all elements in the left child are strictly smaller than the element in the root, all elements in the right child are strictly bigger than it, and both the left and right children are also binary search trees. This is checked by means of auxiliary operations that calculate the minimum and maximum elements in a non-empty search tree, and that are also useful when deleting an element. Again, the most important point is that membership equational logic allows us both to define the corresponding subsort by means of membership assertions (we consider five cases in the specification below) and to assign typings in the best possible way to all the operations defined for this data type.

Although we could parameterize binary search trees just with respect to a strict total order given by the theory **STOSET** (introduced in Section 8.3.1 and recalled at the beginning of this chapter), we specify here the version of search trees containing in the nodes pairs formed by a key and its associated contents, so that we think of search trees as *dictionaries*. The search tree structure is with respect to a strict total order on keys, but contents can be over an arbitrary sort. When we insert a pair $\langle K, C \rangle$ and the key K already appears in the tree in a pair $\langle K, C' \rangle$, insertion takes place by combining the contents C' and C . This combination can be replacing the first with the second, just forgetting the second, addition if the trees are used to implement multisets and the C s represent multiplicities, etc. Therefore, as part of the requirement

parameter theory for the contents we will have an associative binary operator `combine` on the sort `Contents`.

```
fth CONTENTS is
  sort Contents .
  op combine : Contents Contents -> Contents [assoc] .
endfth
```

In principle, a pair construction should be enough to put together the information that we have just described, but we will import the module for search trees in the specifications of more complex data structures that happen to be particular cases of search trees, such as AVL and red-black trees. In those cases, it is important to add new information in the nodes: for AVL trees, one needs the depth of the tree hanging in each node, while for red-black trees one needs the appropriate node color. Therefore, taking this into account, it is important to define a data type that is *extensible*, and we have considered *records* for this, defined in the following module `RECORD`. A record is defined as a collection of pairs consisting of a field name and an associated value. Notice that the associative and commutative union operator is denoted by `_,_` and is declared at the level of kinds, because it is partial in the sense that the union of two records is not a record unless some additional constraints are satisfied. These constraints will be made explicit by means of memberships, which also require the associative operator to be declared at the kind level (see Section 24.2.8).

```
fmod RECORD is
  sort Record .
  op null : -> Record [ctor] .
  op _,_ : [Record] [Record] -> [Record] [ctor assoc comm id: null] .
endfm
```

In the `SEARCH-TREE` module below, after importing the binary trees instantiated with a `Record` view, we define the fields for keys (with syntax `key:_`) and for contents (with syntax `contents:_`), together with corresponding projection operations that extract the appropriate values from records that have those fields, belonging to a subsort `SearchRecord` of search records. But notice that these operations, as well as the operations on trees, can be applied to records that have additional fields, unknown yet at this time, by using a variable `Rec` that takes care of “the rest of the record.” The auxiliary operations `numContents` and `numKeys` are used to make sure that a search record has exactly one field `contents` and one field `key`, respectively.

This construction of adding fields to records is adding new data to the sort `Record`, whose elements are the data in the nodes of the trees; in this way, we are also adding new data to the `NeBinTree{Record}` and `BinTree{Record}` sorts of binary trees. For this reason, we have imported the module `BIN-TREE` (after instantiating it with the view `Record`) in `extending` mode.

```

view Record from TRIV to RECORD is
  sort Elt to Record .
endv

```

In addition to operations for insertion and deletion, we have a `lookup` operation that returns the contents associated with a given key, when the key appears in the tree. However, this last operation is partial, not being defined when the key does not appear in the tree; this partiality cannot be handled by means of a subsort, because the partiality condition depends on the concrete values of the arguments. Instead, we use the technique of declaring this operation as total with coarity a supersort of the expected sort, containing a special value `not-found` which is returned by the `lookup` operation when the key does not appear in the tree. This is done by instantiating the parameterized module `MAYBE`, introduced in Section 8.3.3, with the following view

```

view Contents from TRIV to CONTENTS is
  sort Elt to Contents .
endv

```

and renaming the constant `maybe` to `not-found`.

The operations on binary search trees are specified as usual, by structural induction, and in the non-empty case by comparing the given key K with the key in the root of the tree and distinguishing the three cases according to whether K is smaller than, equal to, or bigger than the root key.

```

fmod SEARCH-TREE{X :: STOSET, Y :: CONTENTS} is
  extending BIN-TREE{Record} .
  protecting MAYBE{Contents}{Y} * (op maybe to not-found) .

  sorts SearchRecord{X, Y} SearchTree{X, Y} NeSearchTree{X, Y} .
  subsort SearchRecord{X, Y} < Record .
  subsorts NeSearchTree{X, Y} < SearchTree{X, Y} < BinTree{Record} .
  subsort NeSearchTree{X, Y} < NeBinTree{Record} .

  --- Search records, used as nodes in search trees.
  var Rec : [Record] .
  var K : X$Elt .
  var C : Y$Contents .

  op key:_ : X$Elt -> Record [ctor] .
  op key : Record ~> X$Elt .
  op numKeys : Record -> Nat .
  eq numKeys(key: K, Rec) = 1 + numKeys(Rec) .
  eq numKeys(Rec) = 0 [owise] .
  ceq key(Rec, key: K) = K if numKeys(Rec, key: K) = 1 .

  op contents:_ : Y$Contents -> Record [ctor] .
  op numContents : Record -> Nat .

```

```

op contents : Record ~> Y$Contents .
eq numContents(contents: C, Rec) = 1 + numContents(Rec) .
eq numContents(Rec) = 0 [owise] .
ceq contents(Rec, contents: C) = C
  if numContents(Rec, contents: C) = 1 .

cmb Rec : SearchRecord{X, Y}
  if numContents(Rec) = 1 /\ numKeys(Rec) = 1 .

--- Definition of binary search trees.
ops min max : NeSearchTree{X, Y} -> SearchRecord{X, Y} .

var SRec : SearchRecord{X, Y} .
vars L R : SearchTree{X, Y} .
vars L' R' : NeSearchTree{X, Y} .
var C' : Y$Contents .

mb empty : SearchTree{X, Y} .
mb empty [SRec] empty : NeSearchTree{X, Y} .
cmb L' [SRec] empty : NeSearchTree{X, Y}
  if key(max(L')) < key(SRec) .
cmb empty [SRec] R' : NeSearchTree{X, Y}
  if key(SRec) < key(min(R')) .
cmb L' [SRec] R' : NeSearchTree{X, Y}
  if key(max(L')) < key(SRec) /\ key(SRec) < key(min(R')) .

eq min(empty [SRec] R) = SRec .
eq min(L' [SRec] R) = min(L') .
eq max(L [SRec] empty) = SRec .
eq max(L [SRec] R') = max(R') .

--- Operations for binary search trees.
op insert : SearchTree{X, Y} X$Elt Y$Contents
  -> SearchTree{X, Y} .
op lookup : SearchTree{X, Y} X$Elt -> Maybe{Contents}{Y} .
op delete : SearchTree{X, Y} X$Elt -> SearchTree{X, Y} .
op find : SearchTree{X, Y} X$Elt -> Bool .

eq insert(empty, K, C) = empty [key: K, contents: C] empty .
ceq insert(L [Rec, key: K, contents: C] R, K, C')
  = L [Rec, key: K, contents: combine(C, C')] R
  if numKeys(Rec) = 0 /\ numContents(Rec) = 0 .
ceq insert(L [SRec] R, K, C) = insert(L, K, C) [SRec] R
  if K < key(SRec) .
ceq insert(L [SRec] R, K, C) = L [SRec] insert(R, K, C)
  if key(SRec) < K .

eq lookup(empty, K) = not-found .
ceq lookup(L [SRec] R, K) = C

```

```

    if key(SRec) = K /\ C := contents(SRec) .
ceq lookup(L [SRec] R, K) = lookup(L, K) if K < key(SRec) .
ceq lookup(L [SRec] R, K) = lookup(R, K) if key(SRec) < K .

eq delete(empty, K) = empty .
ceq delete(L [SRec] R, K) = delete(L, K) [SRec] R
    if K < key(SRec) .
ceq delete(L [SRec] R, K) = L [SRec] delete(R, K)
    if key(SRec) < K .
ceq delete(empty [SRec] R, K) = R if key(SRec) = K .
ceq delete(L [SRec] empty, K) = L if key(SRec) = K .
ceq delete(L' [SRec] R', K) = L' [min(R')] delete(R', key(min(R')))
    if key(SRec) = K .

eq find(empty, K) = false .
ceq find(L [SRec] R, K) = true if key(SRec) = K .
ceq find(L [SRec] R, K) = find(L, K) if K < key(SRec) .
ceq find(L [SRec] R, K) = find(R, K) if key(SRec) < K .
endfm

```

We instantiate this parameterized module, in such a way that keys become integers and contents become strings, by means of the following views:

```

view StringAsContents from CONTENTS to STRING is
  sort Contents to String .
  op combine to _+_ .
endv

view IntAsStoset from STOSSET to INT is
  sort Elt to Int .
endv

fmod SEARCH-TREE-TEST is
  protecting SEARCH-TREE{IntAsStoset, StringAsContents} .
endfm

Maude> red insert(insert(empty, 1, "a"), 2, "b") .
result NeSearchTree{IntAsStoset, StringAsContents}:
  empty
  [key: 1, contents: "a"]
  (empty [key: 2, contents: "b"] empty)

Maude> red lookup(insert(insert(insert(empty, 1, "a"),
                                2, "b"),
                                1, "c"),
                  1) .
result String: "ac"

```

10.10 AVL trees

It is well known that in order to have better efficiency on search trees one has to keep them balanced. One nice solution to this problem is provided by AVL trees; these are binary search trees satisfying the additional constraint that in each node the difference between the depth of its children is at most one. This constraint guarantees that the depth of the tree is always logarithmic with respect to the number of nodes, thus obtaining a logarithmic cost for the operations of search, lookup, insertion, and deletion, assuming that the last two are implemented in such a way that they keep the properties of the balanced tree [195, 46, 381].

As we have already anticipated in Section 10.9, it is convenient to have in each node as additional data the depth of the tree having this node as root, so that comparing the depths of children to check the balance property of the AVL trees becomes very quick. This is accomplished by importing the module `SEARCH-TREE` of search trees and adding a `depth` field to the record structure, together with the corresponding projection and the auxiliary operation `numDepths` that is used in defining an appropriate subsort `AVLRecord` of `SearchRecord`.

```
fmod AVL{X :: STOSSET, Y :: CONTENTS} is
  extending SEARCH-TREE{X, Y} .
  --- Add depth to search records.
  var N : Nat .
  var Rec : Record .

  sort AVLRecord{X, Y} .
  subsort AVLRecord{X, Y} < SearchRecord{X, Y} .
  op depth:_ : Nat -> Record [ctor] .
  op numDepths : Record -> Nat .
  op depth : Record ~> Nat .
  eq numDepths(depth: N, Rec) = 1 + numDepths(Rec) .
  eq numDepths(Rec) = 0 [owise] .
  ceq depth(Rec, depth: N) = N if numDepths(Rec) = 0 .

  var SRec : SearchRecord{X, Y} .
  cmb SRec : AVLRecord{X, Y} if numDepths(SRec) = 1 .
```

The sort `AVL{X, Y}` of AVL trees is a subsort of the sort `SearchTree{X, Y}` of search trees, defined by means of additional membership assertions; in the specification below, just two memberships are enough, one for the empty tree and the other for non-empty AVL trees. Notice the use of the *symmetric difference* operator `sd` on natural numbers; the result of this operation applied to two natural numbers is the result of subtracting the least from the greatest of the two (see Section 9.2). The commutativity of this operation is very convenient here, because we do not need to care about which one of the two trees is higher.

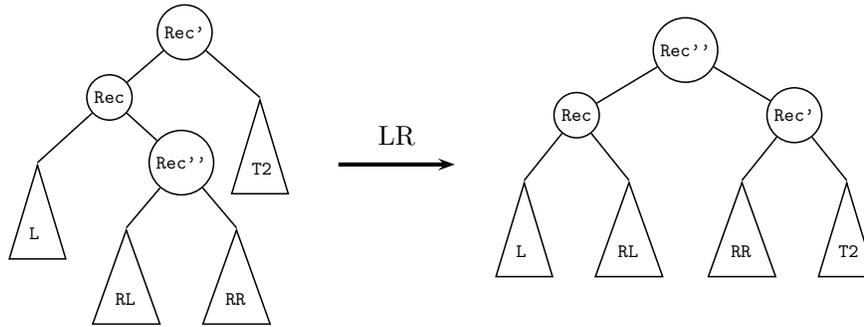


Fig. 10.1. Left-right rotation in an AVL tree

```

sorts NeAVL{X, Y} AVL{X, Y} .
subsorts NeAVL{X, Y} < AVL{X, Y} < SearchTree{X, Y} .
subsorts NeAVL{X, Y} < NeSearchTree{X, Y} .
vars AVLRec AVLRec' AVLRec'' : AVLRecord{X, Y} .
vars L R L' R' RL RR LR LL T1 T2 : AVL{X, Y} .
var ST : NeSearchTree{X, Y} .

mb empty : AVL{X, Y} .
cmb ST : NeAVL{X, Y}
  if L [AVLRec] R := ST /\ sd(depth(L), depth(R)) <= 1
    /\ 1 + max(depth(L), depth(R)) = depth(AVLRec) .

```

For lookup we use the same operation as for search trees, imported from the module `SEARCH-TREE`; on the other hand, insertion and deletion have to be redefined so that they keep the AVL properties. They work as in the general case, by comparing the given key with the one in the root, but the final result is built by means of an auxiliary `join` operation that checks that the difference between the depths of the two children is less than one, using again the symmetric difference operator `sd`; when this is not the case, appropriate *rotation* operations are invoked. It is enough to have a left rotation `lRotate` and a right rotation `rRotate`. This is quite similar to the typical imperative or object-oriented versions of these operations [195, 46, 381]. For example, the second equation for `lRotate` is illustrated in Figure 10.1.

```

op insertAVL : AVL{X, Y} X$Elt Y$Contents -> NeAVL{X, Y} .
op deleteAVL : X$Elt AVL{X, Y} -> AVL{X, Y} .
op depthAVL : AVL{X, Y} -> Nat .
op buildAVL : AVL{X, Y} Record AVL{X, Y} ~> AVL{X, Y} .
op join : AVL{X, Y} Record AVL{X, Y} ~> AVL{X, Y} .
op lRotate : AVL{X, Y} AVLRecord{X, Y} AVL{X, Y} ~> AVL{X, Y} .
op rRotate : AVL{X, Y} AVLRecord{X, Y} AVL{X, Y} ~> AVL{X, Y} .

vars K K' : X$Elt .

```

```

vars C C' : Y$Contents .

eq insertAVL(empty, K, C)
  = buildAVL(empty, (depth: 0, key: K, contents: C), empty) .
ceq insertAVL(L [Rec, key: K, contents: C'] R, K, C)
  = L [Rec, key: K, contents: combine(C, C')] R
  if numKeys(Rec) = 0 /\ numContents(Rec) = 0
    /\ numDepths(Rec) = 1 .
ceq insertAVL(L [AVLRec] R, K, C)
  = join(insertAVL(L, K, C), AVLRec, R)
  if K < key(AVLRec) .
ceq insertAVL(L [AVLRec] R, K, C)
  = join(L, AVLRec, insertAVL(R, K, C))
  if key(AVLRec) < K .

eq depthAVL(empty) = 0 .
eq depthAVL(L [AVLRec] R) = depth(AVLRec) .

ceq buildAVL(T1, (Rec, depth: N), T2)
  = T1 [Rec, depth: (max(depthAVL(T1), depthAVL(T2)) + 1)] T2
  if numDepths(Rec) = 0 /\ numKeys(Rec) = 1
    /\ numContents(Rec) = 1 .

ceq join(T1, AVLRec, T2) = buildAVL(T1, AVLRec, T2)
  if sd(depthAVL(T1), depthAVL(T2)) <= 1 .
ceq join(T1, AVLRec, T2) = lRotate(T1, AVLRec, T2)
  if depthAVL(T1) = depthAVL(T2) + 2 .
ceq join(T1, AVLRec, T2) = rRotate(T1, AVLRec, T2)
  if depthAVL(T1) + 2 = depthAVL(T2) .

ceq lRotate(L [AVLRec] R, AVLRec', T2)
  = buildAVL(L, AVLRec, buildAVL(R, AVLRec', T2))
  if depthAVL(L) >= depthAVL(R) .
ceq lRotate(L [AVLRec] R, AVLRec', T2)
  = buildAVL(buildAVL(L, AVLRec, RL), AVLRec'',
    buildAVL(RR, AVLRec', T2))
  if depthAVL(L) < depthAVL(R) /\ RL [AVLRec''] RR := R .

ceq rRotate(T1, AVLRec, L [AVLRec'] R)
  = buildAVL(buildAVL(T1, AVLRec, L), AVLRec', R)
  if depthAVL(L) <= depthAVL(R) .
ceq rRotate(T1, AVLRec, L [AVLRec'] R)
  = buildAVL(buildAVL(T1, AVLRec, LL), AVLRec'',
    buildAVL(LR, AVLRec', R))
  if depthAVL(L) > depthAVL(R) /\ LL [AVLRec''] LR := L .

--- deleteAVL and auxiliary ops.
sort Pair{X, Y} .      --- used for deleteAVLMax
op pair : AVL{X, Y} AVLRecord{X, Y} -> Pair{X, Y} [ctor].

```

```

op deleteAVLMax : NeAVL{X, Y} -> Pair{X, Y} .

eq deleteAVL(K, empty) = empty .
ceq deleteAVL(K, empty [AVLRec] R) = R
  if K = key(AVLRec) .
ceq deleteAVL(K, L [AVLRec] R) = join(L', AVLRec', R)
  if K = key(AVLRec)
    /\ pair(L', AVLRec') := deleteAVLMax(L)
  [owise] .
ceq deleteAVL(K, L [AVLRec] R) = join(L, AVLRec, deleteAVL(K, R))
  if key(AVLRec) < K .
ceq deleteAVL(K, L [AVLRec] R) = join(deleteAVL(K, L), AVLRec, R)
  if K < key(AVLRec) .

eq deleteAVLMax(L [AVLRec] empty) = pair(L, AVLRec) .
ceq deleteAVLMax(L [AVLRec] R) = pair(join(L, AVLRec, R'), AVLRec')
  if pair(R', AVLRec') := deleteAVLMax(R)
  [owise] .
endfm

```

In the following example of instantiation we use the same views, namely, `IntAsStoset` and `StringAsContents`, from the previous section.

```

fmod AVL-TEST is
  protecting AVL{IntAsStoset, StringAsContents} .
endfm

Maude> red insertAVL(
  insertAVL(
    insertAVL(
      insertAVL(
        insertAVL(insertAVL(empty, 1, "a"),
          2, "b"),
        3, "c"),
      4, "d"),
    5, "e"),
  6, "f") .
result NeAVL{IntAsStoset, StringAsContents}:
  ((empty [key: 1, contents: "a", depth: 1] empty)
   [key: 2, contents: "b", depth: 2]
   (empty [key: 3, contents: "c", depth: 1] empty))
[key: 4, contents: "d", depth:3]
(empty
 [key: 5, contents: "e", depth: 2]
 (empty [key: 6, contents: "f", depth: 1] empty))

```

The AVL tree obtained as result is displayed in Figure 10.2.

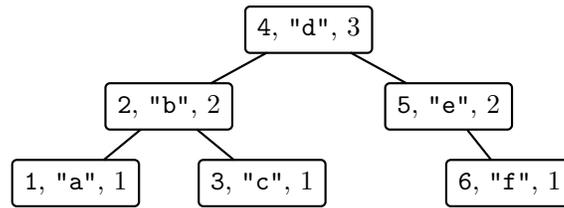


Fig. 10.2. An example of an AVL tree

10.11 2-3-4 trees

Other solutions to the problem of keeping balanced search trees are provided by *2-3 trees*, which are not treated here, and *2-3-4 trees*, whose specification we consider in this section. These search trees generalize binary search trees to a version of general trees of degree 4, so that a non-leaf node can have either 2, 3 or 4 children, and nodes can hold more than one value. The number of values in the node depends on the number of children; for example, there are two different values (let us call N1 the smallest of the two, and N2 the greatest) in the node when it has three children. Moreover, the values in the children are well organized with respect to the values in the node; in the same example, all the values in the first child must be strictly smaller than N1, all the values in the second child must be strictly greater than N1 and strictly smaller than N2, and all the values in the third child must be strictly greater than N2 (for the case of two children this organization coincides with binary search trees, and it is easy to see how it is generalized for more than three children). Furthermore, the children must have exactly the same depth, and recursively they have to satisfy the same properties. As expected, all of these properties can be stated by means of membership assertions, as shown below.

Since these trees need a different set of constructors, they have no direct relationship to binary search trees. Moreover, in order to simplify the presentation we just parameterize the specification with respect to the theory `STOSET` requiring a strict total order, that is, we consider only values in the nodes, instead of keys and associated values as we did in previous sections.

```

fmod 234TREES{T :: STOSET} is
  protecting NAT .
  sort Ne234Tree?{T} 234Tree?{T} Ne234Tree{T} 234Tree{T} .
  subsort Ne234Tree?{T} < 234Tree?{T} .
  subsort Ne234Tree{T} < 234Tree{T} < 234Tree?{T} .
  subsort Ne234Tree{T} < Ne234Tree?{T} .

  op empty234 : -> 234Tree{T} [ctor] .
  op _[_]_ : 234Tree?{T} T$Elt 234Tree?{T} -> Ne234Tree?{T} [ctor] .
  op _<_>_<_>_ : 234Tree?{T} T$Elt 234Tree?{T} T$Elt 234Tree?{T}
    -> Ne234Tree?{T} [ctor] .

```

```

op _[_]_[_]_[_]_ : 234Tree?{T} T$Elt 234Tree?{T} T$Elt
  234Tree?{T} T$Elt 234Tree?{T} -> Ne234Tree?{T} [ctor] .

vars N N1 N2 N3 : T$Elt .
vars TL TLM TC TRM TR : 234Tree{T} .

cmb TL [ N ] TR : Ne234Tree{T}
  if greaterKey(N, TL) /\ smallerKey(N, TR)
    /\ depth(TL) = depth(TR) .
cmb TL < N1 > TC < N2 > TR : Ne234Tree{T}
  if N1 < N2
    /\ greaterKey(N1, TL) /\ smallerKey(N1, TC)
    /\ greaterKey(N2, TC) /\ smallerKey(N2, TR)
    /\ depth(TL) = depth(TC) /\ depth(TC) = depth(TR) .
cmb TL { N1 } TLM { N2 } TRM { N3 } TR : Ne234Tree{T}
  if N1 < N2 /\ N2 < N3
    /\ greaterKey(N1, TL) /\ smallerKey(N1, TLM)
    /\ greaterKey(N2, TLM) /\ smallerKey(N2, TRM)
    /\ greaterKey(N3, TRM) /\ smallerKey(N3, TR)
    /\ depth(TL) = depth(TLM) /\ depth(TL) = depth(TRM)
    /\ depth(TL) = depth(TR) .

--- Auxiliary operations

op depth : 234Tree?{T} -> Nat .
op greaterKey : T$Elt 234Tree{T} -> Bool .
--- true if first argument is greater than all keys in the tree.
op smallerKey : T$Elt 234Tree{T} -> Bool .
--- true if first argument is smaller than all keys in the tree.
op maxKey : Ne234Tree{T} -> T$Elt .
--- greatest element in a tree.
op minKey : Ne234Tree{T} -> T$Elt .
--- smallest element in a tree.

```

Although the number of cases to consider increases according to all the possible different nodes and situations, the specification of the `find` operation is immediate.

```

op find : T$Elt 234Tree{T} -> Bool .

eq find(M, empty234) = false .
ceq find(M, T1 [N1] T2) = find(M, T1) if M < N1 .
eq find(M, T1 [M] T2) = true .
ceq find(M, T1 [N1] T2) = find(M, T2) if N1 < M .
ceq find(M, T1 < N1 > T2 < N2 > T3) = find(M, T1) if M < N1 .
eq find(M, T1 < M > T2 < N2 > T3) = true .
ceq find(M, T1 < N1 > T2 < N2 > T3) = find(M, T2)
  if N1 < M /\ M < N2 .
eq find(M, T1 < N1 > T2 < M > T3) = true .

```

```

ceq find(M, T1 < N1 > T2 < N2 > T3) = find(M, T3) if N2 < M .
ceq find(M, T1 { N1 } T2 { N2 } T3 { N3 } T4) = find(M, T1)
  if M < N1 .
eq find(M, T1 { M } T2 { N2 } T3 { N3 } T4) = true .
ceq find(M, T1 { N1 } T2 { N2 } T3 { N3 } T4) = find(M, T2)
  if N1 < M /\ M < N2 .
eq find(M, T1 { N1 } T2 { M } T3 { N3 } T4) = true .
ceq find(M, T1 { N1 } T2 { N2 } T3 { N3 } T4) = find(M, T3)
  if N2 < M /\ M < N3 .
eq find(M, T1 { N1 } T2 { N2 } T3 { M } T4) = true .
ceq find(M, T1 { N1 } T2 { N2 } T3 { N3 } T4) = find(M, T4)
  if N3 < M .

```

On the other hand, even though the main ideas of insertion are quite simple, the details of its implementation become much lengthier than expected, requiring several auxiliary operations and several equations to treat the different cases arising from combining the different constructors. Even worse is the implementation of deletion, which needs a *zillion* of equations to deal with all possible cases. All these details are not shown here and can be found in the set of book examples in <http://maude.cs.uiuc.edu> and in the companion cd-rom.

We do some reductions with an example of instantiation that uses the view `IntAsStoset` from `STOSET` to `INT`.

```

fmod 234TREES-TEST is
  protecting 234TREES{IntAsStoset} .

  op tree : -> 234Tree{IntAsStoset} .

  eq tree
    = insert(100, insert(90, insert(15, insert(80,
      insert(70, insert(40, insert(50, insert(20,
        insert(60, insert(30, insert(10, empty234)))))))))) .
endfm

Maude> red tree .
result Ne234Tree{IntAsStoset}:
  ((empty234 {10} empty234 {15} empty234 {20} empty234)
    [30]
    (empty234 [40] empty234))
  [50]
  ((empty234 [60] empty234)
    [70]
    (empty234 {80} empty234 {90} empty234 {100} empty234))

Maude> red find(30, tree) .
result Bool: true

Maude> red delete(30, tree) .

```

```

result Ne234Tree{IntAsStoset}:
  (empty234 < 10 > empty234 < 15 > empty234)
  {20}
  empty234 [40] empty234
  {50}
  empty234 [60] empty234
  {70}
  (empty234 {80} empty234 {90} empty234 {100} empty234)

```

10.12 Red-black trees

Yet another solution to the problem of keeping search trees balanced are *red-black* search trees. These are standard binary search trees that satisfy several additional constraints that are related to a color (hence the name!) that can be associated with each node (in some presentations, to the edges). One can think of red-black trees as a binary representation of 2-3-4 search trees, and this provides helpful intuition.

Since the color is additional information in each node, we again make use of the record construction described in Section 10.9, defining a new subsort `RBRecord` of `SearchRecord`.

```

fmod RB-TREES{X :: STOSSET, Y :: CONTENTS} is
  extending SEARCH-TREE{X, Y} .
  --- Add color to search records.
  var Rec : Record .
  var Co : Color .

  sorts Color RBRecord{X, Y} .
  subsort RBRecord{X, Y} < SearchRecord{X, Y} .
  ops r b : -> Color [ctor].
  op color:_ : Color -> Record [ctor] .
  op numColors : Record -> Nat .
  op color : Record ~> Color .
  eq numColors(color: Co, Rec) = 1 + numColors(Rec) .
  eq numColors(Rec) = 0 [owise] .
  ceq color(Rec, color: Co) = Co if numColors(Rec) = 0 .

  var SRec : SearchRecord{X, Y} .
  cmb SRec : RBRecord{X, Y} if numColors(SRec) = 1 .

```

Once more, memberships allow a faithful specification of all the constraints.

```

sorts NeRBTree{X, Y} RBTree{X, Y} .
subsort NeRBTree{X, Y} < RBTree{X, Y} < SearchTree{X, Y} .
subsort NeRBTree{X, Y} < NeSearchTree{X, Y} .

```

```

var RBRec : RBRecord{X, Y} .
vars ST RBTL? RBTR? : SearchTree{X, Y} .

mb empty : RBTree{X, Y} .
cmb ST : NeRBTree{X, Y}
  if RBTL? [RBRec] RBTR? := ST /\ color(RBRec) = b
    /\ blackDepth(RBTR?) = blackDepth(RBTR?)
    /\ blackBalance(RBTL?) /\ blackBalance(RBTR?)
    /\ not twoRed(RBTL?) /\ not twoRed(RBTR?) .

--- Auxiliary operations.
op blackDepth : BinTree{Record} ~> Nat .
op blackBalance : BinTree{Record} -> Bool .
op twoRed : BinTree{Record} ~> Bool .

```

The specification of the insertion and deletion operations is quite long, making use of several additional auxiliary operations. All the corresponding details can be found in the set of book examples in <http://maude.cs.uiuc.edu> and in the companion cd-rom.

In the following example of instantiation we use again the two views `IntAsStoset` and `StringAsContents`, that were introduced in Section 10.9 above.

```

fmod RB-TREES-TEST is
  protecting RB-TREES{IntAsStoset, StringAsContents} .
  op tree : -> RBTree{IntAsStoset, StringAsContents} .
  eq tree = insertRB(3, "mi",
    insertRB(1, "do",
      insertRB(5, "sol",
        insertRB(2, "re",
          insertRB(4, "fa", empty)))))) .
endfm

Maude> red tree .
result NeRBTree{IntAsStoset, StringAsContents}:
  (empty [key: 1, contents: "do", color: b] empty)
  [key: 2, contents: "re", color: b]
  ( (empty [key: 3, contents: "mi", color: r] empty)
    [key: 4, contents: "fa", color: b]
    (empty [key: 5, contents: "sol", color: r] empty))

Maude> red delete(4, delete(3, tree)) .
result NeRBTree{IntAsStoset, StringAsContents}:
  (empty [key: 1, contents: "do", color: b] empty)
  [key: 2, contents: "re", color: b]
  (empty [key: 5, contents: "sol", color: b] empty)

```

10.13 Efficiency concerns

As we have already mentioned in some examples, we have not been especially concerned about efficiency in the specification of the previous data structures in Maude.

There are a number of techniques to improve the efficiency, like the systematic use of *unconditional* equations with the help of the `if.then.else.fi` operator and the `owise` attribute (see Section 24.1.4). Some of these techniques (for example, the use of `owise`) have the drawback of making formal reasoning about the specifications much more difficult or even unfeasible with the currently available tools, like the ITP tool (see Section 23.1.1). Since we were interested in formally verifying some properties of these data structures, we have used simpler although less efficient specifications; some formal proofs appear in the paper [235], from which this chapter has been adapted.

There is also the tradeoff between efficiency and very precise typing by means of memberships; these require typechecking during execution. If one is willing to work with less refined types, as it is the case with other functional languages, one can forget about most of the memberships that appear in our specifications, thus obtaining another considerable speedup.

The main moral of this discussion is that within the same logical framework we have a *spectrum of possibilities*: we may place the emphasis on formal specification and reasoning, as we have done in this chapter, or we may instead focus on efficient programming solutions. And we can relate both kinds of solutions by using refinement techniques. This gives us a way to negotiate the usual tensions between ease of proof and efficient implementation.

Object-Based Programming

Distributed systems can be naturally modeled in Maude as multisets of entities, loosely coupled by some suitable communication mechanism. An important example is object-based distributed systems in which the entities are objects, each with a unique identity, and the communication mechanism is message passing.

Core Maude supports the modeling of object-based systems by providing a predefined module `CONFIGURATION` that declares sorts representing the essential concepts of object, message, and configuration, along with a notation for object syntax that serves as a common language for specifying object-based systems. In addition, there is an *object-message fair* rewriting strategy that is well suited for executing object system configurations. To specify an object-based system, the user can import `CONFIGURATION` and then define the particular objects, messages, and rules for interaction that are of interest. In addition to simple asynchronous message passing, Maude also supports complex patterns of synchronous interaction that can be used to model higher-level communication abstractions. The user is also free to define his/her own notation for configurations and objects, and can still take advantage of the object-message rewriting strategy, simply by making the appropriate declarations. All this is explained in detail below.

Furthermore, Maude also supports *external objects*, so that objects inside a Maude configuration can interact with different kinds of objects outside it. At present, the external objects directly supported are internet sockets; but through them it is possible to interact with other external objects. In addition, sockets make possible distributed programming with rewrite rules. External objects are discussed in Section 11.4. A substantial case study using sockets to build a mobile language is presented in Chapter 17.

As discussed in Chapter 21, Full Maude provides additional support for object-oriented programming with classes, subclassing, and convenient abbreviations for rule syntax.

11.1 Configurations

The predefined module `CONFIGURATION` in the file `prelude.maude` provides basic sorts and constructors for modeling object-based systems.

```

mod CONFIGURATION is
  *** basic object system sorts
  sorts Object Msg Configuration .

  *** construction of configurations
  subsort Object Msg < Configuration .
  op none : -> Configuration [ctor] .
  op __ : Configuration Configuration -> Configuration
        [ctor config assoc comm id: none] .

```

The basic sorts needed to describe an object system are: `Object`, `Msg` (messages), and `Configuration`. A configuration is a multiset of objects and messages that represents (a snapshot of) a possible system state. Configurations are formed by multiset union (represented by empty syntax, `__`) starting from singleton objects and messages. The empty configuration is represented by the constant `none`. The attribute `config` declares that configurations constructed with `__` support the special object-message fair rewriting behavior (see Section 11.2).

A typical configuration will have the form

$$\langle Ob-1 \rangle \dots \langle Ob-k \rangle \langle Mes-1 \rangle \dots \langle Mes-n \rangle$$

where $\langle Ob-1 \rangle, \dots, \langle Ob-k \rangle$ are objects, $\langle Mes-1 \rangle, \dots, \langle Mes-n \rangle$ are messages, and the order is immaterial.

In general, a rewrite rule for an object system has the form

$$\begin{aligned}
 & r1 \langle Ob-1 \rangle \dots \langle Ob-k \rangle \langle Mes-1 \rangle \dots \langle Mes-n \rangle \\
 & \Rightarrow \langle Ob'-1 \rangle \dots \langle Ob'-j \rangle \langle Ob-k+1 \rangle \dots \langle Ob-m \rangle \langle Mes'-1 \rangle \dots \langle Mes'-p \rangle .
 \end{aligned}$$

where $\langle Ob'-1 \rangle, \dots, \langle Ob'-j \rangle$ are updated versions of $\langle Ob-1 \rangle, \dots, \langle Ob-j \rangle$ for $j \leq k$, $\langle Ob-k+1 \rangle, \dots, \langle Ob-m \rangle$ are newly created objects, and $\langle Mes'-1 \rangle, \dots, \langle Mes'-p \rangle$ are new messages. An important special case are rules with a single object and at most one message on the lefthand side. These are called *asynchronous* rules. They directly model asynchronous distributed interactions. Rules involving multiple objects are called *synchronous*; they are used to model higher-level communication abstractions.

The user is free to define any object or message syntax that is convenient. However, for uniformity in identifying objects and message receivers, the adopted convention is that *the first argument of an object or message constructor should be an object's name*. This facilitates defining object system rewriting strategies independently of the particular choice of syntax and is essential for using Maude's object-message fair rewriting strategy.

The remainder of the `CONFIGURATION` module provides an object syntax that serves as a common notation that can be used by developers of object-based system specifications. This syntax is also used by Full Maude (see Chapter 21). For this purpose four new sorts are introduced: `Oid` (object identifiers), `Cid` (class identifiers), `Attribute` (a named element of an object's state), and `AttributeSet` (multisets of attributes). Further details about the `CONFIGURATION` module are discussed later in Section 11.4.

```

*** Maude object syntax
sorts Oid Cid .
sorts Attribute AttributeSet .
subsort Attribute < AttributeSet .
op none : -> AttributeSet [ctor] .
op _,_ : AttributeSet AttributeSet -> AttributeSet
      [ctor assoc comm id: none] .
op <_:_|_> : Oid Cid AttributeSet -> Object [ctor object] .
endm

```

In this syntax, objects have the general form

$$\langle O : C \mid \langle att-1 \rangle, \dots, \langle att-k \rangle \rangle$$

where O is an object identifier, C is a class identifier, and $\langle att-1 \rangle, \dots, \langle att-k \rangle$ are the object's attributes. Attribute sets are formed from singleton attributes by a multiset union operator `_,_` with identity `none` (the empty multiset). The `object` attribute in the `<_:_|_>` operator declares that objects made with this constructor have object-message fair rewriting behavior (see Section 11.2).

Although the user is free to define the syntax of elements of sort `Attribute` according to taste, we will follow the standard Maude notation in most of our examples. The module `BANK-ACCOUNT` illustrates the use of the Maude object syntax to define simple bank account objects. Note that by defining the attribute `bal` with syntax `bal :_` we are able to write account objects as $\langle A : Account \mid bal : N \rangle$.

```

mod BANK-ACCOUNT is
  protecting INT .
  inc CONFIGURATION .
  op Account : -> Cid [ctor] .
  op bal :_ : Int -> Attribute [ctor gather (&)] .
  ops credit debit : Oid Nat -> Msg [ctor] .
  vars A B : Oid .
  vars M N N' : Nat .

  rl [credit] :
    < A : Account | bal : N >
    credit(A, M)
    => < A : Account | bal : N + M > .

  crl [debit] :

```

```

< A : Account | bal : N >
debit(A, M)
=> < A : Account | bal : N - M >
if N >= M .
endm

```

The class identifier for bank account objects is `Account`. Each account object has a single attribute named `bal` of sort `Nat` (the account balance). There are two message constructors `credit` and `debit`, each taking an object identifier (the receiver) and a number (the amount to credit or debit). The rule labeled `credit` describes the processing of a credit message and the rule labeled `debit` describes the processing of a debit message. Suppose that constants `A-001`, `A-002`, and `A-003` of sort `Oid` have been declared. Then, the following is an example of a bank account configuration.

```

< A-001 : Account | bal : 300 >
< A-002 : Account | bal : 250 >
< A-003 : Account | bal : 1250 >
debit(A-001, 200)
debit(A-002, 400)
debit(A-003, 300)
credit(A-002, 300)

```

Note that the messages `debit(A-001, 200)` and `debit(A-003, 300)` can be delivered concurrently, either before or after the other messages. However, the message `debit(A-002, 400)` cannot be delivered until after `credit(A-002, 300)` has been delivered, due to the balance condition for the `debit` rule.

The `credit` and `debit` rules are examples of asynchronous message passing rules involving one object and one message on the lefthand side. In these examples no new objects are created and no new messages are sent.

In order to combine the `debit(A-003, 300)` and `credit(A-002, 300)` messages so that the delivery of these two messages becomes a single atomic transaction, we could define a new message constructor `from_to_transfer_`. The rule for handling a transfer message involves the joint participation of two bank accounts in the transfer, as well as the transfer message. This is an example of a synchronous rule.

```

op from_to_transfer_ : Oid Oid Nat -> Msg [ctor] .
crl [transfer] :
  (from A to B transfer M)
  < A : Account | bal : N >
  < B : Account | bal : N' >
  => < A : Account | bal : N - M >
     < B : Account | bal : N' + M >
  if N >= M .

```

Now we could replace

```
debit(A-003, 300) credit(A-002,300)
```

by

```
from A-003 to A-002 transfer 300
```

in the example configuration. The module `BANK-ACCOUNT-TEST` declares the object identifiers introduced above and defines a configuration constant `bankConf`.

```
mod BANK-ACCOUNT-TEST is
  ex BANK-ACCOUNT .
  ops A-001 A-002 A-003 : -> Obj .
  op bankConf : -> Configuration .
  eq bankConf
    = < A-001 : Account | bal : 300 >
      debit(A-001, 200)
      debit(A-001, 150)
    < A-002 : Account | bal : 250 >
      debit(A-002, 400)
    < A-003 : Account | bal : 1250 >
      (from A-003 to A-002 transfer 300) .
endm
```

From the specification we see that only one of the `debit` messages for `A-001` can be processed. Using the default rewriting strategy we find that the message `debit(A-001, 150)` is processed first in this strategy.

```
Maude> rew in BANK-ACCOUNT-TEST : bankConf .
result Configuration:
  debit(A-001, 200)
  < A-001 : Account | bal : 150 >
  < A-002 : Account | bal : 150 >
  < A-003 : Account | bal : 950 >
```

We use the search command to confirm that it is possible to process the message `debit(A-001, 200)` as well, where the `=>!` symbol indicates that we are searching for states reachable from `bankConf` that cannot be further rewritten (see Sections 6.4.3 and 25.4).

```
Maude> search bankConf =>! C:Configuration debit(A-001, 150) .
search in BANK-ACCOUNT-TEST : bankConf
=>! C:Configuration debit(A-001, 150) .
```

```
Solution 1 (state 8)
states: 9 rewrites: 49 in 0ms cpu (0ms real) (~ rews/sec)
C:Configuration --> < A-001 : Account | bal : 100 >
  < A-002 : Account | bal : 150 >
  < A-003 : Account | bal : 950 >
```

```
No more solutions.
states: 9 rewrites: 49 in 0ms cpu (0ms real) (~ rews/sec)
```

The `BANK-MANAGER` module below illustrates asynchronous message passing with object creation.

```

mod BANK-MANAGER is
  inc BANK-ACCOUNT .
  op Manager : -> Cid [ctor] .
  op new-account : Oid Oid Nat -> Msg [ctor] .
  vars O C : Oid .
  var N : Nat .
  rl [new] :
    < O : Manager | none >
    new-account(O, C, N)
    => < O : Manager | none >
        < C : Account | bal : N > .
endm

```

To open a new account, one sends a message to the bank manager with the account name and initial balance, for example, `new-account(A-000, A-004, 100)`. Of course, in a real system more care would be needed to assure unique account identities. To see the bank manager in action, we define the following module.

```

mod BANK-MANAGER-TEST is
  ex BANK-MANAGER .
  ops A-001 A-002 A-003 A-004 : -> Oid .
  op mgrConf : -> Configuration .
  eq mgrConf
    = < A-001 : Account | bal : 300 >
      < A-004 : Manager | none >
      new-account(A-004, A-002, 250)
      new-account(A-004, A-003, 1250) .
endm

```

Then, we rewrite the configuration `mgrConf`:

```

Maude> rew in BANK-MANAGER-TEST : mgrConf .
result Configuration:
  < A-001 : Account | bal : 300 >
  < A-002 : Account | bal : 250 >
  < A-003 : Account | bal : 1250 >
  < A-004 : Manager | none >

```

The relationships between all the modules involved in this example are illustrated in Figure 11.1, where the different types of arrows correspond to the different modes of importation: single arrow for `including`, double arrow for `extending`, and triple arrow for `protecting`.

The examples above illustrate object-based programming in Maude using the common object syntax. Notice that message constructors obey the “first argument is an object identifier” convention. Alternative object syntax is also

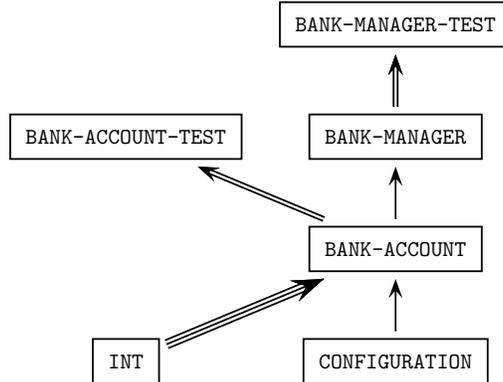


Fig. 11.1. Importation graph of bank modules

possible, by defining an associative and commutative configuration constructor and suitable object and message syntax. It is of course also possible not to use the `config` attribute when defining the multiset union operator, but this will prevent taking advantage of object-message fair rewriting (see Section 11.2). As an example (not using for the moment the `config` attribute, to illustrate different forms of rewriting with objects), we model a *ticker*, the classic example of an actor [1, 351]. First we specify the configurations, objects, and messages of the actor world in the module `ACTOR-CONF`. Actor configurations (of sort `AConf`) are multisets of actors (of sort `Actor`) and messages (of sort `Msg`). Messages are constructed uniformly from an actor identifier and a message body. Thus we introduce sorts `Aid` (actor identifier) and `MsgBody`, and a message constructor `_<|_`.

```

mod ACTOR-CONF is
  sorts Actor Msg AConf .
  subsorts Actor Msg < AConf .
  op none : -> AConf [ctor] .
  op _&_ : AConf AConf -> AConf [ctor assoc comm id: none] .
  *** actor messages
  sorts Aid MsgBody .
  op _<|_ : Aid MsgBody -> Msg [ctor] .
endm

```

A ticker maintains a counter that it updates in response to a `tick` message. `Ticker(T:Aid, N:Nat)` is an actor with identifier `T:Aid` and counter value `N:Nat`. The ticker sends the current value of its counter in response to a `timeReq` message.

```

mod TICKER is
  including ACTOR-CONF .
  protecting NAT .

```

```

op Ticker : Aid Nat -> Actor [ctor] .
op tick : -> MsgBody [ctor] .
op timeReq : Aid -> MsgBody [ctor] .
op timeReply : Nat -> MsgBody [ctor] .

vars T C : Aid .
var N : Nat .
rl Ticker(T, N) (T <| tick)
=> Ticker(T, s N) (T <| tick) .
rl Ticker(T, N) (T <| timeReq(C))
=> Ticker(T, N) (C <| timeReply(N)) .
endm

```

To test the ticker we define actor identifiers for the ticker, `myticker`, a customer, `me`, and an initial configuration with one ticker, one `tick` message, and a `timeReq` message from `me`.

```

mod TICKER-TEST is
  extending TICKER .
  ops myticker me : -> Aid [ctor] .
  op tConf : -> AConf .
  eq tConf
    = Ticker(myticker, 0)
      (myticker <| tick)
      (myticker <| timeReq(me)) .
endm

```

If we ask Maude to rewrite the configuration `tConf` without placing an upper bound on the number of rewrites, Maude will go on forever. This is because there will always be a `tick` message in the configuration, and the ticker can always process this message. Thus we rewrite with an upper bound of, say, 10 rewrites.

```

Maude> rew [10] tConf .
rewrite [10] in TICKER-TEST : tConf myticker <| timeReq(me) .
result AConf:
  (myticker <| tick) (me <| timeReply(1)) Ticker(myticker, 9)

```

We see that the `timeReq` message was processed after just one `tick` was processed.

An interesting property of this configuration is that the reply to the `timeReq` message can contain an arbitrarily large natural number, since any number of `ticks` could be processed before the `timeReq`. For particular numbers this can be checked using the `search` command.

```

Maude> search [1] tConf =>+ tc:AConf me <| timeReply(100) .
search [1] in TICKER-TEST :
  tConf =>+ tc:AConf me <| timeReply(100) .

```

```
Solution 1 (state 5152)
states: 5153 rews: 5153 in 0ms cpu (285ms real)(~ rews/sec)

tc:AConf --> (myticker <| tick) Ticker(myticker, 100)
```

Notice that we used the search relation \Rightarrow^+ (one or more steps) rather than $\Rightarrow!$ (terminating rewrites) since there are no terminal configurations starting from `tConf`. Moreover, we have searched only for the first (`[1]`) solution.

There are two important considerations regarding object systems that are not illustrated by the preceding examples: *uniqueness of object names* and *fairness of message delivery*. To illustrate some of the issues we elaborate the ticker example by defining a ticker factory that creates tickers, and a ticker-customer. The ticker factory accepts requests for new tickers `newReq(c)` where `c` is the customer's name. When such a request is received, a ticker is created and its name is sent to the requesting customer (`newReply(o(a, i))`). To make sure that each ticker has a fresh (unused) name, the ticker factory keeps a counter. It generates ticker names of the form `o(a, i)`, where `a` is the factory name and `i` is the counter value. The counter is incremented each time a ticker is created. This is just one possible method for assuring unique names for dynamically created objects. If objects are only created by factories that use the above method for generating names, then starting from a configuration of objects with unique names (not of the form `o(a, i)`) the unique name property will be preserved.

```
mod TICKER-FACTORY is
  inc TICKER .
  op TickerFactory : Aid Nat -> Actor [ctor] .
  ops newReq newReply : Aid -> MsgBody [ctor] .
  op o : Aid Nat -> Aid [ctor] .

  vars A C : Aid .
  vars I J : Nat .
  rl [newReq] :
    TickerFactory(A, I) (A <| newReq(C))
    => TickerFactory(A, s I) (C <| newReply(o(A, I)))
      Ticker(o(A, I), 0) (o(A, I) <| tick) .
endm
```

A ticker customer knows the name of a ticker factory. It asks for a ticker, waits for a reply, asks the ticker for the time, waits for a reply, increments its reply counter (used just for the user to monitor customer service) and repeats this process.

```
mod TICKER-CUSTOMER is
  inc TICKER-FACTORY .
  ops Cust Cust1 Cust2 : Aid Aid Nat -> Actor [ctor] .

  vars C TF T : Aid .
```

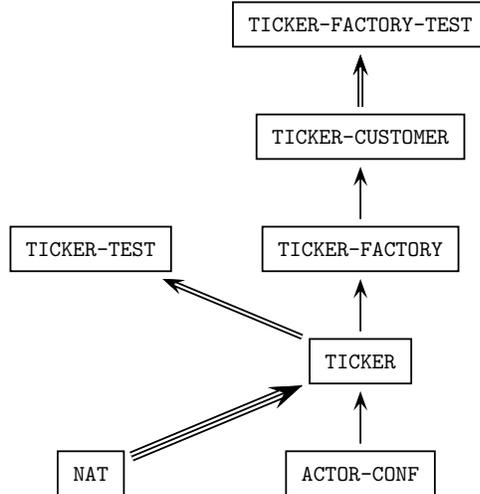


Fig. 11.2. Importation graph of ticker modules

```

vars N M : Nat .

r1 [req] :
  Cust(C, TF, N)
  => Cust1(C, TF, N) (TF <| newReq(C)) .

r1 [newReply] :
  Cust1(C, TF, N) (C <| newReply(T))
  => Cust2(C, TF, N) (T <| timeReq(C)) .

r1 [timeReply] :
  Cust2(C, TF, N) (C <| timeReply(M))
  => Cust(C, TF, s N) .
endm

```

Now we define a test configuration with a ticker factory and two customers. The importation graph of all the modules involved at this point is shown in Figure 11.2.

```

mod TICKER-FACTORY-TEST is
  ex TICKER-CUSTOMER .
  ops tf c1 c2 : -> Aid [ctor] .
  ops ic1 ic2 : -> AConf .
  eq ic1 = TickerFactory(tf, 0) Cust(c1, tf, 0) .
  eq ic2 = ic1 Cust(c2, tf, 0) .
endm

```

Rewriting this configuration using the `rewrite` command with a bound of 40 results in one ticker being created, and ticking away, while customer `c2` is not given an opportunity to execute at all.

```
Maude> rew [40] ic2 .
rewrite [40] in TICKER-FACTORY-TEST : ic2 .
rewrites: 42 in 0ms cpu (0ms real) (~ rewrites/second)
result AConf:
  (o(tf, 0) <| tick)
  Ticker(o(tf, 0), 35) TickerFactory(tf, 1)
  Cust(c1, tf, 1) Cust(c2, tf, 0)
```

In contrast, rewriting using the `frewrite` strategy with the same bound of 40, several tickers are created, however only the first one gets `tick` messages delivered.

```
Maude> frew [40] ic2 .
frewrite [40] in TICKER-FACTORY-TEST : ic2 .
rewrites: 42 in 0ms cpu (1ms real) (~ rewrites/second)
result (sort not calculated):
  (o(tf, 0) <| tick) (o(tf, 1) <| tick)
  (o(tf, 2) <| tick) (o(tf, 3) <| tick)
  (o(tf, 4) <| tick) (o(tf, 5) <| tick)
  (o(tf, 6) <| tick)
  (o(tf, 6) <| timeReq(c1))
  Ticker(o(tf, 0), 6) Ticker(o(tf, 1), 0)
  Ticker(o(tf, 2), 0) Ticker(o(tf, 3), 0)
  Ticker(o(tf, 4), 0) Ticker(o(tf, 5), 0)
  Ticker(o(tf, 6), 0)
  TickerFactory(tf, 7)
  ((tf <| newReq(c2))
  Cust1(c2, tf, 3)) Cust2(c1, tf, 3)
```

The number of rewrites reported by Maude includes both equational and rule rewrites. In the examples above there were 2 equational rewrites (the two equations defining the initial configuration `ic2` and its subconfiguration `ic1`) and 40 rule rewrites. If you execute the command

```
Maude> set profile on .
```

(see Section 24.1.4) before rewriting and then execute

```
Maude> show profile .
```

you will discover that executing the `rewrite` command the rule delivering the `tick` message is used 35 times and the other rules are each used once, while executing the `frewrite` command the `tick` rule is executed only 6 times and each of the other rules are executed between 6 and 8 times.

Turning profiling on substantially reduces performance, so you will want to turn it off

```
Maude> set profile off .
```

when you have found out what you want to know.

Note that `frewrite` uses a fair rewriting strategy, but since it does not know about objects, messages, and configurations, it can only follow a position-fair strategy. As we will explain in the next section, in order to enable the object-message fair rewriting we need only do three things:

- give to the constructor of object and message configurations the `config` attribute,
- give to the message constructor the `message` attribute, and
- give to each object constructor the `object` attribute.

To maintain the separate rewriting semantics we also modify the name of each module by putting an `O` at the front (except for `ACTOR-CONF` which we rename `ACTOR-O-CONF`). Thus we modify the configuration, actor, and message constructor declarations as follows.

```
mod ACTOR-O-CONF is
  ...
  op __ : AConf AConf -> AConf [ctor config assoc comm id: none] .
  op _<|_ : Aid MsgBody -> Msg [ctor message] .
  ...
endm

mod O-TICKER is
  ...
  op Ticker : Aid Nat -> Actor [ctor object] .
  ...
endm

mod O-TICKER-FACTORY is
  ...
  op TickerFactory : Aid Nat -> Actor [ctor object] .
  ...
endm

mod O-TICKER-CUSTOMER is
  ...
  ops Cust Cust1 Cust2 : Aid Aid Nat -> Actor [ctor object] .
  ...
endm
```

Now the `frewrite` command will use *object-message fair rewriting*, as explained in detail in the next section. The counting of object-message rewrites has two aspects: for the purposes of the rewrite argument given to `frewrite`, a visit to a configuration that results in one or more rewrites counts as a single rewrite; though for other accounting purposes all rewrites are counted.

For example, with an upper bound of 40 as above, thirteen tickers are created. To simplify the output we show the results for rewriting with a bound of 20.

```
Maude> frew [20] ic2 .
frewrite [20] in 0-TICKER-FACTORY-TEST : ic2 .
rewrites: 76 in 0ms cpu (1ms real) (~ rewrites/second)
result (sort not calculated):
  (o(tf, 0) <| tick) (o(tf, 1) <| tick)
  (o(tf, 2) <| tick) (o(tf, 3) <| tick)
  (o(tf, 4) <| tick) (o(tf, 5) <| tick)
  Ticker(o(tf, 0), 11) Ticker(o(tf, 1), 11)
  Ticker(o(tf, 2), 7) Ticker(o(tf, 3), 7)
  Ticker(o(tf, 4), 3) Ticker(o(tf, 5), 3)
  TickerFactory(tf, 6)
  ((tf <| newReq(c1)) Cust1(c1, tf, 3))
  (tf <| newReq(c2)) Cust1(c2, tf, 3)
```

Notice that each ticker gets a chance to tick (tickers created later will show less time passed), and each customer is treated fairly. In fact using profiling we find that the `tick` rule is used 42 times (which is the total of the counts for the six tickers created), while the other rules are used 6-8 times and there are 2 equational rewrites as before.

Suppose that we try to violate the unique name condition, for example by adding a copy of customer `c1` to the test configuration. When Maude discovers this (it may take a few rewrites), a warning is issued.

```
Maude> frew [4] ic2 Cust(c1, tf, 0) .
Warning: saw duplicate object: Cust1(c1, tf, 0)
frewrite [4] in 0-TICKER-FACTORY-TEST : ic2 Cust(c1, tf, 0) .
rewrites: 8 in 0ms cpu (0ms real) (~ rewrites/second)
result AConf:
  (c1 <| newReply(o(tf, 0))) (c1 <| newReply(o(tf, 1)))
  (c2 <| newReply(o(tf, 2)))
  (o(tf, 0) <| tick) (o(tf, 1) <| tick) (o(tf, 2) <| tick)
  Ticker(o(tf, 0), 0) Ticker(o(tf, 1), 0) Ticker(o(tf, 2), 0)
  TickerFactory(tf, 3)
  Cust1(c1, tf, 0) Cust1(c1, tf, 0) Cust1(c2, tf, 0)
```

11.2 Object-message fair rewriting

Object-message fair rewriting is a special rewriting strategy associated with configuration constructors which are declared with the `config` attribute. Configuration constructors must be associative and commutative, and may optionally have an identity element. The empty syntax constructors in the `CONFIGURATION` and `ACTOR-0-CONF` modules above (which have been given the `config` attribute) are examples of valid configuration constructors, but

such default syntax can easily be changed by renaming the `--` operator (see Section 8.2.2). Configurations only have their special behavior with respect to arguments that are constructed using operators that are object or message constructors, that is, they are declared with the `object` or `message` attribute. Such object and message constructors must have at least one argument. Examples include the Maude object constructor in `CONFIGURATION`, the various actor constructors imported into `O-TICKER-FACTORY-TEST`, all of which have been given the `object` attribute, and the actor message constructor which has been given the `message` attribute (which can be abbreviated as `msg`).

An operator can have at most one of the three attributes: `config`, `object`, and `message`. For object constructors, the first argument is considered to be the object's name. For message constructors, the first argument is considered to be the message's target or addressee. There may be multiple configuration, object and message constructors. A rule is considered to be an *object-message rule* if the following requirements hold:

1. Its lefthand side has a configuration constructor on top with two arguments A and B ,
2. A and B are stable (that is, they cannot change their top symbol under a substitution),
3. A has a message constructor on top,
4. B has an object constructor on top, and
5. The first arguments of A and B are identical.

For example, the rules `newReply` and `timeReply` in the `O-TICKER-CUSTOMER` module are object-message rules (because configurations are associative and commutative A and B can appear in the rule in either order) while the rule labeled `req` is not, because there is no message term, only an object, in its lefthand side. This rule will be applied in the rewriting that happens after all the enabled object-message rules have been applied, as discussed below.

The object-message fair behavior appears with the command `frewrite` (and at the metalevel with the descent function `metaFrewrite`—see Section 14.5.3). When the fair traversal attempts to perform a single rewrite on a term headed by a configuration constructor, the following happens:

1. Arguments headed by object constructors are collected. It is a runtime error for more than one object to have the same name.
2. For each object, messages with its name as first argument are collected and placed in a queue.
3. Any remaining arguments are placed on a remainder list.
4. For each object, and for each message in its queue, an attempt is made to deliver the message by performing a rewrite with an object-message rule. If all applicable rules fail, the message is placed on the remainder list. If a rule succeeds, the righthand side is constructed, reduced, and the result is searched for the object. Any other arguments in the result configuration go onto the remainder list. If the object cannot be found, any messages

left in its queue go onto the remainder list. Once its message queue is exhausted, the object itself is placed on the remainder list.

5. A new term is constructed using the configuration constructor on top of the arguments in the remainder list. This is reduced, and a single rewrite using the non-object-message rules is attempted.

There is no restriction on object names, other than uniqueness. An object may change its object constructor during the course of a rewrite and delivery of any remaining message will still be attempted.¹ If the configuration constructor changes during the course of a rewrite, the resulting term is considered alien, and does not participate any further in the object-message rewriting for the original term. The order in which objects are considered and messages are delivered is system-dependent, but note that newly created messages are not delivered until some future visit to the configuration (though all arguments including new messages and alien configurations could potentially participate in the single non-object-message rewrite attempt). Message delivery is “just” rather than “fair”: in order for message delivery to be guaranteed, an object must always be willing to accept the message.² If multiple object-message rules contain the same message constructor, they are tried in a round-robin fashion. Non-object-message rules are also tried in a round-robin fashion for the single non-object-message rewrite attempt.

The counting of object-message rewrites is nonstandard: for the purposes of the rewrite argument given to `frewrite`, a visit to a configuration that results in one or more rewrites counts as a single rewrite, though for other accounting purposes all rewrites are counted. Finally, for tracing, profiling, and breakpoints only, there is a fake rewrite at the top of the configuration in the case that object-message rewriting takes place but the single non-object-message rewrite attempt fails. It is not included in the reported rewrite total, but it is inserted to keep tracing consistent.

11.3 Example: data agents

In this section we give an example of a simple distributed dataset in which each agent in a collection of data agents manages a local version of a global data dictionary that maps keys to values. An agent may only have part of the data locally, and must contact other agents to get the value of a key that is not in its local version. To simplify the presentation, we assume that data agents work in pairs.

This example illustrates one way of representing request-reply style of object-based programming in Maude, and also a way of representing informa-

¹ Assuming, as it should be the case, that both object constructors have been declared with the `object` attribute.

² There are program transformations that internalize conditions on message delivery to ensure a stronger fairness condition [238].

tion about the state of the task an object is working on when it needs to make one or more requests to other objects in order to answer a request itself. As in the ticker example, we define a uniform syntax for messages. Here, messages have both a receiver and a sender in addition to a message body, and are constructed with the `msg` constructor. The technique for maintaining task information is to define a sort `Request` and a `requests` attribute that holds the set of pending requests. The constant `empty` indicates that an object has no pending request. The request `w4(O:Oid, C:Oid, MB:MsgBody)` indicates that the object is processing a message from `C:Oid` with body `MB:MsgBody` and is waiting for a message from `O:Oid`.

The module `DATA-AGENTS-CONF` extends `CONFIGURATION` with the uniform message syntax and the specification of the sort `Request`.

```
mod DATA-AGENTS-CONF is
  ex CONFIGURATION .
  *** my msg syntax
  sort MsgBody .
  op msg : Oid Oid MsgBody -> Msg [ctor message] .
  *** agents may be pending on requests
  sort Request .
  op w4 : Oid Oid MsgBody -> Request [ctor] .
endm
```

A data agent stores a dictionary, mapping keys to data elements. To specify such dictionaries, we use the predefined parameterized module `MAP` (see Section 9.13), renaming the main sort as well as the lookup and update operators as follows:

```
MAP{K, V} * (sort Map{K, V} to Dict{K, V},
            op _[_] to lookup,
            op insert to update) .
```

Remember that the constant `undefined` is the result returned by the lookup operators when the map is not defined on the given key.

We split the specification of data agents into two modules: the parameterized functional module `DATA-AGENTS-INTERFACE`, which defines the interface, and the parameterized system module `DATA-AGENTS`, which gives the rules for agent behavior. This illustrates a technique for modularizing object-based system specifications in order to allow the same interface to be shared by more than one “implementation” (rule set). We already applied this technique in the specification of a vending machine as a system module in Section 6.1. Notice also that `DATA-AGENTS-CONF` is imported in `extending` mode, because we add data to the old sorts, but without making further identifications (the interface module has no equation).

```
mod DATA-AGENTS-INTERFACE{K :: TRIV, V :: TRIV} is
  ex DATA-AGENTS-CONF .
```

```

*** messages
op getReq : K$Elt -> MsgBody [ctor] .
op getReply : K$Elt [V$Elt] -> MsgBody [ctor] .
op setReq : K$Elt V$Elt -> MsgBody [ctor] .
op setReply : K$Elt [V$Elt] -> MsgBody [ctor] .
op tellReq : K$Elt V$Elt -> MsgBody [ctor] .
op tellReply : K$Elt V$Elt -> MsgBody [ctor] .
op lookupReq : K$Elt -> MsgBody [ctor] .
op lookupReply : K$Elt [V$Elt] -> MsgBody [ctor] .
endm

```

In a request-reply style of interaction, message body constructors come in pairs. For example, (`lookupReq`, `lookupReply`) and (`tellReq`, `tellReply`) are the message body pairs used when a customer interacts with a data agent in order to access and set data values. Similarly, (`getReq`, `getReply`) and (`setReq`, `setReply`) constitute the message body pairs for an agent to access and set data values from a pal.

A data agent has class identifier `DataAgent`. In addition to the `requests` attribute, each data agent has a `data` attribute holding the agent's local version of the data dictionary, and a `pal` attribute holding the identifier of the other agent. If `sam` and `joe` are collaborating data agents, then their initial state might look like

```

< sam : DataAgent | data : empty, pal : joe, requests : empty >
< joe : DataAgent | data : empty, pal : sam, requests : empty >

```

The module `DATA-AGENTS` specifies a data agent's behavior by giving a rule for handling each type of message it expects to receive (other messages will simply be ignored).

Since we are adding rules acting on the sort `Configuration`, coming from the `CONFIGURATION` module via `DATA-AGENTS-CONF`, we need to make explicit that such modules are imported in `including` mode. We also import in `protecting` mode the predefined parameterized module `SET`, instantiated with the following `Request` view, to define the sets of requests stored in the `requests` attribute.

```

view Request from TRIV to DATA-AGENTS-CONF is
  sort Elt to Request .
endv

mod DATA-AGENTS{K :: TRIV, V :: TRIV} is
  inc DATA-AGENTS-INTERFACE{K, V} .
  inc DATA-AGENTS-CONF .
  inc CONFIGURATION .
  pr MAP{K, V} * (sort Map{K, V} to Dict{K, V},
                 op _[_] to lookup,
                 op insert to update) .
  pr SET{Request} .

```

```

vars A O C : Oid .
var D : Dict{K, V} .
var Key : K$Elt .
vars Val Val' : [V$Elt] .
var Atts : AttributeSet .
var RS : Set{Request} .

*** class structure
op DataAgent : -> Cid [ctor] .
op data :_ : Dict{K, V} -> Attribute [ctor] .
op pal :_ : Oid -> Attribute [ctor] .
op requests :_ : Set{Request} -> Attribute [ctor] .

*** lookup request
rl [lookup] :
  < A : DataAgent | data : D, pal : O, requests : RS >
  msg(A, C, lookupReq(Key))
  => if lookup(D, Key) == undefined
    then < A : DataAgent | data : D, pal : O,
          requests : (RS, w4(O, C, lookupReq(Key))) >
          msg(O, A, getReq(Key))
    else < A : DataAgent | data : D, pal : O, requests : RS >
          msg(C, A, lookupReply(Key, lookup(D, Key)))
    fi .

*** lookup request missing data from pal
rl [getReq] :
  < A : DataAgent | data : D, pal : O, Atts >
  msg(A, O, getReq(Key))
  => < A : DataAgent | data : D, pal : O, Atts >
      msg(O, A, getReply(Key, lookup(D, Key))) .

*** receive lookup requested missing data from pal
rl [getReply] :
  < A : DataAgent | data : D, pal : O,
    requests : (RS, w4(O, C, lookupReq(Key))) >
  msg(A, O, getReply(Key, Val))
  => < A : DataAgent | pal : O, requests : RS,
    data : if Val == undefined
            then D
            else update(Key, Val, D)
          fi >
      msg(C, A, lookupReply(Key, Val)) .

*** tell request
rl [tell] :
  < A : DataAgent | data : D, requests : RS, pal : O >
  msg(A, C, tellReq(Key, Val))

```

```

=> if lookup(D, Key) == undefined
  then < A : DataAgent |
        data : D,
        requests : (RS, w4(0, C, tellReq(Key, Val))),
        pal : 0 >
    msg(0, A, setReq(Key, Val))
  else < A : DataAgent | data : update(Key, Val, D),
        requests : RS, pal : 0 >
    msg(C, A, tellReply(Key, Val))
  fi .

*** request update for missing data from pal
rl [setReq] :
  < A : DataAgent | data : D, pal : 0, Atts >
  msg(A, 0, setReq(Key, Val))
  => if lookup(D, Key) == undefined
    then < A : DataAgent | data : D, pal : 0, Atts >
        msg(0, A, setReply(Key, undefined))
    else < A : DataAgent | data : update(Key, Val, D),
        pal : 0, Atts >
        msg(0, A, setReply(Key, Val))
    fi .

*** receive requested update for missing data from pal
rl [setReply] :
  < A : DataAgent | data : D, pal : 0,
    requests : (RS, w4(0, C, tellReq(Key, Val))) >
  msg(A, 0, setReply(Key, Val'))
  => < A : DataAgent | pal : 0, requests : RS,
    data : if Val' == undefined
            then update(Key, Val, D)
            else D
            fi >
    msg(C, A, tellReply(Key, Val)) .
endm

```

The rule labeled `lookup` specifies how an agent handles a `lookupReq` message. The agent first looks to see if its local dictionary contains the requested entry. If `lookup(D, Key) == undefined`, then a `getReq` is sent to the `pal` and the agent waits for a reply, remembering the pending lookup request (`w4(0, C, lookupReq(Key))`). If the agent has the requested entry, then it is returned in a `lookupReply` message.

The rules labeled `getReq` and `getReply` specify how agents exchange dictionary entries. An agent can always answer a `getReq` message, since the `Atts` variable will match any status attribute. The agent simply replies with the result, possibly `undefined`, of looking up the requested key in its local dictionary. An agent only expects a `getReply` message if it has made a request, and this only happens if the agent is trying to handle a

`lookupReq` message. Thus the rule only matches if the agent has the appropriate request `w4(0, C, lookupReq(Key))` in its `requests` attribute. The agent records the received reply with `update(Key, Val, D)` when this reply is not `undefined`, and in any case sends it on to the customer with the message `msg(C, A, lookupReply(Key, Val))`.

The rules labeled `tell`, `setReq`, and `setReply` specify how an agent handles a `tellReq` message, following a protocol similar to the one described for the lookup request.

Note that in the case of agents with just these three attributes, using the `Atts` variable of sort `AttributeSet` or the `requests : RS` expression, with `RS` a variable of sort `Set{Request}`, are equivalent ways of saying that the rule matches any set of requests. The first way is more extensible, in that the rule would still work for agents belonging to a subclass of `DataAgent` that uses additional attributes.

To test the data agent specification, we define a module `AGENT-TEST`. This module defines object identifiers `sam` and `joe` for data agents, and `me` to name an external customer. It also defines an initial configuration containing two agents named `sam` and `joe` with empty data dictionaries, and two initial `tellReq` messages for each agent. We take both keys and data elements to be quoted identifiers, by instantiating the parameterized `DATA-AGENTS` module with the predefined `Qid` view.

```
mod AGENT-TEST is
  ex DATA-AGENTS{Qid, Qid} .
  ops sam joe me : -> Qid [ctor] .
  op iconf : -> Configuration .
  eq iconf
    = < sam : DataAgent | data : empty,
      pal : joe, requests : empty >
      msg(sam, me, tellReq('a, 'bc))
      msg(sam, me, tellReq('d, 'ef))
    < joe : DataAgent | data : empty,
      pal : sam, requests : empty >
      msg(joe, me, tellReq('g, 'hi))
      msg(joe, me, tellReq('j, 'kl)) .
  endm
```

The importation graph of all the modules involved in this example is shown in Figure 11.3, where the three different types of arrows correspond to the three different modes of importation.

The following are results from test runs. First we rewrite the initial configuration `iconf`, resulting in a configuration in which the agents have updated appropriately their data, and there is one reply for each `tellReq` message.

```
Maude> rew iconf .
result Configuration:
  < sam : DataAgent | data : ('a |-> 'bc, 'd |-> 'ef),
```

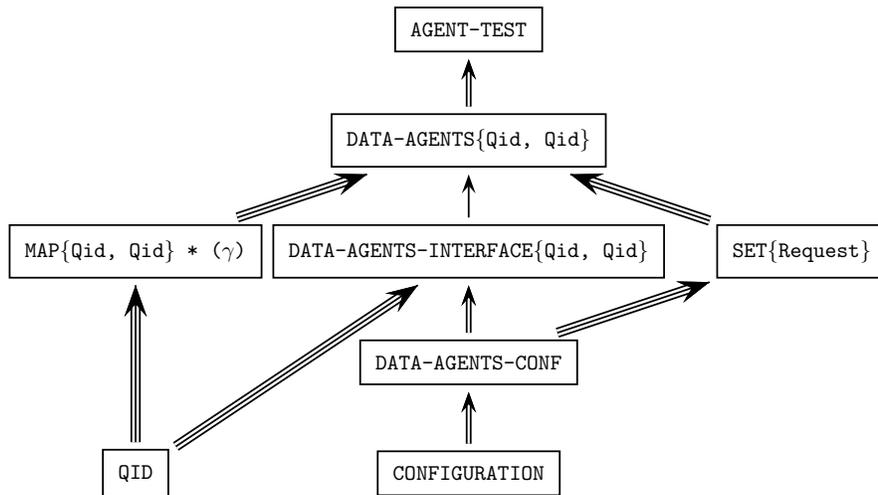


Fig. 11.3. Importation graph of data-agents modules

```

    pal : joe, requests : empty >
  < joe : DataAgent | data : ('g |-> 'hi, 'j |-> 'kl),
    pal : sam, requests : empty >
  msg(me, sam, tellReply('a, 'bc))
  msg(me, sam, tellReply('d, 'ef))
  msg(me, joe, tellReply('g, 'hi))
  msg(me, joe, tellReply('j, 'kl))

```

Next we try adding a lookup request and discover that, using Maude's default rewriting strategy, the lookup request is delivered before the tell requests, so the reply is undefined.

```

Maude> rew iconf msg(sam, me, lookupReq('a)) .
result Configuration:
  < sam : DataAgent | data : ('a |-> 'bc, 'd |-> 'ef),
    pal : joe, requests : empty >
  < joe : DataAgent | data : ('g |-> 'hi, 'j |-> 'kl),
    pal : sam, requests : empty >
  msg(me, sam, tellReply('a, 'bc))
  msg(me, sam, tellReply('d, 'ef))
  msg(me, sam, lookupReply('a, undefined))
  msg(me, joe, tellReply('g, 'hi))
  msg(me, joe, tellReply('j, 'kl))

```

To see if a good answer can be obtained, we use the `search` command to look for a state in which there is a `lookupReply` with data entry different from `undefined`.

```
Maude> search iconf msg(sam, me, lookupReq('a))
=>! msg(me, sam, lookupReply('a, Q:Qid)) C:Configuration .
```

```
Solution 1 (state 1081)
C:Configuration -->
  < sam : DataAgent | data : ('a |-> 'bc, 'd |-> 'ef),
    pal : joe, requests : empty >
  < joe : DataAgent | data : ('g |-> 'hi, 'j |-> 'kl),
    pal : sam, requests : empty >
  msg(me, sam, tellReply('a, 'bc))
  msg(me, sam, tellReply('d, 'ef))
  msg(me, joe, tellReply('g, 'hi))
  msg(me, joe, tellReply('j, 'kl))
Q:Qid --> 'bc
```

No more solutions.

Indeed, there is just one such reply.

Notice that two collaborating agents may get inconsistent data, that is, different values for the same key, if they receive simultaneously tell requests for the same key. We may use the `search` command to illustrate how this may happen.

```
Maude> search iconf
  msg(sam, me, tellReq('m, 'no))
  msg(joe, me, tellReq('m, 'pq))
=>! C:Configuration
  < sam : DataAgent |
    data : ('m |-> Q:Qid, R:Dict{Qid, Qid}),
    Atts:AttributeSet >
  < joe : DataAgent |
    data : ('m |-> Q':Qid, R':Dict{Qid, Qid}),
    Atts':AttributeSet >
  such that Q:Qid /= Q':Qid .
```

```
Solution 1 (state 5117)
C:Configuration -->
  msg(me, sam, tellReply('a, 'bc))
  msg(me, sam, tellReply('d, 'ef))
  msg(me, sam, tellReply('m, 'no))
  msg(me, joe, tellReply('g, 'hi))
  msg(me, joe, tellReply('j, 'kl))
  msg(me, joe, tellReply('m, 'pq))
Atts:AttributeSet --> pal : joe, requests : empty
R:Dict{Qid,Qid} --> 'a |-> 'bc, 'd |-> 'ef
Q:Qid --> 'no
Atts':AttributeSet --> pal : sam, requests : empty
R':Dict{Qid,Qid} --> 'g |-> 'hi, 'j |-> 'kl
Q':Qid --> 'pq
```

No more solutions.

Note the use of the `such that` condition to filter `search` solutions (see Section 6.4.3 and 25.4).

11.4 External objects

This section explains Maude’s support for rewriting with external objects and an implementation of sockets as the first such external objects.

Configurations that want to communicate with external objects must contain at least one *portal*, where

```
sort Portal .
subsort Portal < Configuration .
op <> : -> Portal [ctor] .
```

is part of the predefined module `CONFIGURATION` in the file `prelude.maude`. Rewriting with external objects is started by the external rewrite command `erewrite` (abbreviated `erew`) which is like `frewrite` (see Sections 6.4 and 11.2) except that it allows messages to be exchanged with external objects that do not reside in the configuration. Currently the command `erewrite` has some severe limitations, which might be fixed in future releases:

1. Maude only checks for external rewrites when no “internal” rewrites are possible, so if, for example, there is a clock tick rewrite rule that is always enabled, external rewrites won’t take place.
2. Rewrites that involve messages entering or leaving the configuration do not show up in tracing, profiling or rewrite counts.
3. It is possible to have bad interactions with break points and the debugger.
4. There is a potential race condition with `^C`.

Note that, even if there are no more rewrites possible, `erewrite` may not terminate; if there are requests made to external objects that have not yet been fulfilled because of waiting for external events from the operating system, the Maude interpreter will suspend until at least one of those events occurs, at which time rewriting will resume. While the interpreter is suspended, the command `erewrite` may be aborted with `^C`. External objects created by an `erewrite` command do not survive to the next `erewrite`. If a message to an external object is ill-formed or inappropriate, or the external object is not ready for it, it just stays in the configuration for future acceptance or for debugging purposes.

11.4.1 Sockets

The first example of external objects is *sockets*, which are accessed using the messages declared in the following `SOCKET` module, included in the file `socket.maude` which is part of the Maude distribution.

```

mod SOCKET is
  protecting STRING .
  including CONFIGURATION .

  op socket : Nat -> Oid [ctor] .

  op createClientTcpSocket : Oid Oid String Nat -> Msg
    [ctor msg format (b o)] .
  op createServerTcpSocket : Oid Oid Nat Nat -> Msg
    [ctor msg format (b o)] .
  op createdSocket : Oid Oid Oid -> Msg [ctor msg format (m o)] .

  op acceptClient : Oid Oid -> Msg [ctor msg format (b o)] .
  op acceptedClient : Oid Oid String Oid -> Msg
    [ctor msg format (m o)] .

  op send : Oid Oid String -> Msg [ctor msg format (b o)] .
  op sent : Oid Oid -> Msg [ctor msg format (m o)] .

  op receive : Oid Oid -> Msg [ctor msg format (b o)] .
  op received : Oid Oid String -> Msg [ctor msg format (m o)] .

  op closeSocket : Oid Oid -> Msg [ctor msg format (b o)] .
  op closedSocket : Oid Oid String -> Msg [ctor msg format (m o)] .

  op socketError : Oid Oid String -> Msg [ctor msg format (r o)] .

  op socketManager : -> Oid [special (...)] .
endm

```

Currently only IPv4 TCP sockets are supported; other protocol families and socket types may be added in the future. The external object named by the constant `socketManager` is a factory for socket objects.

To create a client socket, you send `socketManager` a message

```
createClientTcpSocket(socketManager, ME, ADDRESS, PORT)
```

where `ME` is the name of the object the reply should be sent to, `ADDRESS` is the name of the server you want to connect to (say “www.google.com”), and `PORT` is the port you want to connect to (say 80 for HTTP connections). You may also specify the name of the server as an IPv4 dotted address or as “localhost” for the same machine where the Maude system is running on.

The reply will be either

```
createdSocket(ME, socketManager, NEW-SOCKET-NAME)
```

or

```
socketError(ME, socketManager, REASON)
```

where `NEW-SOCKET-NAME` is the name of the newly created socket and `REASON` is the operating system's terse explanation of what went wrong.

You can then send data to the server with a message

```
send(SOCKET-NAME, ME, DATA)
```

which elicits either

```
sent(ME, SOCKET-NAME)
```

or

```
closedSocket(ME, SOCKET-NAME, REASON)
```

Notice that all errors on a client socket are handled by closing the socket.

Similarly, you can receive data from the server with a message

```
receive(SOCKET-NAME, ME)
```

which elicits either

```
received(ME, SOCKET-NAME, DATA)
```

or

```
closedSocket(ME, SOCKET-NAME, REASON)
```

When you are done with the socket, you can close it with a message

```
closeSocket(SOCKET-NAME, ME)
```

with reply

```
closedSocket(ME, SOCKET-NAME, "")
```

Once a socket has been closed, its name may be reused, so sending messages to a closed socket can cause confusion and should be avoided.

Notice that TCP does not preserve message boundaries, so sending `"one"` and `"two"` might be received as `"on"` and `"etwo"`. Delimiting message boundaries is the responsibility of the next higher-level protocol, such as HTTP. We will present an implementation of buffered sockets in Section 11.4.2 which solves this problem.

The following modules implement an updated version of the five rule HTTP/1.0 client from the paper "Towards Maude 2.0" [69] that is now executable. The first module defines some auxiliary operations on strings.

```
fmod STRING-OPS is
pr STRING .
var S : String .
op extractHostName : String -> String .
op extractPath : String -> String .
op extractHeader : String -> String .
```

```

op extractBody : String -> String .

eq extractHostName(S)
  = if find(S, "/", 0) == notFound
    then S
    else substr(S, 0, find(S, "/", 0))
    fi .

eq extractPath(S)
  = if find(S, "/", 0) == notFound
    then "/"
    else substr(S, find(S, "/", 0), length(S))
    fi .

eq extractHeader(S)
  = substr(S, 0, find(S, "\r\n\r\n", 0) + 4) .
eq extractBody(S)
  = substr(S, find(S, "\r\n\r\n", 0) + 4, length(S)) .
endfm

```

The second module requests one web page to a HTTP server.

```

mod HTTP/1.0-CLIENT is
  pr STRING-OPS .
  inc SOCKET .
  sort State .
  ops idle connecting sending receiving closing : -> State [ctor] .
  op state:_ : State -> Attribute [ctor] .
  op requester:_ : Oid -> Attribute [ctor] .
  op url:_ : String -> Attribute [ctor] .
  op stored:_ : String -> Attribute [ctor] .

  op HttpClient : -> Cid .
  op httpClient : -> Oid .
  op dummy : -> Oid .

  op getPage : Oid Oid String -> Msg [msg ctor] .
  op gotPage : Oid Oid String String -> Msg [msg ctor] .

  vars H R R' TS : Oid .
  vars U S ST : String .

```

First, we try to connect to the server using port 80, updating the state and the `requester` attribute with the new server.

```

r1 [getPage] :
  getPage(H, R, U)
  < H : HttpClient |
    state: idle, requester: R', url: S, stored: "" >
  => < H : HttpClient |

```

```

state: connecting, requester: R, url: U, stored: "" >
createClientTcpSocket(socketManager, H,
extractHostName(U), 80) .

```

Once we are connected to the server (we have received a `createdSocket` message), we send a GET message (from the HTTP protocol) requesting the page. When the message is sent, we wait for a response.

```

r1 [createdSocket] :
createdSocket(H, socketManager, TS)
< H : HttpClient |
state: connecting, requester: R, url: U, stored: "" >
=> < H : HttpClient |
state: sending, requester: R, url: U, stored: "" >
send(TS, H, "GET " + extractPath(U) + " HTTP/1.0\r\nHost: " +
extractHostName(U) + "\r\n\r\n") .

r1 [sent] :
sent(H, TS)
< H : HttpClient |
state: sending, requester: R, url: U, stored: "" >
=> < H : HttpClient |
state: receiving, requester: R, url: U, stored: "" >
receive(TS, H) .

```

While the page is not complete, we receive data and append it to the string on the `stored` attribute. When the page is completed, the server closes the socket, and then we show the page information by means of the `gotPage` message.

```

r1 [received] :
received(H, TS, S)
< H : HttpClient |
state: receiving, requester: R, url: U, stored: ST >
=> receive(TS, H)
< H : HttpClient | state: receiving,
requester: R, url: U, stored: (ST + S) > .

r1 [closedSocket] :
closedSocket(H, TS, S)
< H : HttpClient |
state: receiving, requester: R, url: U, stored: ST >
=> gotPage(R, H, extractHeader(ST), extractBody(ST)) .

```

We use a special operator `start` to represent the initial configuration. It receives the server URL we want to connect to. Notice the occurrence of the portal `<>` in such initial configuration.

```

op start : String -> Configuration .
eq start(S)

```

```

= <>
  getPage(httpClient, dummy, S)
  < httpClient : HttpClient | state: idle, requester: dummy,
    url: "", stored: "" > .
endm

```

Now we can get pages from servers, say “www.google.com”, by using the following Maude command:

```
Maude> erew start("www.google.com") .
```

It is also possible to have optional bounds on the `erewrite` command, and then use the continuation commands to get more results, like, for example,

```
Maude> erew [1, 2] start("www.google.com") .
Maude> cont 1 .
```

To have communication between two Maude interpreter instances, one of them must take the server role and offer a service on a given port; generally ports below 1024 are protected. You cannot in general assume that a given port is available for use. To create a server socket, you send `socketManager` a message

```
createServerTcpSocket(socketManager, ME, PORT, BACKLOG)
```

where `PORT` is the port number and `BACKLOG` is the number of queue requests for connection that you will allow (5 seems to be a good choice). The response is either

```
createdSocket(ME, socketManager, SERVER-SOCKET-NAME)
```

or

```
socketError(ME, socketManager, REASON)
```

Here `SERVER-SOCKET-NAME` refers to a server socket. The only thing you can do with a server socket (other than close it) is to accept clients, by means of the following message:

```
acceptClient(SERVER-SOCKET-NAME, ME)
```

which elicits either

```
acceptedClient(ME, SERVER-SOCKET-NAME, ADDRESS, NEW-SOCKET-NAME)
```

or

```
socketError(ME, socketManager, REASON)
```

Here ADDRESS is the originating address of the client and NEW-SOCKET-NAME is the name of the socket you use to communicate with that client. This new socket behaves just like a client socket for sending and receiving. Note that an error in accepting a client does not close the server socket. You can always reuse the server socket to accept new clients until you explicitly close it.

The following modules illustrate a very naive two-way communication between two Maude interpreter instances. The issues of port availability and message boundaries are deliberately ignored for the sake of illustration (and thus if you are unlucky this example could fail).

The first module describes the behavior of the server.

```

mod FACTORIAL-SERVER is
  inc SOCKET .
  pr CONVERSION .
  op _! : Nat -> NzNat .
  eq 0 ! = 1 .
  eq (s N) ! = (s N) * (N !) .

  op Server : -> Cid .
  op aServer : -> Oid .

  vars 0 LISTENER CLIENT : Oid .
  var A : AttributeSet .
  var N : Nat .
  vars IP DATA S : String .

```

Using the following rules, the server waits for clients. If one client is accepted, the server waits for messages from it. When the message arrives, the server converts the received data to a natural number, computes its factorial, converts it into a string, and finally sends this string to the client. Once the message is sent, the server closes the socket with the client.

```

rl [createdSocket] :
  < 0 : Server | A > createdSocket(0, socketManager, LISTENER)
  => < 0 : Server | A > acceptClient(LISTENER, 0) .

rl [acceptedClient] :
  < 0 : Server | A > acceptedClient(0, LISTENER, IP, CLIENT)
  => < 0 : Server | A > receive(CLIENT, 0)
  acceptClient(LISTENER, 0) .

rl [received] :
  < 0 : Server | A > received(0, CLIENT, DATA)
  => < 0 : Server | A >
    send(CLIENT, 0, string(rat(DATA, 10)!, 10)) .

rl [sent] :
  < 0 : Server | A > sent(0, CLIENT)
  => < 0 : Server | A > closeSocket(CLIENT, 0) .

```

```

r1 [closedSocket] :
  < 0 : Server | A > closedSocket(0, CLIENT, S)
  => < 0 : Server | A > .
endm

```

The Maude command that initializes the server is as follows, where the configuration includes the portal <>.

```

Maude> erew <>
  < aServer : Server | none >
  createServerTcpSocket(socketManager, aServer, 8811, 5) .

```

The second module describes the behavior of the clients.

```

mod FACTORIAL-CLIENT is
  inc SOCKET .
  op Client : -> Cid .
  op aClient : -> Oid .

  vars 0 CLIENT : Oid .
  var A : AttributeSet .

```

Using the following rules, the client connects to the server (clients must be created after the server), sends a message representing a number,³ and then waits for the response. When the response arrives, there are no blocking messages and rewriting ends.

```

r1 [createdSocket] :
  < 0 : Client | A > createdSocket(0, socketManager, CLIENT)
  => < 0 : Client | A > send(CLIENT, 0, "6") .

r1 [sent] :
  < 0 : Client | A > sent(0, CLIENT)
  => < 0 : Client | A > receive(CLIENT, 0) .
endm

```

The initial configuration for the client will be as follows, again with portal <>.

```

Maude> erew <>
  < aClient : Client | none >
  createClientTcpSocket(socketManager,
    aClient, "localhost", 8811) .

```

Almost everything in the socket implementation is done in a nonblocking way; so, for example, if you try to open a connection to some webserver and that webserver takes 5 minutes to respond, other rewriting and transactions happen in the meanwhile as part of the same command `erewrite`. The one exception is DNS resolution, which is done as part of the `createClientTcpSocket` message handling and which cannot be nonblocking without special tricks.

³ In this quite simple example, it is always "6".

11.4.2 Buffered sockets

As we said before, TCP does not preserve message boundaries; to guarantee it we may use a filter class `BufferedSocket`, defined in the module `BUFFERED-SOCKET`, which is described here. We interact with buffered sockets in the same way we interact with sockets, with the only difference that all messages in the module `SOCKET` have been capitalized to avoid confusion. Thus, to create a client with a buffered socket, you send `socketManager` a message

```
CreateClientTcpSocket(socketManager, ME, ADDRESS, PORT)
```

instead of a message

```
createClientTcpSocket(socketManager, ME, ADDRESS, PORT).
```

All the messages have exactly the same declarations, the only difference being their initial capitalization:

```
op CreateClientTcpSocket : Oid Oid String Nat -> Msg
  [ctor msg format (b o)] .
op CreateServerTcpSocket : Oid Oid Nat Nat -> Msg
  [ctor msg format (b o)] .
op CreatedSocket : Oid Oid Oid -> Msg [ctor msg format (m o)] .

op AcceptClient : Oid Oid -> Msg [ctor msg format (b o)] .
op AcceptedClient : Oid Oid String Oid -> Msg
  [ctor msg format (m o)] .

op Send : Oid Oid String -> Msg [ctor msg format (b o)] .
op Sent : Oid Oid -> Msg [ctor msg format (m o)] .

op Receive : Oid Oid -> Msg [ctor msg format (b o)] .
op Received : Oid Oid String -> Msg [ctor msg format (m o)] .

op CloseSocket : Oid Oid -> Msg [ctor msg format (b o)] .
op ClosedSocket : Oid Oid String -> Msg [ctor msg format (m o)] .

op SocketError : Oid Oid String -> Msg [ctor msg format (r o)] .
```

Thus, apart from this small difference, we interact with buffered sockets in exactly the same way we do with sockets, the boundary control being completely transparent to the user.

When a buffered socket is created, in addition to the socket object through which the information will be sent, a `BufferedSocket` object is also created on each side of the socket (one in each one of the configurations between which the communication is established). All messages sent through a buffered socket are manipulated before they are sent through the socket underneath. When a message is sent through a buffered socket, a mark is placed at the end of it;

the `BufferedSocket` object at the other side of the socket stores all messages received on a buffer, in such a way that when a message is requested the marks placed indicate which part of the information received must be given as the next message.

An object of class `BufferedSocket` has two attributes: `read`, of sort `String`, which stores the messages read, and `bState`, which indicates whether the filter is idle or active.

```
op BufferedSocket : -> Cid [ctor] .
op read :_ : String -> Attribute [ctor gather(&)] .
op bState :_ : BState -> Attribute [ctor gather(&)] .
sort BState .
ops idle active : -> BState [ctor] .
```

The identifiers of the `BufferedSocket` objects are marked with a `b` operator, i.e., the buffers associated with a socket `SOCKET` have identifier `b(SOCKET)`. Note that there is a `BufferedSocket` object on each side of the socket, that is, there are two objects with the same identifier, but in different configurations.

```
op b : Oid -> Oid [ctor] .
```

A buffered socket object understands capitalized versions of the messages a socket object understands. For most of them, it just converts them into the corresponding uncapitalized message. There are messages `AcceptClient`, `CloseSocket`, `CreateServerTcpSocket`, and `CreateClientTcpSocket` with the same arities as the corresponding socket messages, with the following rules.

```
vars SOCKET NEW-SOCKET SOCKET-MANAGER 0 : Oid .
vars ADDRESS IP IP' DATA S S' REASON : String .
var Atts : AttributeSet .
vars PORT BACKLOG N : Nat .

r1 [createServerTcpSocket] :
  CreateServerTcpSocket(SOCKET-MANAGER, 0, PORT, BACKLOG)
  => createServerTcpSocket(SOCKET-MANAGER, 0, PORT, BACKLOG) .

r1 [acceptClient] :
  AcceptClient(SOCKET, 0)
  => acceptClient(SOCKET, 0) .

r1 [closeSocket] :
  CloseSocket(b(SOCKET), SOCKET-MANAGER)
  => closeSocket(SOCKET, SOCKET-MANAGER) .

r1 [createClientTcpSocket] :
  CreateClientTcpSocket(SOCKET-MANAGER, 0, ADDRESS, PORT)
  => createClientTcpSocket(SOCKET-MANAGER, 0, ADDRESS, PORT) .
```

Note that in these cases the buffered-socket versions of the messages are just translated into the corresponding socket messages.

A `BufferedSocket` object can also convert an uncapitalized message into the capitalized one. The rule `socketError` shows this:

```
r1 [socketError] :
  socketError(0, SOCKET-MANAGER, REASON)
=> SocketError(0, SOCKET-MANAGER, REASON) .
```

`BufferedSocket` objects are created and destroyed when the corresponding sockets are. Thus, we have rules

```
r1 [acceptedClient] :
  acceptedClient(0, SOCKET, IP', NEW-SOCKET)
=> AcceptedClient(0, b(SOCKET), IP', b(NEW-SOCKET))
   < b(NEW-SOCKET) : BufferedSocket |
      bState : idle, read : "" > .

r1 [createdSocket] :
  createdSocket(0, SOCKET-MANAGER, SOCKET)
=> < b(SOCKET) : BufferedSocket | bState : idle, read : "" >
   CreatedSocket(0, SOCKET-MANAGER, b(SOCKET)) .

r1 [closedSocket] :
  < b(SOCKET) : BufferedSocket | Atts >
  closedSocket(SOCKET, SOCKET-MANAGER, DATA)
=> ClosedSocket(b(SOCKET), SOCKET-MANAGER, DATA) .
```

Once a connection has been established, and a `BufferedSocket` object has been created on each side, messages can be sent and received. When a `Send` message is received, the buffered socket sends a `send` message with the same data plus a mark⁴ to indicate the end of the message.

```
r1 [send] :
  < b(SOCKET) : BufferedSocket | bState : active, Atts >
  Send(b(SOCKET), 0, DATA)
=> < b(SOCKET) : BufferedSocket | bState : active, Atts >
   send(SOCKET, 0, DATA + "#") .

r1 [sent] :
  < b(SOCKET) : BufferedSocket | bState : active, Atts >
  sent(0, SOCKET)
=> < b(SOCKET) : BufferedSocket | bState : active, Atts >
   Sent(0, b(SOCKET)) .
```

The key is then in the reception of messages. A `BufferedSocket` object is always listening to the socket. It sends a `receive` message at start up and puts all the received messages in its buffer. Notice that a buffered socket goes from `idle` to `active` in the `buffer-start-up` rule. A `Receive` message is

⁴ We use the character '#' as mark; therefore, the user data sent through the sockets should not contain such a character.

then handled if there is a complete message in the buffer, that is, if there is a mark on it, and results in the reception of the first message in the buffer, which is removed from it.

```

r1 [buffer-start-up] :
  < b(SOCKET) : BufferedSocket | bState : idle, Atts >
  => < b(SOCKET) : BufferedSocket | bState : active, Atts >
      receive(SOCKET, b(SOCKET)) .

r1 [received] :
  < b(SOCKET) : BufferedSocket |
      bState : active, read : S, Atts >
  received(b(SOCKET), 0, DATA)
  => < b(SOCKET) : BufferedSocket |
      bState : active, read : (S + DATA), Atts >
      receive(SOCKET, b(SOCKET)) .

cr1 [Received] :
  < b(SOCKET) : BufferedSocket |
      bState : active, read : S, Atts >
  Receive(b(SOCKET), 0)
  => < b(SOCKET) : BufferedSocket |
      bState : active, read : S', Atts >
      Received(0, b(SOCKET), DATA)
  if N := find(S, "#", 0)
  /\ DATA := substr(S, 0, N)
  /\ S' := substr(S, N + 1, length(S)) .

```

The BUFFERED-SOCKET module is used in the specification of Mobile Maude, a mobile agent language based on Maude, which is discussed in detail in Chapter 17.

Model Checking Invariants Through Search

A rewrite theory, specified in Maude as a system module, provides an executable mathematical model of a concurrent system. We can use the Maude specification to *simulate* the concurrent system so specified. But we can do more. Under appropriate conditions we can *check* that our mathematical model satisfies some important properties, or obtain a useful counterexample showing that the property in question is violated. This kind of *model-checking analysis* can be quite general. Chapter 13 will explain how, under appropriate finite reachability assumptions, we can model check any linear time temporal logic (LTL) property of a system specified in Maude as a system module. This chapter focuses on a simpler, yet very useful, model-checking capability, namely, the model checking of *invariants*, which can be accomplished just by using the `search` command.

12.1 Invariants

Invariants are the most common and useful *safety properties*, that is, properties stating that something bad should never happen. Given a transition system and an initial state s_0 , an *invariant* I is a predicate defining a subset of states meeting two properties:

- it contains s_0 , and
- it contains any state reachable from s_0 through a finite number of transitions.

Therefore, an invariant is a predicate defining a set of states that contains all the states reachable from s_0 . If an invariant holds, that is, if it is truly an invariant satisfying the two properties above, then we know that something “bad” can never happen, namely, the negation $\neg I$ of the invariant is impossible. In other words, we view $\neg I$ as a bad property that should never happen, and I as a good property we want to ensure.

Given a rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$ specified in Maude as a system module, we can define an invariant I , yielding a decidable set of states, by:

1. choosing a given kind k in Σ as the kind of states and an initial state `init` in it; and
2. specifying an *equationally-defined Boolean condition* I , so that the invariant holds of a state if and only if such a state satisfies the condition I .

The transition system implicit in this setting is one in which a one-step transition between two states, that is, between two elements $[t], [t'] \in T_{\Sigma/E, k}$, exists if and only if there is a representative $t_0 \in [t]$ and a one-step rewrite with the rules R , $t_0 \rightarrow^1 t'_0$, such that $t'_0 \in [t']$. We introduce the notation

$$\mathcal{R}, \text{init} \models \square I$$

to state that the transition system associated with \mathcal{R} with state kind k and initial state `init` satisfies the invariant I .

12.2 Model checking of invariants

The key question now is: how can we automatically *verify* that an invariant I holds? The answer is very simple. Assuming that $\mathcal{R} = (\Sigma, E, \phi, R)$ satisfies reasonable executability conditions, namely, that E and R are finite sets, (Σ, E) is ground Church-Rosser and terminating,¹ and the rules R are ground coherent with E , and assuming, further, that all the conditions for rules in R are purely equational, then I holds if and only if the search command

```
search init =>* x:k such that I(x:k) /= true .
```

has no solutions. Indeed, having no solutions exactly means that on `init`, and on all states reachable from it, the predicate I evaluates to `true`, that is, that I is an invariant. Assuming that I has been fully defined in all cases (which is always easy with the `owise` feature, described in Section 4.5.4), so that it always evaluates to either `true` or `false`, we could instead give the command

```
search init =>* x:k such that not I(x:k) .
```

Consider, for example, a simple clock that marks the hours of the day. Such a clock can be specified with the system module

```
mod SIMPLE-CLOCK is
  protecting INT .
  sort Clock .
  op clock : Int -> Clock [ctor] .
  var T : Int .
  rl clock(T) => clock((T + 1) rem 24) .
endm
```

¹ As usual, the ground Church-Rosser and termination assumptions should be understood *modulo* any axioms $A \subseteq E$ which in Maude are declared as equational attributes of operators.

A natural initial state is the state `clock(0)`. Note that, in principle, the clock could be in an infinite number of states, such as `clock(7633157)` or `clock(-33457129)`. The point, however, is that if our initial state is `clock(0)`, then only states `clock(T)` with times T such that $0 \leq T < 24$ can be reached. This suggests making the predicate $0 \leq T < 24$ an invariant of our clock system.

Using simple linear arithmetic reasoning, we can express the negation of such an invariant as the predicate $(T < 0)$ or $(T \geq 24)$; thus, we can automatically verify that our simple clock satisfies the invariant by giving the command:

```
Maude> search in SIMPLE-CLOCK : clock(0) =>* clock(T)
      such that T < 0 or T >= 24 .
```

No solution.

```
states: 24  rewrites: 216 in 0ms cpu (2ms real) (~ rews/sec)
```

If, as it is the case in this clock example, the number of states reachable from the initial state is *finite*, then search commands of this kind provide a *decision procedure* for the satisfaction of invariants. That is, in finite time Maude will either find no solutions to a search for a state violating the invariant, or will find a state violating the invariant together with a sequence of rewrites from the initial state to it, that is, a counterexample.

But what if the number of states reachable from the initial state is *infinite*? In such a case, *if* the invariant I is violated, the `search` command will terminate in finite time yielding a counterexample. Termination is guaranteed by the breadth-first nature of the search. The point is that such a counterexample is a *reachable* state; therefore, there is a finite sequence of rewrites from the initial state to such a violating state. Since there is only a finite number of rules R , and therefore a finite number of ways that each state can be rewritten, even though the number of reachable states is infinite, the number of states reachable from the initial state by a sequence of rewrites of length less than a given bound is always finite. This bounded subset is always explored in finite time by the `search` command. This means that, for systems where the set of reachable states is infinite, search becomes a *semi-decision procedure* for detecting the *violation* of an invariant. That is, if the invariant is violated, we are guaranteed to get a counterexample; but, if it is not violated, we will search forever, never finding it.

We can illustrate the semi-decision procedure nature of search for the verification of invariant failures with a simple infinite-state example of processes and resources. This example has some similarities with the classical dining philosophers problem, but it is on the one hand simpler (processes and resources have no identities or topology), and on the other hand more complex, since the number of processes and resources can grow dynamically in an unbounded manner.

```
mod PROCS-RESOURCES is
```

```

sorts State Resources Process .
subsorts Process Resources < State .
ops res null : -> Resources [ctor] .
op p : Resources -> Process [ctor] .
op _ : Resources Resources -> Resources
    [ctor assoc comm id: null] .
op _ : State State -> State [ctor ditto] .

r1 [acq1] : p(null) res => p(res) .
r1 [acq2] : p(res) res => p(res res) .
r1 [rel] : p(res res) => p(null) res res .
r1 [dup1] : p(null) res => p(null) res p(null) res .
endm

```

The state is a soup (multiset) of processes and resources. Each process needs to acquire two resources. Originally, each process `p` contains the `null` state; but if a resource `res` is available, it can acquire it (rule `acq1`). If a second resource becomes available, it can also acquire it (rule `acq2`). After acquiring both resources and using them, the process can release them (rule `rel`). Furthermore, the number of processes and resources can grow in an unbounded manner by the duplication of each process-resource pair (rule `dup1`).

One invariant we might like to verify about this system is *deadlock freedom*. There are two ways to model check this property: one completely straightforward, and another requiring some extra work. The straightforward manner is to give the search command

```
Maude> search in PROCS-RESOURCES : res p(null) =>! X:State .
```

```
Solution 1 (state 1)
```

```
states: 3 rewrites: 2 in 0ms cpu (0ms real) (~ rews/sec)
X:State --> p(res)
```

```
Solution 2 (state 5)
```

```
states: 9 rewrites: 9 in 0ms cpu (1ms real) (~ rews/sec)
X:State --> p(res) p(res)
```

```
Solution 3 (state 13)
```

```
states: 19 rewrites: 26 in 0ms cpu (3ms real) (~ rews/sec)
X:State --> p(res) p(res) p(res)
```

```
Solution 4 (state 25)
```

```
states: 34 rewrites: 56 in 0ms cpu (4ms real) (~ rews/sec)
X:State --> p(res) p(res) p(res) p(res)
```

```
Solution 5 (state 43)
```

```
states: 55 rewrites: 104 in 0ms cpu (23ms real) (~ rews/sec)
X:State --> p(res) p(res) p(res) p(res) p(res)
```

```
.....
```

```

Solution 20 (state 1649)
states: 1770 rewrites: 5640 in 20ms cpu (67ms real)
      (282000 rews/sec)
X:State --> p(res) p(res) p(res) p(res) p(res) p(res) p(res) p(res)
           p(res) p(res) p(res) p(res) p(res) p(res) p(res) p(res)
           p(res) p(res) p(res) p(res)
.....

```

Maude will indeed continue printing all the solutions it finds. But since there is an infinite number of deadlock states, it may be preferable to specify in advance a bound on the number of solutions, giving, for example, a command like the following, that looks for at most 5 solutions.

```
Maude> search [5] in PROCS-RESOURCES : res p(null) =>! X:State .
```

The nice thing about model checking deadlock freedom this way is that there is no need to explicitly specify the invariant as a Boolean predicate. This is because the negation of the invariant is by definition the set of deadlock states, which is what the `search` command with the `=>!` qualification precisely looks for.

If one wishes, one can, with a little more work, perform an equivalent model checking of the same property by using an explicit enabledness predicate. Of course, this cannot be done in the original module, because such a predicate has not been defined, but this is easy enough to do:

```

mod PROCS-RESOURCES-ENABLED is
  protecting PROCS-RESOURCES .
  protecting BOOL .
  op enabled : State -> Bool .
  eq enabled(p(null) res X:State) = true .
  eq enabled(p(res) res X:State) = true .
  eq enabled(p(res res) X:State) = true .
  eq enabled(X:State) = false [otherwise] .
endm

```

One can then give the command

```
Maude> search [5] in PROCS-RESOURCES-ENABLED : res p(null)
      =>* X:State such that enabled(X:State) /= true .
```

getting the following 5 solutions:

```

Solution 1 (state 1)
states: 2 rewrites: 4 in 0ms cpu (0ms real) (~ rews/sec)
X:State --> p(res)

```

```

Solution 2 (state 5)
states: 6 rewrites: 15 in 0ms cpu (0ms real) (~ rews/sec)

```

```

X:State --> p(res) p(res)

Solution 3 (state 13)
states: 14 rewrites: 41 in 0ms cpu (0ms real) (~ rews/sec)
X:State --> p(res) p(res) p(res)

Solution 4 (state 25)
states: 26 rewrites: 87 in 0ms cpu (1ms real) (~ rews/sec)
X:State --> p(res) p(res) p(res) p(res)

Solution 5 (state 43)
states: 44 rewrites: 160 in 0ms cpu (1ms real) (~ rews/sec)
X:State --> p(res) p(res) p(res) p(res) p(res)

```

12.3 Bounded model checking of invariants

In cases where either the set of reachable states is infinite, or it is finite but too large to be explored in reasonable time due to time and memory limitations, *bounded model checking* is an appealing formal analysis method. The idea of bounded model checking is that we model check a property, not for all reachable states, but only for those states reachable within a certain *depth bound*, that is, reachable by a sequence of transitions of bounded length. Of course, our analysis is not complete anymore, since we may fail to find a counterexample lying at greater depth. However, bounded model checking can be quite effective in finding counterexamples and it is a widely used procedure. Bounded model checking of invariants is supported in Maude by means of the *bounded search command*.

Consider the following specification of a readers-writers system.

```

mod READERS-WRITERS is
  sort Nat .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .

  sort Config .
  op <_,_> : Nat Nat -> Config [ctor] . --- readers/writers

  vars R W : Nat .

  rl < 0, 0 > => < 0, s(0) > .
  rl < R, s(W) > => < R, W > .
  rl < R, 0 > => < s(R), 0 > .
  rl < s(R), W > => < R, W > .
endm

```

A state is represented by a tuple $\langle R, W \rangle$ indicating the number R of readers and the number W of writers accessing a critical resource. Readers and writers

can leave the resource at any time, but writers can only gain access to it if nobody else is using it, and readers only if there are no writers.

Taking $\langle 0, 0 \rangle$ as the initial state, this simple system satisfies two important invariants, namely:

- *mutual exclusion*: readers and writers never access the resource simultaneously: only readers or only writers can do so at any given time.
- *one writer*: at most one writer will be able to access the resource at any given time.

We could try model checking these two invariants in two different ways:

- we can represent the invariants *implicitly* by representing their negations through *patterns*; or
- we can represent them *explicitly* as Boolean predicates.

To model check our two invariants using an implicit representation we could use the commands

```
Maude> search < 0, 0 > =>* < s(N:Nat), s(M:Nat) > .
```

```
Maude> search < 0, 0 > =>* < N:Nat, s(s(M:Nat)) > .
```

since the negation of the first invariant corresponds to the simultaneous presence of both readers and writers, which is exactly captured by the first pattern $\langle s(N:Nat), s(M:Nat) \rangle$; whereas the negation of the fact that zero or at most one writer should be present at any given time is exactly captured by the second pattern $\langle N:Nat, s(s(M:Nat)) \rangle$.

The problem with the above two search commands is that, since the number of readers allowed is unbounded, the set of reachable states is infinite and the commands never terminate. We can instead perform bounded model checking of these two invariants by giving a depth bound, for example 10^5 , with the commands:

```
Maude> search [1, 100000] in READERS-WRITERS :
      < 0, 0 > =>* < s(N:Nat), s(M:Nat) > .
```

No solution.

```
states: 100002 rewrites: 200001 in 312460ms cpu (636669ms real)
      (640 rews/sec)
```

```
Maude> search [1, 100000] in READERS-WRITERS :
      < 0, 0 > =>* < N:Nat, s(s(M:Nat)) > .
```

No solution.

```
states: 100002 rewrites: 200001 in 70600ms cpu (623434ms real)
      (2832 rews/sec)
```

As the reader can observe, these computations take quite a long time. Notice that the terms appearing during the search grow very quickly. A very

simple way of improving performance in this case is using the `iter` attribute for the `s` operator.

```
op s : Nat -> Nat [ctor iter] .
```

Then, we obtain a much better performance:

```
Maude> search [1, 100000] in READERS-WRITERS :
      < 0, 0 > =>* < s(N:Nat), s(M:Nat) > .

No solution.
states: 100002 rewrites: 200001 in 610ms cpu (1298ms real)
      (327870 rews/sec)
```

```
Maude> search [1, 100000] in READERS-WRITERS :
      < 0, 0 > =>* < N:Nat, s(s(M:Nat)) > .

No solution.
states: 100002 rewrites: 200001 in 400ms cpu (1191ms real)
      (500002 rews/sec)
```

In the following section we will use some formal tools for checking properties about the `READERS-WRITERS` module. Since some of these tools cannot handle the `iter` attribute, we use the module as shown above.

12.4 Verifying infinite-state systems through abstractions

The bounded model checking of our two invariants for the readers and writers example up to depth 10^6 greatly increases our confidence that the invariants hold, but, as already mentioned, bounded model checking is an incomplete procedure that falls short of being a proof: a counterexample at greater depth could exist.

Can we actually fully verify such invariants in an automatic way? One possible method is to verify the invariants through search not on the original infinite-state system, but on a finite-state *abstraction* of it, that is, on an appropriate quotient of the original system whose set of reachable states is finite. The paper [256] describes a simple method for, given a rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$, defining an abstraction \mathcal{A} of it: *just add equations*. That is, we can define our abstract theory as a rewrite theory $\mathcal{A} = (\Sigma, E \cup G, \phi, R)$, with G a set of extra equations powerful enough to collapse the infinite set of reachable states into a finite set. This method can be used (with an additional deadlock-freedom requirement) to verify not just invariants, but in fact any LTL formula (see Section 13.4).

Of course, the equations G we add cannot be arbitrary. First of all, they should respect all the necessary executability assumptions, so that in

$\mathcal{A} = (\Sigma, E \cup G, \phi, R)$ the equations $E \cup G$ should be ground Church-Rosser and terminating,² and the rules R should be ground coherent with $E \cup G$. Furthermore, the equations G should be *invariant-preserving* with respect to the invariants that we want to model check; that is, for any such invariant I and for any two ground terms t, t' denoting states such that $t =_{E \cup G} t'$, it should be the case that $E \vdash I(t) = I(t')$.

A first key observation is that, if both \mathcal{R} and \mathcal{A} *protect* the sort `Bool`, that is, the only canonical forms of that sort are `true` and `false`, and `true` \neq `false`, then the equations G are invariant-preserving. A second key observation is that we may be able to automatically check that a module protects the sort `Bool` by:

1. checking that it has no equations with `true` or `false` in the lefthand side;
2. checking that it is ground confluent and sort-decreasing with the Church-Rosser Checker (CRC) tool, described in Section 23.1.3;
3. checking that it is terminating with the Maude Termination Tool (MTT), described in Section 23.1.2; and
4. checking that it is sufficiently complete with the Sufficient Completeness Checker (SCC) tool, described in Section 23.1.5.

Indeed, since `true` and `false` are the only constructors of sort `Bool`, (4) ensures the “no junk” part of protection, whereas (1)–(3) ensure the “no confusion,” `true` \neq `false` part.

For invariant verification, the key property that an abstraction meeting these requirements satisfies is that, if I is one of the invariants preserved by G , then we have the implication

$$\mathcal{A}, \text{init} \models \Box I \implies \mathcal{R}, \text{init} \models \Box I$$

Therefore, if we can verify the invariant on \mathcal{A} , we are sure that it also holds on \mathcal{R} . However, the fact that we find a counterexample in \mathcal{A} does not necessarily mean that a counterexample exists for \mathcal{R} : it could be a spurious counterexample, caused by \mathcal{A} being too coarse of an abstraction, and therefore having no counterpart in \mathcal{R} . In such cases, one possible approach is to refine our abstraction by imposing fewer equations.

We can illustrate these ideas by defining an abstraction of our readers-writers system. In order to check that the equations in our abstraction preserve the invariants, we need an *explicit* representation of those invariants. Since at present the CRC and MTT tools do not handle predefined modules like `BOOL`, we use instead a sort `NewBool`.

```
mod READERS-WRITERS-PREDS is
  protecting READERS-WRITERS .
  sort NewBool .
  ops tt ff : -> NewBool [ctor] .
  ops mutex one-writer : Config -> NewBool [frozen] .
```

² Again, possibly *modulo* equational attributes $A \subseteq E \cup G$.

```

eq mutex(< s(N:Nat), s(M:Nat) >) = ff .
eq mutex(< 0, N:Nat >) = tt .
eq mutex(< N:Nat, 0 >) = tt .
eq one-writer(< N:Nat, s(s(M:Nat)) >) = ff .
eq one-writer(< N:Nat, 0 >) = tt .
eq one-writer(< N:Nat, s(0) >) = tt .
endm

```

We can now define our abstraction, in which all the states having more than one reader and no writers are identified with the state having exactly one reader and no writer.

```

mod READERS-WRITERS-ABS is
  including READERS-WRITERS-PREDS .
  including READERS-WRITERS .
  eq < s(s(N:Nat)), 0 > = < s(0), 0 > .
endm

```

where the second `including` importation is needed because the `READERS-WRITERS` module is not protected, but would be assumed protected by default (because it is protected in `READERS-WRITERS-PREDS`) unless we explicitly declare it in `including mode` (see Section 8.1.3).

In order to check both the executability and the invariant-preservation properties of this abstraction, since we have no equations with either `tt` or `ff` in their lefthand side, we now just need to check:

1. that the equations in both `READERS-WRITERS-PREDS` and `READERS-WRITERS-ABS` are ground confluent, sort-decreasing, and terminating;
2. that the equations in both `READERS-WRITERS-PREDS` and `READERS-WRITERS-ABS` are sufficiently complete; and
3. that the rules in both `READERS-WRITERS-PREDS` and `READERS-WRITERS-ABS` are ground coherent with respect to their equations.

Regarding termination, since the equations of `READERS-WRITERS-ABS` contain those of the module `READERS-WRITERS-PREDS`, it is enough to check the termination of the equations in the former. The MTT tool, using the AProVE termination tool, checks this automatically.

Once the `READERS-WRITERS-ABS` and `READERS-WRITERS-PREDS` modules are available in Full Maude (see Section 19.1), we can check confluence of the equations by invoking the CRC as follows:

```

Maude> (check Church-Rosser READERS-WRITERS-PREDS .)
Church-Rosser checking of READERS-WRITERS-PREDS
Checking solution:
  All critical pairs have been joined. The specification is
  locally-confluent.
The specification is sort-decreasing.

Maude> (check Church-Rosser READERS-WRITERS-ABS .)

```

Church-Rosser checking of READERS-WRITERS-ABS

Checking solution:

All critical pairs have been joined. The specification is locally-confluent.

The specification is sort-decreasing.

which finishes task (1).

Regarding (2), the SCC tool gives us:

```
Maude> (scc READERS-WRITERS-PREDS .)
Checking sufficient completeness of READERS-WRITERS-PREDS ...
Success: READERS-WRITERS-PREDS is sufficiently complete under the
assumption that it is weakly-normalizing, confluent, and
sort-decreasing.
```

```
Maude> (scc READERS-WRITERS-ABS .)
Checking sufficient completeness of READERS-WRITERS-ABS ...
Success: READERS-WRITERS-ABS is sufficiently complete under the
assumption that it is weakly-normalizing, confluent, and
sort-decreasing.
```

This leaves us with task (3), where the Coherence Checker (ChC) tool, described in Section 23.1.4, responds as follows:

```
Maude> (check coherence READERS-WRITERS-PREDS .)
Coherence checking of READERS-WRITERS-PREDS
Coherence checking solution:
All critical pairs have been rewritten and all equations
are non-constructor.
The specification is coherent.
```

```
Maude> (check coherence READERS-WRITERS-ABS .)
Coherence checking of READERS-WRITERS-ABS
Coherence checking solution:
The following critical pairs cannot be rewritten:
cp < s(0), 0 > => < s(N*@:Nat), 0 > .
```

Of course, ground coherence means that all ground instances of this pair can be rewritten by a one-step rewrite up to canonical form by the equations. We can reason by cases and decompose this critical pair into two:

```
cp < s(0), 0 > => < s(0), 0 > .
cp < s(0), 0 > => < s(s(N:Nat)), 0 > .
```

Using the `reduce` command we can check that the canonical form of the term $\langle s(s(N:Nat)), 0 \rangle$ is $\langle s(0), 0 \rangle$. Therefore, all we need to do is to check that $\langle s(0), 0 \rangle$ rewrites to *itself* (up to canonical form) in one step. We can do this check by giving the command:

```
Maude> search in READERS-WRITERS-ABS : < s(0), 0 > =>1 X:Config .
```

```
Solution 1 (state 0)
states: 1 rewrites: 2 in 0ms cpu (26ms real) (~ rews/sec)
X:Config --> < s(0), 0 >
```

```
Solution 2 (state 1)
states: 2 rewrites: 3 in 0ms cpu (124ms real) (~ rews/sec)
X:Config --> < 0, 0 >
```

```
No more solutions.
```

We have therefore completed all the checks (1)–(3) and can now automatically verify the two invariants on the abstract system, and therefore conclude that they hold in our original infinite-state readers-writers system, by executing the `search` commands:

```
Maude> search in READERS-WRITERS-ABS :
      < 0, 0 > =>* C:Config such that mutex(C:Config) = ff .
```

```
No solution.
states: 3 rewrites: 9 in 0ms cpu (0ms real) (~ rews/sec)
```

```
Maude> search in READERS-WRITERS-ABS :
      < 0, 0 > =>* C:Config such that one-writer(C:Config) = ff .
```

```
No solution.
states: 3 rewrites: 9 in 0ms cpu (0ms real) (~ rews/sec)
```

LTL Model Checking

As pointed out in Section 1.4, given a Maude system module, we can distinguish two levels of specification:

- a *system specification* level, provided by the rewrite theory specified by that system module which defines the behavior of the system, and
- a *property specification* level, given by some property (or properties) φ that we want to state and prove about our module.

This chapter discusses how a specific property specification logic, *linear temporal logic* (LTL), and a decision procedure for it, *model checking*, can be used to prove properties when the set of states reachable from an initial state in a system module is finite. It also explains how this is supported in Maude by its `MODEL-CHECKER` module, and other related modules, including the `SAT-SOLVER` module that can be used to check both satisfiability of an LTL formula and LTL tautologies. These modules are found in the file `model-checker.maude`.

Temporal logic allows specification of properties such as safety properties (ensuring that something bad never happens) and liveness properties (ensuring that something good eventually happens). These properties are related to the *infinite behavior* of a system. There are different temporal logics [64]; we focus on linear temporal logic [226, 64], because of its intuitive appeal, widespread use, and well-developed proof methods and decision procedures.

13.1 LTL formulas and the LTL module

Given a set AP of *atomic propositions*, we define the formulas of the *propositional linear temporal logic* $LTL(AP)$ inductively as follows:

- **True:** $\top \in LTL(AP)$.
- **Atomic propositions:** If $p \in AP$, then $p \in LTL(AP)$.
- **Next operator:** If $\varphi \in LTL(AP)$, then $\bigcirc\varphi \in LTL(AP)$.
- **Until operator:** If $\varphi, \psi \in LTL(AP)$, then $\varphi \mathcal{U} \psi \in LTL(AP)$.

- **Boolean connectives:** If $\varphi, \psi \in \text{LTL}(AP)$, then the formulas $\neg\varphi$, and $\varphi \vee \psi$ are in $\text{LTL}(AP)$.

Other LTL connectives can be defined in terms of the above minimal set of connectives as follows:

- Other Boolean connectives:
 - **False:** $\perp = \neg\top$
 - **Conjunction:** $\varphi \wedge \psi = \neg((\neg\varphi) \vee (\neg\psi))$
 - **Implication:** $\varphi \rightarrow \psi = (\neg\varphi) \vee \psi$.
- Other temporal operators:
 - **Eventually:** $\diamond\varphi = \top \mathcal{U} \varphi$
 - **Henceforth:** $\square\varphi = \neg\diamond\neg\varphi$
 - **Release:** $\varphi \mathcal{R} \psi = \neg((\neg\varphi) \mathcal{U} (\neg\psi))$
 - **Unless:** $\varphi \mathcal{W} \psi = (\varphi \mathcal{U} \psi) \vee (\square\varphi)$
 - **Leads-to:** $\varphi \rightsquigarrow \psi = \square(\varphi \rightarrow (\diamond\psi))$
 - **Strong implication:** $\varphi \Rightarrow \psi = \square(\varphi \rightarrow \psi)$
 - **Strong equivalence:** $\varphi \Leftrightarrow \psi = \square(\varphi \leftrightarrow \psi)$.

The LTL syntax, in a typewriter approximation of the above mathematical syntax, is supported in Maude by the following LTL functional module (in the file `model-checker.maude`).

```
fmod LTL is
protecting BOOL .
sorts Formula .

*** primitive LTL operators
ops True False : -> Formula [ctor format (g o)] .
op ~_ : Formula -> Formula [ctor prec 53 format (r o d)] .
op _/\_ : Formula Formula -> Formula
  [comm ctor gather (E e) prec 55 format (d r o d)] .
op _\/_ : Formula Formula -> Formula
  [comm ctor gather (E e) prec 59 format (d r o d)] .
op 0_ : Formula -> Formula [ctor prec 53 format (r o d)] .
op _U_ : Formula Formula -> Formula
  [ctor prec 63 format (d r o d)] .
op _R_ : Formula Formula -> Formula
  [ctor prec 63 format (d r o d)] .

*** defined LTL operators
op _->_ : Formula Formula -> Formula
  [gather (e E) prec 65 format (d r o d)] .
op _<->_ : Formula Formula -> Formula [prec 65 format (d r o d)] .
op <>_ : Formula -> Formula [prec 53 format (r o d)] .
op []_ : Formula -> Formula [prec 53 format (r d o d)] .
op _W_ : Formula Formula -> Formula [prec 63 format (d r o d)] .
op _|->_ : Formula Formula -> Formula [prec 63 format (d r o d)] .
op _=>_ : Formula Formula -> Formula
```

```

[gather (e E) prec 65 format (d r o d)] .
op _<=>_ : Formula Formula -> Formula [prec 65 format (d r o d)] .

vars f g : Formula .

eq f -> g = ~ f \\/ g .
eq f <-> g = (f -> g) /\ (g -> f) .
eq <> f = True U f .
eq [] f = False R f .
eq f W g = (f U g) \\/ [] f .
eq f l-> g = [] (f -> (<> g)) .
eq f => g = [] (f -> g) .
eq f <=> g = [] (f <-> g) .

*** negative normal form
eq ~ True = False .
eq ~ False = True .
eq ~ ~ f = f .
eq ~ (f \\/ g) = ~ f /\ ~ g .
eq ~ (f /\ g) = ~ f \\/ ~ g .
eq ~ 0 f = 0 ~ f .
eq ~ (f U g) = (~ f) R (~ g) .
eq ~ (f R g) = (~ f) U (~ g) .
endfm

```

Note that, for the moment, no set AP of atomic propositions has been specified in the LTL module. Section 13.2 explains how such atomic propositions are defined for a given system module M , and Section 13.3 explains how they are added to the LTL module as a subsort `Prop` of `Formula` in the `MODEL-CHECKER` module.

Note that the nonconstructor connectives have been defined in terms of more basic constructor connectives in the first set of equations. But since there are good reasons to put an LTL formula in *negative normal form* by pushing the negations next to the atomic propositions (this is specified by the second set of equations) we need to consider also the *duals* of the basic connectives \top , \circ , \mathcal{U} , and \vee (respectively, `True`, `0_`, `_U_`, and `_\/_`) as constructors. That is, we need to also have as constructors the dual connectives: \perp , \mathcal{R} , and \wedge (respectively, `False`, `_R_`, and `_/_`). Note that \circ is self-dual.

13.2 Associating Kripke structures to rewrite theories

Since the models of temporal logic are Kripke structures, we need to explain how we can associate a Kripke structure to the rewrite theory specified by a Maude system module M .

Kripke structures are the natural models for propositional temporal logic. Essentially, a Kripke structure is a (total) *unlabeled transition system* to which we have added a collection of unary state predicates on its set of states.

A binary relation $R \subseteq A \times A$ on a set A is called *total* if and only if for each $a \in A$ there is at least one $a' \in A$ such that $(a, a') \in R$. If R is not total, it can be made total by defining $R^\bullet = R \cup \{(a, a) \in A^2 \mid \nexists a' \in A (a, a') \in R\}$.

A *Kripke structure* is a triple $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L)$ such that A is a set, called the set of *states*, $\rightarrow_{\mathcal{A}}$ is a total binary relation on A , called the *transition relation*, and $L : A \rightarrow \mathcal{P}(AP)$ is a function, called the *labeling function*, associating to each state $a \in A$ the set $L(a)$ of those *atomic propositions* in AP that *hold* in the state a .

The semantics of $LTL(AP)$ is defined by means of a *satisfaction relation*

$$\mathcal{A}, a \models \varphi$$

between a Kripke structure \mathcal{A} having AP as its atomic propositions, a state $a \in A$, and an LTL formula $\varphi \in LTL(AP)$. Specifically, $\mathcal{A}, a \models \varphi$ holds if and only if for each path $\pi \in Path(\mathcal{A})_a$ the *path satisfaction relation*

$$\mathcal{A}, \pi \models \varphi$$

holds, where we define the set $Path(\mathcal{A})_a$ of *computation paths* starting at state a as the set of functions of the form $\pi : \mathbb{N} \rightarrow A$ such that $\pi(0) = a$ and, for each $n \in \mathbb{N}$, we have $\pi(n) \rightarrow_{\mathcal{A}} \pi(n+1)$.

We can define the path satisfaction relation (for any path, beginning at any state) inductively as follows:

- We always have $\mathcal{A}, \pi \models_{LTL} \top$.
- For $p \in AP$,

$$\mathcal{A}, \pi \models_{LTL} p \Leftrightarrow p \in L(\pi(0)).$$

- For $\bigcirc \varphi \in LTL(A)$,

$$\mathcal{A}, \pi \models_{LTL} \bigcirc \varphi \Leftrightarrow \mathcal{A}, s; \pi \models_{LTL} \varphi,$$

where $s : \mathbb{N} \rightarrow \mathbb{N}$ is the successor function, and where $(s; \pi)(n) = \pi(s(n)) = \pi(n+1)$.

- For $\varphi \mathcal{U} \psi \in LTL(\mathcal{A})$,

$$\mathcal{A}, \pi \models_{LTL} \varphi \mathcal{U} \psi \Leftrightarrow$$

$$(\exists n \in \mathbb{N})$$

$$((\mathcal{A}, s^n; \pi \models_{LTL} \psi) \wedge ((\forall m \in \mathbb{N}) m < n \Rightarrow \mathcal{A}, s^m; \pi \models_{LTL} \varphi)).$$

- For $\neg \varphi \in LTL(AP)$,

$$\mathcal{A}, \pi \models_{LTL} \neg \varphi \Leftrightarrow \mathcal{A}, \pi \not\models_{LTL} \varphi.$$

- For $\varphi \vee \psi \in LTL(AP)$,

$$\mathcal{A}, \pi \models_{LTL} \varphi \vee \psi \Leftrightarrow \mathcal{A}, \pi \models_{LTL} \varphi \text{ or } \mathcal{A}, \pi \models_{LTL} \psi.$$

How can we associate a Kripke structure to the rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$ specified by a Maude system module M ? We just need to make explicit two things:

- the intended *kind* k of states in the signature Σ , and
- the relevant *state predicates*, that is, the relevant set AP of atomic propositions.

In general, the state predicates need not be part of the *system specification* and therefore they need not be specified in our system module M . They are typically part of the *property specification*. This is because the state predicates need not be related to the operational semantics of M : they are just certain *predicates* about the states of the system specified by M that are needed to specify some *properties*.

Therefore, after choosing a given kind, say $[Foo]$, in M as our kind for states we can specify the relevant state predicates in a module M -PREDS protecting M according to the following general pattern:

```
mod M-PREDS is
  protecting M .
  including SATISFACTION .
  subsort Foo < State .
  ...
endm
```

where the dots ‘...’ indicate the part in which the syntax and semantics of the relevant state predicates are specified, as further explained in what follows.

The module $SATISFACTION$ (contained in the `model-checker.maude` file) is very simple, and has the following specification:

```
fmod SATISFACTION is
  protecting BOOL .
  sorts State Prop .
  op _|=_ : State Prop -> Bool [frozen] .
endfm
```

where the sorts $State$ and $Prop$ are unspecified. However, by importing $SATISFACTION$ into M -PREDS and giving the subsort declaration

```
subsort Foo < State .
```

all terms of sort Foo in M are also made terms of sort $State$. Note that we then have the kind identity $[Foo] = [State]$.

The operator

```
op _|=_ : State Prop -> Bool [frozen] .
```

is crucial to define the semantics of the relevant state predicates in M -PREDS. Each such state predicate is declared as an operator of sort $Prop$. In standard LTL propositional logic, the set AP of atomic propositions is assumed to be a set of *constants*. In Maude, however, we can define *parametric* state predicates, that is, operators of sort $Prop$ which need not be constants, but may have one or more sorts as parameter arguments. We then define the *semantics* of such state predicates (when the predicate holds) by appropriate equations.

We can illustrate all this by means of a simple mutual exclusion example. Suppose that our original system module *M* is the following module `MUTEX`, in which two processes, one named *a* and another named *b*, can be either waiting or in their critical section, and take turns accessing their critical section by passing each other a different *token* (either `$` or `*`).

```

mod MUTEX is
  sorts Name Mode Proc Token Conf .
  subsorts Token Proc < Conf .
  op none : -> Conf [ctor] .
  op _ _ : Conf Conf -> Conf [ctor assoc comm id: none] .
  ops a b : -> Name [ctor] .
  ops wait critical : -> Mode [ctor] .
  op [_,_] : Name Mode -> Proc [ctor] .
  ops * $ : -> Token [ctor] .
  rl [a-enter] : $ [a, wait] => [a, critical] .
  rl [b-enter] : * [b, wait] => [b, critical] .
  rl [a-exit] : [a, critical] => [a, wait] * .
  rl [b-exit] : [b, critical] => [b, wait] $ .
endm

```

Our obvious kind for states is the kind `[Conf]` of configurations. In order to state the desired safety and liveness properties we need state predicates telling us whether a process is waiting or is in its critical section. We can make these predicates *parametric* on the name of the process and define their semantics as follows:

```

mod MUTEX-PREDS is
  protecting MUTEX .
  including SATISFACTION .
  subsort Conf < State .
  op crit : Name -> Prop .
  op wait : Name -> Prop .
  var N : Name .
  var C : Conf .
  var P : Prop .
  eq [N, critical] C |= crit(N) = true .
  eq [N, wait] C |= wait(N) = true .
  eq C |= P = false [owise] .
endm

```

Note the equations, defining when each of the two parametric state predicates holds in a given state.

The above example illustrates a general method by which desired state predicates for a module *M* are defined in a *protecting* extension, say `M-PREDS`, of *M* which imports `SATISFACTION`. One specifies the desired states by choosing a sort in *M* and declaring it as a subsort of `State`. One then defines the syntax of the desired state predicates as operators of sort `Prop`, and defines their semantics by means of a set of equations that specify for what states

a given state predicate evaluates to **true**. We assume that those equations, when added to those of \mathbb{M} , are (ground) Church-Rosser and terminating, and, furthermore, that \mathbb{M} 's equational part is *protected* when the new operators and equations defining the state predicates are added.

Since we should *protect* **BOOL**, it is important to make sure that satisfaction of state predicates is *fully defined*. This can be checked with Maude's Sufficient Completeness Checker (SCC) tool. This means that we should give equations for when the predicates are **true** and when they are **false**. In practice, however, this can often be reduced to specifying *when a predicate is true* by means of (possibly conditional) equations of the general form,

$$t \models p(v_1, \dots, v_n) = \text{true} \text{ if } C$$

because we can use the **owise** feature (described in Section 4.5.4) to cover all the remaining cases, when it is false, with an equation

$$x : \text{State} \models p(y_1, \dots, y_n) = \text{false} \text{ [owise]}.$$

In some cases, however—for example, because we want to perform reasoning using formal tools which do not accept **owise** declarations—we may fully define the true and false cases of a predicate not by using the **owise** attribute, but by explicit (possibly conditional) equations of the more general form,

$$t \models p(v_1, \dots, v_n) = \text{bexp} \text{ if } C,$$

where *bexp* is an arbitrary Boolean expression.

We are now ready to associate with a system module \mathbb{M} specifying a rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$ (with a selected kind k of states and with state predicates Π defined by means of equations D in a protecting extension \mathbb{M} -PREDS) a Kripke structure whose atomic predicates are specified by the set

$$AP_{\Pi} = \{\theta(p) \mid p \in \Pi, \theta \text{ ground substitution}\},$$

where by convention we use the simplified notation $\theta(p)$ to denote the ground term $\theta(p(x_1, \dots, x_n))$. This defines a labeling function L_{Π} on the set of states $T_{\Sigma/E,k}$ assigning to each $[t] \in T_{\Sigma/E,k}$ the set of atomic propositions

$$L_{\Pi}([t]) = \{\theta(p) \in AP_{\Pi} \mid (E \cup D) \vdash (\forall \emptyset) t \models \theta(p) = \text{true}\}.$$

The Kripke structure we are interested in is then

$$\mathcal{K}(\mathcal{R}, k)_{\Pi} = (T_{\Sigma/E,k}, (\rightarrow_{\mathcal{R}}^1)^{\bullet}, L_{\Pi}),$$

where $(\rightarrow_{\mathcal{R}}^1)^{\bullet}$ denotes the total relation extending the one-step \mathcal{R} -rewriting relation $\rightarrow_{\mathcal{R}}^1$ among states of kind k , that is, $[t] \rightarrow_{\mathcal{R}}^1 [t']$ holds if and only if there are $u \in [t]$ and $u' \in [t']$ such that u' is the result of applying one of the rules in R to u at some position. Under the usual assumptions that E is (ground) Church-Rosser and terminating (possibly modulo some axioms A contained in E) and R is (ground) coherent relative to E (again, possibly modulo A), u can always be chosen to be the canonical form of t under the equations E .

13.3 LTL model checking

Suppose that, given a system module M specifying a rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$, we have:

- chosen a kind k in M as our kind of states;
- defined some state predicates Π and their semantics in a module, say M -PREDS, protecting M by the method already explained in Section 13.2.

Then, as explained in Section 13.2, this defines a Kripke structure $\mathcal{K}(\mathcal{R}, k)_\Pi$ on the set of atomic propositions AP_Π . Given an initial state $[t] \in T_{\Sigma/E, k}$ and an LTL formula $\varphi \in LTL(AP_\Pi)$ we would like to have a procedure to decide the satisfaction relation

$$\mathcal{K}(\mathcal{R}, k)_\Pi, [t] \models \varphi.$$

In general this relation is undecidable. It becomes decidable if two conditions hold:

1. the set of states in $T_{\Sigma/E, k}$ that are *reachable* from $[t]$ by rewriting is *finite*,¹ and
2. the rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$ specified by M plus the equations D defining the predicates Π are such that:
 - both E and $E \cup D$ are (ground) Church-Rosser and terminating, perhaps modulo some axioms A , with $(\Sigma, E) \subseteq (\Sigma \cup \Pi, E \cup D)$ a protecting extension, and
 - R is (ground) coherent relative to E (again, perhaps modulo some axioms A).

Under these assumptions, both the state predicates Π and the transition relation $\rightarrow_{\mathcal{R}}^1$ are *computable* and, given the finite reachability assumption, we can then settle the above satisfaction problem using a *model-checking procedure*.

Specifically, Maude uses an on-the-fly LTL model-checking procedure of the style described in [64]. The basis of this procedure is the following. Each LTL formula φ has an associated Büchi automaton B_φ whose acceptance ω -language is exactly that of the behaviors satisfying φ . We can then reduce the satisfaction problem

$$\mathcal{K}(\mathcal{R}, k)_\Pi, [t] \models \varphi$$

to the *emptiness problem* of the language accepted by the *synchronous product* of $B_{\neg\varphi}$ and (the Büchi automaton associated with) $(\mathcal{K}(\mathcal{R}, k)_\Pi, [t])$. The formula φ is satisfied if and only if such a language is empty. The model-checking procedure checks emptiness by looking for a counterexample, that is, an infinite computation belonging to the language recognized by the synchronous

¹ Note that this can happen even when $T_{\Sigma/E, k}$ is an infinite set: there is no requirement that $T_{\Sigma/E, k}$ is finite.

product. For a detailed explanation of this general method of on-the-fly LTL model checking, see [64]; and for a discussion of the specific techniques used in the Maude LTL model-checker implementation, see [139].

This makes clear our interest in obtaining the *negative normal form* of a formula $\neg\varphi$, since we need it to build the Büchi automaton $B_{\neg\varphi}$. For efficiency purposes we need to make $B_{\neg\varphi}$ as small as possible. The following module LTL-SIMPLIFIER (also in the `model-checker.maude` file) tries to further simplify the negative normal form of the formula $\neg\varphi$ in the hope of generating a smaller Büchi automaton $B_{\neg\varphi}$. This module is optional (the user may choose to include it or not when doing model checking) but tends to help building a smaller $B_{\neg\varphi}$.

```
fmod LTL-SIMPLIFIER is
  including LTL .

  *** The simplifier is based on:
  ***   Kousha Etessami and Gerard J. Holzman,
  ***   "Optimizing Buchi Automata", CONCUR 2000, LNCS 1877.
  *** We use the Maude sort system to do much of the work.

  sorts TrueFormula FalseFormula PureFormula PE-Formula PU-Formula .
  subsort TrueFormula FalseFormula < PureFormula <
          PE-Formula PU-Formula < Formula .

  op True : -> TrueFormula [ctor ditto] .
  op False : -> FalseFormula [ctor ditto] .
  op _/\_ : PE-Formula PE-Formula -> PE-Formula [ctor ditto] .
  op _/\_ : PU-Formula PU-Formula -> PU-Formula [ctor ditto] .
  op _/\_ : PureFormula PureFormula -> PureFormula [ctor ditto] .
  op _\/_ : PE-Formula PE-Formula -> PE-Formula [ctor ditto] .
  op _\/_ : PU-Formula PU-Formula -> PU-Formula [ctor ditto] .
  op _\/_ : PureFormula PureFormula -> PureFormula [ctor ditto] .
  op 0_ : PE-Formula -> PE-Formula [ctor ditto] .
  op 0_ : PU-Formula -> PU-Formula [ctor ditto] .
  op 0_ : PureFormula -> PureFormula [ctor ditto] .
  op _U_ : PE-Formula PE-Formula -> PE-Formula [ctor ditto] .
  op _U_ : PU-Formula PU-Formula -> PU-Formula [ctor ditto] .
  op _U_ : PureFormula PureFormula -> PureFormula [ctor ditto] .
  op _U_ : TrueFormula Formula -> PE-Formula [ctor ditto] .
  op _U_ : TrueFormula PU-Formula -> PureFormula [ctor ditto] .
  op _R_ : PE-Formula PE-Formula -> PE-Formula [ctor ditto] .
  op _R_ : PU-Formula PU-Formula -> PU-Formula [ctor ditto] .
  op _R_ : PureFormula PureFormula -> PureFormula [ctor ditto] .
  op _R_ : FalseFormula Formula -> PU-Formula [ctor ditto] .
  op _R_ : FalseFormula PE-Formula -> PureFormula [ctor ditto] .

  vars p q r s : Formula .
  var pe : PE-Formula .
```

```

var pu : PU-Formula .
var pr : PureFormula .

*** Rules 1, 2 and 3; each with its dual.
eq (p U r) /\ (q U r) = (p /\ q) U r .
eq (p R r) \/ (q R r) = (p \/ q) R r .
eq (p U q) \/ (p U r) = p U (q \/ r) .
eq (p R q) /\ (p R r) = p R (q /\ r) .
eq True U (p U q) = True U q .
eq False R (p R q) = False R q .

*** Rules 4 and 5 do most of the work.
eq p U pe = pe .
eq p R pu = pu .

*** An extra rule in the same style.
eq 0 pr = pr .

*** We also use the rules from:
***   Fabio Somenzi and Roderick Bloem,
***   "Efficient Buchi Automata from LTL Formulae",
***   p247-263, CAV 2000, LNCS 1633.
***   that are not subsumed by the previous system.

*** Four pairs of duals.
eq 0 p /\ 0 q = 0 (p /\ q) .
eq 0 p \/ 0 q = 0 (p \/ q) .
eq 0 p U 0 q = 0 (p U q) .
eq 0 p R 0 q = 0 (p R q) .
eq True U 0 p = 0 (True U p) .
eq False R 0 p = 0 (False R p) .
eq (False R (True U p)) \/ (False R (True U q))
  = False R (True U (p \/ q)) .
eq (True U (False R p)) /\ (True U (False R q))
  = True U (False R (p /\ q)) .

*** <= relation on formula
op _<=_ : Formula Formula -> Bool [prec 75] .

eq p <= p = true .
eq False <= p = true .
eq p <= True = true .

ceq p <= (q /\ r) = true if (p <= q) /\ (p <= r) .
ceq p <= (q \/ r) = true if p <= q .
ceq (p /\ q) <= r = true if p <= r .
ceq (p \/ q) <= r = true if (p <= r) /\ (q <= r) .

ceq p <= (q U r) = true if p <= r .

```

```

ceq (p R q) <= r = true if q <= r .
ceq (p U q) <= r = true if (p <= r) /\ (q <= r) .
ceq p <= (q R r) = true if (p <= q) /\ (p <= r) .
ceq (p U q) <= (r U s) = true if (p <= r) /\ (q <= s) .
ceq (p R q) <= (r R s) = true if (p <= r) /\ (q <= s) .

*** conditional rules depending on <= relation
ceq p /\ q = p if p <= q .
ceq p \/ q = q if p <= q .
ceq p /\ q = False if p <= ~ q .
ceq p \/ q = True if ~ p <= q .
ceq p U q = q if p <= q .
ceq p R q = q if q <= p .
ceq p U q = True U q if p /= True /\ ~ q <= p .
ceq p R q = False R q if p /= False /\ q <= ~ p .
ceq p U (q U r) = q U r if p <= q .
ceq p R (q R r) = q R r if q <= p .
endfm

```

Suppose that all the requirements listed above to perform model checking are satisfied. How do we then model check a given LTL formula in Maude for a given initial state $[t]$ in a module M ? We define a new module, say M -CHECK, according to the following pattern:

```

mod M-CHECK is
  protecting M-PREDS .
  including MODEL-CHECKER .
  including LTL-SIMPLIFIER . *** optional
  op init : -> k .          *** optional
  eq init = t .            *** optional
endm

```

The declaration of a constant `init` of the kind of states is not necessary: it is a matter of convenience, since the initial state `t` may be a large term.

The module `MODEL-CHECKER` (in the file `model-checker.maude`) is as follows:

```

fmod MODEL-CHECKER is
  protecting QID .
  including SATISFACTION .
  including LTL .
  subsort Prop < Formula .

  *** transitions and results
  sorts RuleName Transition TransitionList ModelCheckResult .
  subsort Qid < RuleName .
  subsort Transition < TransitionList .
  subsort Bool < ModelCheckResult .
  ops unlabeled deadlock : -> RuleName .

```

```

op {_,_} : State RuleName -> Transition [ctor] .
op nil : -> TransitionList [ctor] .
op __ : TransitionList TransitionList -> TransitionList
      [ctor assoc id: nil] .
op counterexample :
      TransitionList TransitionList -> ModelCheckResult [ctor] .

op modelCheck : State Formula ~> ModelCheckResult
      [special (...)] .
endfm

```

Its key operator is `modelCheck` (whose `special` attribute has been omitted here), which takes a state and an LTL formula and returns either the Boolean `true` if the formula is satisfied, or a counterexample when it is not satisfied. Note that the sort `Prop` coming from the `SATISFACTION` module is declared as a subsort of `Formula` in `LTL`. In each concrete use of `MODEL-CHECKER`, such as that in `M-CHECK` above, the atomic propositions in `Prop` will have been specified in a module such as `M-PREDS`.

Let us illustrate the use of this operator with our `MUTEX` example. Following the pattern described above, we can define the module

```

mod MUTEX-CHECK is
  protecting MUTEX-PREDS .
  including MODEL-CHECKER .
  including LTL-SIMPLIFIER .
  ops initial1 initial2 : -> Conf .
  eq initial1 = $ [a, wait] [b, wait] .
  eq initial2 = * [a, wait] [b, wait] .
endm

```

The relationships between all the modules involved at this point is illustrated in Figure 13.1, where a single arrow represents an `including` importation and a triple arrow represents a `protecting` importation.

We are then ready to model check different LTL properties of `MUTEX`. The first obvious property to check is mutual exclusion:

```

Maude> red modelCheck(initial1, [] ~(crit(a) /\ crit(b))) .
reduce in MUTEX-CHECK :
  modelCheck(initial1, []~ (crit(a) /\ crit(b))) .
result Bool: true

Maude> red modelCheck(initial2, [] ~(crit(a) /\ crit(b))) .
reduce in MUTEX-CHECK :
  modelCheck(initial2, []~ (crit(a) /\ crit(b))) .
result Bool: true

```

We can also model check the strong liveness property that if a process waits infinitely often, then it is in its critical section infinitely often:

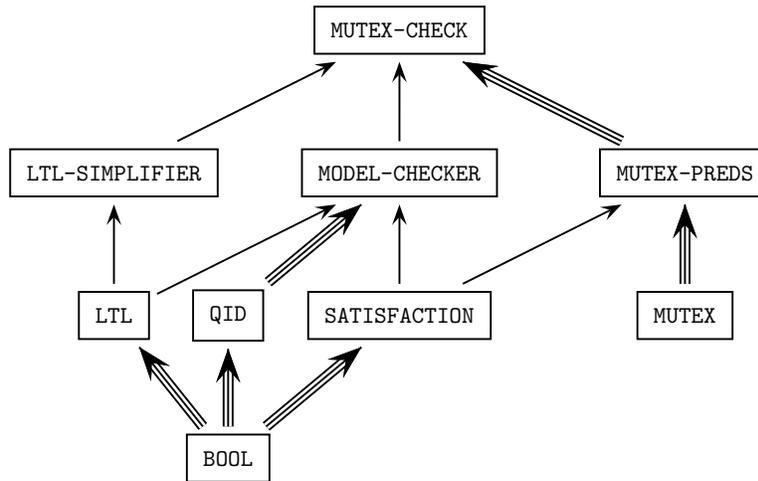


Fig. 13.1. Importation graph of model-checking modules

```

Maude> red modelCheck(initial1, ([]<> wait(a) -> ([]<> crit(a))) .
reduce in MUTEX-CHECK :
  modelCheck(initial1, []<> wait(a) -> []<> crit(a)) .
result Bool: true
  
```

```

Maude> red modelCheck(initial1, ([]<> wait(b) -> ([]<> crit(b))) .
reduce in MUTEX-CHECK :
  modelCheck(initial1, []<> wait(b) -> []<> crit(b)) .
result Bool: true
  
```

```

Maude> red modelCheck(initial2, ([]<> wait(a) -> ([]<> crit(a))) .
reduce in MUTEX-CHECK :
  modelCheck(initial2, []<> wait(a) -> []<> crit(a)) .
result Bool: true
  
```

```

Maude> red modelCheck(initial2, ([]<> wait(b) -> ([]<> crit(b))) .
reduce in MUTEX-CHECK :
  modelCheck(initial2, []<> wait(b) -> []<> crit(b)) .
result Bool: true
  
```

Of course, not all properties are true. Therefore, instead of a success we can get a *counterexample* showing why a property fails. Suppose that we want to check whether, beginning in the state `initial1`, process `b` will always be waiting. We then get the counterexample:

```

Maude> red modelCheck(initial1, [] wait(b)) .
reduce in MUTEX-CHECK : modelCheck(initial1, []wait(b)) .
result ModelCheckResult:
  
```

```

counterexample({$ [a, wait] [b, wait], 'a-enter}
              {[a, critical] [b, wait], 'a-exit}
              {* [a, wait] [b, wait], 'b-enter},
              {[a, wait] [b, critical], 'b-exit}
              {$ [a, wait] [b, wait], 'a-enter}
              {[a, critical] [b, wait], 'a-exit}
              {* [a, wait] [b, wait], 'b-enter})

```

The main constructors used in the syntax of a counterexample term are:

```

op {_,_} : State RuleName -> Transition .
op nil : -> TransitionList [ctor] .
op __ : TransitionList TransitionList -> TransitionList
      [ctor assoc id: nil] .
op counterexample :
      TransitionList TransitionList -> ModelCheckResult [ctor] .

```

Therefore, a counterexample is a pair consisting of two lists of transitions, where the first corresponds to a finite path beginning in the initial state, and the second describes a loop. This is because, if an LTL formula φ is not satisfied by a finite Kripke structure, it is always possible to find a counterexample for φ having the form of a path of transitions followed by a cycle [64]. Note that each transition is represented as a *pair*, consisting of a state and the label of the rule applied to reach the next state.

Let us finish this section with an example of how *not to use* the model checker. Consider the following specification:

```

mod MODEL-CHECK-BAD-EX is
  including MODEL-CHECKER .
  extending LTL .
  sort Foo .
  op a : -> Foo .
  op f : Foo -> Foo .
  rl a => f(a) .

  subsort Foo < State .
  op p : -> Prop .
endm

```

This module describes an *infinite* transition system of the form

$$a \rightarrow f(a) \rightarrow f(f(a)) \rightarrow f(f(f(a))) \rightarrow \dots,$$

where the property p is not satisfied anywhere. Therefore the state a does not satisfy, e.g., the property $\Box p$. However, if we try to invoke Maude with the command

```
Maude> red in MODEL-CHECK-BAD-EX : modelCheck(a, []p) .
```

we run into a nonterminating computation.

Making explicit that p does not hold in a by adding the equation

```
eq (a |= p) = false .
```

does not help. We run into the same problem even if the formula does not contain a temporal operator: `modelCheck(a, p)` does not terminate either. The reason is that the set of states reachable from `a` is *not* finite, and hence one of the main requirements for the model-checking algorithm is not satisfied.

13.4 Equational abstractions

As already explained in Section 12.4, when a system, specified as a rewrite theory $\mathcal{R} = (\Sigma, E, \phi, R)$ in a Maude system module, cannot be model checked, either because it is infinite-state or because it is finite-state but it is too large to be model checked in practice, an appealing possibility is to try to model check a finite-state *equational abstraction* of it. That is, we try to model check a rewrite theory $\mathcal{A} = (\Sigma, E \cup G, \phi, R)$ obtained from \mathcal{R} by adding some extra equations G to \mathcal{R} .

In order for \mathcal{A} to be a correct abstraction that we can model check in practice, the equations $E \cup G$ should be ground Church-Rosser and terminating (possibly modulo A), and the rules R should be ground coherent with $E \cup G$ (again, possibly modulo A). Furthermore, the equations G should *preserve the atomic propositions*² involved in the formulas that we want to model check [256]. This last requirement means that for any two ground terms t, t' denoting states, if $t =_{E \cup G} t'$, then the states $[t]_E$ and $[t']_E$ satisfy exactly the *same* atomic propositions in \mathcal{R} . A further technical requirement for LTL model checking is that the theory \mathcal{R} should be *deadlock free*, at least for the states reachable from an initial state; that is, that we cannot reach a terminating state. For rewrite theories \mathcal{R} with no rules with rewrites in their conditions, this requirement does not involve any practical loss of generality, since if \mathcal{R} is not deadlock free, we can always transform it into another theory bisimilar to it that is deadlock free, as described in [256] and Section 16.3.

Under these assumptions, it is proved in [256] that for any LTL formula φ involving those atomic propositions and any initial state `init` we have the implication

$$\mathcal{A}, \text{init} \models \varphi \implies \mathcal{R}, \text{init} \models \varphi.$$

Therefore, if we verify by model checking that φ holds for \mathcal{A} , then we can be sure that it also holds for \mathcal{R} . However, a counterexample for \mathcal{A} may or may not lift to a counterexample for \mathcal{R} . We presented in Section 12.4 an example of equational abstraction and illustrated its use in model checking invariants. Here we present another example showing how, more generally, we can use equational abstractions to model check LTL properties.

² We assume here that the equations E already contain those defining the semantics of the atomic propositions.

Our example is a simplified version of Lamport's bakery protocol. This is an infinite-state protocol that achieves mutual exclusion between processes by the usual method common in bakeries and deli shops: there is a number dispenser, and customers are served in sequential order according to the number that they hold. A simple Maude specification for the case of two processes is as follows:

```

mod BAKERY is
  protecting NAT .
  sorts Mode BState .
  ops sleep wait crit : -> Mode [ctor] .
  op <_,_,_,_> : Mode Nat Mode Nat -> BState [ctor] .

  op initial : -> BState .
  eq initial = < sleep, 0, sleep, 0 > .

  vars P Q : Mode .
  vars X Y : Nat .

  rl [p1_sleep] : < sleep, X, Q, Y > => < wait, s Y, Q, Y > .
  rl [p1_wait] : < wait, X, Q, 0 > => < crit, X, Q, 0 > .
  crl [p1_wait] : < wait, X, Q, Y > => < crit, X, Q, Y >
    if not (Y < X) .
  rl [p1_crit] : < crit, X, Q, Y > => < sleep, 0, Q, Y > .

  rl [p2_sleep] : < P, X, sleep, Y > => < P, X, wait, s X > .
  rl [p2_wait] : < P, 0, wait, Y > => < P, 0, crit, Y > .
  crl [p2_wait] : < P, X, wait, Y > => < P, X, crit, Y >
    if Y < X .
  rl [p2_crit] : < P, X, crit, Y > => < P, X, sleep, 0 > .
endm

```

In this module, states are represented by terms of sort `BState`, which are constructed by a 4-tuple operator `<_,_,_,_>`; the first two components describe the status of the first process (the mode it is currently in, and its priority as given by the number according to which it will be served), and the last two components the status of the second process. The rules describe how each process passes from being sleeping to waiting, from waiting to its critical section, and then back to sleeping.

We may wish to verify two basic properties about this protocol, namely:

- *mutual exclusion*, that is, the two processes are never simultaneously in their critical section, and
- *liveness*, that is, whenever a process enters the waiting mode, it will eventually enter its critical section.

Since the set of states reachable from `initial` is *infinite*, we should model check these properties using an abstraction. We can define the abstraction by adding to the equations of `BAKERY` a set G of additional equations defining a

quotient of the set of states. We can do so in the following module extending BAKERY by equations and leaving the rules unchanged:

```

mod ABSTRACT-BAKERY is
  including BAKERY .
  vars P Q : Mode .
  vars X Y : Nat .
  eq < P, 0, Q, s s Y > = < P, 0, Q, s 0 > .
  eq < P, s s X, Q, 0 > = < P, s 0, Q, 0 > .
  eq < P, s s s X, Q, s s s Y > = < P, s s X, Q, s s Y > .
  eq < P, s s s X, Q, s s 0 > = < P, s s 0, Q, s 0 > .
  eq < P, s s s X, Q, s 0 > = < P, s s 0, Q, s 0 > .
  eq < P, s s 0, Q, s s Y > = < P, s 0, Q, s 0 > .
  eq < P, s 0, Q, s s Y > = < P, s 0, Q, s 0 > .
endm

```

Three key questions are:

- Is the set of states now finite?
- Does this abstraction correspond to a rewrite theory whose equations are ground confluent, sort-decreasing, and terminating?
- Are the rules still ground coherent?

The check of termination for the ABSTRACT-BAKERY module follows from that for the bigger module ABSTRACT-BAKERY-PREDS, which is discussed later.

When we give to the Maude Church-Rosser Checker (CRC) tool a version without predefined modules of the ABSTRACT-BAKERY module to check local confluence we get:

```

Maude> (check Church-Rosser ABSTRACT-BAKERY .)
Church-Rosser checking of ABSTRACT-BAKERY
Checking solution:
  All critical pairs have been joined. The specification is
  locally-confluent.
The specification is sort-decreasing.

```

It is also clear that the set of states is now *finite*, since in the canonical forms obtained with these equations the natural numbers possible in the state can never be greater than $s(s(0))$ (i.e., 2). In fact, it is easy to see that the equivalence on states defined by the above equations is: $\langle P, N, Q, M \rangle \equiv \langle P', N', Q', M' \rangle$ iff

- $P = P'$ and $Q = Q'$,
- $N = 0$ iff $N' = 0$,
- $M = 0$ iff $M' = 0$,
- $M < N$ iff $M' < N'$.

This leaves us with the ground coherence question. Checking with Maude's Coherence Checker (ChC) a version without predefined modules, in which `true` and `false` are replaced by `tt` and `ff`, respectively, gives us:

```

Maude> (check coherence ABSTRACT-BAKERY .)
Coherence checking of ABSTRACT-BAKERY
Coherence checking solution:
The following critical pairs cannot be rewritten:
cp < sleep, 0, Q@:Mode, s 0 >
  => < wait, s s s Y*@:Nat, Q@:Mode, s s Y*@:Nat > .
cp < sleep, s 0, Q@:Mode, s 0 >
  => < wait, s s s Y*@:Nat, Q@:Mode, s s Y*@:Nat > .
cp < P@:Mode, s 0, sleep, 0 >
  => < P@:Mode, s s X*@:Nat, wait, s s s X*@:Nat > .
cp < P@:Mode, s s 0, sleep, s 0 >
  => < P@:Mode, s s X*@:Nat, wait, s s s X*@:Nat > .
ccp < wait, s s 0, Q@:Mode, s 0 >
  => < crit, s s 0, Q@:Mode, s 0 >
  if s 0 < s s s X*@:Nat = ff .
ccp < wait, s s 0, Q@:Mode, s 0 >
  => < crit, s s 0, Q@:Mode, s 0 >
  if s s 0 < s s s X*@:Nat = ff .
ccp < wait, s s X*@:Nat, Q@:Mode, s s Y*@:Nat >
  => < crit, s s X*@:Nat, Q@:Mode, s s Y*@:Nat >
  if s s s Y*@:Nat < s s s X*@:Nat = ff .
ccp < P@:Mode, s 0, wait, s 0 >
  => < P@:Mode, s 0, crit, s 0 >
  if s s Y*@:Nat < s 0 = tt .
ccp < P@:Mode, s 0, wait, s 0 >
  => < P@:Mode, s 0, crit, s 0 >
  if s s Y*@:Nat < s s 0 = tt .
ccp < P@:Mode, s s X*@:Nat, wait, s s Y*@:Nat >
  => < P@:Mode, s s X*@:Nat, crit, s s Y*@:Nat >
  if s s s Y*@:Nat < s s s X*@:Nat = tt .

```

To interpret these pairs the first key observation is that both NAT and BOOL are *protected* in ABSTRACT-BAKERY. This follows from the above check for confluence, plus the (postponed) proof of termination, plus the following sufficient completeness check,

```

Maude> (scc ABSTRACT-BAKERY .)
Success: ABSTRACT-BAKERY is sufficiently complete under the
assumption that it is weakly-normalizing, confluent, and
sort-decreasing.

```

plus the observation that no equations involving either the successor function or zero, or tt or ff have been added in ABSTRACT-BAKERY. But since NAT and BOOL are protected, the only pairs with satisfiable conditions are the following:

```

cp < sleep, 0, Q@:Mode, s 0 >
  => < wait, s s s Y*@:Nat, Q@:Mode, s s Y*@:Nat > .
cp < sleep, s 0, Q@:Mode, s 0 >
  => < wait, s s s Y*@:Nat, Q@:Mode, s s Y*@:Nat > .
cp < P@:Mode, s 0, sleep, 0 >

```

```

=> < P@:Mode, s s X*@:Nat, wait, s s s X*@:Nat > .
cp < P@:Mode, s s 0, sleep, s 0 >
=> < P@:Mode, s s X*@:Nat, wait, s s s X*@:Nat > .
ccp < wait, s s X*@:Nat, Q@:Mode, s s Y*@:Nat >
=> < crit, s s X*@:Nat, Q@:Mode, s s Y*@:Nat >
  if s s s Y*@:Nat < s s s X*@:Nat = ff .
ccp < P@:Mode, s s X*@:Nat, wait, s s Y*@:Nat >
=> < P@:Mode, s s X*@:Nat, crit, s s Y*@:Nat >
  if s s s Y*@:Nat < s s s X*@:Nat = tt .

```

all of which can be inductively rewritten. We can illustrate the method of inductive proof with the first unconditional and the first conditional pair.

The first unconditional pair is:

```

cp < sleep, 0, Q@:Mode, s 0 >
=> < wait, s s s Y*@:Nat, Q@:Mode, s s Y*@:Nat > .

```

We can first inductively prove the equation

```

eq < wait, s s s Y:Nat, Q:Mode, s s Y:Nat >
= < wait, 2, Q, 1 > .

```

in the module ABSTRACT-BAKERY. We can induct on $Y:Nat$, which gives us the following two goals:

```

eq < wait, s s s 0, Q:Mode, s s 0 > = < wait, 2, Q, 1 > .

ceq < wait, s s s s Y:Nat, Q:Mode, s s s Y:Nat >
= < wait, 2, Q, 1 >
  if < wait, s s s Y:Nat, Q:Mode, s s Y:Nat >
  = < wait, 2, Q, 1 > .

```

These two goals can be easily proved either using the ITP, or directly in Maude by simplifying the first goal to a syntactic identity, and by applying the *Theorem of Constants* to the second goal and adding the premise (instantiated with a constant) as an extra lemma to simplify the conclusion (also instantiated with a constant) to a syntactic identity.

We can then check that the above critical pair fills in by giving the `search` command:

```

Maude> search in ABSTRACT-BAKERY :
      < sleep, 0, Q, 1 > =>1 X:BState .

Solution 1 (state 1)
states: 2  rewrites: 1 in 0ms cpu (54ms real) (~ rews/sec)
X:BState --> < wait, 2, Q, 1 >

No more solutions.

```

Similarly, consider the first conditional critical pair, where, using the first equation among the inductive lemmas below

```

eq s X < s Y = X < Y .
eq 0 < s X = true .
eq s X < 0 = false .
eq X < s X = true .
eq s X < X = false .
ceq X < s Y = true if X < Y .
ceq s X < Y = false if X < Y = false .

```

we can simplify its condition as follows:

```

ccp < wait, s s X:Nat, Q:Mode, s s Y:Nat >
=> < crit, s s X:Nat, Q:Mode, s s Y:Nat >
if Y:Nat < X:Nat = ff .

```

Using the *Theorem of Constants*, we can convert the variables X and Y into constants a and b and add an equation assuming the condition $b < a = \text{ff}$. Then, using also the above equations as extra lemmas, we can fill in this conditional critical pair by giving the `search` command:

```

Maude> search in ABSTRACT-BAKERY :
      < wait, s s a, Q, s s b > =>1 X:BState .

```

```

Solution 1 (state 1)
X:BState --> < crit, s_2(a), Q, s_2(b) >

```

No more solutions.

What about *state predicates*? Are they preserved by the abstraction? In order to specify the desired mutual exclusion and liveness properties, it is enough to specify in Maude the following state predicates:

```

mod BAKERY-PREDS is
  protecting BAKERY .
  including SATISFACTION .
  subsort BState < State .
  ops 1wait 2wait 1crit 2crit : -> Prop [ctor] .
  vars P Q : Mode .
  vars X Y : Nat .
  eq < wait, X, Q, Y > |= 1wait = true .
  eq < sleep, X, Q, Y > |= 1wait = false .
  eq < crit, X, Q, Y > |= 1wait = false .
  eq < P, X, wait, Y > |= 2wait = true .
  eq < P, X, sleep, Y > |= 2wait = false .
  eq < P, X, crit, Y > |= 2wait = false .
  eq < crit, X, Q, Y > |= 1crit = true .
  eq < sleep, X, Q, Y > |= 1crit = false .
  eq < wait, X, Q, Y > |= 1crit = false .
  eq < P, X, crit, Y > |= 2crit = true .
  eq < P, X, sleep, Y > |= 2crit = false .
  eq < P, X, wait, Y > |= 2crit = false .
endm

```

These predicates are then imported without change, together with `ABSTRACT-BAKERY`, in the module:

```
mod ABSTRACT-BAKERY-PREDS is
  protecting ABSTRACT-BAKERY .
  including BAKERY-PREDS .
endm
```

The preservation of these state predicates can be guaranteed if we show that both `BAKERY-PREDS` and `ABSTRACT-BAKERY-PREDS` protect `BOOL`. This follows from the absence of any equations having `true` or `false` in their lefthand sides plus the following facts, all of which are checked by Maude tools (after replacing the predefined modules `NAT` and `BOOL` by equivalent specifications when necessary):

1. both `BAKERY-PREDS` and `ABSTRACT-BAKERY-PREDS` are sufficiently complete;
2. both `BAKERY-PREDS` and `ABSTRACT-BAKERY-PREDS` are locally confluent and sort-decreasing;
3. both `BAKERY-PREDS` and `ABSTRACT-BAKERY-PREDS` are terminating.

The SCC tool checks fact (1):

```
Maude> (scc BAKERY-PREDS .)
Success: BAKERY-PREDS is sufficiently complete under the assumption
  that it is weakly-normalizing, confluent, and sort-decreasing.
```

```
Maude> (scc ABSTRACT-BAKERY-PREDS .)
Success: ABSTRACT-BAKERY-PREDS is sufficiently complete under
  the assumption that it is weakly-normalizing, confluent, and
  sort-decreasing.
```

The CRC tool checks fact (2):

```
Maude> (check Church-Rosser BAKERY-PREDS .)
All critical pairs have been joined. The specification is
  locally-confluent.
The specification is sort-decreasing.
```

```
Maude> (check Church-Rosser ABSTRACT-BAKERY-PREDS .)
All critical pairs have been joined. The specification is
  locally-confluent.
The specification is sort-decreasing.
```

All we have left is checking termination of the equations in `BAKERY-PREDS` and `ABSTRACT-BAKERY-PREDS`, that is, fact (3), plus the pending proof of terminating equations for `ABSTRACT-BAKERY`. But since the equations in `ABSTRACT-BAKERY-PREDS` are precisely the union of those in `ABSTRACT-BAKERY` and `BAKERY-PREDS`, it is enough to check that `ABSTRACT-BAKERY-PREDS` is

terminating. This check succeeds with the Maude Termination Tool (MTT), described in Section 23.1.2.

This finishes all the checks of correctness and executability. The only remaining issue is deadlock freedom, which is required for the correctness of the abstraction. To ensure deadlock freedom we can perform the *automatic module transformation* described in Section 16.3, that preserves all the desired executability properties to obtain a semantically equivalent, deadlock-free version, say DF-BAKERY, of BAKERY. After loading the BAKERY module, we can obtain its deadlock-free version DF-BAKERY by performing this transformation in Full Maude (with the extension described in Section 16.3) as follows:

```
(mod DF-BAKERY is
  protecting DEADLOCK-FREE[BAKERY, BState] .
endm)
```

Note that the kind of states in DEADLOCK-FREE[BAKERY, BState] has changed. It is now [EConfig] and there is an operator

```
op {_} : State -> EConfig .
```

wrapping each state of kind [BState] into a state of kind [EConfig]. This means that we have to slightly redefine our state predicates, since they take now states of kind [EConfig], as follows:

```
(mod DF-BAKERY-PREDS is
  protecting DF-BAKERY .
  including SATISFACTION .
  subsort EConfig < State .
  ops 1wait 2wait 1crit 2crit : -> Prop [ctor] .
  vars P Q : Mode .
  vars X Y : Nat .
  eq {< wait, X, Q, Y >} |= 1wait = true .
  eq {< sleep, X, Q, Y >} |= 1wait = false .
  eq {< crit, X, Q, Y >} |= 1wait = false .
  eq {< P, X, wait, Y >} |= 2wait = true .
  eq {< P, X, sleep, Y >} |= 2wait = false .
  eq {< P, X, crit, Y >} |= 2wait = false .
  eq {< crit, X, Q, Y >} |= 1crit = true .
  eq {< sleep, X, Q, Y >} |= 1crit = false .
  eq {< wait, X, Q, Y >} |= 1crit = false .
  eq {< P, X, crit, Y >} |= 2crit = true .
  eq {< P, X, sleep, Y >} |= 2crit = false .
  eq {< P, X, wait, Y >} |= 2crit = false .
endm)
```

Our desired module DF-ABSTRACT-BAKERY has exactly the same equations as before, but now includes DF-BAKERY instead of BAKERY.

```
(mod DF-ABSTRACT-BAKERY is
  including DF-BAKERY .
```

```

vars P Q : Mode .
vars X Y : Nat .
eq < P, 0, Q, s s Y > = < P, 0, Q, s 0 > .
eq < P, s s X, Q, 0 > = < P, s 0, Q, 0 > .
eq < P, s s s X, Q, s s s Y > = < P, s s X, Q, s s Y > .
eq < P, s s s X, Q, s s 0 > = < P, s s 0, Q, s 0 > .
eq < P, s s s X, Q, s 0 > = < P, s s 0, Q, s 0 > .
eq < P, s s 0, Q, s s Y > = < P, s 0, Q, s 0 > .
eq < P, s 0, Q, s s Y > = < P, s 0, Q, s 0 > .
endm)

```

Finally we can verify our two desired properties as follows:

```

(mod DF-ABSTRACT-BAKERY-PREDS is
  protecting DF-ABSTRACT-BAKERY .
  including DF-BAKERY-PREDS .
endm)

(mod DF-ABSTRACT-BAKERY-CHECK is
  including MODEL-CHECKER .
  including DF-ABSTRACT-BAKERY-PREDS .
endm)

Maude> (red modelCheck({initial}, []~(1crit /\ 2crit)) .)
reduce in DF-ABSTRACT-BAKERY-CHECK :
modelCheck({initial}, []~(1crit /\ 2crit))
result Bool :
  true

Maude> (red modelCheck({initial},
  (1wait |-> 1crit) /\ (2wait |-> 2crit)) .)
reduce in DF-ABSTRACT-BAKERY-CHECK :
modelCheck({initial}, (1wait |-> 1crit) /\ (2wait |-> 2crit))
result Bool :
  true

```

The reason why if an LTL formula holds in an equational abstraction \mathcal{A} it also holds in the original theory \mathcal{R} is that an equational abstraction defines a *simulation map* $q : \mathcal{K}(\mathcal{R}, k)_\Pi \rightarrow \mathcal{K}(\mathcal{A}, k)_\Pi$, mapping each state $[t]_E$ in \mathcal{R} to its corresponding abstract state $[t]_{E \cup G}$ in \mathcal{A} , and such simulation maps reflect satisfaction of LTL formulas (see [256]). We can consider other forms of abstraction based, not just on simulation maps (which require each transition in $\mathcal{K}(\mathcal{R}, k)_\Pi$ to be mimicked by a transition in $\mathcal{K}(\mathcal{A}, k)_\Pi$), but, more generally, on *stuttering simulation maps* $q : \mathcal{K}(\mathcal{R}, k)_\Pi \rightarrow \mathcal{K}(\mathcal{A}, k)_\Pi$, which relate finite sequences of transitions in $\mathcal{K}(\mathcal{R}, k)_\Pi$ and in $\mathcal{K}(\mathcal{A}, k)_\Pi$. Such maps reflect satisfaction of LTL- \bigcirc formulas, that is, of LTL formulas not involving the *next* operator \bigcirc . Therefore, for such formulas a stuttering simulation map can be used to prove the formula for \mathcal{R} by model checking it for \mathcal{A} . Two abstraction techniques based on stuttering simulations, that complement the

method of equational abstraction presented here and are applicable to rewrite theories in general, are the method of theoroidal maps presented in [233] and the state-space reduction technique proposed in [154]. A further, stuttering-bisimulation-based abstraction technique applicable to real-time systems and supported by the Real-Time Maude tool is described in [290].

13.5 The LTL satisfiability and tautology checker

A formula $\varphi \in \text{LTL}(AP)$ is *satisfiable* if there is a Kripke structure $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L)$, with $L : A \rightarrow \mathcal{P}(AP)$, and a computation path π such that $\mathcal{A}, \pi \models \varphi$. Satisfiability of a formula $\varphi \in \text{LTL}(AP)$ is a decidable property. In Maude, the satisfiability decision procedure is supported by the following predefined functional module `SAT-SOLVER` (also in the file `model-checker.maude`).

```
fmod SAT-SOLVER is
  protecting LTL .

  *** formula lists and results
  sorts FormulaList SatSolveResult TautCheckResult .
  subsort Formula < FormulaList .
  subsort Bool < SatSolveResult TautCheckResult .
  op nil : -> FormulaList [ctor] .
  op _;_ : FormulaList FormulaList -> FormulaList
    [ctor assoc id: nil] .
  op model : FormulaList FormulaList -> SatSolveResult [ctor] .

  op satSolve : Formula ~> SatSolveResult [special (...)] .

  op counterexample :
    FormulaList FormulaList -> TautCheckResult [ctor] .
  op tautCheck : Formula ~> TautCheckResult .
  op $invert : SatSolveResult -> TautCheckResult .

  var F : Formula .
  vars L C : FormulaList .
  eq tautCheck(F) = $invert(satSolve(~ F)) .
  eq $invert(false) = true .
  eq $invert(model(L, C)) = counterexample(L, C) .
endfm
```

One can define the desired atomic predicates in a module extending `SAT-SOLVER`, such as, for example,

```
fmod SAT-SOLVER-TEST is
  extending SAT-SOLVER .
  extending LTL .
```

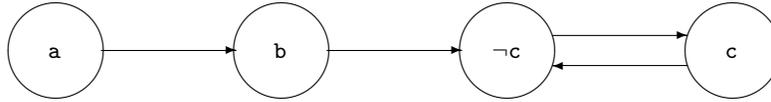


Fig. 13.2. Graphical representation of a Kripke structure

```

ops a b c d e p q r : -> Formula .
endfm

```

The user can then decide the satisfiability of an LTL formula involving those atomic propositions by applying the operator `satSolve` (whose `special` attribute has also been omitted in the module above) to the given formula and evaluating the expression. The resulting solution of sort `SatSolveResult` is then either `false`, if no model exists, or a finite model satisfying the formula. Such a model is described by a comma-separated pair of finite paths of states: an initial path leading to a cycle. Each state is described by a conjunction of atomic propositions or negated atomic propositions, with the propositions not mentioned in the conjunction being “don’t care” ones. For example, we can evaluate

```

Maude> red satSolve(a /\ (0 b) /\ (0 0 ((~ c) /\ [](c \/ (0 c)))) .
reduce in SAT-SOLVER-TEST :
  satSolve(0 0 (~ c /\ [](c \/ 0 c)) /\ (a /\ 0 b)) .
result SatSolveResult: model(a ; b, (~ c) ; c)

```

which is satisfied by a four-state model with `a` holding in the first state, `b` holding in the second, `c` not holding in the third but holding in the fourth, and the fourth state going back to the third, as shown in Figure 13.2.

We call $\varphi \in \text{LTL}(AP)$ a *tautology* if and only if $\mathcal{A}, a \models_{\text{LTL}} \varphi$ holds for every Kripke structure $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L)$ with $L : A \rightarrow \mathcal{P}(AP)$, and every state $a \in A$. It then follows easily that φ is a tautology if and only if $\neg\varphi$ is unsatisfiable. Therefore, the module `SAT-SOLVER` can also be used as a tautology checker. This is accomplished by using the `tautCheck`, `$invert`, and `counterexample` operators and their corresponding equations in `SAT-SOLVER`. The `tautCheck` function returns either `true` if the formula is a tautology, or a finite model that does not satisfy the formula. For example, we can evaluate:

```

Maude> red tautCheck((a => (0 a)) <-> (a => ([] a))) .
reduce in SAT-SOLVER-TEST : tautCheck((a => 0 a) <-> a => []a) .
result Bool: true

Maude> red tautCheck((a -> (0 a)) <-> (a -> ([] a))) .
reduce in SAT-SOLVER-TEST : tautCheck((a -> 0 a) <-> a -> []a) .
result TautCheckResult: counterexample(a ; a ; (~ a), True)

```

The tautology checker gives us also a *decision procedure for semantic LTL equality*, which is further discussed in [139].

13.6 Verifying LTL properties of imperative concurrent programs

How can the rewriting logic approach that we have been developing in the previous sections be applied to the verification of *imperative concurrent programs*, say in an imperative language \mathcal{L} ? Essentially we have to do three things:

1. specify precisely the *semantics* of the imperative concurrent language \mathcal{L} as a *rewrite theory* $\mathcal{R}_{\mathcal{L}}$,
2. choose an appropriate kind k of states in $\mathcal{R}_{\mathcal{L}}$ and define appropriate state predicates Π in an equational theory extending that of $\mathcal{R}_{\mathcal{L}}$ in protecting mode,
3. express the desired verification of properties of a program P in \mathcal{L} as the satisfaction of, for example, an LTL formula (or formulas) φ by the Kripke structure $\mathcal{K}(\mathcal{R}_{\mathcal{L}}, k)_{\Pi}$, and apply some verification or proof method (model checking, search, abstraction, deductive proof, etc.) to either verify or disprove φ .

13.6.1 The semantics of a simple parallel language

We illustrate step (1) in the above process by defining in Maude the semantics of a simple parallel language as a rewrite theory. The language and its specification are exactly those in [139]. First, a model of the memory itself has to be developed; then the syntax of the programs used by the processes is defined. All this can be done in a series of functional modules that we proceed to present in detail.

In the following MEMORY module, a memory is represented as a set of pairs formed by an identifier and an integer.

```
fmod MEMORY is
  protecting INT .
  protecting QID .
  sorts Memory .
  op none : -> Memory [ctor] .
  op _- : Memory Memory -> Memory [ctor assoc comm id: none] .
  op [_,_] : Qid Int -> Memory [ctor] .
endfm
```

Then we define an equality test comparing the contents of a named memory location to a given integer, in another functional module.

```
fmod TESTS is
  protecting MEMORY .
  sort Test .
  op _=_ : Qid Int -> Test [ctor] .
  op eval : Test Memory ~> Bool .
```

```

var Q : Qid .
var M : Memory .
vars N N' : Int .
eq eval(Q = N, [Q, N'] M) = N == N' .
endfm

```

The following functional module `SEQUENTIAL` provides syntax for a very simple sequential programming language. We can abstract certain terminating, or potentially nonterminating, program fragments as constants of sorts `UserStatement` and `LoopingUserStatement`, respectively.

```

fmod SEQUENTIAL is
protecting TESTS .
sorts UserStatement LoopingUserStatement Program .
subsort LoopingUserStatement < UserStatement < Program .
op skip : -> Program [ctor] .
op _;_ : Program Program -> Program [ctor prec 61 assoc id: skip] .
op _:=_ : Qid Int -> Program [ctor] .
op if_then-fi : Test Program -> Program [ctor] .
op while_do_od : Test Program -> Program [ctor] .
op repeat_forever : Program -> Program [ctor] .
endfm

```

Using the above modules, we can then define our simple parallel language in a system module `PARALLEL`. The *global state* is a triple consisting of:

1. a “soup” (set) of processes;
2. the shared memory; and
3. a process identifier recording the last process that touched the memory or, in any event, performed some computation; this is used to express fairness properties.

Processes themselves are *pairs* having a process identifier and a program.

```

mod PARALLEL is
protecting SEQUENTIAL .
sorts Pid Process Soup MachineState .
subsort Process < Soup .
op [_,_] : Pid Program -> Process [ctor] .
op empty : -> Soup [ctor] .
op _|_ : Soup Soup -> Soup [ctor prec 61 assoc comm id: empty] .
op {_,_,_} : Soup Memory Pid -> MachineState [ctor] .

vars P R : Program .
var S : Soup .
var U : UserStatement .
var L : LoopingUserStatement .
vars I J : Pid .
var M : Memory .
var Q : Qid .

```

```

vars N X : Int .
var T : Test .

```

The operational semantics of this programming language is given by just six rules. The first two rules deal with terminating and potentially nonterminating user statements. Note that there is no need to give a rule for skip, because it is the identity element for the sequential composition operator `_;_`.

```

r1 {[I, U ; R] | S, M, J} => {[I, R] | S, M, I} .

```

```

r1 {[I, L ; R] | S, M, J} => {[I, L ; R] | S, M, I} .

```

```

r1 {[I, (Q := N) ; R] | S, [Q, X] M, J}
=> {[I, R] | S, [Q, N] M, I} .

```

```

r1 {[I, if T then P fi ; R] | S, M, J}
=> {[I, if eval(T, M) then P else skip fi ; R] | S, M, I} .

```

```

r1 {[I, while T do P od ; R] | S, M, J}
=> {[I, if eval(T, M) then (P ; while T do P od)
     else skip fi ; R] | S, M, I} .

```

```

r1 {[I, repeat P forever ; R] | S, M, J}
=> {[I, P ; repeat P forever ; R] | S, M, I} .

```

```

endm

```

13.6.2 Model checking Dekker's algorithm

We illustrate steps (2) and (3) in the verification of imperative concurrent programs, as well as the use of the Maude model checker, with the example of Dekker's algorithm, one of the earliest correct solutions to the mutual exclusion problem. The Maude specification of this algorithm is exactly as in [139].

The algorithm assumes processes that execute concurrently on a shared memory machine and communicate with each other through shared variables. There are two processes: `p1` and `p2`. Process `p1` sets a Boolean variable `c1` to 1 to indicate that it wishes to enter its critical section. Process `p2` does the same with variable `c2`. If one process, after setting its variable to 1, finds that the variable of its competitor is 0, then it enters its critical section right away. In case of a tie (both variables set to 1) the tie is broken using a variable `turn` that takes values in `{1, 2}`.

We can then define the two processes for Dekker's algorithm as programs in our simple language, as well as the desired initial state, in the following module extending `PARALLEL`.

```

mod DEKKER is
  extending PARALLEL .
  subsort Int < Pid .

```

```

op crit : -> UserStatement .
op rem : -> LoopingUserStatement .
ops p1 p2 : -> Program .
op initialMem : -> Memory .
op initial : -> MachineState .

eq p1
= repeat
  'c1 := 1 ;
  while 'c2 = 1 do
    if 'turn = 2 then
      'c1 := 0 ;
      while 'turn = 2 do skip od ;
      'c1 := 1
    fi
  od ;
  crit ;
  'turn := 2 ;
  'c1 := 0 ;
  rem
  forever .

eq p2
= repeat
  'c2 := 1 ;
  while 'c1 = 1 do
    if 'turn = 1 then
      'c2 := 0 ;
      while 'turn = 1 do skip od ;
      'c2 := 1
    fi
  od ;
  crit ;
  'turn := 1 ;
  'c2 := 0 ;
  rem
  forever .

eq initialMem = ['c1, 0] ['c2, 0] ['turn, 1] .
eq initial = { [1, p1] | [2, p2], initialMem, 0 } .
endm

```

In this module the fragments of code for the critical section and for the remaining part of the program are respectively abstracted as constants `crit` and `rem`. We assume that `crit` is *terminating*, but no such assumption is made about `rem`. This is achieved by declaring subsorts (in module `SEQUENTIAL`) and constants,

```

subsorts LoopingUserStatement < UserStatement < Program .

```

```

op crit : -> UserStatement .
op rem  : -> LoopingUserStatement .

```

and by semantic rules where a `UserStatement` not in `LoopingUserStatement` always terminates, but a `LoopingUserStatement` may not terminate.

To specify relevant properties of Dekker's algorithm we define three state predicates parameterized by the process identifier: `enterCrit`, when the process enters its critical section; `in-rem`, when it is in its `rem` part; and `exec`, when the process has just executed.

```

mod DEKKER-CHECK is
  protecting DEKKER .
  including MODEL-CHECKER .
  including LTL-SIMPLIFIER .

  subsort MachineState < State .
  ops enterCrit in-rem exec : Pid -> Prop .
  var M : Memory .
  var R : Program .
  var S : Soup .
  vars I J : Pid .
  eq {[I, crit ; R] | S, M, J} |= enterCrit(I) = true .
  eq {[I, rem ; R] | S, M, J} |= in-rem(I) = true .
  eq {S, M, J} |= exec(J) = true .
endm

```

In the following reduction examples making use of the model checker, we will get a bit more of information about the number of states by means of the setting:

```
Maude> set verbose on .
```

We first verify that the *mutual exclusion property* is indeed satisfied.

```

Maude> reduce in DEKKER-CHECK :
  modelCheck(initial, []~ (enterCrit(1) /\ enterCrit(2))) .
ModelChecker: Property automaton has 2 states.
ModelCheckerSymbol: Examined 263 system states.
rewrites: 1156 in 60ms cpu (91ms real) (19266 rews/sec)
result Bool: true

```

However, the *strong liveness property* that executing infinitely often implies entering one's critical section infinitely often fails. The Maude LTL model checker returns a counterexample as follows, where we have only shown the first state in the counterexample.

```

Maude> reduce in DEKKER-CHECK :
  modelCheck(initial, []<> exec(1) -> []<> enterCrit(1)) .
ModelChecker: Property automaton has 3 states.
ModelCheckerSymbol: Examined 16 system states.

```

```

rewrites: 148 in 10ms cpu (7ms real) (14800 rews/sec)
result ModelCheckResult:
  counterexample({
    { [1, repeat 'c1 := 1 ; while 'c2 = 1 do if 'turn = 2
      then 'c1 := 0 ; while 'turn = 2 do skip od ;
        'c1 := 1 fi od ; crit ; 'turn := 2 ; 'c1 := 0 ; rem
        forever] |
      [2, repeat 'c2 := 1 ; while 'c1 = 1 do if 'turn = 1
        then 'c2 := 0 ; while 'turn = 1 do skip od ;
          'c2 := 1 fi od ; crit ; 'turn := 1 ; 'c2 := 0 ; rem
          forever],
        ['c1,0] ['c2,0] ['turn,1], 0 }, unlabeled }
    { ... })

```

Moreover, since `rem` may not terminate, the weaker liveness property that if *both* `p1` and `p2` execute infinitely often, then both enter their critical sections infinitely often also fails, as shown by the following counterexample returned by Maude. Again, we only show the first state for the counterexample that in full occupies three pages.

```

Maude> reduce in DEKKER-CHECK :
  modelCheck(initial, []<> exec(1) /\ []<> exec(2)
    -> []<> enterCrit(1) /\ []<> enterCrit(2)) .
ModelChecker: Property automaton has 7 states.
ModelCheckerSymbol: Examined 236 system states.
rewrites: 1463 in 70ms cpu (215ms real) (20900 rews/sec)
result ModelCheckResult:
  counterexample({
    { [1, repeat 'c1 := 1 ; while 'c2 = 1 do if 'turn = 2
      then 'c1 := 0 ; while 'turn = 2 do skip od ;
        'c1 := 1 fi od ; crit ; 'turn := 2 ; 'c1 := 0 ; rem
        forever] |
      [2, repeat 'c2 := 1 ; while 'c1 = 1 do if 'turn = 1
        then 'c2 := 0 ; while 'turn = 1 do skip od ;
          'c2 := 1 fi od ; crit ; 'turn := 1 ; 'c2 := 0 ; rem
          forever],
        ['c1,0] ['c2,0] ['turn,1], 0 }, unlabeled }
    { ... })

```

What *does* hold is the more subtle weak liveness property that if `p1` and `p2` both get to execute infinitely often, then if `p1` is infinitely often out of its `rem` section, then `p1` enters its critical section infinitely often. Of course, the symmetric statement holds true for `p2`.

```

Maude> reduce in DEKKER-CHECK :
  modelCheck(initial, []<> exec(1) /\ []<> exec(2)
    -> []<> ~ in-rem(1) -> []<> enterCrit(1)) .
ModelChecker: Property automaton has 5 states.
ModelCheckerSymbol: Examined 263 system states.

```

```
rewrites: 1661 in 80ms cpu (223ms real) (20762 rews/sec)
result Bool: true
```

The above Dekker algorithm example illustrates a general capability to model check in Maude *any* program (or abstraction of a program) having finitely many reachable states in *any* programming language: we just have to define in Maude the language’s rewriting semantics and the state predicates.

PARALLEL is a toy, yet nontrivial, language; but the same methodology works in fact very well for “real” languages such as Java and the JVM. By “very well” we mean that the program analysis tools, such as search for failures of invariants and model checking of LTL properties, that are obtained *for free* by defining the semantics of a real language in Maude compare favorably with state-of-the-art language-specific tools using standard benchmarks (see Section 23.2.5, and [156, 153]).

13.7 Crossing the river (revisited)

In Section 7.8 we showed how to solve in Maude the classic puzzle in which a shepherd takes a wolf, a goat and a cabbage across a river using a boat with only two seats. While crossing the river was modelled by means of rules, the facts that the goat and the wolf would eat, respectively, the cabbage and the goat if they were left alone by the shepherd were modelled with equations; we showed that those equations gave rise to coherence problems and explained how to solve them. Here we present an alternative solution in which such equations disappear altogether and a safe path is found with the help of the model checker.

The idea is simple: we only specify in Maude the transitions that correspond to crossing the river, and use atomic propositions in LTL to take care of unwanted situations. Thus, the system specification level is captured by the module

```
mod RIVER-CROSSING-2 is
  sorts Side Group .

  ops left right : -> Side [ctor] .
  op change : Side -> Side .

  --- shepherd, wolf, goat, cabbage
  ops s w g c : Side -> Group [ctor] .
  op _ : Group Group -> Group [ctor assoc comm] .
  op initial : -> Group .

  vars S S' : Side .

  eq change(left) = right .
  eq change(right) = left .
```

```

eq initial = s(left) w(left) g(left) c(left) .

rl [shepherd-alone] : s(S) => s(change(S)) .
rl [wolf] : s(S) w(S) => s(change(S)) w(change(S)) .
rl [goat] : s(S) g(S) => s(change(S)) g(change(S)) .
rl [cabbage] : s(S) c(S) => s(change(S)) c(change(S)) .
endm

```

and the property specification level by

```

mod RIVER-CROSSING-2-PROP is
  protecting RIVER-CROSSING-2 .
  including MODEL-CHECKER .

  subsort Group < State .
  ops disaster success : -> Prop .

  vars S S' S'' : Side .
  ceq (w(S) g(S) s(S') c(S'') |= disaster) = true if S /= S' .
  ceq (w(S'') g(S) s(S') c(S) |= disaster) = true if S /= S' .
  eq (s(right) w(right) g(right) c(right) |= success) = true .
endm

```

That is, a state satisfies the `disaster` condition if either the wolf or the goat can eat, whereas the `success` property only holds if everybody is on the right riverbank.

Note now that the model checker only returns paths that are counterexamples of properties. Then, to find a safe path we need to find a formula that somehow expresses the negation of the property we are interested in: a counterexample will then witness a safe path for the shepherd. For this problem, if no safe path exists then it should be true that whenever `success` is reached a disastrous state must have been traversed before:

$$\langle \rangle \text{ success} \rightarrow (\langle \rangle \text{ disaster} \wedge ((\sim \text{ success}) \cup \text{ disaster}))$$

We can use the model checker to ask for a counterexample to this formula, that will correspond to a safe path.

```

Maude> red modelCheck(initial,
  <> success -> (<> disaster /\ ((~ success) U disaster))) .
result ModelCheckResult: counterexample(
  {s(left) w(left) l(left) c(left), 'lamb}
  {s(right) w(left) l(right) c(left), 'shepherd}
  {s(left) w(left) l(right) c(left), 'wolf}
  {s(right) w(right) l(right) c(left), 'lamb}
  {s(left) w(right) l(left) c(left), 'cabbage}
  {s(right) w(right) l(left) c(right), 'shepherd}
  {s(left) w(right) l(left) c(right), 'lamb}
  {s(right) w(right) l(right) c(right), 'lamb}

```

```

{s(left) w(right) l(left) c(right), 'shepherd}
{s(right) w(right) l(left) c(right), 'wolf}
{s(left) w(left) l(left) c(right), 'lamb}
{s(right) w(left) l(right) c(right), 'cabbage}
{s(left) w(left) l(right) c(left), 'wolf},
{s(right) w(right) l(right) c(left), 'lamb}
{s(left) w(right) l(left) c(left), 'lamb}

```

The first eight states in this path are the same returned by the `search` command in Section 7.8; the remaining ones are there because the model checker always return a counterexample that consists of a path followed by a cycle.

13.8 Other model-checking examples

In Section 17.6 some properties of a Mobile Maude application are model checked. This example is interesting because two levels of reflection (see Chapter 14) are involved: the object level, at which Mobile Maude system code executes, and the metalevel, at which application code is executed.

The model checker can also be executed in Full Maude. This is illustrated with an example in Section 21.8. This example, though quite simple, is interesting in several ways. The use of parameterization is exploited at both the system and the property level. At the system level, it allows a succinct specification of a parametric system. At the property level, it makes possible the specification of the relevant properties for each value of the parameter, also in a very succinct way. This is quite useful, because the property formulas vary as the parameter changes, and the parametric description encompasses an infinite number of instance formulas.

Reflection, Metalevel Computation, and Strategies

Informally, a reflective logic is a logic in which important aspects of its metatheory can be represented at the object level in a consistent way, so that the object-level representation correctly simulates the relevant metatheoretic aspects. In other words, a reflective logic is a logic which can be faithfully interpreted in itself. Maude’s language design and implementation make systematic use of the fact that rewriting logic is reflective [76, 66, 77, 78]. This makes the metatheory of rewriting logic accessible to the user in a clear and principled way. However, since a naive implementation of reflection can be computationally expensive, a good implementation must provide efficient ways of performing reflective computations. This chapter explains how this is achieved in Maude through its predefined `META-LEVEL` module, that can be found in the `prelude.maude` file.

14.1 Reflection and metalevel computation

Rewriting logic is reflective in a precise mathematical way, namely, there is a finitely presented rewrite theory \mathcal{U} that is *universal* in the sense that we can represent in \mathcal{U} any finitely presented rewrite theory \mathcal{R} (including \mathcal{U} itself) as a term $\overline{\mathcal{R}}$, any terms t, t' in \mathcal{R} as terms $\overline{t}, \overline{t'}$, and any pair (\mathcal{R}, t) as a term $\langle \overline{\mathcal{R}}, \overline{t} \rangle$, in such a way that we have the following equivalence

$$\mathcal{R} \vdash t \longrightarrow t' \Leftrightarrow \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \longrightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle.$$

Since \mathcal{U} is representable in itself, we can achieve a “reflective tower” with an arbitrary number of levels of reflection:

$$\mathcal{R} \vdash t \rightarrow t' \Leftrightarrow \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \rightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle \Leftrightarrow \mathcal{U} \vdash \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \overline{t} \rangle} \rangle \rightarrow \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \overline{t'} \rangle} \rangle \dots$$

In this chain of equivalences we say that the first rewriting computation takes place at level 0, the second at level 1, and so on. In a naive implementation, each step up the reflective tower comes at considerable computational

cost, because simulating a single step of rewriting at one level involves many rewriting steps one level up. It is therefore important to have systematic ways of lowering the levels of reflective computations as much as possible, so that a rewriting subcomputation happens at a higher level in the tower only when this is strictly necessary.

In Maude, key functionality of the universal theory \mathcal{U} has been efficiently implemented in the functional module `META-LEVEL`. This module includes the modules `META-VIEW`, `META-MODULE`, and `META-TERM`. As an overview,

- in the module `META-TERM`, Maude terms are metarepresented as elements of a data type `Term` of terms;
- in the module `META-MODULE`, Maude modules are metarepresented as terms in a data type `Module` of modules;
- in the module `META-VIEW`, Maude views are metarepresented as terms in a data type `View` of views; and
- in the module `META-LEVEL`,
 - operations `upModule`, `upTerm`, `downTerm`, and others allow moving between reflection levels;
 - the process of reducing a term to canonical form using Maude’s `reduce` command is metarepresented by a built-in function `metaReduce`;
 - the processes of rewriting a term in a system module using Maude’s `rewrite` and `frewrite` commands are metarepresented by built-in functions `metaRewrite` and `metaFrewrite`;
 - the process of applying (without extension) a rule of a system module at the top of a term is metarepresented by a built-in function `metaApply`;
 - the process of applying (with extension) a rule of a system module at any position of a term is metarepresented by a built-in function `metaXApply`;
 - the process of matching (without extension) two terms at the top is reified by a built-in function `metaMatch`;
 - the process of matching (with extension) a pattern to any subterm of a term is reified by a built-in function `metaXmatch`;
 - the process of searching for a term satisfying some conditions starting in an initial term is reified by built-in functions `metaSearch` and `metaSearchPath`; and
 - parsing and pretty-printing of a term in a module, as well as key sort operations such as comparing sorts in the subsort ordering of a signature, are also metarepresented by corresponding built-in functions.

The functions `metaReduce`, `metaApply`, `metaXApply`, `metaRewrite`, `metaFrewrite`, `metaMatch`, and `metaXmatch` are called *descent functions*, since they allow us to descend levels in the reflective tower. The paper [70] provides a formal definition of the notion of *descent function*, and a detailed explanation of how they can be used to achieve a systematic, conservative way of lowering the levels of reflective computations.

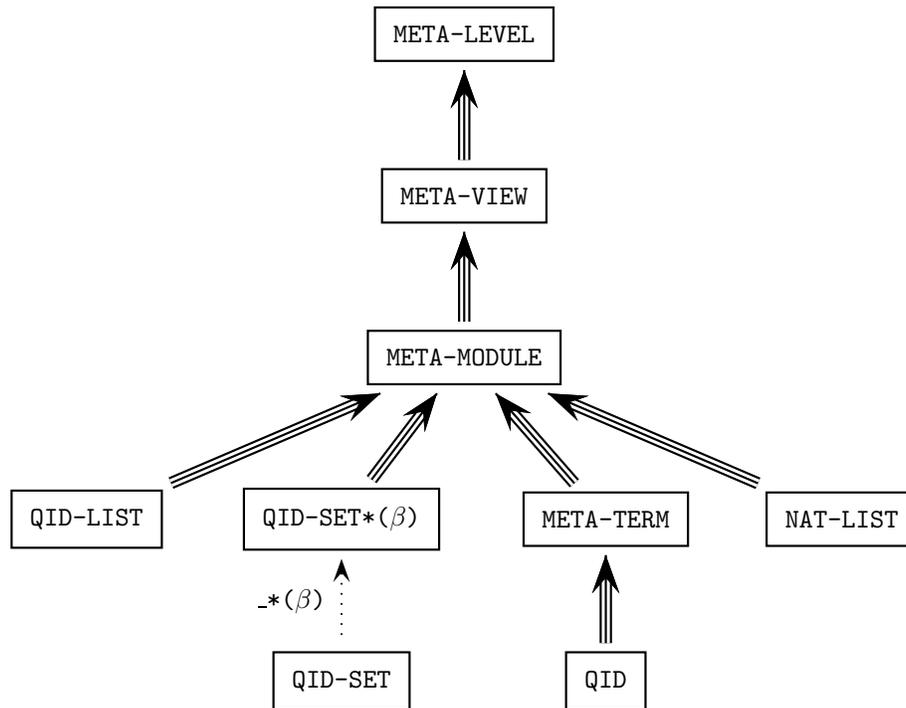


Fig. 14.1. Importation graph of metalevel modules

The importation graph in Figure 14.1 shows the relationships between all the modules in the metalevel. The modules NAT-LIST and QID-LIST provide lists of natural numbers and quoted identifiers, respectively (see Section 9.12.1), and the module QID-SET provides sets of quoted identifiers (see Section 9.12.2). Notice that QID-SET is imported (in protecting mode) with renaming

```
(op empty to none, op _,_ to _;- [prec 43])
```

abbreviated to β in the figure.

14.2 The META-TERM module

14.2.1 Metarepresenting sorts and kinds

In the META-TERM module, sorts and kinds are metarepresented as data in specific subsorts of the sort Qid of quoted identifiers.

A term of sort `Sort` is any quoted identifier not containing the following characters: `‘:’`, `‘.’`, `‘[’`, and `‘]’`. Moreover, the characters `‘{’`, `‘}’`, and `‘,’` can only appear in structured sort names (see Section 3.3). For example, `’Bool`, `’NzNat`, `a’{X’}`, `a’{X’,Y’}`, `a’{b’,c’{d’}’}’{e’}`, and `a’{(‘}` are terms of sort `Sort`.

An element of sort `Kind` is a quoted identifier of the form `’’[SortList’]` where `SortList` is a single identifier formed by a list of unquoted elements of sort `Sort` separated by backquoted commas. For example, `’’[Bool’]` and `’’[NzNat’,Zero’,Nat’]` are valid elements of the sort `Kind`. Note the use of backquotes to force them to be single identifiers.

Since commas and square brackets are used to metarepresent kinds, these characters are forbidden in sort names, in order to avoid undesirable ambiguities. Periods and colons are also forbidden, due to the metarepresentation of constants and variables, as explained in the next section.

Since operator declarations can use both sorts and kinds, we denote by `Type` the union of `Sort` and `Kind`.

```
sorts Sort Kind Type .
subsorts Sort Kind < Type < Qid.
op <Qids> : -> Sort [special (...)] .
op <Qids> : -> Kind [special (...)] .
```

Remember from the introduction of Chapter 9 that `<Qids>` is a special operator declaration used to represent sets of constants that are not algebraically constructed, but are instead associated with appropriate C++ code by “hooks” which are specified following the `special` attribute; see the functional module `META-TERM` in file `prelude.mauve` for the details omitted here.

14.2.2 Metarepresenting terms

In the module `META-TERM`, terms are metarepresented as elements of the data type `Term` of terms. The base cases in the metarepresentation of terms are given by subsorts `Constant` and `Variable` of the sort `Qid`.

```
sorts Constant Variable Term .
subsorts Constant Variable < Qid Term .
op <Qids> : -> Constant [special (...)] .
op <Qids> : -> Variable [special (...)] .
```

Constants are quoted identifiers that contain the constant’s name and its type separated by a `‘.’`, e.g., `’0.Nat`. Similarly, variables contain their name and type separated by a `‘:’`, e.g., `’N:Nat`. Appropriate selectors then extract their names and types.

```
op getName : Constant -> Qid .
op getName : Variable -> Qid .
op getType : Constant -> Type .
op getType : Variable -> Type .
```


1. terms in equations, membership axioms, and rules are now metarepresented as we have already explained in Section 14.2.2;
2. in the metarepresentation of modules and theories we follow a fixed order in introducing the different kinds of declarations for sorts, subsort relations, equations, etc., whereas in the user syntax there is considerable flexibility for introducing such different declarations in an interleaved and piecemeal way;
3. there is no need for variable declarations—in fact, there is no syntax for metarepresenting them—and
4. sets of identifiers—used in declarations of sorts—are metarepresented as sets of quoted identifiers built with an associative and commutative operator `_;`.

The syntax for the top-level operators metarepresenting functional and system modules and functional and system theories (just modules in general) is as follows, where `Header` means just an identifier in the case of non-parameterized modules or an identifier together with a list of parameter declarations in the case of a parameterized module.

```

sorts FModule SModule FTheory STheory Module .
subsorts FModule < SModule < Module .
subsorts FTheory < STheory < Module .
sort Header .
subsort Qid < Header .
op {_} : Qid ParameterDeclList -> Header [ctor] .
op fmod_is_sorts_.....endfm : Header ImportList SortSet
  SubsortDeclSet OpDeclSet MembAxSet EquationSet -> FModule
  [ctor gather (& & & & & &)] .
op mod_is_sorts_.....endm : Header ImportList SortSet
  SubsortDeclSet OpDeclSet MembAxSet EquationSet RuleSet
  -> SModule [ctor gather (& & & & & &)] .
op fth_is_sorts_.....endfth : Qid ImportList SortSet SubsortDeclSet
  OpDeclSet MembAxSet EquationSet -> FTheory
  [ctor gather (& & & & & &)] .
op th_is_sorts_.....endth : Qid ImportList SortSet SubsortDeclSet
  OpDeclSet MembAxSet EquationSet RuleSet -> STheory
  [ctor gather (& & & & & &)] .

```

Appropriate selectors then extract from the metarepresentation of modules the metarepresentations of their names, imported submodules, and declared sorts, subsorts, operators, memberships, equations, and rules.

```

op getName : Module -> Qid .
op getImports : Module -> ImportList .
op getSorts : Module -> SortSet .
op getSubsorts : Module -> SubsortDeclSet .
op getOps : Module -> OpDeclSet .
op getMbs : Module -> MembAxSet .
op getEqs : Module -> EquationSet .

```

```
op getRls : Module -> RuleSet .
```

Without going into all the syntactic details, we show only the operators used to metarepresent sets of sorts and kinds, conditions, equations, and rules. The complete syntax used for metarepresenting modules can be found in the module META-MODULE in the file `prelude.maude`.

```
sorts EmptyTypeSet NeSortSet NeKindSet
      NeTypeSet SortSet KindSet TypeSet .
subsort EmptyTypeSet < SortSet KindSet < TypeSet < QidSet .
subsort Sort < NeSortSet < SortSet .
subsort Kind < NeKindSet < KindSet .
subsort Type NeSortSet NeKindSet < NeTypeSet < TypeSet NeQidSet .
op none : -> EmptyTypeSet [ctor] .
op ;_ : TypeSet TypeSet -> TypeSet
      [ctor assoc comm id: none prec 43] .
op ;_ : SortSet SortSet -> SortSet [ctor ditto] .
op ;_ : KindSet KindSet -> KindSet [ctor ditto] .

sorts EqCondition Condition .
subsort EqCondition < Condition .
op nil : -> EqCondition [ctor] .
op == : Term Term -> EqCondition [ctor prec 71] .
op := : Term Sort -> EqCondition [ctor prec 71] .
op :=_ : Term Term -> EqCondition [ctor prec 71] .
op => : Term Term -> Condition [ctor prec 71] .
op /\_ : EqCondition EqCondition -> EqCondition
      [ctor assoc id: nil prec 73] .
op /\_ : Condition Condition -> Condition
      [ctor assoc id: nil prec 73] .

sorts Equation EquationSet .
subsort Equation < EquationSet .
op eq=_[_] : Term Term AttrSet -> Equation [ctor] .
op ceq=_if[_] : Term Term EqCondition AttrSet -> Equation
      [ctor] .
op none : -> EquationSet [ctor] .
op __ : EquationSet EquationSet -> EquationSet
      [ctor assoc comm id: none] .

sorts Rule RuleSet .
subsort Rule < RuleSet .
op rl=>[_] : Term Term AttrSet -> Rule [ctor] .
op crl=>_if[_] : Term Term Condition AttrSet -> Rule [ctor] .
op none : -> RuleSet [ctor] .
op __ : RuleSet RuleSet -> RuleSet [ctor assoc comm id: none] .
```

For example, we show here the metarepresentations of the modules introduced in Section 6.1 VENDING-MACHINE-SIGNATURE and VENDING-MACHINE.

```

fmod 'VENDING-MACHINE-SIGNATURE is
  nil
  sorts 'Coin ; 'Item ; 'Marking .
  subsort 'Coin < 'Marking .
  subsort 'Item < 'Marking .
  op '___ : 'Marking 'Marking -> 'Marking
    [assoc comm id('null.Marking)] .
  op 'a : nil -> 'Item [format('b! 'o)] .
  op 'null : nil -> 'Marking [none] .
  op '$ : nil -> 'Coin [format('r! 'o)] .
  op 'q : nil -> 'Coin [format('r! 'o)] .
  op 'c : nil -> 'Item [format('b! 'o)] .
  none
  none
endfm

mod 'VENDING-MACHINE is
  including 'VENDING-MACHINE-SIGNATURE .
  sorts none .
  none
  none
  none
  none
  rl 'M:Marking => '___['M:Marking, 'q.Coin] [label('add-q)] .
  rl 'M:Marking => '___['M:Marking, '$.Coin] [label('add-)] .
  rl '$.Coin => 'c.Item [label('buy-c)] .
  rl '$.Coin => '___['a.Item, 'q.Coin] [label('buy-a)] .
  rl '___['q.Coin, '___['q.Coin, '___['q.Coin, 'q.Coin]]
    => '$.Coin [label('change)] .
endm

```

Since `VENDING-MACHINE-SIGNATURE` has no list of imported submodules, no membership axioms, and no equations, those fields are filled, respectively, with the constants `nil` of sort `ImportList`, `none` of sort `MembAxSet`, and `none` of sort `EquationSet`. Similarly, since the module `VENDING-MACHINE` has no subsort declarations and no operator declarations, those fields are filled, respectively, with the constants `none` of sort `SubsortDeclSet` and `none` of sort `OpDeclSet`. Variable declarations are not metarepresented, but rather each variable is metarepresented in its “on the fly”-declaration form, i.e., with its sort or kind.

As mentioned above, parameterized modules are also metarepresented through the notion of a *header*, which is either an identifier (for non-parameterized modules) or an identifier together with a list of parameter declarations (for parameterized modules). Such parameter declarations are metarepresented again with a syntax similar to the user syntax.

```

sorts ParameterDecl NeParameterDeclList ParameterDeclList .
subsorts ParameterDecl < NeParameterDeclList < ParameterDeclList .
op _::_ : Sort ModuleExpression -> ParameterDecl .

```

```

op nil : -> ParameterDeclList [ctor] .
op __ : ParameterDeclList ParameterDeclList -> ParameterDeclList
      [ctor assoc id: nil prec 121] .

```

Module expressions involving renamings and summations can also be metarepresented with the expected syntax:

```

sort ModuleExpression .
subsort Qid < ModuleExpression .
op +_ : ModuleExpression ModuleExpression -> ModuleExpression
      [ctor assoc comm] .
op *_(_) : ModuleExpression RenamingSet -> ModuleExpression
          [ctor prec 39 format (d d s n++i n--i d)] .

sorts Renaming RenamingSet .
subsort Renaming < RenamingSet .
op sort_to_ : Qid Qid -> Renaming [ctor] .
op op_to_[_] : Qid Qid AttrSet -> Renaming
              [ctor format (d d d d s d d d)] .
op op_->_to_[_] : Qid TypeList Type Qid AttrSet -> Renaming
                 [ctor format (d d d d d d d s d d d)] .
op label_to_ : Qid Qid -> Renaming [ctor] .
op __ : RenamingSet RenamingSet -> RenamingSet
      [ctor assoc comm prec 43 format (d d ni d)] .

```

Finally, the instantiation of a parameterized module is metarepresented as follows:

```

op _{_] : ModuleExpression ParameterList -> ModuleExpression
        [ctor prec 37].

sort EmptyCommaList NeParameterList ParameterList .
subsorts Sort < NeParameterList < ParameterList .
subsort EmptyCommaList < GroundTermList ParameterList .
op empty : -> EmptyCommaList [ctor] .
op __ : ParameterList ParameterList -> ParameterList [ctor ditto] .

```

The rules for constructing parameterized metamodules and instantiating parameterized modules existing in the database reflect the object-level rules. In particular, bound parameters are permitted; for example, the following term metarepresents a parameterized module:

```

fmod 'PARMODEX{'X :: 'TRIV} is
  including 'MAP{'String, 'X} .
  sorts 'Foo .
  none
  none
  none
  none
endfm

```

Although, as we will see in the following section, views can be metarepresented as terms of the `View` sort, it is not possible to use the views constructed at the metalevel in module expressions. The views used in the module expressions occurring in metamodules must have been declared at the object level, so that they are present in the database of modules and views declared in the given session. Such views are written in quoted form within metamodule expressions, like `'String` in `'MAP{'String, 'X}` in the example above.

Note that terms of sort `Module` can be metarepresented again, yielding then a term of sort `Term`, and this can be iterated an arbitrary number of times. This is in fact necessary when a metalevel computation has to operate at higher levels.

14.4 The META-VIEW module: Metarepresenting views

In the module `META-VIEW`, which imports `META-MODULE`, views are metarepresented in a syntax very similar to their original user syntax.

```
sort View .
op view_from_to_is_endv : Header ModuleExpression ModuleExpression
  SortMappingSet OpMappingSet -> View [ctor gather (& & & & &)
  format (d d d d d d n++i ni n--i d)] .
```

The first argument corresponds to the name of the view, while the second and third are module expressions corresponding to the source (usually a theory) and target (usually a module) of the view, respectively. The fourth and fifth arguments are the sort mappings and the operator mappings defining the view.

The following syntax defines sets of sort mappings in a way completely similar to the user syntax.

```
sorts SortMapping SortMappingSet .
subsort SortMapping < SortMappingSet .
op sort_to_ : Sort Sort -> SortMapping [ctor] .
op none : -> SortMappingSet [ctor] .
op _ : SortMappingSet SortMappingSet -> SortMappingSet
  [ctor assoc comm id: none format (d ni d)] .
eq S:SortMapping S:SortMapping = S:SortMapping .
```

Analogously, the following syntax is used to define set of operator mappings.

```
sorts OpMapping OpMappingSet .
subsort OpMapping < OpMappingSet .
op (op_to_) : Qid Qid -> OpMapping [ctor] .
op (op:_->to_) : Qid TypeList Type Qid -> OpMapping [ctor] .
op (op_to_term_) : Term Term -> OpMapping [ctor] .
op none : -> OpMappingSet [ctor] .
```

```

op __ : OpMappingSet OpMappingSet -> OpMappingSet
  [ctor assoc comm id: none format (d ni d)] .
eq 0:OpMapping 0:OpMapping = 0:OpMapping .

```

Finally, appropriate selectors are used to extract from the metarepresentation of a view the corresponding components, namely, the metarepresentations of its name, of its source, of its target, of its set of sort mappings, and of its set of operator mappings.

```

op getName : View -> Qid .
op getFrom : View -> ModuleExpression .
op getTo : View -> ModuleExpression .
op getSortMappings : View -> SortMappingSet .
op getOpMappings : View -> OpMappingSet .

```

For example, the metarepresentation of the view `RingToRat` (see Section 8.3.2) from the theory `RING` to the predefined `RAT` module is as follows:

```

view 'RingToRat from 'RING to 'RAT is
  sort 'Ring to 'Rat .
  op 'z to '0 .
  op 'e.Ring to term 's_['0.Zero] .
endv

```

Then, we can extract some components of this metarepresented view:

```

Maude> reduce in META-VIEW :
  getFrom(view 'RingToRat from 'RING to 'RAT is
    sort 'Ring to 'Rat .
    op 'z to '0 .
    op 'e.Ring to term 's_['0.Zero] .
  endv) .
result Sort: 'RING

Maude> reduce in META-VIEW :
  getOpMappings(view 'RingToRat from 'RING to 'RAT is
    sort 'Ring to 'Rat .
    op 'z to '0 .
    op 'e.Ring to term 's_['0.Zero] .
  endv) .
result OpMappingSet:
  op 'z to '0 .
  op 'e.Ring to term 's_['0.Zero] .

```

14.5 The META-LEVEL module: Metalevel operations

The `META-LEVEL` module, which imports `META-VIEW`, has several built-in descent functions that provide useful and efficient ways of reducing metalevel

computations to object-level ones, as well as several useful operations on sorts and kinds. Since, in general, these operations take among their arguments the metarepresentations of modules, sorts, kinds, terms, and so on, the `META-LEVEL` module also provides several built-in functions for moving conveniently between reflection levels. Notice that most of the operations in the module `META-LEVEL` are partial (as explicitly stated by using the arrow `~>` in the corresponding operator declaration). This is due to the fact that they do not make sense on terms that, although may be of the correct sort, for example, `Module` or `Term`, either are not correct metarepresentations of modules or are not correct metarepresentations of terms in the module provided as another argument.

Concerning partial operations, the criteria used to choose between using a supersort for the result and having an operator map to a kind is as follows.

If the error return value is built from constructors, say

```
op noParse : Nat -> ResultPair? [ctor] .
op ambiguity : ResultPair ResultPair -> ResultPair? [ctor] .
```

it goes to a supersort. In some sense these are not errors, but merely exceptions or out-of-band results for which there is a carefully defined semantics.

The kind is reserved for nonconstructors which may not be able to reduce at all on illegal arguments, like, for example, in the function (notice the form of the arrow)

```
op metaParse : Module QidList Type? ~> ResultPair? [special (...)] .
```

In this second case, an expression that does not evaluate to the appropriate sort represents a real error.

So, for example, a call to `metaParse` with an ill-formed module would produce an unreduced term `metaParse(...)` in the kind, whereas a call to `metaParse` with valid arguments but a list of tokens that could not be parsed to a term of the desired type in the metamodule would produce a term `noParse(...)` of sort `ResultPair?` indicating where the parse first failed.

14.5.1 Moving between reflection levels: `upModule`, `upTerm`, `downTerm`, and others

For a module \mathcal{R} that *has already been loaded* into Maude, the operations `upSorts`, `upSubsortDecl`, `upOpDecls`, `upMbs`, `upEqs`, `upRls`, and `upModule` take as arguments the metarepresentation of the name of \mathcal{R} and a Boolean value b , and return, respectively, the metarepresentations of the module \mathcal{R} , of its sorts, subsort declarations, operator declarations, membership axioms, equations, and rules. If the second argument of these functions is `true`, then the resulting metarepresentations will include the corresponding statements that \mathcal{R} imports from its submodules; but if the second argument is `false`, the resulting metarepresentations will only contain the metarepresentations of the statements explicitly declared in \mathcal{R} .

```

op upModule : Qid Bool ~> Module [special (...)] .
op upSorts : Qid Bool ~> SortSet [special (...)] .
op upSubsortDecls : Qid Bool ~> SubsortDeclSet [special (...)] .
op upOpDecls : Qid Bool ~> OpDeclSet [special (...)] .
op upMbs : Qid Bool ~> MembAxSet [special (...)] .
op upEqs : Qid Bool ~> EquationSet [special (...)] .
op upRls : Qid Bool ~> RuleSet [special (...)] .

```

We give below simple examples of using these functions. Note that, since `BOOL` is automatically imported by all modules, its equations are shown when `upEqs` is called with `true` as its second argument. For the same reason, the metarepresentation of the `VENDING-MACHINE-SIGNATURE` module includes an `including` declaration that was not explicit in that module. Here, and in the rest of this section, we assume that the modules `NUMBERS` and `SIEVE` from Chapter 4, as well as the modules `VENDING-MACHINE-SIGNATURE` and `VENDING-MACHINE` from Chapter 6, have already been loaded into Maude.

```

Maude> reduce in META-LEVEL :
      upModule('VENDING-MACHINE-SIGNATURE, false) .
result FModule:
  fmod 'VENDING-MACHINE-SIGNATURE is
    including 'BOOL .
    sorts 'Coin ; 'Item ; 'Marking .
    subsort 'Coin < 'Marking .
    subsort 'Item < 'Marking .
    op '$ : nil -> 'Coin [format('r! 'o)] .
    op '__ : 'Marking 'Marking -> 'Marking
      [assoc comm id('null.Marking)] .
    op 'a : nil -> 'Item [format('b! 'o)] .
    op 'c : nil -> 'Item [format('b! 'o)] .
    op 'null : nil -> 'Marking [none] .
    op 'q : nil -> 'Coin [format('r! 'o)] .
    none
    none
  endfm

Maude> reduce in META-LEVEL : upEqs('VENDING-MACHINE, true) .
result EquationSet:
  eq '_and_['true.Bool, 'A:Bool] = 'A:Bool [none] .
  eq '_and_['A:Bool, 'A:Bool] = 'A:Bool [none] .
  eq '_and_['A:Bool, '_xor_['B:Bool, 'C:Bool]]
    = '_xor_['_and_['A:Bool, 'B:Bool], '_and_['A:Bool, 'C:Bool]]
    [none] .
  eq '_and_['false.Bool, 'A:Bool] = 'false.Bool [none] .
  eq '_or_['A:Bool, 'B:Bool]
    = '_xor_['_and_['A:Bool, 'B:Bool], '_xor_['A:Bool, 'B:Bool]]
    [none] .
  eq '_xor_['A:Bool, 'A:Bool] = 'false.Bool [none] .
  eq '_xor_['false.Bool, 'A:Bool] = 'A:Bool [none] .

```

```

eq 'not_['A:Bool] = '_xor_['true.Bool, 'A:Bool] [none] .
eq '_implies_['A:Bool, 'B:Bool]
  = 'not_['_xor_['A:Bool, '_and_['A:Bool, 'B:Bool]]] [none] .

Maude> reduce in META-LEVEL : upEqs('VENDING-MACHINE, false) .
result EquationSet: (none).EquationSet

Maude> reduce in META-LEVEL : upRls('VENDING-MACHINE, true) .
result RuleSet:
  r1 '$.Coin => 'c.Item [label('buy-c)] .
  r1 '$.Coin => '[_['q.Coin, 'a.Item] [label('buy-a)] .
  r1 'M:Marking => '[_['$.Coin, 'M:Marking] [label('add-)] .
  r1 'M:Marking => '[_['q.Coin, 'M:Marking] [label('add-q)] .
  r1 '[_['q.Coin, 'q.Coin, 'q.Coin, 'q.Coin] => '$.Coin
    [label('change)] .

```

In addition to the `upModule` operator, there is another operator allowing the use of an already loaded module at the metalevel. This operator is defined in the module `META-MODULE` as follows:

```

op [_] : Qid -> Module .
eq [Q:Qid] = (th Q:Qid is including Q:Qid .
             sorts none . none none none none endth) .

```

This operator is just syntactic sugar for accessing the corresponding module. Notice that the module is not moved up to the metalevel as `upModule` does, it is just a way of referring to it, and therefore more efficient.

The `META-LEVEL` module also provides a function `upImports` that takes as argument the metarepresentation of the name of a module \mathcal{R} . When \mathcal{R} is already in the Maude module database, then `upImports` returns the metarepresentation of its list of imported submodules. The function `upImports` does not take a Boolean argument, as the previous `up`-functions, since it is not useful to ask for the list of imported submodules of a flattened module.

```

op upImports : Qid ~> ImportList [special (...)] .

```

In the same way, the `META-LEVEL` module provides a function `upView` that takes as argument the metarepresentation of the name of a view; when such a view is in the Maude view database, then `upView` returns the corresponding metarepresentation.

```

op upView : Qid ~> View [special (...)] .

```

As a simple example, let us consider the view `String0` from the predefined theory `DEFAULT` to the predefined module `STRING`, all of them provided in `prelude.maude`; then,

```

Maude> reduce in META-LEVEL : upView('String0) .
result View:
  view 'String0 from 'DEFAULT to 'STRING is

```

```

    sort 'Elt to 'String .
    op '0.Elt to term '"".String .
endv

```

Finally, the META-LEVEL module introduces two polymorphic functions. The function `upTerm` takes a term t and returns the metarepresentation of its canonical form. The function `downTerm` takes the metarepresentation of a term t as its first argument and a term t' as its second argument, and returns the canonical form of t , if t is a term in the same kind as t' ; otherwise, it returns the canonical form of t' .

```

op upTerm : Universal -> Term [poly (1) special (...)] .
op downTerm : Term Universal -> Universal
    [poly (2 0) special (...)] .

```

As simple examples, we can use the function `upTerm` to obtain the metarepresentation of the term `f(a, f(b, c))` in the module UP-DOWN-TEST below, and the function `downTerm` to recover the term `f(a, f(b, c))` from its metarepresentation.

```

fmod UP-DOWN-TEST is
  protecting META-LEVEL .
  sort Foo .
  ops a b c d : -> Foo .
  op f : Foo Foo -> Foo .
  op error : -> [Foo] .
  eq c = d .
endfm

Maude> reduce in UP-DOWN-TEST : upTerm(f(a, f(b, c))) .
result GroundTerm: 'f['a.Foo,'f['b.Foo,'d.Foo]]

```

Notice in the previous example that the given argument has been reduced before obtaining its metarepresentation, more specifically, the subterm `c` has become `d`. In the following examples we can observe the same behavior with respect to `downTerm`.

```

Maude> reduce in UP-DOWN-TEST :
    downTerm('f['a.Foo,'f['b.Foo,'c.Foo]], error) .
result Foo: f(a, f(b, d))

Maude> reduce in UP-DOWN-TEST :
    downTerm(upTerm(f(a, f(b, c))), error) .
result Foo: f(a, f(b, d))

```

In our last example, we show the result of `downTerm` when its first argument does not correspond to the metarepresentation of a term in the module UP-DOWN-TEST; notice the constant `e` in the metarepresented term that does not correspond to a declared constant in the module.

```
Maude> reduce in UP-DOWN-TEST :
      downTerm('f['a.Foo,'f['b.Foo,'e.Foo]], error) .
Advisory: could not find a constant e of
      sort Foo in meta-module UP-DOWN-TEST.
result [Foo]: error
```

Due to the failure in moving down the metarepresented term given as first argument, the result is the term given as second argument, namely, `error`, which was declared in the module `UP-DOWN-TEST` as a constant of kind `[Foo]`.

14.5.2 Simplifying: `metaReduce` and `metaNormalize`

`metaReduce`

The (partial) operation `metaReduce` takes as arguments the metarepresentation of a module \mathcal{R} and the metarepresentation of a term t .

```
sort ResultPair .
op {_,_} : Term Type -> ResultPair [ctor] .
op metaReduce : Module Term ~> ResultPair [special (...)] .
```

When t is a term in \mathcal{R} , `metaReduce($\overline{\mathcal{R}}$, \bar{t})` returns the metarepresentation of the canonical form of t , using the equations in \mathcal{R} , together with the metarepresentation of its corresponding sort or kind. The reduction strategy used by `metaReduce` coincides with that of the `reduce` command (see Sections 4.9 and 25.2).

As said above, in general, when either the first argument of `metaReduce` is a term of sort `Module` but not a correct metarepresentation $\overline{\mathcal{R}}$ of an object module \mathcal{R} , or the second argument is not the correct metarepresentation \bar{t} of a term t in \mathcal{R} , the operation `metaReduce` is undefined, that is, the term `metaReduce(u,v)` does not reduce and it does not get evaluated to a term of sort `ResultPair`, but only to an expression in the kind `[ResultPair]`.

Appropriate selectors extract from the result pairs their two components:

```
op getTerm : ResultPair -> Term .
op getType : ResultPair -> Type .
```

Using `metaReduce` we can simulate at the metalevel the primes computation example at the end of Section 4.4.7.

```
Maude> reduce in META-LEVEL :
      metaReduce(upModule('SIEVE, false),
        'show_upto_['primes.NatList, 's^10['0.Zero]]) .
result ResultPair:
  {'_._['s^2['0.Zero], 's^3['0.Zero], 's^5['0.Zero],
    's^7['0.Zero], 's^11['0.Zero], 's^13['0.Zero],
    's^17['0.Zero], 's^19['0.Zero], 's^23['0.Zero],
    's^29['0.Zero]],
  'NatList}
```

We can also insert a new element into an empty map of the type declared in the module PARMODEX at the end of Section 14.3 as follows:

```
Maude> red in META-LEVEL :
  metaReduce(
    fmod 'PARMODEX{'X :: 'TRIV} is
      including 'MAP{'String, 'X} .
      sorts 'Foo .
      none
      none
      none
      none
    endfm,
    'insert['"foo".String, 'A:X$Elt,
            'empty.Map{'String',X'}]) .
result ResultPair:
  {'_|->_'["foo".String, 'A:X$Elt], 'Entry{'String',X'}}
```

Notice that the module expression 'MAP{'String, 'X}' has a bound parameter X, which appears also in the sort X\$Elt in the on-the-fly declaration of the variable A:X\$Elt.

`metaNormalize`

The (partial) operation `metaNormalize` takes as arguments the metarepresentation of a module \mathcal{R} and the metarepresentation of a term t .

```
op metaNormalize : Module Term ~> ResultPair [special (...)] .
```

When t is a term in \mathcal{R} , `metaNormalize($\overline{\mathcal{R}}$, \overline{t})` returns the metarepresentation of the normal form of t with respect to the equational theory consisting of the equational attributes of the operators in t , without doing any simplification or rewriting with respect to equations or rules in \mathcal{R} , together with the metarepresentation of its corresponding sort or kind. For example, from the declarations in the predefined NAT module

```
op s_ : Nat -> NzNat [ctor iter special (...)] .
op _+_ : NzNat Nat -> NzNat [assoc comm prec 33 special (...)] .
op _+_ : Nat Nat -> Nat [ditto] .
```

we know that the successor operator satisfies the `iter` theory (see Section 4.4.2) and that the addition operator is associative and commutative (see Section 4.4.1). With this information it is easy to make sense of the following results:

```
Maude> red in META-LEVEL :
  metaNormalize(upModule('NAT, false), 's_['s_['0.Zero]]) .
result ResultPair: {'s_~2['0.Zero], 'NzNat}

Maude> red in META-LEVEL :
```

```

    metaNormalize(upModule('NAT, false),
      '[_+_['s_['s_['0.Zero]], '0.Zero]] .
result ResultPair: {'[_+_['0.Zero, 's_^2['0.Zero]], 'NzNat}

Maude> red in META-LEVEL :
    metaNormalize(upModule('NAT, false),
      '[_+_['0.Zero, '[_+_['s_['s_['0.Zero]], '0.Zero]]) .
result ResultPair: {'[_+_['0.Zero, '0.Zero, 's_^2['0.Zero]], 'NzNat}

```

Notice that associative terms are flattened and, if they are also commutative, the subterms are sorted with respect to an internal order. Notice also that in the last two examples the subterm `'0.Zero` does not disappear. This is because 0 is not declared as an identity element for `[_+_`.

14.5.3 Rewriting: `metaRewrite` and `metaFrewrite`

`metaRewrite`

The (partial) operation `metaRewrite` takes as arguments the metarepresentation of a module \mathcal{R} , the metarepresentation of a term t , and a value b of the sort `Bound`, i.e., either a natural number or the constant `unbounded`.

```

sort Bound .
subsort Nat < Bound .
op unbounded :-> Bound [ctor] .
op metaRewrite : Module Term Bound ~> ResultPair [special (...)] .

```

The operation `metaRewrite` is entirely analogous to `metaReduce`, but instead of using only the equational part of a module it now uses both the equations and the rules to rewrite the term. The reduction strategy used by `metaRewrite` coincides with that of the `rewrite` command (see Sections 6.4 and 25.2). That is, the result of `metaRewrite(\mathcal{R} , \bar{t} , b)` is the metarepresentation of the term obtained from t after at most b applications of the rules in \mathcal{R} using the `rewrite` strategy, together with the metarepresentation of its corresponding sort or kind. When the value `unbounded` is given as the third argument, no bound is imposed to the number of rewrites, and rewriting proceeds to the bitter end.

Using `metaRewrite` we can redo at the metalevel the examples in Section 6.4.

```

Maude> reduce in META-LEVEL :
    metaRewrite(upModule('VENDING-MACHINE, false),
      '[_[_['$.Coin, '[_[_['$.Coin, '[_[_['q.Coin, 'q.Coin]]], 1) .
result ResultPair:
  {'[_[_['$.Coin, '$.Coin, 'q.Coin, 'q.Coin, 'q.Coin], 'Marking}

Maude> reduce in META-LEVEL :
    metaRewrite(upModule('VENDING-MACHINE, false),

```

```

      '___['$.Coin, '___['$.Coin, '___['q.Coin, 'q.Coin]]], 2) .
result ResultPair:
  {'___['$.Coin, '$.Coin, '$.Coin, 'q.Coin, 'q.Coin, 'q.Coin],
   'Marking}

```

`metaFrewrite`

Position fair rewriting, which was described in Section 6.4, is metarepresented by the operation `metaFrewrite`. This (partial) operation takes as arguments the metarepresentation of a module, the metarepresentation of a term, a value of sort `Bound`, and a natural number.

```

op metaFrewrite : Module Term Bound Nat ~> ResultPair
[special (...)] .

```

The reduction strategy used by `metaFrewrite` coincides with that of the `frewrite` command in Maude, except that a final (semantic) sort calculation is performed at the end in order to produce a correct `ResultPair`. That is, `frewrite($\overline{\mathcal{R}}$, \bar{t} , b , n)` results in the metarepresentation of the term obtained from t after at most b applications of the rules in \mathcal{R} using the `frewrite` strategy, with at most n rewrites at each entitled position on each traversal of a subject term, together with the metarepresentation of its corresponding sort or kind. When the value `unbounded` is given as the third argument, no bound is imposed to the number of rewrites.

Using `metaFrewrite` we can redo at the metalevel the examples in Section 6.4.

```

Maude> reduce in META-LEVEL :
  metaFrewrite(upModule('VENDING-MACHINE, false),
    '___['$.Coin, '___['$.Coin, '___['q.Coin, 'q.Coin]]],
    1, 1) .
result ResultPair:
  {'___['$.Coin, 'q.Coin, 'q.Coin, 'c.Item], 'Marking}

Maude> reduce in META-LEVEL :
  metaFrewrite(upModule('VENDING-MACHINE, false),
    '___['$.Coin, '___['$.Coin, '___['q.Coin, 'q.Coin]]],
    12, 1) .
result ResultPair:
  {'___['$.Coin, '$.Coin, '$.Coin, '$.Coin, 'q.Coin, 'q.Coin,
    'q.Coin, 'q.Coin, 'q.Coin, 'q.Coin, 'q.Coin, 'q.Coin,
    'q.Coin, 'a.Item, 'c.Item],
   'Marking}

```

14.5.4 Applying rules: `metaApply` and `metaXapply`

`metaApply`

The (partial) operation `metaApply` takes as arguments the metarepresentation of a module, the metarepresentation of a term, the metarepresentation of a

rule label, the metarepresentation of a set of assignments (possibly empty) defining a partial substitution, and a natural number.

```

sorts Assignment Substitution .
subsort Assignment < Substitution .
op _<_ : Variable Term -> Assignment [ctor prec 63] .
op none : -> Substitution [ctor] .
op _;_ : Substitution Substitution -> Substitution
      [assoc comm id: none prec 65] .

sort ResultTriple ResultTriple? .
subsort ResultTriple < ResultTriple? .
op {_,_,_} : Term Type Substitution -> ResultTriple [ctor] .
op failure : -> ResultTriple? [ctor] .
op metaApply : Module Term Qid Substitution Nat ~> ResultTriple?
      [special (...)] .

```

The operation $\text{metaApply}(\overline{\mathcal{R}}, \bar{t}, \bar{l}, \sigma, n)$ is evaluated as follows:

1. the term t is first fully reduced using the equations in \mathcal{R} ;
2. the resulting term is matched at the top against all rules with label l in \mathcal{R} partially instantiated with σ , with matches that fail to satisfy the condition of their rule discarded;
3. the first n successful matches are discarded; if there is an $(n+1)$ th match, its rule is applied using that match and the steps 4 and 5 below are taken; otherwise **failure** is returned;
4. the term resulting from applying the given rule with the $(n+1)$ th match is fully reduced using the equations in \mathcal{R} ;
5. the triple formed by the metarepresentation of the resulting fully reduced term, the metarepresentation of its corresponding sort or kind, and the metarepresentation of the substitution used in the reduction is returned.

The **failure** value should not be confused with the “undefined” value for the **metaApply** operation. As already mentioned before for descent functions in general, this operation is partial because it does not make sense on some nonvalid arguments that are terms of the appropriate sort but are not correct metarepresentations. However, even if all arguments are valid in this sense, the intended rule application may fail, either because there is no match or because the match does not satisfy the corresponding rule condition, and then **failure** is used to represent this situation, which is important to distinguish from ill-formed invocations, for example, for error recovery purposes.

Note also that, according to the information in step 3 above, the last argument of **metaApply** is a natural number used to enumerate (starting from 0) all the possible solutions of the intended rule application. For efficiency, the different solutions should be generated in order, that is, starting with the argument 0 and increasing it until a failure is obtained, indicating that there are no more solutions.

Appropriate selectors extract from the result triples their three components:

```
op getTerm : ResultTriple -> Term .
op getType : ResultTriple -> Type .
op getSubstitution : ResultTriple -> Substitution .
```

As an example, we can force at the metalevel the rewriting of the term $\$$ in the module VENDING-MACHINE, so that only the rule buy-c is used, and only once.

```
Maude> reduce in META-LEVEL :
      metaApply(upModule('VENDING-MACHINE, false),
                '$.Coin, 'buy-c, none, 0) .
result ResultTriple: {'c.Item, 'Item, none}
```

Similarly, we can force the rewriting of the same term so that this time only the rule add-\$ is applied.

```
Maude> reduce in META-LEVEL :
      metaApply(upModule('VENDING-MACHINE, false),
                '$.Coin, 'add-$, none, 0) .
result ResultTriple:
  {'_['$.Coin, '$.Coin], 'Marking, 'M:Marking <- '$.Coin}
```

However, using `metaApply`, we cannot force the term $q \$$ to be rewritten with the rule buy-c, since its lefthand side, $\$$, does not match (without extension) this term. In this case, we should use instead the `metaXapply` operation described below.

```
Maude> reduce in META-LEVEL :
      metaApply(upModule('VENDING-MACHINE, false),
                '_['q.Coin, '$.Coin], 'buy-c, none, 0) .
result ResultTriple?: (failure).ResultTriple?
```

metaXapply

The (partial) operation `metaXapply` takes as arguments the metarepresentation of a module, the metarepresentation of a term, the metarepresentation of a rule label, the metarepresentation of a set of assignments (possibly empty) defining a partial substitution, a natural number, a Bound value, and another natural number.

The operation `metaXapply`($\overline{\mathcal{R}}$, \bar{t} , \bar{l} , σ , n , b , m) is evaluated as the function `metaApply` but using extension (see Section 4.8) and in any possible position, not only at the top. The arguments n and b can be used to localize the part of the term where the rule application can take place:

- n is the lower bound on depth in terms of nested operators, and should be set to 0 to start searching from the top, while

- the `Bound` argument b indicates the upper bound, and should be set to `unbounded` to have no cut off.

Notice that nested occurrences of an operator with the `assoc` attribute are counted as a single operator for depth purposes, that is, matching takes place on the *flattened term* (see Section 4.8). The same idea applies to `iter` operators (see section 4.4.2): a whole stack of an `iter` operator counts as a single operator. Furthermore, because of matching with extension, the solution may have an extra layer, as illustrated in the matching examples at the end of Section 14.5.5.

The last `Nat` argument m in `metaXapply`($\overline{\mathcal{R}}$, \bar{t} , \bar{l} , σ , n , b , m), as in the case of the operation `metaApply`, is the solution number, used to enumerate multiple solutions. The first solution is 0, and they should again be generated in order for efficiency.

The result of `metaXapply` has an additional component, giving the context (a term with a single “hole”, represented `[]`) inside the given term t , where the rewriting has taken place. The sort `NeCTermList` represents nonempty lists of terms with exactly one “hole,” that is, exactly one term of sort `Context`, the rest being of sort `Term`. The sort `GTermList` is the supersort of `NeCTermList` and `TermList` needed for the `assoc` attribute (hidden in the following declarations in the `ditto` attribute) to make sense.

```

sorts Context NeCTermList GTermList .
subsorts Context < NeCTermList < GTermList .
subsort TermList < GTermList .

op [] : -> Context [ctor] .
op __ : TermList NeCTermList -> NeCTermList [ctor ditto] .
op __ : NeCTermList TermList -> NeCTermList [ctor ditto] .
op __ : GTermList GTermList -> GTermList [ctor ditto] .
op _[] : Qid NeCTermList -> Context [ctor] .

sorts Result4Tuple Result4Tuple? .
subsort Result4Tuple < Result4Tuple? .
op {_,_,_,_} : Term Type Substitution Context -> Result4Tuple
  [ctor] .
op failure : -> Result4Tuple? [ctor] .

op metaXapply :
  Module Term Qid Substitution Nat Bound Nat ~> Result4Tuple?
  [special (...)] .

```

Appropriate selectors extract from the result 4-tuples their four components:

```

op getTerm : Result4Tuple -> Term .
op getType : Result4Tuple -> Type .
op getSubstitution : Result4Tuple -> Substitution .
op getContext : Result4Tuple -> Context .

```

As an example, we can force at the metalevel the rewriting of the term $\$ q$ in the module VENDING-MACHINE so that only the rule buy-c is used (compare with the last metaApply example).

```
Maude> reduce in META-LEVEL :
      metaXapply(upModule('VENDING-MACHINE, false),
        '_[q.Coin, '$.Coin], 'buy-c, none, 0, unbounded, 0) .
result Result4Tuple:
  {'_[q.Coin, 'c.Item], 'Marking, none, '_[q.Coin, []]}
```

Notice the fragment `'_[q.Coin, []]` of the result, providing the context where the rule has been applied. Since this is the only possible solution, if we request the “next” solution (by increasing to 1 the last argument), the result will be a failure.

```
Maude> reduce in META-LEVEL :
      metaXapply(upModule('VENDING-MACHINE, false),
        '_[q.Coin, '$.Coin], 'buy-c, none, 0, unbounded, 1) .
result Result4Tuple?: (failure).Result4Tuple?
```

14.5.5 Matching: metaMatch and metaXmatch

The (partial) operation `metaMatch` takes as arguments the metarepresentation of a module, the metarepresentations of two terms, the metarepresentation of a condition, and a natural number.

```
sort Substitution? .
subsort Substitution < Substitution? .
op noMatch : -> Substitution? [ctor] .
op metaMatch : Module Term Term Condition Nat ~> Substitution?
  [special (...)] .
```

The operation `metaMatch`($\overline{\mathcal{R}}$, \bar{t} , \bar{t}' , $Cond$, n) tries to match at the top the terms t and t' in the module \mathcal{R} in such a way that the resulting substitution satisfies the condition $Cond$. The last argument is used to enumerate possible matches. If the matching attempt is successful, the result is the corresponding substitution; otherwise, `noMatch` is returned. The generalization to `metaXmatch` follows exactly the same ideas as for `metaXapply`. Notice that the operation `metaMatch` provides the metalevel counterpart of the object-level command `match`, while the operation `metaXmatch` provides a generalization of the object-level command `xmatch` (see Sections 4.7, 4.8, and 25.3) in that it is possible to specify min and max depths (in terms of theory layers) and search for proper subterms that do not belong to the top theory layer. The object-level behavior of the `xmatch` command is obtained by setting both min and max depth to 0.

```
sorts MatchPair MatchPair? .
subsort MatchPair < MatchPair? .
```

```

op {_,_} : Substitution Context -> MatchPair [ctor] .
op noMatch : -> MatchPair? [ctor] .
op metaXmatch :
  Module Term Term Condition Nat Bound Nat ~> MatchPair?
  [special (...)] .

```

Appropriate selectors extract from the result pairs their two components:

```

op getSubstitution : MatchPair -> Substitution .
op getContext : MatchPair -> Context .

```

In the following examples, we try to match the pattern `M:Marking $` with the term `$ q c a` in several different ways:

- at the top, asking for the first solution,

```

Maude> reduce in META-LEVEL :
  metaMatch(upModule('VENDING-MACHINE, false),
    '_[M:Marking, '$.Coin],
    '_[$.Coin, 'q.Coin, 'a.Item, 'c.Item],
    nil, 0) .
result Assignment:
  'M:Marking <- '_[q.Coin, 'a.Item, 'c.Item]

```

- at the top, asking for the second solution (that does not exist in this example)

```

Maude> reduce metaMatch(upModule('VENDING-MACHINE, false),
  '_[M:Marking, '$.Coin],
  '_[$.Coin, 'q.Coin, 'a.Item, 'c.Item],
  nil, 1) .
result Substitution?: (noMatch).Substitution?

```

- anywhere, asking for the first solution,

```

Maude> reduce metaXmatch(upModule('VENDING-MACHINE, false),
  '_[M:Marking, '$.Coin],
  '_[$.Coin, 'q.Coin, 'a.Item, 'c.Item],
  nil, 0, unbounded, 0) .
result MatchPair:
  {M:Marking <- '_[q.Coin, 'a.Item, 'c.Item], []}

```

- anywhere, asking for the second solution,

```

Maude> reduce metaXmatch(upModule('VENDING-MACHINE, false),
  '_[M:Marking, '$.Coin],
  '_[$.Coin, 'q.Coin, 'a.Item, 'c.Item],
  nil, 0, unbounded, 1) .
result MatchPair:
  {M:Marking <- '_[a.Item, 'c.Item], '_[q.Coin, []]}

```

- at the top, asking for the first solution satisfying a given condition (that again does not exist),

```
Maude> reduce metaMatch(upModule('VENDING-MACHINE, false),
  '__['M:Marking, '$.Coin],
  '__['$.Coin, 'q.Coin, 'a.Item, 'c.Item],
  'M:Marking = 'a.Item, 0) .
result Substitution?: (noMatch).Substitution?
```

- anywhere, asking for the first solution satisfying a given condition,

```
Maude> reduce metaXmatch(upModule('VENDING-MACHINE, false),
  '__['M:Marking, '$.Coin],
  '__['$.Coin, 'q.Coin, 'a.Item, 'c.Item],
  'M:Marking = 'a.Item, 0, unbounded, 0) .
result MatchPair:
  {'M:Marking <- 'a.Item, '__['__['q.Coin, 'c.Item], []]}
```

As mentioned in the previous section, when matching with extension, the solution may have an extra layer. Let us consider, for example, the following module:

```
fmod METAXMATCH-EX is
  pr META-LEVEL .
  op foo : QidSet ~> QidSet .
endfm
```

Then we take at the metalevel the pattern `_;_('A, QS:QidSet)` and the (flattened) subject term `foo(;_('A, 'B, 'C))`, and ask for matches with extension under at most 1 theory layer, as shown in the following reductions:

```
Maude> red metaXmatch(upModule('METAXMATCH-EX, false),
  upTerm(('A ; QS:QidSet)),
  upTerm(foo('A ; 'B ; 'C)), nil, 0, 1, 0) .
result MatchPair: {'QS:QidSet <- '[_;_['B.Sort, 'C.Sort], 'foo[[]]}

Maude> red metaXmatch(upModule('METAXMATCH-EX, false),
  upTerm(('A ; QS:QidSet)),
  upTerm(foo('A ; 'B ; 'C)), nil, 0, 1, 1) .
result MatchPair: {'QS:QidSet <- 'C.Sort, 'foo[';_['B.Sort, []]}

Maude> red metaXmatch(upModule('METAXMATCH-EX, false),
  upTerm(('A ; QS:QidSet)),
  upTerm(foo('A ; 'B ; 'C)), nil, 0, 1, 2) .
result MatchPair: {'QS:QidSet <- 'B.Sort, 'foo[';_['C.Sort, []]}

Maude> red metaXmatch(upModule('METAXMATCH-EX, false),
  upTerm(('A ; QS:QidSet)),
  upTerm(foo('A ; 'B ; 'C)), nil, 0, 1, 3) .
result MatchPair?: (noMatch).MatchPair?
```

Obviously, there is no match at the top, but under one theory layer (the `foo` operator) we have `_;_('A, 'B, 'C)`. The first solution is the expected one,

with the variable `QS:QidSet` matching the subterm `_;_('B, 'C)`. However, in the next two solutions we see that we also have the variable `QS:QidSet` matching either the fragment `'C` or `'B` while the other fragment goes into the extension. Then the context in the solution has 2 theory layers but this is just a feature of matching with extension: some solutions will have an extra layer.

As another example of this situation, let us consider the following reductions:

```
Maude> reduce in META-LEVEL :
      metaXmatch(upModule('METAXMATCH-EX, false),
        upTerm(s N:Nat), upTerm(prec(s_^2(0))), nil, 0, 1, 0) .
result MatchPair: {'N:Nat <- 's_['0.Zero], 'prec[[]]}

Maude> red metaXmatch(upModule('METAXMATCH-EX, false),
      upTerm(s N:Nat), upTerm(prec(s_^2(0))), nil, 0, 1, 1) .
result MatchPair: {'N:Nat <- '0.Zero, 'prec['s_ [[]]]}
```

Here the context in the first solution has one theory layer while the context in the second has two, but the actual matching problem solved (with extension), namely, $s \ N \leq? \ s \ ^2(0)$ under the single theory layer provided by the operator `prec` is the same in both reductions.

14.5.6 Searching: `metaSearch` and `metaSearchPath`

`metaSearch`

The operation `metaSearch` takes as arguments the metarepresentation of a module, the metarepresentation of the starting term for search, the metarepresentation of the pattern to search for, the metarepresentation of a condition to be satisfied, the metarepresentation of the kind of search to carry on, a `Bound` value, and a natural number.

```
op metaSearch :
  Module Term Term Condition Qid Bound Nat ~> ResultTriple?
  [special (...)] .
```

The searching strategy used by `metaSearch` coincides with that of the object-level `search` command in Maude (see Sections 6.4 and 25.4). The `Qid` values that are allowed as arguments are: `'*` for a search involving zero or more rewrites (corresponding to `=>*` in the `search` command), `'+` for a search consisting in one or more rewrites (`=>+`), and `'!` for a search that only matches canonical forms (`=>!`). The `Bound` argument indicates the maximum depth of the search, and the `Nat` argument is the solution number. To indicate a search consisting in exactly one rewrite, we set the maximum depth of the search to the number 1.

Using `metaSearch` we can redo at the metalevel the last example in Section 6.4. The results give us the answer to the question: if I have already inserted one dollar and three quarters in the vending machine, can I get two cakes and an apple? The answer is yes; in fact, there are several ways.

```

Maude> reduce in META-LEVEL :
  metaSearch(upModule('VENDING-MACHINE, false),
    '__['$.Coin, 'q.Coin, 'q.Coin,'q.Coin],
    '__['c.Item, 'a.Item, 'c.Item, 'M:Marking],
    nil, '+, unbounded, 0) .
result ResultTriple:
  {'__['q.Coin,'q.Coin,'q.Coin,'q.Coin,'a.Item,'c.Item,'c.Item],
   'Marking,
   'M:Marking <- '__['q.Coin, 'q.Coin, 'q.Coin, 'q.Coin]}

Maude> reduce in META-LEVEL :
  metaSearch(upModule('VENDING-MACHINE, false),
    '__['$.Coin, 'q.Coin, 'q.Coin, 'q.Coin],
    '__['c.Item, 'a.Item, 'c.Item, 'M:Marking],
    nil, '+, unbounded, 1) .
result ResultTriple:
  {'__['a.Item, 'c.Item, 'c.Item],
   'Marking,
   'M:Marking <- 'null.Marking}

```

metaSearchPath

The operation `metaSearchPath` is complementary to `metaSearch` and carries out the same search, but instead of returning the final state and matching substitution it returns the sequence of states and rules on a path starting with the reduced initial state and leading to (but not including) the final state.

```

op metaSearchPath :
  Module Term Term Condition Qid Bound Nat ~> Trace?
  [special (...)] .

```

The sort `Trace` is used to represent the path as a list of triples by means of the following syntax:

```

sorts TraceStep Trace Trace? .
subsorts TraceStep < Trace < Trace? .
op {_,_,_} : Term Type Rule -> TraceStep [ctor] .
op nil : -> Trace [ctor] .
op __ : Trace Trace -> Trace [ctor assoc id: nil format (d n d)] .
op failure : -> Trace? [ctor] .

```

We run again the same two examples as above, with the following results.

```

Maude> reduce in META-LEVEL :
  metaSearchPath(upModule('VENDING-MACHINE, false),
    '__['$.Coin, 'q.Coin, 'q.Coin,'q.Coin],
    '__['c.Item, 'a.Item, 'c.Item, 'M:Marking],
    nil, '+, unbounded, 0) .
result Trace:
  {'__['$.Coin,'q.Coin,'q.Coin,'q.Coin],

```

```

    'Marking,
    r1 'M:Marking => '___['$.Coin,'M:Marking] [label('add-$)] .}
  {'___['$.Coin,'$.Coin,'q.Coin,'q.Coin,'q.Coin],
   'Marking,
   r1 'M:Marking => '___['$.Coin,'M:Marking] [label('add-$)] .}
  {'___['$.Coin,'$.Coin,'$.Coin,'q.Coin,'q.Coin,'q.Coin],
   'Marking,
   r1 '$.Coin => 'c.Item [label('buy-c)] .}
  {'___['$.Coin,'$.Coin,'q.Coin,'q.Coin,'q.Coin,'c.Item],
   'Marking,
   r1 '$.Coin => 'c.Item [label('buy-c)] .}
  {'___['$.Coin,'q.Coin,'q.Coin,'q.Coin,'c.Item,'c.Item],
   'Marking,
   r1 '$.Coin => '___[q.Coin,'a.Item] [label('buy-a)] .}

Maude> reduce in META-LEVEL :
      metaSearchPath(upModule('VENDING-MACHINE, false),
        '___['$.Coin, 'q.Coin, 'q.Coin, 'q.Coin],
        '___['c.Item, 'a.Item, 'c.Item, 'M:Marking],
        nil, '+, unbounded, 1) .
result Trace:
  {'___['$.Coin,'q.Coin,'q.Coin,'q.Coin],
   'Marking,
   r1 'M:Marking => '___['$.Coin,'M:Marking] [label('add-$)] .}
  {'___['$.Coin,'$.Coin,'q.Coin,'q.Coin,'q.Coin],
   'Marking,
   r1 '$.Coin => 'c.Item [label('buy-c)] .}
  {'___['$.Coin,'q.Coin,'q.Coin,'q.Coin,'c.Item],
   'Marking,
   r1 '$.Coin => '___[q.Coin,'a.Item] [label('buy-a)] .}
  {'___[q.Coin,'q.Coin,'q.Coin,'q.Coin,'a.Item,'c.Item],
   'Marking,
   r1 '___[q.Coin,'q.Coin,'q.Coin,'q.Coin] => '$.Coin
     [label('change)] .}
  {'___['$.Coin,'a.Item,'c.Item],
   'Marking,
   r1 '$.Coin => 'c.Item [label('buy-c)] .}

```

The operations `metaSearchPath` and `metaSearch` share caching, so calling one after the other on the same arguments only performs a single search.

14.5.7 Parsing and pretty-printing: `metaParse` and `metaPrettyPrint`

`metaParse`

The (partial) operation `metaParse` takes as arguments the metarepresentation of a module, a list of quoted identifiers metarepresenting a list of tokens, and a value of the sort `Type?`, i.e., either the metarepresentation of a component or the constant `anyType`.

```

sort Type? .
subsort Type < Type? .
op anyType : -> Type? [ctor] .
sort ResultPair? .
subsort ResultPair < ResultPair? .
op noParse : Nat -> ResultPair? [ctor] .
op ambiguity : ResultPair ResultPair -> ResultPair? [ctor] .
op metaParse : Module QidList Type? ~> ResultPair? [special (...)] .

```

The operation `metaParse` reflects the `parse` command in Maude (see Section 3.9.4); that is, it tries to parse the given list of tokens as a term of the given type in the module given as first argument; the constant `anyType` allows any component. If `metaParse` succeeds, it returns the metarepresentation of the parsed term with its corresponding sort or kind. Otherwise, it returns:

- `noParse(n)` if there was no parse, where *n* is the index of the first bad token (counting from 0), or the number of tokens in the case of unexpected end of input; or
- `ambiguity(r1, r2)` if there were multiple parses, where *r*₁ and *r*₂ are the result pairs corresponding to two distinct parses.

These are simple examples of using `metaParse`:

```

Maude> reduce in META-LEVEL :
      metaParse(upModule('VENDING-MACHINE, false),
                '$ 'q 'q 'q, 'Marking) .
result ResultPair:
  {'_['_['$.Coin, '_['_['q.Coin, '_['_['q.Coin, 'q.Coin]]], 'Marking}

Maude> reduce in META-LEVEL :
      metaParse(upModule('VENDING-MACHINE, false),
                '$ 'q 'd 'q, 'Marking) .
result ResultPair?: noParse(2)

```

`metaPrettyPrint`

The (partial) operation `metaPrettyPrint` takes as arguments the metarepresentations of a module \mathcal{R} and of a term *t* together with a set of printing options, and it returns a list of quoted identifiers that metarepresents the string of tokens produced by pretty-printing the term *t* in the signature of \mathcal{R} . In the event of an error an empty list of quoted identifiers is returned.

```

op metaPrettyPrint : Module Term PrintOptionSet ~> QidList
  [special (...)] .

```

Pretty-printing a term involves more than just naively using the mixfix syntax for operators. Precedence and gathering information and the relative positions of underscores in an operator and its parent in the term must be considered to determine whether parentheses need to be inserted around any given subterm to avoid ambiguity. If there is ad-hoc overloading in the module,


```
= metaPrettyPrint(M:Module, T:Term,
  mixfix flat format number rat) .
```

For example,

```
Maude> reduce in META-LEVEL :
  metaPrettyPrint(upModule('VENDING-MACHINE, false),
    '_[ '$.Coin, '_[ 'q.Coin, '_[ 'q.Coin, 'q.Coin]]) .
result NeTypeList:
  '\r '\! '$ '\o '\r '\! 'q '\o '\r '\! 'q '\o '\r '\! 'q '\o
```

14.5.8 Sort operations

The META-LEVEL module also provides in a built-in way commonly needed operations on the poset of sorts of a given module.

All these operations, related to sorts and kinds, take as first argument a term of sort `Module`. Assuming that this term is indeed the metarepresentation of a module, the remaining arguments might be terms representing sorts or kinds that do not correspond to sorts or kinds declared in such a module; in this case, the operation is undefined.

In the following we include descriptions together with simple examples of using these operations.

`sortLeq`

The operation `sortLeq` takes as arguments the metarepresentation of a module \mathcal{R} and the metarepresentations of two types, that is, either sorts or kinds.

```
op sortLeq : Module Type Type ~> Bool [special (...)] .
```

According to whether the types passed to `sortLeq` as arguments are metarepresented sorts or kinds, we can distinguish the following cases:

- Assume first that both types given as arguments are two sorts s and s' . Let S be the set of sorts in \mathcal{R} and let $\leq_{\mathcal{R}}$ be its subsort relation. When $s, s' \in S$, `sortLeq` returns `true` if $s \leq_{\mathcal{R}} s'$ and `false` otherwise. For example,

```
Maude> reduce in META-LEVEL :
  sortLeq(upModule('NUMBERS, false), 'Zero, 'Nat) .
result Bool: true

Maude> reduce in META-LEVEL :
  sortLeq(upModule('NUMBERS, false), 'Zero, 'NzNat) .
result Bool: false
```

- If both types given as arguments are kinds in \mathcal{R} , then `sortLeq` returns `false` when both kinds are different and `true` when they are equal. For example,

```

Maude> reduce in META-LEVEL :
      sortLeq(upModule('NUMBERS, false), '[Zero', '[Nat']) .
result Bool: true

Maude> reduce in META-LEVEL :
      sortLeq(upModule('NUMBERS, false), '[Zero', '[Bool']) .
result Bool: false

```

- If one type is one sort in \mathcal{R} and the other one is a kind in \mathcal{R} , then `sortLeq` checks whether the given sort belongs to the given kind or not. For example,

```

Maude> reduce in META-LEVEL :
      sortLeq(upModule('NUMBERS, false), '[Zero', 'Bool) .
result Bool: false

Maude> reduce in META-LEVEL :
      sortLeq(upModule('NUMBERS, false), 'Zero, '[NatSet']) .
result Bool: true

```

sameKind

The operation `sameKind` takes as arguments the metarepresentation of a module \mathcal{R} and the metarepresentations of two types, that is, either sorts or kinds.

```

op sameKind : Module Type Type ~> Bool [special (...)] .

```

Let S be the set of sorts in \mathcal{R} and let $\leq_{\mathcal{R}}$ be its subsort relation. When the two types passed as arguments to `sameKind` are sorts $s, s' \in S$, the operation `sameKind` returns `true` if s and s' belong to the same connected component in the subsort ordering $\leq_{\mathcal{R}}$, that is, if they belong to the same kind, and `false` otherwise. When the two arguments are kinds in \mathcal{R} , `sameKind` returns `true` when they are indeed the same, and `false` otherwise. Finally, when one argument is one sort and the other is a kind, this operation checks whether the sort belongs to the kind.

For example, we have the following reductions about sorts and kinds in the module `NUMBERS`.

```

Maude> reduce in META-LEVEL :
      sameKind(upModule('NUMBERS, false), 'Zero, 'NzNat) .
result Bool: true

Maude> reduce in META-LEVEL :
      sameKind(upModule('NUMBERS, false), 'Zero, 'Nat3) .
result Bool: false

Maude> reduce in META-LEVEL :
      sameKind(upModule('NUMBERS, false), '[Zero', '[NzNat']) .
result Bool: true

Maude> reduce in META-LEVEL :
      sameKind(upModule('NUMBERS, false), '[Zero', 'NzNat) .
result Bool: true

```

completeName

The operation `completeName` takes as arguments the metarepresentation of a module \mathcal{R} and the metarepresentation of a sort s or a kind k . When its second argument is the metarepresentation of a sort s , it returns the same metarepresentation of s . But if its second argument is the metarepresentation of a kind k , then it returns the metarepresentation of the complete name of k in \mathcal{R} , i.e., the metarepresentation of the comma-separated list of the maximal elements of the corresponding connected component.

```
op completeName : Module Type ~> Type [special (...)] .
```

For example,

```
Maude> reduce in META-LEVEL :
      completeName(upModule('NUMBERS, false), 'Zero) .
result Sort: 'Zero

Maude> reduce in META-LEVEL :
      completeName(upModule('NUMBERS, false), '['Zero']) .
result Kind: '['NatSeq',NatSet']
```

getKind and getKinds

The operation `getKind` takes as arguments the metarepresentation of a module \mathcal{R} and the metarepresentation of a type, i.e., a sort or a kind. When its second argument is the metarepresentation of a type in \mathcal{R} , it returns the metarepresentation of the complete name of the corresponding kind.

```
op getKind : Module Type ~> Kind [special (...)] .
```

For example,

```
Maude> reduce in META-LEVEL :
      getKind(upModule('NUMBERS, false), 'Zero) .
result Kind: '['NatSeq',NatSet']

Maude> reduce in META-LEVEL :
      getKind(upModule('NUMBERS, false), '['Zero']) .
result Kind: '['NatSeq',NatSet']
```

The operation `getKinds` takes as its only argument the metarepresentation of a module \mathcal{R} and returns the metarepresentation of the set of kinds declared in \mathcal{R} , with kinds metarepresented using their complete names.

```
op getKinds : Module ~> KindSet [special (...)] .
```

For example,

```
Maude> reduce in META-LEVEL : getKinds(upModule('NUMBERS, false)) .
result NeKindSet: '['Bool'] ; '['Nat3'] ; '['NatSeq',NatSet']
```

lesserSorts

The operation `lesserSorts` takes as arguments the metarepresentation of a module \mathcal{R} and the metarepresentation of a type, i.e., a sort or a kind.

```
op lesserSorts : Module Type ~> SortSet [special (...)] .
```

Let S be the set of sorts in \mathcal{R} . When $s \in S$, `lesserSorts` returns the metarepresentation of the set of sorts strictly smaller than s in S . For example,

```
Maude> reduce in META-LEVEL :
  lesserSorts(upModule('NUMBERS, false), 'Nat) .
result NeSortSet: 'NzNat ; 'Zero

Maude> reduce in META-LEVEL :
  lesserSorts(upModule('NUMBERS, false), 'Zero) .
result EmptyTypeSet: (none).EmptyTypeSet

Maude> reduce in META-LEVEL :
  lesserSorts(upModule('NUMBERS, false), 'NatSeq) .
result NeSortSet: 'Nat ; 'NzNat ; 'Zero
```

When the second argument of `lesserSorts` metarepresents a kind in \mathcal{R} , this operation returns the metarepresentation of the set of all sorts in such kind. For example,

```
Maude> reduce in META-LEVEL :
  lesserSorts(upModule('NUMBERS, false), '['NatSeq']) .
result NeSortSet: 'Nat ; 'NatSeq ; 'NatSet ; 'NzNat ; 'Zero

Maude> reduce in META-LEVEL :
  lesserSorts(upModule('NUMBERS, false), '['Bool']) .
result Sort: 'Bool
```

leastSort

The operation `leastSort` takes as arguments the metarepresentation of a module \mathcal{R} and the metarepresentation of a term t , and it returns the metarepresentation of the least sort or kind of t in \mathcal{R} , obtained without reducing the term, that is, the memberships in the module are used to get the information, but equations are not used to reduce the term.

```
op leastSort : Module Term ~> Type [special (...)] .
```

For example,

```
Maude> reduce in META-LEVEL :
  leastSort(upModule('NUMBERS, false), 'p['s_['zero.Zero]]) .
result Sort: 'Nat
```

glbSorts

The operation `glbSorts` takes as arguments the metarepresentation of a module \mathcal{R} and the metarepresentations of two types, that is, either sorts or kinds.

```
op glbSorts : Module Type Type ~> TypeSet [special (...)] .
```

According to whether the types passed to `glbSorts` as arguments are metarepresented sorts or kinds, we can distinguish the following cases:

- If both types given as arguments are sorts in \mathcal{R} , then `glbSorts` returns the metarepresentation of the set (which can be empty) consisting of the largest sorts that are common subsorts of the two given sorts, that is, the set of maximal lower bounds of the two sorts; when this set is a singleton set $\{s\}$, then s will be the greatest lower bound of the two sorts, thus the operation name `glbSorts`.

For example, we have the following reductions concerning sorts in the module `NUMBERS`.

```
Maude> reduce in META-LEVEL :
      glbSorts(upModule('NUMBERS, false), 'Zero, 'Nat) .
result Sort: 'Zero

Maude> reduce in META-LEVEL :
      glbSorts(upModule('NUMBERS, false), 'NatSet, 'NatSeq) .
result Sort: 'Nat

Maude> reduce in META-LEVEL :
      glbSorts(upModule('NUMBERS, false), 'NzNat, 'NzNat) .
result Sort: 'NzNat

Maude> reduce in META-LEVEL :
      glbSorts(upModule('NUMBERS, false), 'Zero, 'NzNat) .
result EmptyTypeSet: (none).EmptyTypeSet

Maude> reduce in META-LEVEL :
      glbSorts(upModule('NUMBERS, false), 'NzNat, 'Bool) .
result EmptyTypeSet: (none).EmptyTypeSet
```

- If both types given as arguments are kinds in \mathcal{R} , then `glbSorts` returns the empty set when both kinds are different, and the metarepresentation of the kind (using the corresponding complete name) when both kinds are equal. For example,

```
Maude> reduce in META-LEVEL :
      glbSorts(upModule('NUMBERS, false), '['Nat'], '['Bool']) .
result EmptyTypeSet: (none).EmptyTypeSet

Maude> reduce in META-LEVEL :
      glbSorts(upModule('NUMBERS, false), '['Nat'], '['NatSeq']) .
result Kind: '['NatSeq',NatSet']
```

- If one type is one sort in \mathcal{R} and the other one is a kind in \mathcal{R} , then `glbSorts` returns the metarepresentation of the sort when the sort belongs to the kind, and the empty set otherwise. For example,

```
Maude> reduce in META-LEVEL :
      glbSorts(upModule('NUMBERS, false), '[Nat', 'Bool) .
result EmptyTypeSet: (none).EmptyTypeSet

Maude> reduce in META-LEVEL :
      glbSorts(upModule('NUMBERS, false), '[NatSeq', 'Zero) .
result Sort: 'Zero

Maude> reduce in META-LEVEL :
      glbSorts(upModule('NUMBERS, false), 'NzNat, '[NatSet]) .
result Sort: 'NzNat
```

`maximalSorts` and `minimalSorts`

The operations `maximalSorts` and `minimalSorts` take as arguments the metarepresentation of a module \mathcal{R} and the metarepresentation of a kind k . If k is a kind in \mathcal{R} , `maximalSorts` returns the metarepresentation of the set of the maximal sorts in the connected component of k , while `minimalSorts` returns the metarepresentation of the set of its minimal sorts.

```
op maximalSorts : Module Kind ~> SortSet [special (...)] .
op minimalSorts : Module Kind ~> SortSet [special (...)] .
```

For example,

```
Maude> reduce in META-LEVEL :
      maximalSorts(upModule('NUMBERS, false), '[Zero]) .
result NeSortSet: 'NatSeq ; 'NatSet

Maude> reduce in META-LEVEL :
      minimalSorts(upModule('NUMBERS, false), '[Zero]) .
result NeSortSet: 'Zero ; 'NzNat
```

`maximalAritySet`

The operation `maximalAritySet` takes as arguments the metarepresentation of a module \mathcal{R} , the metarepresentation of an operator f in \mathcal{R} , the metarepresentation of an arity (list of types) for f and the metarepresentation of a sort s , and then computes the set of maximal arities (lists of types) that f could take and have a sort $s' \leq_{\mathcal{R}} s$. This result might be the empty set if s is small or f is only defined at the kind level.

Notice that the result of this operation is a *set of lists* of types, which is built by means of the following syntax, extending the syntax for building lists of types that we only show partially here and whose full specification can be found in the module `META-MODULE` in the file `prelude.maude` available with the Maude distribution.

```

sort NeTypeList TypeList .
op nil : -> TypeList [ctor] .
op __ : TypeList TypeList -> TypeList [ctor ditto] .

sort TypeListSet .
subsort TypeList TypeSet < TypeListSet .
op _;_ : TypeListSet TypeListSet -> TypeListSet [ctor ditto] .
eq T:TypeList ; T:TypeList = T:TypeList .

op maximalAritySet : Module Qid TypeList Sort ~> TypeListSet
  [special (...)] .

```

Let us consider for example the operator `_+_` in the module `NUMBERS`, where it is overloaded by means of the following declarations:

```

op _+_ : Nat Nat -> Nat [assoc comm].
op _+_ : NzNat Nat -> NzNat [ditto] .
op _+_ : Nat3 Nat3 -> Nat3 [comm] .

```

With this information, we obtain the following reductions concerning this operator:

```

Maude> reduce in META-LEVEL :
  maximalAritySet(upModule('NUMBERS, false),
    '_+_ , 'NzNat 'NzNat, 'NzNat) .
result TypeListSet: 'Nat 'NzNat ; 'NzNat 'Nat

Maude> reduce in META-LEVEL :
  maximalAritySet(upModule('NUMBERS, false),
    '_+_ , 'Nat 'Nat, 'NzNat) .
result TypeListSet: 'Nat 'NzNat ; 'NzNat 'Nat

Maude> reduce in META-LEVEL :
  maximalAritySet(upModule('NUMBERS, false),
    '_+_ , 'Nat 'Nat, 'Nat) .
result NeTypeList: 'Nat 'Nat

Maude> reduce in META-LEVEL :
  maximalAritySet(upModule('NUMBERS, false),
    '_+_ , 'Nat3 'Nat3, 'Nat3) .
result NeTypeList: 'Nat3 'Nat3

```

Notice that if the operator f and the list of types passed as arguments to `maximalAritySet` do not match, then the result is an error, which is represented as a non-reduced term in a metalevel kind. We have for instance the following example where we have omitted the lengthy metarepresentation of the `NUMBERS` module.

```

Maude> reduce in META-LEVEL :
  maximalAritySet(upModule('NUMBERS, false),
    '_+_ , 'Nat3 'Nat3, 'NzNat) .

```

```

result [GTermList,ParameterList,QidList,
       TypeListSet,Type?,ModuleExpression,Header]:
maximalAritySet(fmod 'NUMBERS is ... endfm,
               '_+_,'Nat3 'Nat3, 'NzNat)

```

14.5.9 Other metalevel operations: wellFormed

The operation `wellFormed` can take as arguments the metarepresentation of a module \mathcal{R} , or the metarepresentation of a module \mathcal{R} and a term t , or the metarepresentation of a module \mathcal{R} and a substitution σ . In the first case, it returns `true` if \mathcal{R} is a well-formed module, and `false` otherwise. In the second case, if t is a well-formed term in \mathcal{R} , it returns `true`; otherwise, it returns `false`. Finally, in the third case, if σ is a well-formed substitution in \mathcal{R} , it returns `true`; otherwise, it returns `false`.

```

op wellFormed : Module -> Bool [special (...)] .
op wellFormed : Module Term ~> Bool [special (...)] .
op wellFormed : Module Substitution ~> Bool [special (...)] .

```

Note that the first operation is total, while the other two are partial (notice the form of the arrow in the declarations). The reason is that the last two are not defined when the term given as first argument does not represent a module, and then it does not make sense to check whether a term or substitution is well formed with respect to such a wrong “module.” For example,

```

Maude> reduce in META-LEVEL :
      wellFormed(upModule('NUMBERS, false)) .
result Bool: true

Maude> reduce in META-LEVEL :
      wellFormed(upModule('NUMBERS, false), 'p['zero.Zero]) .
result Bool: true

Maude> reduce in META-LEVEL :
      wellFormed(upModule('NUMBERS, false),
                 's_['zero.Zero, 'zero.Zero]) .
Advisory: could not find an operator s_ with appropriate domain
in meta-module NUMBERS when trying to interpret metaterm
's_['zero.Zero,'zero.Zero].
result Bool: false

Maude> reduce in META-LEVEL :
      wellFormed(upModule('NUMBERS, false),
                 'N:Zero <- 'zero.Zero) .
result Bool: true

Maude> reduce in META-LEVEL :
      wellFormed(upModule('NUMBERS, false),
                 'N:Nat <- 'p['zero.Zero]) .
result Bool: false

```

```
Maude> reduce in META-LEVEL :
      wellFormed(upModule('NUMBERS, false),
        'N:Zero <- 's_['zero.Zero,'zero.Zero]) .
Advisory: could not find an operator s_ with appropriate domain
          in meta-module NUMBERS when trying to interpret metaterm
          's_['zero.Zero,'zero.Zero].
result Bool: false
```

14.6 Internal strategies

System modules in Maude are rewrite theories that do not need to be Church-Rosser and terminating. Therefore, we need to have good ways of controlling the rewriting inference process—which in principle could not terminate or go in many undesired directions—by means of adequate *strategies*. In Maude, thanks to its reflective capabilities, strategies can be made *internal* to the system. That is, they can be defined using statements in a normal module in Maude, and can be reasoned about as with statements in any other module. In general, strategies are defined in extensions of the META-LEVEL module by using `metaReduce`, `metaApply`, `metaXApply`, etc., as building blocks.

We illustrate some of these possibilities by implementing the following strategies for controlling the execution of the rules in the VENDING-MACHINE module in Section 6.1:

1. insert either a dollar or a quarter in the vending machine;
2. only buy cakes, and buy as many cakes as possible, with the coins already inserted;
3. only buy either cakes or apples, and buy at most n of them, with the coins already inserted;
4. buy the same number of apples and cakes, and buy as many as possible, with the coins already inserted.

Consider the module BUYING-STRATS below, which imports the META-LEVEL module.

```
fmod BUYING-STRATS is
  protecting META-LEVEL .
```

The function `insertCoin` below defines the strategy (1): it expects as first argument either `'add-q` or `'add-$`, for inserting a quarter or a dollar, respectively, and as second argument the metarepresentation of the marking of a vending machine, and it applies once the rule corresponding to the given label. The rules `add-q` and `add-$` are applied using the descent function `metaXApply`. A rule cannot be applied when the result of `metaXApply`-ing the rule is not a term of sort `Result4Tuple`. Note the use of a matching equation in the condition to simplify the presentation of the righthand side of the equation (see Section 4.3), as well as the use of the statement attribute `owise` (see Section 4.5.4) to define the function `insertCoin` for unexpected cases.

```

var T : Term .
var Q : Qid .
var N : Nat .
vars BuyItem? BuyCake? Change? : [Result4Tuple].

op insertCoin : Qid Term -> Term .

ceq insertCoin(Q, T)
  = if BuyItem? :: Result4Tuple
    then getTerm(BuyItem?)
    else T
  fi
  if (Q == 'add-q or Q == 'add-$)
    /\ BuyItem? := metaXapply(upModule('VENDING-MACHINE, false),
      T, Q, none, 0, unbounded, 0) .

eq insertCoin(Q, T) = T [owise] .

```

The function `onlyCakes` below defines the strategy (2): it applies the rule `buy-c` as many times as possible, applying the rule `change` whenever it is necessary. In particular, if the rule `buy-c` can be applied, then there is a recursive call to the function `onlyCakes` with the term resulting from its application. If the rule `buy-c` cannot be applied, then the application of the rule `change` is attempted. If the rule `change` can be applied, then there is a recursive call to the function `onlyCakes` with the term resulting from the `change` rule application. Otherwise, the argument is returned unchanged. The rules `buy-c` and `change` are also applied using the descent function `metaXapply`.

```

op onlyCakes : Term -> Term .

ceq onlyCakes(T)
  = if BuyCake? :: Result4Tuple
    then onlyCakes(getTerm(BuyCake?))
    else (if Change? :: Result4Tuple
          then onlyCakes(getTerm(Change?))
          else T
        fi)
  fi
  if BuyCake? := metaXapply(upModule('VENDING-MACHINE, false),
    T, 'buy-c, none, 0, unbounded, 0)
    /\ Change? := metaXapply(upModule('VENDING-MACHINE, false),
      T, 'change, none, 0, unbounded, 0) .

```

The function `onlyNitems` defines the strategy (3): it applies either the rule `buy-c` or `buy-a` (but not both) at most n times. As expected, the rules are applied using the descent function `metaXapply`. Note the use of the symmetric difference operator `sd` (see Section 9.2) to decrement N .

```

op onlyNitems : Term Qid Nat -> Term .

```

```

ceq onlyNitems(T, Q, N)
= if N == 0
  then T
  else (if BuyItem? :: Result4Tuple
        then onlyNitems(getTerm(BuyItem?), Q, sd(N, 1))
        else (if Change? :: Result4Tuple
              then onlyNitems(getTerm(Change?), Q, N)
              else T
              fi)
        fi)
  fi
if (Q == 'buy-c or Q == 'buy-a)
  /\ BuyItem? := metaXapply(upModule('VENDING-MACHINE, false),
                           T, Q, none, 0, unbounded, 0)
  /\ Change? := metaXapply(upModule('VENDING-MACHINE, false),
                           T, 'change, none, 0, unbounded, 0) .

eq onlyNitems(T, Q, N) = T [owise] .

```

Finally, the function `cakesAndApples` defines the strategy (4): it applies the rule `buy-c` as many times as the rule `buy-a`. To define this function, we use an auxiliary Boolean function `buyItem?` that determines whether a given rule (`buy-c` or `buy-a`) can be applied. In the definition of `cakesAndApples` the Boolean function `buyItem?` is used to check if the rule `buy-a` can be applied after applying the rule `buy-c`. When the answer is `true`, then `buy-c` and `buy-a` are applied once, using the function `onlyNitems` with the appropriate arguments, and the function `cakesAndApples` is applied again to the result.

```

op cakesAndApples : Term -> Term .
op buyItem? : Term Qid -> Bool .

ceq buyItem?(T, Q)
= if BuyItem? :: Result4Tuple
  then true
  else (if Change? :: Result4Tuple
        then buyItem?(getTerm(Change?), Q)
        else false
        fi)
  fi
if (Q == 'buy-c or Q == 'buy-a)
  /\ BuyItem? := metaXapply(upModule('VENDING-MACHINE, false),
                           T, Q, none, 0, unbounded, 0)
  /\ Change? := metaXapply(upModule('VENDING-MACHINE, false),
                           T, 'change, none, 0, unbounded, 0) .

eq buyItem?(T, Q) = false [owise] .

eq cakesAndApples(T)

```

```

= if buyItem?(T, 'buy-c)
  then (if buyItem?(onlyNitems(T, 'buy-c, 1), 'buy-a)
        then cakesAndApples(onlyNitems(onlyNitems(T, 'buy-c, 1),
                                          'buy-a, 1))
        else T
        fi)
  else T
  fi .
endfm

```

As examples, we apply below the buying strategies (2–4) to spend in different ways the same amount of money: three dollars and a quarter.

```

Maude> reduce in BUYING-STRATS :
  onlyCakes('__['$.Coin, '$.Coin, '$.Coin, 'q.Coin]) .
result GroundTerm: '__['q.Coin, 'c.Item, 'c.Item, 'c.Item]

Maude> reduce in BUYING-STRATS :
  onlyNitems('__['$.Coin, '$.Coin, '$.Coin, 'q.Coin],
             'buy-a, 3) .
result GroundTerm:
  '__['q.Coin, 'q.Coin, 'q.Coin, 'q.Coin, 'a.Item, 'a.Item, 'a.Item]

Maude> reduce in BUYING-STRATS :
  cakesAndApples('__['$.Coin, '$.Coin, '$.Coin, 'q.Coin]) .
result GroundTerm: '__['$.Coin, 'q.Coin, 'q.Coin, 'a.Item, 'c.Item]

```

There is in fact great freedom for defining many different types of strategies, or even many different strategy languages inside Maude. As illustrated above with simple examples, this can be done in a completely user-definable way, so that users are not limited by a fixed and closed particular strategy language. Another example is presented in Section 21.7. See [66] for a general methodology for defining internal strategy languages using reflection, and [68, 70] for other examples of rewriting strategies defined in Maude.

However, the great freedom of defining internal strategies at the metalevel is purchased at some cost. First, some familiarity with Maude’s metalevel features is required; and second, some cost in performance is incurred in comparison with what might be possible in a direct implementation using Maude’s rewrite engine. To address these two issues, a strategy language for Maude, that can be used entirely at the object level, has been proposed and has been implemented in prototype form [234]. Work on an implementation of this strategy language at the level of the Maude rewrite engine has already begun. We expect that this object-level strategy language will be available in future Maude releases.

Unification and Variant Generation

15.1 Introduction

Unification is the solving of equations, either in *free algebras* of the form $T_{\Sigma}(X)$, or in *relatively free algebras* modulo a set E of equations, that is, in algebras of the form $T_{\Sigma/E}(X)$. The first case is sometimes called *syntactic unification*. The second case is sometimes called *E-unification*, or, if E is not explicitly mentioned, *equational unification*, or *semantic unification*.

In solving any equation, such as, for example,

$$f(x, h(y)) = f(g(y), z)$$

we look for *instances* of the variables appearing in the equation that make both sides equal. Variables can of course be instantiated by *substitutions*. A substitution that makes both sides of the equation equal, that is, a solution of the equation, is called a *unifier*. For example, if we are solving the above equation in the free algebra $T_{\Sigma}(X)$ with X a countable set of variables and with Σ having a single sort (unsorted unification), the substitution $\sigma = \{x \mapsto g(y), z \mapsto h(y)\}$ is a unifier, and indeed the so-called *most general unifier*, so that for any other unifier ρ there exists a substitution μ such that $\rho = \sigma; \mu$, where $\sigma; \mu$ denotes composition of substitutions in diagrammatic order. That is, any other solution of the equation is an *instance* of the most general solution provided by σ .

Of course, some equations may not have syntactic unifiers, but may have semantic unifiers *modulo* some equations E . Consider, for example, the equation

$$f(h(y), x) = f(g(y), z)$$

which obviously does not have any solution in $T_{\Sigma}(X)$. It does, however, have a solution in $T_{\Sigma/C}(X)$, where C is the commutativity axiom $f(x, y) = f(y, x)$. Indeed, the exact same substitution σ solving the first equation $f(x, h(y)) =$

$f(g(y), z)$ in a syntactic way, is now a unifier solving the second equation $f(h(y), x) = f(g(y), z)$ modulo C , because we have $f(h(y), g(y)) =_C f(g(y), h(y))$.

Unification is a fundamental deductive mechanism used in many automated deduction tasks (see Section 15.6 for a discussion of some of them). It is also very important in combining the paradigms of functional programming and logic programming (in the Prolog sense of “logic programming”). Furthermore, in the context of Maude, unification can be very useful to reason not only about equational theories (functional modules or theories), but also, as explained in Section 15.6.2, about rewrite theories (system modules or theories).

Therefore, it is very useful to have an efficient implementation of unification available in Core Maude, which is what this chapter describes. Specifically, we explain both how order-sorted unification modulo frequently occurring equational axioms and variant-based unification are supported in Maude.

15.2 Order-sorted unification modulo a set of axioms

Although the most general equational theories supported by Maude are membership equational theories, to obtain *practical unification algorithms*, allowing us to effectively compute the solutions of an equational unification problem, it is important to restrict ourselves to *order-sorted* equational theories. Furthermore, for an arbitrary set of equations E no unification algorithm may be known; even if one is known, the number of solutions may be *infinite*.

This suggests a *hybrid* approach, in which we take advantage of Maude’s structuring of a module’s equations into equational axioms Ax , such as associativity, and/or commutativity, and/or identity, and equations E , which are assumed to be coherent,¹ confluent, and terminating modulo Ax . We can then consider order-sorted equational theories of the form $(\Sigma, E \cup Ax)$ and *decompose* the $E \cup Ax$ -unification problem into *two* problems: one of Ax -unification, and another of $E \cup Ax$ -unification that uses an Ax -unification algorithm as a subroutine. This decomposition, as well as a similar one for membership equational theories, is explained in Section 15.6. The point of

¹ Coherence of E modulo Ax is very closely related to the notion of coherence of rules relative to equations explained in Section 6.3; see [200] for the precise definition of coherence in the equational case. The main role of coherence of E modulo Ax is to get the effect of rewriting in Ax -equivalence classes with E . For example, for $Ax = AC$, coherence modulo AC is easily achieved by adding to each equation in E with a top AC function symbol in its lefthand side a similar equation with an extra “extension variable” argument added to the AC function symbol, as explained in Section 4.8. Section 4.8 also explains how, for rewriting modulo axioms Ax supported by Maude, Maude automatically performs such a coherence completion in an implicit, built-in way using extension variables and “extension aware” Ax -matching algorithms; see also Section 19.5.

this decomposition is that Ax -unification needs to be built-in at the level of Core Maude's C++ implementation for efficiency purposes and $E \cup Ax$ -unification can then be built on top of Ax -unification. Since the axioms Ax are well-known and unification algorithms exist for them, the task of building in efficient Ax -unification algorithms, although highly nontrivial, becomes manageable. Maude 2.6 implemented $E \cup Ax$ -unification in Maude itself, but Maude 2.7 implements $E \cup Ax$ -unification also in Core Maude's C++ level.

We can, however, define the basic concepts of order-sorted E -unification in full generality, even though in the Core Maude implementation E will always be a particular combination of equational axioms declared as equational attributes in a module or a theory.

Given an order-sorted equational theory (Σ, E) , an E -unification problem consists of a nonempty set of unification equations of the form $t =^? t'$, written in the notation

$$t_1 =^? t'_1 \wedge \dots \wedge t_n =^? t'_n$$

where $n \geq 1$ and the “conjunction” operator \wedge is assumed to be associative and commutative.

Given such an E -unification problem, an E -unifier for it is an order-sorted substitution² $\sigma : \text{Vars}(t_1, t'_1, \dots, t_n, t'_n) \rightarrow T_\Sigma(X)$ (where we assume that the set X of variables contains a countable number of variables for each sort) such that, for all $i = 1, \dots, n$,

$$E \vdash (\forall Y_i) \sigma(t_i) = \sigma(t'_i),$$

where $Y_i = \text{Vars}(\sigma(t_i), \sigma(t'_i))$, that is, all the equations $(\forall Y_i) \sigma(t_i) = \sigma(t'_i)$ can be deduced in (membership) equational logic from the set of equations E .

A set of unifiers \mathcal{U} is called a *complete set of E -unifiers* for a given E -unification problem $t_1 =^? t'_1 \wedge \dots \wedge t_n =^? t'_n$ iff for any other E -unifier ρ of the same E -unification problem there exists a substitution μ and a unifier $\sigma \in \mathcal{U}$ such that $\rho =_E \sigma; \mu$, that is, for each variable x in the domain of ρ we have $E \vdash \rho(x) = \mu(\sigma(x))$. A complete set of E -unifiers \mathcal{U} is called *minimal* if any proper subset of \mathcal{U} fails to be complete.

For an order-sorted equational theory (Σ, E) , unification is said to be *finitary* if for any E -unification problem there is always a finite complete set of unifiers \mathcal{U} . Similarly, unification for (Σ, E) is called *unitary* if it is finitary and for any E -unification problem a minimal complete set of unifiers is always either empty or a singleton set.

We say that (Σ, E) has a *unification algorithm* if there is an algorithm generating a complete set of E -unifiers for any E -unification problem in (Σ, E) .

Unlike unsorted syntactic unification, which always either fails or has a single most general unifier, order-sorted syntactic unification is *not* necessarily unitary, that is, there is in general no single most general unifier. What

² That is, a sort-preserving mapping from $\text{Vars}(t_1, t'_1, \dots, t_n, t'_n)$, the set of all variables appearing in the terms t_i or t'_i , to $T_\Sigma(X)$.

exists (if Σ is finite) is a finite minimal complete set of syntactic unifiers. For some commonly occurring theories having a unification algorithm, such as the theory A of associativity of a binary function symbol, it is well-known that unification is not finitary: in general an infinite number of solutions may exist. However, for other theories, such as commutativity or associativity-commutativity, unification is finitary, both when Σ is unsorted and when Σ is order-sorted (and finite).

15.3 Theories currently supported

As mentioned in Section 15.2, a practical way of dealing with order-sorted equational unification is to consider order-sorted theories of the form $(\Sigma, E \cup Ax)$, with Ax a set of commonly occurring axioms, declared in Maude as equational attributes (see Section 4.4.1), and E the remaining equations specified with the `eq` or `ceq` keywords. We can then decompose the $E \cup Ax$ -unification problem into an Ax -unification problem and an $E \cup Ax$ -unification problem that uses an Ax -unification algorithm as a subroutine. In such a decomposition, the efficiency of the Ax -unification algorithm becomes crucial.

Maude currently provides an order-sorted Ax -unification algorithm for all order-sorted theories $(\Sigma, E \cup Ax)$ such that the order-sorted signature Σ is *preregular* modulo Ax (see Sections 3.8 and 24.2.5) and the axioms Ax associated to function symbols are as follows:

- there can be arbitrary function symbols and constants with no equational attributes;
- the `iter` equational attribute can be declared for some unary symbols;
- different equational attributes can be declared for some binary function symbols:
 - the `comm` attribute; usually referred to as C (for commutative),
 - the `assoc comm` attributes; usually referred to as AC (for associative and commutative),
 - the `assoc comm id`: attributes; usually referred to as ACU (for associative and commutative with identity, or unit),
 - the `comm id`: attributes; usually referred to as CU (for commutative and identity, or unit),
 - the `id`: attribute; usually referred to as U (for identity, or unit),
 - the `left id`: attribute; usually referred to as Ul (for left identity, or unit), and
 - the `right id`: attribute; usually referred to as Ur (for right identity, or unit),

but no other equational attributes must be given for such symbols.

Explicitly excluded are theories with binary function symbols having any combination of:

- the `idem` attribute, or

- the `assoc` attribute without `comm`.

The reason for excluding the `assoc` attribute without `comm` is the already-mentioned fact that associative unification is not finitary.

If we give to Maude a unification problem in a functional module of the form `fmod` (Σ, Ax) `endfm` where Σ and Ax satisfy the above requirements, we get a complete set of order-sorted unifiers modulo the theory (Σ, Ax) . If, instead, we give the same problem to Maude in the functional module `fmod` $(\Sigma, E \cup Ax)$ `endfm`, then the equations E are ignored and we still get a complete set of order-sorted unifiers modulo the theory (Σ, Ax) . Similarly, if we provide the same unification problem in a functional theory `fth` $(\Sigma, E \cup Ax)$ `endfth`, a system module `mod` $(\Sigma, E \cup Ax, R)$ `endm` or a system theory `th` $(\Sigma, E \cup Ax, R)$ `endth`, we again get a complete set of order-sorted unifiers modulo the theory (Σ, Ax) . All this is consistent with the decomposition idea mentioned above: to deal with order-sorted $E \cup Ax$ -unification, other methods, that use the Ax -unification algorithm as a component, can later be defined as we explain in Section 15.6.

Maude is even more tolerant than this. The user can give to Maude a unification problem in a functional module (or functional theory, or system module, or system theory) of the form `fmod` $(\Sigma, E \cup M \cup Ax \cup Ax')$ `endfm` (or the analogous specification in the other cases), where (Σ, Ax) satisfies the conditions mentioned above, but M is an optional set of *membership axioms* (that is, $(\Sigma, E \cup M \cup Ax \cup Ax')$ can be a membership equational theory and not just an order-sorted theory), and the axioms Ax' violate those requirements (as explained in conditions (i)–(iii) above). Then what will happen is:

1. As before, the additional equations E (or rules R) are completely ignored, and the membership axioms M are likewise ignored.
2. If a unification problem involves the occurrence of a symbol satisfying axioms Ax' at the root position of a *non-ground subterm*, the unification process will *fail* and a warning will be printed.
3. If a unification problem involves the occurrence of symbols satisfying axioms Ax' , but all such occurrences are always in *ground* subterms of the problem, then this special case of $Ax \cup Ax'$ -unification is handled by Maude and the corresponding $Ax \cup Ax'$ -unifiers are returned.

Furthermore, the functional module `fmod` $(\Sigma, E \cup M \cup Ax \cup Ax')$ `endfm` (or the analogous functional theory or system module or theory) may import predefined modules such as `BOOL` or `NAT`, so that function symbols in such predefined modules can also be used in unification problems.

15.4 The `unify` command

Given a functional module or theory, or a system module or theory, $\langle ModId \rangle$, the user can give to Maude a unification command of the form:

```
unify [ n ] in <ModId> :
  <Term-1> =? <Term'-1> /\ ... /\ <Term-k> =? <Term'-k> .
```

where $k \geq 1$; n is an optional argument providing a bound on the number of unifiers requested, so that if the cardinality of the set of unifiers is greater than the specified bound, the unifiers beyond that bound are omitted; and $\langle ModId \rangle$ can be any module or theory declared in the current Maude session (as usual, if no module is mentioned, the current module is used).

For a simple example of syntactic order-sorted unification problem illustrating:

- the use of the `unify` command;
- the use of the predefined operator `_^_` in the `NAT` module, representing exponentiation on natural numbers; and
- the in general non-unitary nature of order-sorted unification,

we can define the module

```
fmod UNIFICATION-EX1 is
  protecting NAT .
  op f : Nat Nat -> Nat .
  op f : NzNat Nat -> NzNat .
  op f : Nat NzNat -> NzNat .
endfm
```

and then give to Maude the following command:

```
Maude> unify f(X:Nat, Y:Nat) ^ B:NzNat =? A:NzNat ^ f(Y:Nat, Z:Nat) .
```

```
Solution 1
X:Nat --> #1:Nat
Y:Nat --> #2:NzNat
B:NzNat --> f(#2:NzNat, #3:Nat)
A:NzNat --> f(#1:Nat, #2:NzNat)
Z:Nat --> #3:Nat
```

```
Solution 2
X:Nat --> #1:NzNat
Y:Nat --> #2:Nat
B:NzNat --> f(#2:Nat, #3:NzNat)
A:NzNat --> f(#1:NzNat, #2:Nat)
Z:Nat --> #3:NzNat
```

The next example in the same module illustrates the use of the `unify` command with a unification problem consisting of two unification equations:

```
Maude> unify f(X:Nat, Y:NzNat) =? f(Z:NzNat, U:Nat)
  /\ V:NzNat =? f(X:Nat, U:Nat) .
```

```
Solution 1
```

```

X:Nat --> #1:NzNat
Y:NzNat --> #2:NzNat
Z:NzNat --> #1:NzNat
U:Nat --> #2:NzNat
V:NzNat --> f(#1:NzNat, #2:NzNat)

```

Note that, as already mentioned, we could instead invoke the `unify` command in a functional or system module or theory having additional equations, memberships, or rules, which are always ignored. For example, we could have instead declared the system theory

```

th UNIFICATION-EX2 is
  protecting NAT .
  op f : Nat Nat -> Nat .
  op f : NzNat Nat -> NzNat .
  op f : Nat NzNat -> NzNat .
  eq f(f(N:Nat, M:Nat), K:Nat) = f(N:Nat, M:Nat) .
  rl f(N:Nat, M:Nat) => 0 .
endth

```

so that, if we give again the same `unify` commands above, we will obtain *exactly* the same sets of order-sorted unifiers as for the `UNIFICATION-EX1` module.

The above examples illustrate a further point about the form of the returned unifiers, namely, that in each assignment $X \rightarrow t$ in a unifier, the variables appearing in the term t are always *fresh* variables of the form $\#n:Sort$. The user is required *not to use* variables of this form in the submitted unification problem. A warning is printed if this requirement is violated:

```

Maude> unify in NAT : X:Nat ^ #1:Nat =? #2:Nat .
Warning: unsafe variable name #1:Nat in unification problem.

```

15.4.1 Associative-commutative (AC) unification examples

The use of a bound on the number of unifiers, as well as the use of the associative-commutative (AC) operator `+` in the predefined `NAT` module (see Section 9.2), plus the fact that even small AC-unification problems can generate a large number of unifiers are all illustrated by the following command:

```

Maude> unify [100] in NAT :
      X:Nat + X:Nat + Y:Nat =? A:Nat + B:Nat + C:Nat .

```

```

Solution 1
X:Nat --> #1:Nat + #2:Nat + #3:Nat + #5:Nat + #6:Nat + #8:Nat
Y:Nat --> #4:Nat + #7:Nat + #9:Nat
A:Nat --> #1:Nat + #1:Nat + #2:Nat + #3:Nat + #4:Nat
B:Nat --> #2:Nat + #5:Nat + #5:Nat + #6:Nat + #7:Nat
C:Nat --> #3:Nat + #6:Nat + #8:Nat + #8:Nat + #9:Nat
...

```

```

Solution 100
X:Nat --> #1:Nat + #2:Nat + #3:Nat + #4:Nat
Y:Nat --> #5:Nat
A:Nat --> #1:Nat + #1:Nat + #2:Nat
B:Nat --> #2:Nat + #3:Nat
C:Nat --> #3:Nat + #4:Nat + #4:Nat + #5:Nat

```

The following unification command in the predefined `CONVERSION` module (see Section 9.9) illustrates a further point on the handling of built-in constants and functions. Built-in constants, even though infinite in number (all strings, all quoted identifiers, all natural numbers, and so on), are handled and can be used in unification problems. But built-in functions are not considered built-in for unification purposes; therefore, built-in evaluation of such functions is *not* performed during the unification.

```

Maude> unify in CONVERSION :
      X:String < "foo" + Y:Char =?
      Z:String + string(pi) < "foo" + Z:String .

```

```

Solution 1
X:String --> #1:Char + string(pi)
Y:Char --> #1:Char
Z:String --> #1:Char

```

The handling of unification problems in modules with operators whose equational axioms are excluded from the current unification algorithm can be illustrated by means of the following module:

```

fmod UNIFICATION-EX3 is
  protecting NAT .
  op f : Nat Nat -> Nat [assoc] .
endfm

```

As already mentioned, a unification problem using such an associative function symbol `f` in a non-ground subterm will mean that the unification attempt fails and a warning is issued:

```

Maude> unify f(f(X:Nat, Y:Nat), Z:Nat) =? f(A:Nat, B:Nat) .
Warning: Term f(X:Nat, Y:Nat, Z:Nat) is non-ground and unification
for its top symbol is not currently supported.

```

Instead, if all symbols satisfying unsupported equational axioms Ax' only appear in *ground* subterms of the unification problem, the unification attempt succeeds with the correct set of order-sorted $Ax \cup Ax'$ -unifiers:

```

Maude> unify X:Nat + f(f(41, 42),43) =? Y:Nat + f(41,f(42,43)) .

```

```

Solution 1
X:Nat --> #1:Nat
Y:Nat --> #1:Nat

```

15.4.2 Unification examples with the iter attribute

The following example illustrates the efficiency of order-sorted unification modulo the `iter` theory (in this example in combination with the `comm` theory). Consider the following module:

```
fmod ITER-EXAMPLE is
  sorts NzEvenNat EvenNat OddNat NzNat Nat EvenInt OddInt NzInt Int .
  subsorts OddNat < OddInt NzNat < NzInt < Int .
  subsorts EvenNat < EvenInt Nat < Int .
  subsorts NzEvenNat < NzNat EvenNat < Nat .

  op 0 : -> EvenNat .

  op s : EvenNat -> OddNat [iter] .
  op s : OddNat -> NzEvenNat [iter] .
  op s : Nat -> NzNat [iter] .
  op s : EvenInt -> OddInt [iter] .
  op s : OddInt -> EvenInt [iter] .
  op s : Int -> Int [iter] .

  op _+_ : Int Int -> Int [comm gather (E e)] .
  op _+_ : OddInt OddInt -> EvenInt [ditto] .
  op _+_ : EvenInt EvenInt -> EvenInt [ditto] .
  op _+_ : OddInt EvenInt -> OddInt [ditto] .
  op _+_ : Nat Nat -> Nat [ditto] .
  op _+_ : Nat NzNat -> NzNat [ditto] .
  op _+_ : OddNat OddNat -> NzEvenNat [ditto] .
  op _+_ : NzEvenNat EvenNat -> NzEvenNat [ditto] .
  op _+_ : EvenNat EvenNat -> EvenNat [ditto] .
  op _+_ : OddNat EvenNat -> OddNat [ditto] .
endfm
```

We can then give the unification commands:

```
Maude> unify in ITER-EXAMPLE :
      s^1000000(X:OddNat) =? s^1000000000001(Y:Int) .
Decision time: 1ms cpu (1ms real)

Solution 1
X:OddNat --> s^99999000001(#1:EvenNat)
Y:Int --> #1:EvenNat
```

and

```
Maude> unify in ITER-EXAMPLE :
      s^1000000(X:OddNat) =? s^1000000000001(Y:Int + Z:Int + W:Int) .
Decision time: 2ms cpu (5ms real)

Solution 1
```

```

X:OddNat --> s^99999000001(#1:OddNat + (#2:OddNat + #3:EvenNat))
W:Int --> #1:OddNat
Z:Int --> #2:OddNat
Y:Int --> #3:EvenNat

```

```

Solution 2
X:OddNat --> s^99999000001(#1:OddNat + (#2:EvenNat + #3:OddNat))
W:Int --> #1:OddNat
Z:Int --> #2:EvenNat
Y:Int --> #3:OddNat

```

```

Solution 3
X:OddNat --> s^99999000001(#1:EvenNat + (#2:OddNat + #3:OddNat))
W:Int --> #1:EvenNat
Z:Int --> #2:OddNat
Y:Int --> #3:OddNat

```

```

Solution 4
X:OddNat --> s^99999000001(#1:EvenNat + (#2:EvenNat + #3:EvenNat))
W:Int --> #1:EvenNat
Z:Int --> #2:EvenNat
Y:Int --> #3:EvenNat

```

As already mentioned, assuming that no bound on the number of unifiers is specified by the user, Maude will always compute a *complete* set of order-sorted unifiers modulo Ax , for Ax the supported equational axioms described in Section 15.3. However, there is no guarantee that the computed set of unifiers is *minimal*, that is, some of the unifiers in the computed set may be redundant, since they could be obtained as instances (modulo Ax) of other unifiers in the set.

15.4.3 Associative-commutative with identity (*ACU*) unification examples

To illustrate the use of the unification command in the presence of *ACU* operators, let us consider yet another version of the vending machine example (first presented in Section 6.1 and in other sections of this document in different forms):

```

mod UNIF-VENDING-MACHINE is
  sorts Coin Item Marking Money State .
  subsort Coin < Money .
  op empty : -> Money .
  op _ : Money Money -> Money [assoc comm id: empty] .
  subsort Money Item < Marking .
  op _ : Marking Marking -> Marking [assoc comm id: empty] .
  op <_> : Marking -> State .
  ops $ q : -> Coin .

```

```

ops a c : -> Item .
var M : Marking .
rl [buy-c] : < M $ > => < M c > .
rl [buy-a] : < M $ > => < M a q > .
eq [change]: q q q q = $ .
endm

```

We can ask whether there is an equational unifier of two configurations, one containing at least two quarters, and another containing at least one dollar.

```

Maude> unify in UNIF-VENDING-MACHINE :
      < q q X:Marking > =? < $ Y:Marking > .

```

```

Solution 1
X:Marking --> $
Y:Marking --> q q

```

```

Solution 2
X:Marking --> $ #1:Marking
Y:Marking --> q q #1:Marking

```

Notice that the computed set of unifiers is not minimal because the first solution is the instance of the second obtained by substituting the variable `#1:Marking` with the constant `empty`. However, Maude computes a minimal set of unifiers modulo axioms when invoked via the variant-based unification command of Section 15.9 below; for instance, take the previous command above and add the word “`variant`” at the beginning to obtain the minimal set (see Section 15.9 for details on how to invoke the command):

```

Maude> variant unify in UNIF-VENDING-MACHINE :
      < q q X:Marking > =? < $ Y:Marking > .

```

```

Solution 1
X:Marking --> $ #1:Marking
Y:Marking --> q q #1:Marking

```

Recall that memberships are discarded completely. For instance, if we modify the previous example to include a membership definition for a new sort `Quarter`, any unification call with that sort may not succeed.

```

mod UNIF-VENDING-MACHINE-MB is
  sorts Coin Item Marking Money State .
  subsort Coin < Money .
  op empty : -> Money .
  op _ : Money Money -> Money [assoc comm id: empty] .
  subsort Money Item < Marking .
  op _ : Marking Marking -> Marking [assoc comm id: empty] .
  op <_> : Marking -> State .
  ops $ q : -> Coin .
  ops a c : -> Item .

```

```

sort Quarter .
subsort Quarter < Coin .
mb q : Quarter .

var M : Marking .
rl [buy-c] : < M $ > => < M c > .
rl [buy-a] : < M $ > => < M a q > .
eq [change]: q q q q = $ .
endm

```

We can ask whether there is an equational unifier of two configurations, one containing at least two quarters, and another containing two quarters and a dollar, but it fails:

```

Maude> unify in UNIF-VENDING-MACHINE-MB :
      < q q X:Marking > =? < $ Y:Quarter Z:Quarter > .

```

No unifier.

despite the fact that instantiating both *Y* and *Z* to *q* is part of a solution in the unification call above. The reason is that the membership is not used during *ACU* unification and therefore the algorithm unification treats the sort *Quarter* as empty.

15.4.4 Unification examples with an identity symbol

Let us illustrate the use of the different combinations of the identity attribute for unification. Let us consider first a module using the *left-id* attribute.

```

mod LEFTID-UNIFICATION-EX is
  sorts Magma Elem .
  subsorts Elem < Magma .
  op _ : Magma Magma -> Magma [gather (E e) left id: e] .
  ops a b c d e : -> Elem .
endm

```

Then the following two unification problems have a different meaning, where we have swapped the position of the variables. First, when we unify two terms where variables of sort *Magma* are at the left of the terms, we have both a syntactic unifier and a unifier modulo identity.

```

Maude> unify in LEFTID-UNIFICATION-EX : X:Magma a =? Y:Magma a a .

```

```

Solution 1
X:Magma --> a
Y:Magma --> e

```

```

Solution 2

```

```
X:Magma --> #1:Magma a
Y:Magma --> #1:Magma
```

When the variables are instead at the right side of the terms of sort `Magma`, there is clearly no unifier, since the term `a a Y:Magma` is parsed as `(a a) Y:Magma` in module `LEFTID-UNIFICATION-EX` due to the attribute `gather (E e)` (see Section 3.9).

```
Maude> unify in LEFTID-UNIFICATION-EX : a X:Magma =? a a Y:Magma .
No unifier.
```

Consider now a similar module but for the right identity.

```
mod RIGHTID-UNIFICATION-EX is
  sorts Magma Elem .
  subsorts Elem < Magma .
  op _ : Magma Magma -> Magma [gather (e E) right id: e] .
  ops a b c d e : -> Elem .
endm
```

When we unify two terms where variables of sort `Magma` are at the left of the terms, there is clearly no unifier, since the term `Y:Magma a a` is parsed this time as `Y:Magma (a a)` in module `RIGHTID-UNIFICATION-EX` due to the attribute `gather (e E)` (see Section 3.9).

```
Maude> unify in RIGHTID-UNIFICATION-EX : X:Magma a =? Y:Magma a a .
No unifier.
```

When the variables are instead at the right side of the terms of sort `Magma`, we have both a syntactic unifier and a unifier modulo identity:

```
Maude> unify in RIGHTID-UNIFICATION-EX : a X:Magma =? a a Y:Magma .
```

```
Solution 1
X:Magma --> a
Y:Magma --> e
```

```
Solution 2
X:Magma --> #1:Magma a
Y:Magma --> #1:Magma
```

Consider now a similar module but with the identity attribute.

```
mod ID-UNIFICATION-EX is
  sorts Magma Elem .
  subsorts Elem < Magma .
  op _ : Magma Magma -> Magma [gather (E e) id: e] .
  ops a b c d e : -> Elem .
endm
```

When we unify two terms where variables of sort `Magma` are at the left of the terms, we have both a syntactic unifier and a unifier modulo identity:

```
Maude> unify in ID-UNIFICATION-EX : X:Magma a =? Y:Magma a a .
```

```
Solution 1
```

```
X:Magma --> a
```

```
Y:Magma --> e
```

```
Solution 2
```

```
X:Magma --> #1:Magma a
```

```
Y:Magma --> #1:Magma
```

When the variables of sort `Magma` are instead at the right side of the terms of sort `Magma`, we only have a unifier modulo identity:

```
Maude> unify in ID-UNIFICATION-EX : a X:Magma =? a a Y:Magma .
```

```
Solution 1
```

```
X:Magma --> a
```

```
Y:Magma --> e
```

And finally, when we add commutativity, we obtain slightly different results.

```
mod COMM-ID-UNIFICATION-EX is
  sorts Magma Elem .
  subsorts Elem < Magma .
  op _ : Magma Magma -> Magma [gather (E e) comm id: e] .
  ops a b c d e : -> Elem .
endm
```

When we unify two terms where variables of sort `Magma` are at the left of the terms, we have both a syntactic unifier and a unifier modulo identity and commutativity, but the latter is duplicated:

```
Maude> unify in COMM-ID-UNIFICATION-EX : X:Magma a =? Y:Magma a a .
```

```
Solution 1
```

```
X:Magma --> a
```

```
Y:Magma --> e
```

```
Solution 2
```

```
X:Magma --> a #1:Magma
```

```
Y:Magma --> #1:Magma
```

```
Solution 3
```

```
X:Magma --> a
```

```
Y:Magma --> e
```

When the variables of sort `Magma` are instead at the right side of the terms of sort `Magma`, we have several unifiers modulo identity and commutativity:

```
Maude> unify in COMMID-UNIFICATION-EX : a X:Magma =? a a Y:Magma .
```

```
Solution 1
X:Magma --> a a
Y:Magma --> a
```

```
Solution 2
X:Magma --> a
Y:Magma --> e
```

```
Solution 3
X:Magma --> a
Y:Magma --> e
```

Note that the first solution is intriguing and is obtained by unifying terms $(X:Magma\ a)$ and $((a\ a)\ Y:Magma)$.

15.5 Unification at the metalevel: metaUnify and metaDisjointUnify

Following the general Maude philosophy, all Maude functionality is as much as possible moved up to the metalevel, so that it becomes available by reflection (see Chapter 14). This is particularly important for unification for two reasons:

1. Many of the formal reasoning applications of unification do require access to unification functions at the metalevel. Consider, for example, the computation of critical pairs to determine if a functional module is locally confluent. This will be done by a function that takes the metarepresentation of the given functional module as data, and then this function will have to invoke the unification function at the metalevel as part of its critical pair subcomputations.
2. The unification algorithm is *theory-dependent*, since a *different* order-sorted unification algorithm is derived for each different signature Σ and combination of axioms Ax . Therefore, unification as a *function*, instead of as a user *command*, must necessarily be at the metalevel, since it must take the given theory (Σ, Ax) as a parameter.

Thus, the unification command is reflected in the `META-LEVEL` module (see Section 14.5) by two descent functions:

```
op metaUnify :
  Module UnificationProblem Nat Nat ~> UnificationPair?
  [special (...)] .
```

```

op metaDisjointUnify :
  Module UnificationProblem Nat Nat ~> UnificationTriple?
  [special (...)] .

```

These two metalevel functions work on unification problems constructed by means of the following signature:

```

sorts UnificandPair UnificationProblem .
subsort UnificandPair < UnificationProblem .
op _=?_ : Term Term -> UnificandPair [ctor prec 71] .
op _/\_ : UnificationProblem UnificationProblem -> UnificationProblem
  [ctor assoc comm prec 73] .

```

The key difference between `metaUnify` and `metaDisjointUnify` is that the latter assumes that the variables in the left and righthand unificands are to be considered *disjoint* even when they are not so, and it generates each solution to the given unification problem not as a single substitution, but as a *pair* of substitutions, one for left unificands and the other for right unificands. This functionality is very useful for applications, such as critical-pair checking or narrowing, where a disjoint copy of the terms or rules involved must always be computed before unification is performed. Indeed, what the `metaDisjointUnify` operation avoids is precisely the need for explicitly computing such disjoint copies. The need for two substitutions in each solution is then obvious, since the terms in the given unification problem need not be made explicitly disjoint, but their (accidentally) common variables must be treated differently, as if they were disjoint.

Since it is convenient to *reuse* variable names from unifiers in new problems, for example in narrowing, this is allowed via the third argument, which is the largest number n appearing in a unificand variable of the form $\#n:Sort$. Then the fresh variables in the computed unifiers will all be numbered from $n + 1$ on.

As is usual for descent functions, the last argument in the function is used to select which result is wanted, starting from 0. Caching is used so that if unifier i has just been returned, requesting unifier $i + 1$ gives rise to an incremental computation.

Results are returned using the following constructors:

```

subsort UnificationPair < UnificationPair? .
subsort UnificationTriple < UnificationTriple? .
op {_,_} : Substitution Nat -> UnificationPair [ctor] .
op {_,_,_} : Substitution Substitution Nat -> UnificationTriple
  [ctor] .

```

as appropriate for the descent function. The final `Nat` component is the largest n occurring in a fresh metavariable of the form $\#n:Sort$. In this way, when we want to reuse variable names from unifiers, the next invocation of the function

can use this parameter to make sure that the new variables generated are always fresh.

When no unifier with a given index exists the constant

```
op noUnifier : -> UnificationPair? [ctor] .
```

or, respectively, the constant

```
op noUnifier : -> UnificationTriple? [ctor] .
```

is returned as appropriate for the corresponding descent function.

We can illustrate the use of these metalevel functions with a few examples. The first one comes from the previous section, but moved up at the metalevel:

```
Maude> reduce in META-LEVEL :
  metaUnify(upModule('UNIFICATION-EX1, false),
    'f['X:Nat, 'Y:NzNat] =? 'f['Z:NzNat, 'U:Nat] /\
    'V:NzNat =? 'f['X:Nat, 'U:Nat], 0, 0) .

result UnificationPair:
  {'U:Nat <- '#1:NzNat ;
   'V:NzNat <- 'f['#2:NzNat, '#1:NzNat] ;
   'X:Nat <- '#2:NzNat ;
   'Y:NzNat <- '#1:NzNat ;
   'Z:NzNat <- '#2:NzNat, 2}
```

The second example shows that we can request fresh variables with arbitrarily large numbering:

```
Maude> reduce in META-LEVEL :
  metaUnify(upModule('NAT, false),
    '[_+'X:Nat, 'Y:Nat] =? '[_+'A:Nat, 'B:Nat],
    10000000000000000000, 0) .

result UnificationPair:
  {'A:Nat <- '[_+'#100000000000000000001:Nat,
   '#1000000000000000000002:Nat] ;
   'B:Nat <- '[_+'#1000000000000000000003:Nat,
   '#1000000000000000000004:Nat] ;
   'X:Nat <- '[_+'#1000000000000000000001:Nat,
   '#1000000000000000000003:Nat] ;
   'Y:Nat <- '[_+'#1000000000000000000002:Nat,
   '#1000000000000000000004:Nat],
  10000000000000000004}
```

The following example shows a similar unification problem but with much smaller numberings in fresh variables, and now involving an invocation of metaDisjointUnify.

```

Maude> reduce in META-LEVEL :
      metaDisjointUnify(upModule('NAT, false),
        '[_+_['X:Nat, 'Y:Nat] =? '[_+_['X:Nat, 'B:Nat], 0, 0) .

result UnificationTriple: {
  'X:Nat <- '[_+_['#1:Nat, '#2:Nat] ;
  'Y:Nat <- '[_+_['#3:Nat, '#4:Nat],
  'B:Nat <- '[_+_['#1:Nat, '#3:Nat] ;
  'X:Nat <- '[_+_['#2:Nat, '#4:Nat], 4}

```

Yet another example shows how using variable names in unification problems with larger numbers than declared by the third argument generates a warning and no reduction.

```

Maude> reduce in META-LEVEL :
      metaUnify(upModule('NAT, false),
        '[_+_['X:Nat, 'Y:Nat] =? '[_+_['#1:Nat, 'Y:Nat], 0, 0) .

Warning: unsafe variable name #1:Nat in unification problem.

result [UnificationPair?]:
  metaUnify(th 'NAT is
    including 'NAT .
    sorts none .
    none
    none
    none
    none
    none
  endth,
  '[_+_['X:Nat, 'Y:Nat] =? '[_+_['#1:Nat, 'Y:Nat], 0, 0)

```

15.6 Some applications of unification

In this section we review briefly some applications that can be developed using a unification infrastructure like the one described in this chapter. We begin by discussing narrowing and narrowing-based unification algorithms. We also explain how narrowing modulo an equational theory can be used for reachability analysis of concurrent systems described by rewrite theories, and, more generally, for symbolic temporal logic model checking of such systems. We then discuss briefly other automated deduction applications, including theorem proving ones.

Chapter 20 contains more information about an implementation of narrowing in Full Maude following the ideas of Section 15.6.2 below.

15.6.1 Narrowing-based unification

If we have a dedicated algorithm (as the one supported by Maude) to solve unification problems in an order-sorted theory (Σ, Ax) , then we can use it as a component to obtain a unification algorithm for theories of the form $(\Sigma, E \cup Ax)$, provided the equations E are unconditional, coherent, confluent and terminating modulo Ax [200].

The technique used under such conditions to obtain an $E \cup Ax$ -unification algorithm from an Ax -unification algorithm is called *narrowing*, and is the obvious generalization of term rewriting to handle logical variables and perform a kind of symbolic execution. In ordinary term rewriting, if we want to apply a rewrite rule, say $l \rightarrow r$, to a term t at position p , the subterm $t|_p$ must be an *instance* of the lefthand side l , that is, there must be a substitution σ such that $t|_p = \sigma(l)$. Instead, in narrowing we can apply the rule $l \rightarrow r$ at a non-variable position p in t , provided the unification problem $t|_p =? l$ (where the variables of l and t are assumed disjoint) has a nonempty set of unifiers. For any such unifier θ we then *narrow* the original term t to the substitution instance under θ of $t[r]_p$. We then write

$$t \rightsquigarrow \theta(t[r]_p)$$

for such a narrowing step. For example, in the standard, unsorted specification of the natural numbers, we can use the equation $x + s(y) = s(x + y)$ as a rewrite rule to narrow the term $x' * (y' + z')$ at position 2 with substitution $\theta = \{x \mapsto y'', y \mapsto z'', y' \mapsto y'', z' \mapsto s(z'')\}$ to get the narrowing step

$$x' * (y' + z') \rightsquigarrow x' * s(y'' + z'').$$

In this example, θ is the most general unifier for the syntactic unification problem $y' + z' =? x + s(y)$. However, in the same way as we can perform rewriting modulo a set of axioms Ax if we have an Ax -matching algorithm, we can likewise perform *narrowing modulo a set Ax of axioms* if we have an Ax -unification algorithm. That is, the unification problems $t|_p =? l$ are now solved, not by syntactic unification, but by Ax -unification.

If a theory $(\Sigma, E \cup Ax)$ satisfies the above coherence, confluence, and termination modulo Ax requirements, we can systematically reduce $E \cup Ax$ -unification problems to narrowing problems as follows:

1. we add a fresh new sort **Truth** to Σ with a constant **tt**;
2. for each top sort of each connected component of sorts we add a binary predicate **eq** of sort **Truth** and add to E the equation $\mathbf{eq}(x, x) = \mathbf{tt}$, where x has such a top sort;
3. we then reduce an $E \cup Ax$ -unification problem $t =? t'$ to the narrowing reachability problem

$$\mathbf{eq}(t, t') \rightsquigarrow^* \mathbf{tt}$$

modulo Ax in the theory extending $(\Sigma, E \cup Ax)$ with these new sorts, operators, and equations, where E and the new equations are used as rewrite rules.

That is, we search for all narrowing paths modulo Ax from $\text{eq}(t, t')$ to tt . Each such path then gives us a unifier of the equation $t =^? t'$, just by composing the unifiers of each narrowing step in the path. As explained in [31], narrowing can be performed not only in order-sorted equational theories, but also in membership equational theories.

The just-described computation of $E \cup Ax$ -unifiers by narrowing modulo Ax yields a complete but in general infinite set of $E \cup Ax$ -unifiers. For the case when $Ax = \emptyset$, some sufficient conditions are known ensuring termination of the basic narrowing strategy (see, e.g., [196, 11, 9]), and therefore ensuring that the complete set of $E \cup Ax$ -unifiers computed by basic narrowing is finite. However, for commonly occurring sets of axioms Ax , such as associativity-commutativity (AC), it is well-known that narrowing modulo AC “almost never terminates” and, furthermore, that narrowing strategies facilitating termination such as basic narrowing are incomplete [373, 82]. Based on the idea of “variants” in [82], a complete, yet quite efficient in terms of its search space, narrowing strategy modulo Ax called *folding variant narrowing* has been proposed in [149, 151]. Furthermore, in [148, 151, 57] sufficient checkable conditions on $(\Sigma, E \cup Ax)$ have been given ensuring that the $E \cup Ax$ -unification algorithm provided by folding variant narrowing modulo Ax is finitary. In Maude 2.6, a narrowing library developed by Santiago Escobar implemented the folding variant narrowing as a component, making $E \cup Ax$ -unification available as part of Full Maude. Instead, Maude 2.7 implements $E \cup Ax$ -unification via folding variant narrowing directly in Core Maude’s C++ level (see Section 15.10).

15.6.2 Symbolic reachability analysis in rewrite theories

A rewrite theory³, say $\mathcal{R} = (\Sigma, E \cup Ax, R)$, specified in Maude as a system module, describes a concurrent system whose states are $E \cup Ax$ -equivalence classes of ground terms, and whose local concurrent transitions are specified by the rules R . When formally analyzing the properties of \mathcal{R} , an important problem is ascertaining for specific patterns t and t' the following *symbolic reachability problem*:

$$\exists X t \longrightarrow^* t'$$

with X the set of variables appearing in t and t' , which for this discussion we may assume are a disjoint union of those in t and those in t' . That is, t and t' symbolically describe sets of concurrent states $\llbracket t \rrbracket$ and $\llbracket t' \rrbracket$ (namely, all the ground substitution instances of t , resp. t' , or, more precisely, the $E \cup Ax$ -equivalence classes associated to such ground instances). And we are asking: is there a state in $\llbracket t \rrbracket$ from which we can *reach* a state in $\llbracket t' \rrbracket$ after a finite number of rewriting steps?

³ All we say here applies also to rewrite theories with an additional freezing function ϕ specifying which arguments of each function symbol are *frozen*.

For example, \mathcal{R} may specify a cryptographic protocol, t may symbolically describe a set of initial states, and t' may likewise describe a set of *attack states*. Then, if the above reachability question can be answered in the affirmative, the protocol \mathcal{R} is insecure against the kinds of attacks described by t' . Furthermore, if the way of answering the reachability question is somehow constructive, we should be able to exhibit a concrete attack as a rewrite sequence violating the security of the protocol.

As explained in [266], provided the rewrite theory $\mathcal{R} = (\Sigma, E \cup Ax, R)$ is *topmost* (that is, all rewrites take place at the root of a term), or, as in the case of *AC* rewriting of object-oriented systems, \mathcal{R} is “essentially topmost,” and the rules R are coherent with E modulo Ax , narrowing with the rules R modulo the equations $E \cup Ax$ gives a constructive, sound, and complete method to solve reachability problems of the form $\exists X t \longrightarrow^* t'$, that is, such a problem has an affirmative answer if and only if we can find a finite narrowing sequence modulo $E \cup Ax$ of the form $t \rightsquigarrow^* \theta(t')$ for some θ . The method is *constructive*, because instantiating t with the composition of the unifiers for each step in the narrowing sequence gives us a concrete rewrite sequence witnessing the existential formula.

Of course, narrowing with R modulo $E \cup Ax$ requires performing $E \cup Ax$ -unification at each narrowing step. As explained in Section 15.6.1, $E \cup Ax$ -unification can itself be performed by narrowing with the equations E modulo Ax , provided E is coherent, confluent, and terminating modulo Ax . Therefore, in performing symbolic reachability analysis in a rewrite theory $\mathcal{R} = (\Sigma, E \cup Ax, R)$ there are usually *two* levels of narrowing and *two* levels of unification: narrowing with R modulo $E \cup Ax$ for reachability, and narrowing with E modulo Ax for unification purposes. Similarly, unification modulo $E \cup Ax$ is performed by narrowing, while unification modulo Ax is usually performed in a built-in way.

This is exactly the approach taken in the Maude-NPA protocol analyzer [144, 146], where cryptographic protocols are formally specified as rewrite theories of the form $\mathcal{R} = (\Sigma, E \cup Ax, R)$, and the formal reachability analysis is performed in a *backwards* way, from an attack state to an initial state. This just means that we perform standard (forwards) reachability analysis with the rewrite theory $\mathcal{R}^{-1} = (\Sigma, E \cup Ax, R^{-1})$, where $R^{-1} = \{r \longrightarrow l \mid (l \longrightarrow r) \in R\}$. The equational theory $E \cup Ax$ typically specifies the algebraic properties of the cryptographic functions used in the given protocol, for example, public key encryption and decryption, exclusive or, modular exponentiation, and so on. Reasoning modulo such algebraic properties is very important to gain high levels of assurance, since it is well-known that some cryptographic protocols that can be proved secure under the standard Dolev-Yao model, in which the cryptographic functions are treated as a “black box,” can actually be broken by an attacker that makes clever use of the algebraic properties of the cryptographic functions of the protocol. Besides using narrowing with rules modulo equations, the Maude-NPA tool uses several state space reduction techniques, including grammars that can describe sets of unreachable states

that need not be explored [144], to drastically reduce the narrowing search space, often from an infinite set of states to a finite set of them, so that finite failure to find an attack becomes an actual proof of security.

Given a rewrite theory $\mathcal{R} = (\Sigma, E \cup Ax, R)$, we may be interested in verifying properties more general than existential questions of the form $\exists X t \longrightarrow^* t'$. Note that we can view such questions as questions about the *violation of an invariant*, because we can regard the set of states $\llbracket t' \rrbracket$ as the *complement* of an invariant set of states, say I , which can be easily specified by an equationally-defined predicate. That is, proving the existential formula $\exists X t \longrightarrow^* t'$ is the same thing as finding a counterexample for the assertion $\mathcal{R}, t \models \Box I$. This is just a temporal logic satisfaction assertion (see Chapter 13), but with the following nonstandard features: (i) the term t does not describe a single initial state, but a possibly infinite set $\llbracket t \rrbracket$ of initial states; and (ii) there is no guarantee that the set of reachable states is finite. Therefore, standard model-checking techniques may not be usable, because of a possible double infinity: in the number of initial states, and in the number of states reachable for each of those initial states. One can also generalize the above reachability question $\mathcal{R}, t \models \Box I$ to questions of the form $\mathcal{R}, t \models \varphi$, with φ a temporal logic formula. The papers [147, 14] show how narrowing can be used (again, both at the level of transitions with rules R and at the level of equations E) to perform *logical model checking* to verify such temporal logic formulas; this is a kind of *symbolic model checking* not in the binary decision diagram sense of “symbolic,” which still remains finite-state, but in a much more general sense in which possibly infinite sets of states are finitely described by patterns with logical variables.

15.6.3 Other automated deduction applications

The automated deduction application *par excellence*, and the one that historically, thanks to Alan Robinson, gave rise to the unification notion is *resolution-based theorem proving* [315]. Subsequent work by Gordon Plotkin [302] made it clear that not just syntactic unification, but unification *modulo* a set of equational axioms Ax is a very useful mechanism supporting theorem proving. Indeed, state-of-the-art resolution-based theorem provers routinely support unification modulo commonly occurring equational theories such as AC . Of course, the use of equational unification need not be restricted to resolution-based theorem provers. For example, the paper [316] shows how narrowing with sequent rules and equational unification can be used in a sequent-based theorem prover in which one can reason *modulo* both the equivalences given by the structural rules for sequents and also Boolean equivalences between formulas.

Yet another important application area is that of formal reasoning methods such as Knuth-Bendix equational completion (and its associated “inductionless induction” theorem-proving methods), checking local confluence of rewrite rules, and checking coherence of a set of rewrite rules with respect to a set

of equations [377]. In all these formal reasoning methods one needs to compute *critical pairs* by unification of a term with a subterm of another term. In particular, tools such as the Maude Church-Rosser Checker (CRC) and Coherence Checker (ChC) (see Sections 23.1.3 and 23.1.4) were before restricted to theories where the only equational axiom supported was commutativity. The present built-in support for unification modulo a wide set Ax of axioms (further extensible to identity axioms by narrowing as explained in Section 15.6.1) has made it possible to have much more general versions of the Maude Church-Rosser Checker and Coherence Checker tools to reason about the confluence and coherence of Maude specifications modulo equational axioms specified as equational attributes in Maude modules and theories [127, 126, 128].

15.7 Endogenous vs. exogenous order-sorted unification algorithms

The current Maude order-sorted unification algorithm modulo axioms Ax is what we might call an *endogenous* algorithm, in the sense that the computation of order-sorted unifiers is intimately integrated with the order-sorted reasoning process, so that unifiers that do not type under the order-sorted typing restrictions are never generated. This makes such an algorithm typically more efficient, because the order-sorted typing restrictions may drastically cut the number of generated unifiers, particularly modulo axioms such as AC where the number of unsorted unifiers can be very large. That is, order-sorted unification, even though it lacks the unitary property of unsorted syntactic unification and is in general more expensive than unsorted unification in the syntactic case, can often be more efficient in the modulo Ax case because of the drastic reductions that can be achieved by order-sorted typing restrictions in the number of Ax -unifiers. Moreover, even in the syntactic case, the efficiency of deductive processes that use order-sorted unification can substantially increase, because order-sorted unification will *fail* more often than unsorted unification, leading to smaller search spaces.

However, from the early papers on order-sorted unification such as, e.g., [326, 254, 331] a more modular, although typically less efficient, approach to order-sorted unification, which we might call *exogenous* has been known. The basic idea is to *reuse* an existing unsorted unification algorithm modulo some axioms Ax (under some conditions on Ax) to compute order-sorted Ax -unifiers in the following way:

1. type information is removed from the order-sorted Ax -unification problem to convert it into an unsorted Ax -unification problem;
2. a complete set of unsorted Ax -unifiers is computed; and
3. the order-sorted Ax -unifiers of the original problem are obtained from the unsorted ones by a process of *filtering* the unsorted unifiers through an order-sorted reasoning process, in which the sorts of the variables in

the original problem are taken into account. Each order-sorted unifier thus obtained is always a *specialization* of a corresponding unsorted one, where the unsorted variables have been specialized to given sorts; however, some unsorted unifiers, perhaps many, may be filtered out by this process and have no corresponding order-sorted unifiers.

For a state-of-the art study of the exogenous approach, allowing very general axioms Ax and proving the correctness of an order-sorted inference system to generate the order-sorted unifiers from the unsorted ones, see [189].

Both the endogenous and the exogenous approaches have their own advantages and disadvantages. The endogenous approach is more efficient, but it requires dedicated algorithms and implementations, so that unsorted unification algorithms and tools cannot be reused. The exogenous algorithms are less efficient because: (i) they can generate many unifiers that may later be discarded; (ii) a separate order-sorted filtering process is needed; and (iii) changes of representation, and even parsing, are required between unsorted and order-sorted representations (particularly when existing unsorted algorithms are reused). However, they are more modular and flexible, so that one can with relatively little effort obtain an order-sorted unification algorithm from an unsorted one.

In Maude we have experimented with, and benefited from, both an exogenous algorithm and the current endogenous one. The exogenous algorithm was developed in collaboration with Evelyn Contejean and Claude Marché from Université Paris-Sud, and involved also the efforts of Prasanna Thati and Joe Hendrix at UIUC. It reused the rich library of unsorted unification algorithms modulo axioms of the CiME system [84], which could be called from Maude in an experimental version. Inside Maude, it used the order-sorted inference system to compute order-sorted unifiers developed by Joe Hendrix and described in [189].

This exogenous algorithm has been extensively used in a previous version of the Maude-NPA tool, and has been shown effective in finding attacks to cryptographic protocols modulo nontrivial equational theories of the form $E \cup Ax$ [142]. The exogenous algorithm has also been extremely useful in *testing* the endogenous one. Because of the large number of unifiers generated and the complex nature of semantic unification algorithms, their testing is a non-trivial matter, and the automation of such testing is quite difficult. Thanks to the exogenous algorithm, and through the efforts of Ralf Sasse and Santiago Escobar, it has been possible to generate large numbers of random unification problems of different sizes in which the sets of unifiers generated by the exogenous and endogenous order-sorted unification algorithms have been automatically compared. This testing uncovered several bugs in an earlier alpha version of the Maude endogenous algorithm, and has also served to evaluate in practice the greater efficiency of the endogenous algorithm developed by Steven Eker.

15.8 Some notes on the implementation of unification

Order-sorted unification is NP-complete in general because Boolean algebra can be encoded as an order-sorted free theory signature and hence satisfiability can be reduced to an order-sorted free theory unification problem. In practice, reasonable performance can be obtained using a binary decision diagram technique to compute sorts for free variables occurring in unsorted unifiers. Furthermore in the AC case, sort information can be pushed into the unsorted unification algorithm and used to prune the Diophantine basis and the choice of subsets drawn from such a basis [137].

The unification theory combination framework and AC -unification algorithm are based on [30], while the Diophantine system solver used by the AC algorithm is based on [83]. The unification algorithm has been thoroughly tested by Santiago Escobar and Ralf Sasse using CiME [84] as an oracle, and has shown better average performance than CiME on the same problems.

The addition of ACU to the theories handled by the dedicated unification algorithm in Maude required substantial changes to the unification infrastructure implemented in previous versions of Maude for C and AC theories because of the problems associated with collapse theories. In this section we give an overview of the techniques used and highlight a novel algorithm for selecting sets of Diophantine basis elements during the computation of ACU unifiers.

15.8.1 Combining unification algorithms

The basic approach to solving unification problems where function symbols are drawn from more than one theory is variable abstraction where *alien subterms*, i.e., subterms headed by a symbol from a theory different from that of the top symbol of the parent term, are replaced by fresh variables to form *pure* unification subproblems which only involve variables and function symbols from a single theory and which can be passed to a unification algorithm for such a theory. Proving termination of combinations of algorithms is nontrivial, as variables are necessarily shared between theories and the unification of variables in one theory can create new unification subproblems in another theory, potentially ad infinitum. Stickel's algorithm [347], which combined the AC and free theories, required an elaborate termination proof by Fages [152]. Boudet et al. [30] proposed a much simpler approach where all unification subproblems and variable bindings in a given theory are solved (and re-solved if another subproblem in that theory is created) simultaneously. This method requires a simultaneous E -unification algorithm for each theory E and was the method implemented in Maude for C , AC , and \emptyset prior to the addition of ACU .

Collapse theories add two major complications to the combination of unification algorithms. Firstly, theory clashes where two terms with top symbols from different theories are required to unify can no longer be treated as a

failure, since if one of the top symbols belongs to a collapse theory, a collapse may occur, yielding solutions. Secondly, *compound cycles*, that is, problems of the form $x_1 =^? t_1(\dots, x_2, \dots), x_2 =^? t_2(\dots, x_3, \dots), \dots, x_n =^? t_n(\dots, x_1, \dots)$ where the terms t_i are pure in different theories, can no longer be treated as failure, since solutions may be possible via collapse.

Several authors have proposed combination schemes that can handle collapse theories. We use a simplified version of an algorithm due to Boudet [29]. The original algorithm also handles nonregular theories but we omit that capability to simplify the implementation. The key idea is that each theory E needs a restricted simultaneous E -unification algorithm which solves the simultaneous unification problem for pure equations that are pure in E but where certain variables may be marked as only being allowed to unify with other variables. A theory clash subproblem $f(\dots) =^? g(\dots)$, is split into a disjunction of two subproblems each of which is a conjunction $x =^? f(\dots) \wedge x =^? g(\dots)$ where x is a fresh variable. In one subproblem x is marked in the f equation and in the other subproblem x is marked in the g equation; either or both branches of the search may return solutions. Restricted unification is also used to break compound cycles. Because we do not handle nonregular theories, Boudet-style variable-elimination algorithms are unnecessary.

Boudet's algorithm assumes that theories are disjoint, i.e., that they do not share function symbols. Because in Maude this is not quite true — identities can contain symbols from other theories — we need to handle a special kind of variable elimination. We illustrate the issue with the following example:

```
fmod UNIFICATION-CYCLE is
  sort S .
  vars X Y : S .
  ops a b c d : -> S .
  op f : S S -> S [assoc comm id: g(c, d)] .
  op g : S S -> S [assoc comm id: f(a, b)] .
endfm

Maude> unify X =? f(Y, a, b) /\ Y =? g(X, c, d) .
```

Here the unification problem would already be in solved form but for the compound cycle formed by the X and Y variables. Restricted unification cannot break this cycle, since neither of the righthand sides can collapse out of their theory. However, putting $Y = g(c, d)$ eliminates Y from the first equation yielding $X = f(a, b)$ which eliminates X from the second equation, thus yielding a solution. This situation is somewhat pathological in Maude programs, and we do not really care about performance in its handling. Maude handles it by looking for this kind of cyclic dependency between theories when the signature is preprocessed and setting a flag so that a brute force variable elimination algorithm will be used to try and break compound cycles at unification time.

15.8.2 Diophantine basis element selection

We solve restricted simultaneous *ACU* unification using an extension of the simultaneous *AC* unification algorithm in [30]. For an *ACU* function symbol f we are presented with a set of flattened pure equations that take the form $f(x_1^{p_1}, \dots, x_n^{p_n}) =? f(y_1^{q_1}, \dots, y_m^{q_m})$ or $x_1 =? f(y_1^{q_1}, \dots, y_m^{q_m})$. Each f -equation yields a Diophantine equation $p_1 X_1 + \dots + p_n X_n = q_1 Y_1 + \dots + q_m Y_m$ or respectively, $X_1 = q_1 Y_1 + \dots + q_m Y_m$ where the X_i 's and Y_i 's are non-negative Diophantine variables. If an original variable is marked in some equation, the corresponding Diophantine variable receives an upper-bound of 1. Also, we may be able to obtain an upper-bound from order-sorting information, using the signature analysis technique in [134].

The general solution to a set of non-negative Diophantine equations is a set of basis elements from which all solutions can be obtained by linear combination. Upper-bound information may trivially eliminate some basis elements from consideration and can be used by the Diophantine solver to terminate the search for basis elements early.

A fresh variable z_k is allocated for each basis element α_k and unifiers are formed by finding sets of basis elements that satisfy certain properties and constructing assignments $x_i \leftarrow f(\dots, z_k^{\alpha_{k,i}}, \dots)$ where k ranges over the indices of the selected basis elements and $\alpha_{k,i}$ is the value of X_i in the basis element α_k .

The criteria for choosing the sets of basis elements is the key difference between *AC* unification, *ACU* unification, and restricted *ACU* unification. With *AC* unification, every selection of basis elements whose sum yields a nonzero value for each X_i and Y_i must be considered. With *ACU* unification that requirement is lifted because of the availability of an identity element. The identity element also means that any assignment including basis element α_k generalizes the same assignment with α_k removed by assigning the identity element to z_k and thus there is a single most general solution, formed by selecting all the basis elements.

In the case of restricted *ACU* unification, we may have upper-bounds on variables because they are marked. In Maude, order-sorted considerations may place upper-bounds on variables, and may also place a lower-bound of 1 on variables where the corresponding original variable has a sort that cannot take the identity element. In order to find a complete set of unifiers we need to find all maximal sets of basis elements whose sum satisfies the upper and lower-bounds on the variables.

Several explicit schemes for searching the subsets of basis elements were tried but the search was typically the dominant cost for *ACU* unification, often rendering the solution of quite modest unification problems impractical. In the current implementation this search is performed symbolically using a Binary Decision Diagram (BDD) [41] based algorithm. A BDD variable is allocated for each basis element, whose value, true or false, denotes whether the basis element is included in the subset. A BDD, called *legal*, is constructed, which

evaluates to true on exactly those valuations that correspond to selections of basis elements that satisfy the upper- and lower-bound constraints on each Diophantine variable. Enforcement of the upper-bounds on the sum is done using dynamic programming and the BDD *ite* operation. Using the BDD *legal*, a second BDD, called *maximal*, is constructed which is true on exactly those valuations where *legal* is true, and changing a false into a true makes *legal* false. These valuations of the BDD variables and thus the subsets of basis elements they encode are then recovered by tracing the paths from the root to the true terminal in *maximal*. This method yielded a dramatic speed up (from hours to milliseconds) on problems of useful size.

15.9 Variants

As explained in Section 15.2, Maude 2.7 features order-sorted unification modulo axioms Ax , including commutativity (C), associativity-commutativity (AC), and associativity-commutativity-identity (ACU). However, order-sorted equational unification in full generality considers a decomposition of an equational theory $(\Sigma, E \cup Ax)$ into *two* problems: one of Ax -unification, and another of $E \cup Ax$ -unification that uses an Ax -unification algorithm as a subroutine. As explained in Section 15.6.1, algorithms for $E \cup Ax$ -unification have been extensively defined by using narrowing-based unification, where for a unification problem $t =^? t'$ we obtain the search space associated to narrowing the term $\text{eq}(t, t')$ using E modulo Ax and search for all paths from $\text{eq}(t, t')$ to the truth constant tt . However, we use the notion of *variants* of a term for generating such a narrowing search space.

Comon-Lundh and Delaune's notion of *variant* [82] characterizes the instances of a term w.r.t. an equational theory $E \cup Ax$ such that the equations E are confluent, terminating, and coherent modulo axioms Ax . The E, Ax -variants of a term t are *pairs* (t', θ) , with θ a substitution and t' the E, Ax -canonical form of $\theta(t)$. A preorder relation of generalization that holds between such pairs provides a notion of *most general* variant and also of completeness of a set of variants. A *complete set of E, Ax -variants* (up to renaming) of a term t is a subset V of E, Ax -variants of t such that, for each substitution σ , there is a variant $(t', \theta) \in V$ and a substitution ρ such that: (i) t' is E, Ax -irreducible, (ii) $\sigma(t) \downarrow_{E, Ax} =_{Ax} \rho(t')$, and (iii) $(\sigma \downarrow_{E, Ax})|_{\text{Var}(t)} =_{Ax} (\theta; \rho)|_{\text{Var}(t)}$.

In order to avoid clashing of algorithms and notions, we have decided that the equations used for variant generation (and variant-based unification) should be identifiable and clearly distinguished from standard equations in Maude. For this purpose we have defined a new attribute for equations: the word **variant**. This implies that if the user wants to use an equation $\mathbf{t} = \mathbf{t}'$ both for variant generation and for simplification, it should be duplicated: `eq $\mathbf{t} = \mathbf{t}'$.` and `eq $\mathbf{t} = \mathbf{t}'$ [variant] .` No equation with the **variant** attribute can have the **owise** attribute. Note that what this allows is a greater flexibility at the operational level when combining variant generation and

simplification: by the above method, an equation can be used for either purpose (declared only once in the appropriate way), or for both, by a double declaration.

For example, consider the following functional module defining the addition function `_+_` on natural numbers built from `0` and `s`:

```
fmod NAT-VARIANT is
  sort Nat .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat .
  vars X Y : Nat .
  eq [base] : 0 + Y = Y [variant] .
  eq [ind] : s(X) + Y = s(X + Y) [variant] .
endfm
```

The term `X + s(0)` has an infinite number of variants w.r.t. those equations, i.e.,

- $(s(0), \{X \mapsto 0\})$,
- $(s(s(Y)), \{X \mapsto s(Y)\})$,
- $(s(s(0)), \{X \mapsto s(0)\})$,
- $(s(s(s(Y))), \{X \mapsto s(s(Y))\})$,
- $(s(s(s(0))), \{X \mapsto s(s(0))\})$, ...

Indeed, there is no finite, complete, most general set of variants for that term. However, the term `0 + X` has a finite number of most general variants w.r.t. those equations, i.e., (X, id) . Obviously, there are many more variants, such as $(0, \{X \mapsto 0\})$, but they are all instances of the most general one.

An equational theory $E \cup Ax$ has the *finite variant property* iff there is a finite complete set of most general variants for each term. This property also ensures the existence of a generic *finitary* $E \cup Ax$ -unification algorithm based on computing variants, as shown in Section 15.10.

Consider the following equational theory for exclusive or.

```
fmod EXCLUSIVE-OR is
  sorts Nat NatSet .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .

  subsort Nat < NatSet .
  op mt : -> NatSet [ctor] .
  op *_ : NatSet NatSet -> NatSet [ctor assoc comm] .

  vars X Y Z : [NatSet] .
  eq [idem] : X * X = mt [variant] .
  eq [idem-Coh] : X * X * Z = Z [variant] .
  eq [id] : X * mt = X [variant] .
endfm
```

This theory has the finite variant property, as proved manually in [82] and automatically in [148]. For instance, the term $X * X$ has a finite set of most general variants, just (mt, id) . And the term $X * Y$ has also a finite, complete set of most general variants:

1. $(X * Y, id)$,
2. $(Z, \{X \mapsto mt, Y \mapsto Z\})$,
3. $(Z, \{X \mapsto Z, Y \mapsto mt\})$,
4. $(Z, \{X \mapsto Z * U, Y \mapsto U\})$,
5. $(Z, \{X \mapsto U, Y \mapsto Z * U\})$,
6. $(mt, \{X \mapsto U, Y \mapsto U\})$, and
7. $(Z_1 * Z_2, \{X \mapsto U * Z_1, Y \mapsto U * Z_2\})$

Note that if variable X in the equational theory is changed from sort `[NatSet]` to `Nat` (or `[Nat]`), then the theory does not have the finite variant property, since every pair of similar elements has to be separately eliminated, whereas now chunks of similar elements can be eliminated at once. Also, note that the symbol `_*` cannot be made *ACU* instead of *AC*, because then the equation $X * X * Z = Z$ is not *ACU*-terminating.⁴

The finite variant property happens to be an undecidable problem [28]. However, a new semi-decision procedure for checking the finite variant property has been developed which works well in practice: it has recently [57] been shown that, in order to prove the finite variant property for an equational theory $(\Sigma, E \cup Ax)$, it is enough to check, for each function symbol $f \in \Sigma$, whether or not each pattern of the form $f(X_1, \dots, X_n)$ has a finite number of variants, where the X_i are distinct variables of the appropriate kind and n is the arity of f . This can be done by attempting to generate all the variants of $f(X_1, \dots, X_n)$ as described in Section 15.9.2 below.

Variants are used for variant-based unification in Section 15.10, and such a variant-based unification is later used in Section 20.3 for symbolic reachability analysis. Before defining variant-based unification, in Section 15.9.1 we introduce the class of equational theories admissible for variant generation, and thus for variant-based unification. We also provide in Section 15.9.2 a command `get variants` for user generation of variants.

15.9.1 Theories supported for variant generation

The equational theories that are admissible for variant generation are as follows. Let `fmod` $(\Sigma, E \cup Ax)$ `endfm` (resp. `fth` $(\Sigma, E \cup Ax, R)$ `endfth`) be an order-sorted functional module (resp. functional theory) where E is a set of equations specified with the `eq` keyword and the attribute `variant`, and Ax is a set of axioms such that (Σ, Ax) satisfies the restrictions explained in Section 15.3. Furthermore, the equations E must satisfy the following extra conditions:

⁴ This is the only equation necessary for *ACU*-coherence and the other two would be eliminated.

- The equations E are unconditional, confluent, terminating, sort-decreasing, and coherent modulo Ax (see also Section 19.5 for coherence details).
- An equation's lefthand side cannot be a variable, and the `owise` feature is not allowed.
- The equational theory must satisfy the finite variant property.

Any system module `mod ($\Sigma, G \cup E \cup Ax, R$) endm` (of system theory `th ($\Sigma, G \cup E \cup Ax, R$) endth`), where G is an additional set of equations (without the `variant` attribute!) and R is a set of rules, is also considered admissible for variant generation if the equational part $(\Sigma, E \cup Ax)$ satisfies the conditions described above. Note that when an equational theory $(\Sigma, G \cup E \cup Ax)$ is entered into Maude, each equation in E (used for variant computation) must include the `variant` attribute. Note that equations in G do not have any restriction, i.e., they can be conditional equations, with the `owise` attribute, etc.

15.9.2 The `get variants` command

Given a module $\langle ModId \rangle$, Maude provides a variant generation command of the form:

```
get variants [ n ] in  $\langle ModId \rangle$  :  $\langle Term \rangle$  .
```

where n is an optional argument providing a bound on the number of variants requested, so that if the cardinality of the set of variants is greater than the specified bound, the variants beyond that bound are omitted; and, as usual, if no module is mentioned, the current module is used.

For example, we can check that the `EXCLUSIVE-OR` module above has the finite variant property by simply generating the variants for the exclusive-or symbol `*`.

```
Maude> get variants in EXCLUSIVE-OR : X * Y .
```

```
Variant #1
[NatSet]: #1:[NatSet] * #2:[NatSet]
X --> #1:[NatSet]
Y --> #2:[NatSet]

Variant #2
NatSet: mt
X --> %1:[NatSet]
Y --> %1:[NatSet]

Variant #3
[NatSet]: %1:[NatSet] * %3:[NatSet]
X --> %1:[NatSet] * %2:[NatSet]
Y --> %2:[NatSet] * %3:[NatSet]
```

```

Variant #4
[NatSet]: %1:[NatSet]
X --> %1:[NatSet] * %2:[NatSet]
Y --> %2:[NatSet]

```

```

Variant #5
[NatSet]: %2:[NatSet]
X --> %1:[NatSet]
Y --> %1:[NatSet] * %2:[NatSet]

```

```

Variant #6
[NatSet]: %1:[NatSet]
X --> mt
Y --> %1:[NatSet]

```

```

Variant #7
[NatSet]: %1:[NatSet]
X --> %1:[NatSet]
Y --> mt

```

No more variants.

The above example illustrates a difference between unifiers returned by the built-in unification modulo axioms and unifiers returned by variant generation or variant-based unification: two forms of fresh variables, the former *#n:Sort* and the new *%n:Sort*. The reasons for this distinction are irrelevant and connected to efficiency of algorithms. Both forms represent fresh variables and both share the same counter for new fresh variables. The user is required not to use variables of these two forms in submitted unification problems (either modulo axioms or variant-based). And when used at the metalevel, the counter for new fresh variables must take into account the numbers used for both forms of fresh variables.

Recall that memberships are discarded completely. For instance, we can modify the previous example to include a membership definition for a new sort `Empty`.

```

fmod EXCLUSIVE-OR-MB is
  sorts Nat NatSet .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .

  subsort Nat < NatSet .
  op mt : -> NatSet [ctor] .
  op *_ : NatSet NatSet -> NatSet [ctor assoc comm] .

  sort Empty .
  subsort Empty < NatSet .
  mb mt : Empty .

```

```

vars X Y Z : [NatSet] .
eq [idem] :      X * X = mt      [variant] .
eq [idem-Coh] : X * X * Z = Z [variant] .
eq [id] :       X * mt = X      [variant] .
endfm

```

We can ask then for the variants of the exclusive-or symbol `*` when restricted to the sort `Empty`, and the results use the variant equations but not the membership information.

```
Maude> get variants in EXCLUSIVE-OR-MB : X:Empty * Y:Empty .
```

```

Variant #1
NatSet: #1:Empty * #2:Empty
X:Empty --> #1:Empty
Y:Empty --> #2:Empty

Variant #2
Empty: mt
X:Empty --> %1:Empty
Y:Empty --> %1:Empty

```

No more variants.

Note that the membership is used to compute the least sort of terms involved in the results (like the constant `mt` above), but is not used during variant generation. For example, this process is not able to instantiate any of the two variables to the constant `mt`.

Consider now the following version of the vending machine to buy apples (`a`) or cakes (`c`) with dollars (`$`) and/or quarters (`q`). The reader can check that the only difference with the `UNIF-VENDING-MACHINE` module in Section 15.4 is the `change` equation, where we have added the attribute `variant` and a variable `M` to make it *ACU*-coherent (see Section 19.5 for details on *ACU*-coherence).

```

mod VARIANT-VENDING-MACHINE is
  sorts Coin Item Marking Money State .
  subsort Coin < Money .
  op empty : -> Money .
  op _ : Money Money -> Money [assoc comm id: empty] .
  subsort Money Item < Marking .
  op _ : Marking Marking -> Marking [assoc comm id: empty] .
  op <_> : Marking -> State .
  ops $ q : -> Coin .
  ops a c : -> Item .
  var M : Marking .
  rl [buy-c] : < M $ > => < M c > .
  rl [buy-a] : < M $ > => < M a q > .

```

```

    eq [change] : q q q q M = $ M [variant] .
endm

```

Note that the `change` equation satisfies the finite variant property, as proved by generating the variants of symbol `---`.

```
Maude> get variants in VARIANT-VENDING-MACHINE : X:Marking Y:Marking .
```

```

Variant #1
State: #1:Marking #2:Marking
X:Marking --> #1:Marking
Y:Marking --> #2:Marking

```

```

Variant #2
State: %1:Marking %2:Marking
X:Marking --> q q q %1:Marking
Y:Marking --> q %2:Marking

```

```

Variant #3
State: %1:Marking %2:Marking
X:Marking --> q q %1:Marking
Y:Marking --> q q %2:Marking

```

```

Variant #4
State: %1:Marking %2:Marking
X:Marking --> q %1:Marking
Y:Marking --> q q q %2:Marking

```

We can also generate the variants of the state `< $ q q X:Marking >` containing at least a dollar and two quarters.

```
Maude> get variants in VARIANT-VENDING-MACHINE : < $ q q X:Marking > .
```

```

Variant #1
State: < $ q q #1:Marking >
X:Marking --> #1:Marking

```

```

Variant #2
State: < $ $ %1:Marking >
X:Marking --> q q %1:Marking

```

These two variants form a finite, complete, and most general set of variants for the given term; for example, the variant

```
{< $ $ q q Y:Marking >, X:Marking --> q q q q Y:Marking}
```

is an instance of the first variant above, i.e., the canonical form `< $ $ q q Y:Marking >` is an instance of the normal form `< $ q q #1:Marking >` of the first variant, and the (normalized version) of the instantiating substitution,

i.e., #1:Marking --> \$ Y:Marking, is an instance of the empty substitution of the first variant.

We can consider a more complex equational theory such as the one of Abelian groups specified in the following module.

```
fmod ABELIAN-GROUP is
  sorts Element .
  op +_ : Element Element -> Element [comm assoc prec 30] .
  op -_ : Element -> Element [prec 20] .
  op 0 : -> Element .
  vars X Y Z : Element .
  eq X + 0 = X [variant] .
  eq X + - X = 0 [variant] .
  eq X + - X + Y = Y [variant] .
  eq - - X = X [variant] .
  eq - 0 = 0 [variant] .
  eq - X + - Y = -(X + Y) [variant] .
  eq -(X + Y) + Y = - X [variant] .
  eq -(- X + Y) = X + - Y [variant] .
  eq - X + - Y + Z = -(X + Y) + Z [variant] .
  eq -(X + Y) + Y + Z = - X + Z [variant] .
endfm
```

The generation of the variants for the addition symbol takes more time and provides 47 variants:

```
Maude> get variants in ABELIAN-GROUP : X + Y .

Variant #1
Element: #1:Element + #2:Element
X --> #1:Element
Y --> #2:Element

Variant #2
Element: %1:Element
X --> 0
Y --> %1:Element
...

Variant #46
Element: %2:Element + - (%3:Element + %4:Element)
X --> %5:Element + - (%1:Element + %3:Element)
Y --> %1:Element + %2:Element + - (%4:Element + %5:Element)

Variant #47
Element: - (%2:Element + %3:Element)
X --> %4:Element + - (%1:Element + %2:Element)
Y --> %1:Element + - (%3:Element + %4:Element)
```

And the minus sign symbol has four variants:

```
Maude> get variants in ABELIAN-GROUP : - X .
```

```
Variant #1
Element: - #1:Element
X --> #1:Element
```

```
Variant #2
Element: %1:Element
X --> - %1:Element
```

```
Variant #3
Element: 0
X --> 0
```

```
Variant #4
Element: %1:Element + - %2:Element
X --> %2:Element + - %1:Element
```

Another interesting feature is that variant generation is incremental and in this way we are able to give partial support to theories that do not have the finite variant property. Let us consider the functional module for addition `NAT-VARIANT` in Section 15.9 that does not have the finite variant property. On the one hand, it is possible to have a term with a finite number of most general variants although the theory does not have the finite variant property. For instance, the term $s(0) + X$ is simplified into $s(X)$.

```
Maude> get variants in NAT-VARIANT : s(0) + X .
```

```
Variant #1
Nat: s(#1:Nat)
X --> #1:Nat
```

On the other hand, we can approximate the number of variants of a term that we suspect does not have a finite number of most general variants. For instance, the term $X + s(0)$ has an infinite number of most general variants and we can approximate that infinite set of variants by including a bound in the command, as it is also done for unification modulo axioms.

```
Maude> get variants [10] in NAT-VARIANT : X + s(0) .
```

```
Variant #1
Nat: #1:Nat + s(0)
X --> #1:Nat
```

```
Variant #2
Nat: s(0)
X --> 0
```

```
...
```

```

Variant #10
Nat: s(s(s(s(s(0))))))
X --> s(s(s(s(0))))

```

Note that if we do not know a priori whether a term has a finite number of most general variants, we can incrementally increase the bound and if we obtain a number of variants smaller than the bound, we know for sure that it had a finite number of most general variants.

When an equational theory satisfies the requirements in Section 15.9.1 except the finite variant property, it is because there are terms with an infinite set of most general variants. Of course, if the user does not provide a bound, Maude cannot return this whole set for anyone of those terms, because it will keep generating one variant after the other; however, when the user provides such a bound, the process will always terminate with Maude returning a finite set of variants. As said above, the number of returned variants can be smaller than the given bound when the term has indeed a finite set of variants, but it will coincide with the bound otherwise.

15.9.3 Variant generation at the metalevel: getVariants

The procedure for variant generation is also available at the metalevel of Maude thanks to the `metaGetVariant` function provided in the `META-LEVEL` module.

```

op metaGetVariant : Module Term TermList Nat Nat ~> Variant?
[special (...)] .

```

The third argument is not used right now and is left for future use; thus, for the time being, the constant `empty` must be used. As in Section 15.5, it is convenient to *reuse* variable names from terms; this is allowed via the fourth argument, which is the largest number n appearing in fresh variables of the form `#n:Sort` or `%n:Sort`. Then the fresh variables in the computed variants will all be numbered from $n + 1$ on. And, as usual for descent functions, the last argument in the function is used to select which result is wanted, starting from 0. Caching is used so that if variant i has just been returned, requesting unifier $i + 1$ gives rise to an incremental computation.

The result sort is defined by means of the following data:

```

sorts Variant Variant? .
subsort Variant < Variant? .
op {_,_,_} : Term Substitution Nat -> Variant [ctor] .
op noVariant : -> Variant? [ctor] .

```

Again, the third argument denotes the largest number n used in the fresh variables appearing in the solutions.

We can illustrate the use of this metalevel function with the variant generation of the configuration `< $ q q X:Marking >` for the first variant.

```
Maude> reduce in META-LEVEL :
  metaGetVariant(upModule('VARIANT-VENDING-MACHINE, true),
    '<_>['_[$.Coin,'q.Coin,'q.Coin,'X:Marking]], empty, 0, 0) .
result Variant: {'<_>['_[$.Coin,'q.Coin,'q.Coin,'#1:Marking]],
  'X:Marking <- '#1:Marking,1}
```

Then the second possible variant:

```
Maude> reduce in META-LEVEL :
  metaGetVariant(upModule('VARIANT-VENDING-MACHINE, true),
    '<_>['_[$.Coin,'q.Coin,'q.Coin,'X:Marking]], empty, 0, 1) .
result Variant: {'<_>['_[$.Coin,'$.Coin,'%1:Marking]],
  'X:Marking <- '[_['q.Coin,'q.Coin,'%1:Marking],1}
```

And there are no more variants.

```
Maude> reduce in META-LEVEL :
  metaGetVariant(upModule('VARIANT-VENDING-MACHINE, true),
    '<_>['_[$.Coin,'q.Coin,'q.Coin,'X:Marking]], empty, 0, 2) .
result Variant?: noVariant
```

15.10 Variant-based equational order-sorted unification

The intimate connection between E , Ax -variants and $E \cup Ax$ -unification is as follows. Suppose that we extend the equational theory $(\Sigma, E \cup Ax)$ to $(\widehat{\Sigma}, \widehat{E} \cup Ax)$ by adding to Σ a new sort **Truth**, not related to any sort in Σ , with a constant **tt**, and, for each top sort of a connected component $[s]$, an operator $\mathbf{eq} : [s] [s] \rightarrow \mathbf{Truth}$; and where \widehat{E} is the result of adding for each top sort $[s]$ an extra (oriented) equation $\mathbf{eq}(x, x) = \mathbf{tt}$ (where x is a variable of sort $[s]$) to E . Then, given any two terms t, t' , if θ is an $E \cup Ax$ -unifier of t and t' , then the E, Ax -canonical forms of $\theta(t)$ and $\theta(t')$ must be Ax -equal and therefore the pair (\mathbf{tt}, θ) must be a variant of the term $\mathbf{eq}(t, t')$. Furthermore, if the term $\mathbf{eq}(t, t')$ has a finite set of most general variants, then we are *guaranteed* that the set of most general $E \cup Ax$ -unifiers of t and t' is *finite*.

At a practical level, variants are generated using narrowing (see Chapter 20 for narrowing capabilities in Maude). Narrowing with oriented equations E (with or without modulo Ax) enjoys well-known completeness results, including the generation of complete sets of unifiers and covering all rewriting sequences from instances of a term using normalized substitutions (i.e., variants). For instance, [196] showed that narrowing with E without axioms enjoyed good completeness results, and [200] showed that narrowing with E modulo axioms Ax enjoyed also good completeness results. But narrowing can be quite inefficient, generating a huge search space, and different narrowing strategies have been devised to reduce the search space while remaining complete, specially for unification purposes (see [11] for a survey on narrowing

termination). The *basic narrowing strategy* of [196] provided a restriction of narrowing that, while being complete, it was terminating for specific classes of theories. However, very little was known about effective narrowing strategies in the modulo case, and some of the known anomalies ring a cautionary note, to the effect that the naive extensions of standard narrowing strategies, for example *basic narrowing modulo AC*, were incomplete [373, 82]. In [151], the *folding variant narrowing strategy* is defined for the modulo case and it is proved to be complete for variants and with good termination properties, providing a finitary and complete unification algorithm for equational unification for the theories described in Section 15.9.1. Moreover, it is even better than the basic narrowing strategy in the case without axioms, since it can terminate for equational theories where basic narrowing cannot (see [151]).

In Maude 2.6, variant generation and variant-based equational unification were implemented in Maude and made available in Full Maude. Instead, in Maude 2.7 variant generation and variant-based equational unification have been implemented in Core Maude's C++ level for efficiency purposes and using the *Ax*-unification algorithm described in Section 15.4. Furthermore, the variant generation and variant-based equational unification available in Maude 2.6 were accepting only equational theories where the righthand side of the equations was a strongly irreducible term (e.g., a variable or a constant), while the current version implements the folding variant narrowing strategy in full generality. By “full generality” we mean not just any equational theory $(\Sigma, E \cup Ax)$ having the finite variant property with *Ax* satisfying the requirements in Section 15.3 (so that unification is *finitary*), but any confluent, coherent, and terminating modulo *Ax* decomposition (Σ, Ax, E) , thus obtaining an incremental generation for the (in general infinite) set of $E \cup Ax$ -unifiers in that case.

The key distinction, now supported for the first time in Maude in full generality, is one between *dedicated* unification algorithms for a limited set of axioms *Ax* and *generic* unification algorithms which can be applied to a much wider range of user-definable theories and can even deal with incremental generation of infinite sets of unifiers.

15.10.1 The variant unify command

Given a module $\langle ModId \rangle$, of the general form `mod ($\Sigma, G \cup E \cup Ax, R$) endm` where $(\Sigma, E \cup Ax)$ satisfies the requirements of Section 15.9.1, Maude provides a command for $E \cup Ax$ -equational unification based on variant generation of the form:

```
variant unify [ n ] in  $\langle ModId \rangle$  :
   $\langle Term-1 \rangle =? \langle Term'-1 \rangle \wedge \dots \wedge \langle Term-k \rangle =? \langle Term'-k \rangle$  .
```

where $k \geq 1$; n is an optional argument providing a bound on the number of unifiers requested, so that if the cardinality of the set of unifiers is greater

than the specified bound, the unifiers beyond that bound are omitted; and, as usual, if no module is mentioned, the current module is used.

Consider again the module `VARIANT-VENDING-MACHINE` introduced in Section 15.9.2. We can ask whether there is an $E \cup Ax$ -equational unifier of two configurations, one containing at least two quarters, and another containing at least one dollar, by invoking the following command:

```
Maude> variant unify in VARIANT-VENDING-MACHINE :
      < q q X:Marking > =? < $ Y:Marking > .
```

```
Unifier #1
X:Marking --> $ %1:Marking
Y:Marking --> q q %1:Marking
```

```
Unifier #2
X:Marking --> q q #1:Marking
Y:Marking --> #1:Marking
```

There are no most general unifiers; for instance, the unifier `X:Marking --> q q`, `Y:Marking --> empty` is an instance of the first solution by using the identity property of the operator for markings.

And we can ask again whether there is an $E \cup Ax$ -equational unifier of the two previous configurations but variable `Y:Marking` also appears in a second unification problem with two new configurations, one containing again at least two quarters, and another with just variable `Y:Marking`.

```
Maude> variant unify in VARIANT-VENDING-MACHINE :
      < q q X:Marking > =? < $ Y:Marking > /\
      < q q Z:Marking > =? < Y:Marking > .
```

```
Unifier #1
X:Marking --> $ %1:Marking
Z:Marking --> %1:Marking
Y:Marking --> q q %1:Marking
```

```
Unifier #2
X:Marking --> $ q q %1:Marking
Z:Marking --> q q %1:Marking
Y:Marking --> $ %1:Marking
```

Similarly to the incremental generation of variants, one can obtain an incremental number of unifiers for a given unification problem. Let us consider again the `NAT-VARIANT` module in Section 15.9 that does not have the finite variant property. On the one hand, it is possible to have a finite number of most general unifiers for a unification problem although the theory does not have the finite variant property. For instance, the unification problem between `s(0) + X` and `s(s(s(0)))` returns just one unifier.

```
Maude> variant unify in NAT-VARIANT : s(0) + X =? s(s(s(0))) .
```

```
Unifier #1
X --> s(s(0))
```

```
No more unifiers.
```

On the other hand, we can approximate the number of unifiers of a unification problem that we suspect does not have a finite number of most general unifiers. For instance, the unification problem between terms $X + s(0)$ and $s(s(s(0)))$ has only one solution $X \mapsto s(s(0))$ and we can obtain that solution by including a bound in the command, as it is also done for variant generation.

```
Maude> variant unify [1] in NAT-VARIANT : X + s(0) =? s(s(s(0))) .
```

```
Unifier #1
X --> s(s(0))
```

However, if we tried to obtain two unifiers, Maude would not stop because it would keep trying to generate a second unifier for a unification problem that has only one unifier, without knowing that it could stop. This differs from the incremental generation of variants (Section 15.9.2), where we can incrementally increase the bound even if the theory does not have the finite variant property and Maude will always stop with a new variant. The problem here is due to the fact that Maude needs to compute the set of variants before computing the set of unifiers; when the equational theory does not have the finite variant property, such a set of variants can be infinite and Maude is not able to complete this computation, even when the user provides a bound, because such a bound refers to the number of requested unifiers, but not to the number of variants, which in this process is just part of the internal process for computing unifiers. As in the example above, this can happen even when there are no further unifiers.

15.10.2 Variant unification at the metalevel: metaVariantUnify and metaVariantDisjointUnify

The procedure for variant-based equational unification is also available at the metalevel by means of the following functions provided in the `META-LEVEL` module.

```
op metaVariantUnify :
  Module UnificationProblem TermList Nat Nat ~> UnificationPair?
  [special (...)] .

op metaVariantDisjointUnify :
  Module UnificationProblem TermList Nat Nat ~> UnificationTriple?
  [special (...)] .
```

The unification problems and the result sort are the same as in Section 15.5.

```

sorts UnificandPair UnificationProblem .
subsort UnificandPair < UnificationProblem .
op _=?_ : Term Term -> UnificandPair [ctor prec 71] .
op _/\_ : UnificationProblem UnificationProblem -> UnificationProblem
      [ctor assoc comm prec 73] .

subsort UnificationPair < UnificationPair? .
subsort UnificationTriple < UnificationTriple? .
op {_,_} : Substitution Nat -> UnificationPair [ctor] .
op {_,_,_} : Substitution Substitution Nat -> UnificationTriple [ctor] .

```

We can illustrate the use of this metalevel function with the variant unification of the two terms shown above: `< q q X:Marking >` and `< $ Y:Marking >`:

```

Maude> reduce in META-LEVEL :
  metaVariantUnify(upModule('VARIANT-VENDING-MACHINE, true),
    '<_>[_][_'q.Coin,'q.Coin,'X:Marking]] =?
    '<_>[_][_'$.Coin,'Y:Marking]],
    empty, 0, 0) .
result UnificationPair: {
  'X:Marking <- '[_][_'$.Coin,'%1:Marking] ;
  'Y:Marking <- '[_][_'q.Coin,'q.Coin,'%1:Marking],1}

```

Metaprogramming Applications

A *metaprogram* is a program that takes programs as inputs and performs some useful computation. It may, for example, *transform* one program into another. Or it may *analyze* such a program with respect to some properties, or perform other useful program-dependent computations. This is of course very useful and very powerful. In Maude, metaprogramming has a logical, reflective semantics. It is just a direct consequence of the fact that both membership equational logic and rewriting logic are reflective logics, and of the efficient exploitation of that fact in the `META-LEVEL` module. That is, we can easily write Maude metaprograms by importing `META-LEVEL` into a module that defines such metaprograms as functions that have `Module` as one of their arguments. Since this is one of the most powerful uses of Maude as a programming language, we present in this chapter three metaprogramming examples of moderate size, yet interesting and nontrivial, that can be helpful as an appetizer and guide to more ambitious metaprogramming tasks. Much more ambitious examples, also freely available for inspection, are the Full Maude system itself (see Chapter 19), which is a metaprogram entirely written in Maude, and the various tools in Maude's formal environment, which are described in Section 23.1.

The first example is a unification algorithm. The unification in question is *order-sorted*, that is, it takes account of sorts, subsorts, and operator overloading, and can solve equations *modulo* the commutativity of some operators. This means that this kind of unification is *parametric* on the signature and equational attributes of the module in which we are performing the unification. In other words, it is a *metaprogram*. An interesting point about this example is the very close way in which the formal inference system defining the unification rules is represented in the metafunction's equational definition. It is therefore a *logical framework* example, in which a (theory-parametric) inference system is represented in Maude's logic, which is used as a framework logic to mechanize the given inference system.

The other two examples illustrate two useful *module transformations*. From a programming point of view, these are examples of *programming in the large*,

that is, of composing or transforming programs to obtain other programs. At the object level, Maude already provides some powerful programming-in-the-large features, such as the module importation, renaming, parameterization, and instantiation by views described in Chapter 8. But from a reflective point of view these are just concrete, special instances of a much more general and open-ended notion of *parameterization*: we can view any metaprogram that takes a module as input and produces a module as a result as a new kind of user-definable parameterized construction. The Full Maude design uses exactly this viewpoint to make Maude's module algebra extensible, and supports several new such constructions, including object-oriented modules (see Chapter 21), and the `TUPLE` and `POWER` constructions (see Section 19.3.1). The two other examples in this chapter illustrate two useful parameterized constructions. The first transforms an object-oriented module into a semantically equivalent version that *instruments* its own execution by keeping a list of the rules that have been applied so far to rewrite the configuration. The second module transformation is quite useful for model-checking purposes: it transforms an arbitrary system module with a chosen kind of states into another module which is deadlock free and whose Kripke structure is bisimilar to that of the original module. Therefore, both modules will satisfy the same LTL formulas with respect to bisimilar initial states.

To illustrate the ease with which new metaprograms can be added to Full Maude as new language features at the object level, we explain in Section 19.7 how both the order-sorted unification algorithm and the deadlock-freedom construction described in this chapter have been added to Full Maude as new features.

16.1 Commutative order-sorted unification

In this section we present the specification of an order-sorted unification algorithm. It yields a complete set of unifiers for a unification problem in which it is assumed that the order-sorted signature of the specification in question can have some operators declared *commutative*,¹ but no other equational axioms are declared as attributes for the operators. Therefore, the order-sorted unification is performed *modulo* the commutativity of certain operators.

The basic idea is to turn each of the textbook-style inference rules for such a unification algorithm into corresponding rewrite rules in Maude. We introduce below some basic concepts, then review the inference system, and finally describe its Maude implementation.

Following [207], we unify using the sort information as soon as possible in order to quickly discard failures. Then, we complete the simplification process and push the assertions of the sorts on the solutions that have been found.

¹ Then, all the subsort-overloaded instances of any such operator must also be declared as commutative; see Section 4.4.6.

The order-sorted unification algorithm for solving commutative equations (pairs of the form $t =_c^? t'$, where there is no distinction between lefthand and righthand sides) can be described as the result of a simplification phase followed by a solving phase. The simplification phase is a sequence of decomposition, merging, and mutation steps, transforming the initial unification problem into an equivalent disjunction of systems of fully decomposed equations of the form $x =_c^? t$, where x is a variable appearing only once in the system. The solving phase consists then in finding the finite set of solutions for the simplified system.

Thus, a unification problem is a set of equations, also called a *system*, denoted (e_1, \dots, e_n) , or a set of systems, also called a *disjunction of systems* and written $S_1 \vee \dots \vee S_n$. The set of variables occurring in a unification problem U is denoted $\text{vars}(U)$.

Given a specification $\mathcal{S} = (\Sigma, E \cup A)$, where A consists of the commutativity property of some operators in Σ , a substitution σ is an \mathcal{S} -solution of the equation $t =_A^? t'$ if and only if $\sigma(t) =_A \sigma(t')$. The set of \mathcal{S} -solutions of an equation $t =_A^? t'$ is denoted $U(t, t', \mathcal{S})$.

The set of variables occurring in a term t is denoted $\text{vars}(t)$. Then, given a substitution $\sigma = \{x_1 \leftarrow t_1, \dots, x_n \leftarrow t_n\}$, we denote by $D(\sigma)$ the set of variables $\{x_1, \dots, x_n\}$, and by $I(\sigma)$ the set $\bigcup_{i=1}^n \text{vars}(t_i)$.

We say that substitution σ is *at least as general* (modulo A) as substitution ρ with respect to a subset X of $D(\sigma)$ if there exists a substitution τ such that $\sigma\tau =_A \rho$ when σ and ρ are restricted to X . In this case we write $\sigma \prec_A \rho [X]$.

A set of substitutions Φ is a complete set of \mathcal{S} -solutions of the equation $t =_A^? t'$ away from the set of variables W such that $\text{vars}(t) \cup \text{vars}(t') \subseteq W$ if and only if

- $\forall \sigma \in \Phi, D(\sigma) \subseteq \text{vars}(t) \cup \text{vars}(t')$ and $I(\sigma) \cap W = \emptyset$;
- $\forall \sigma \in \Phi, \sigma \in U(t, t', \mathcal{S})$; and
- $\forall \rho \in U(t, t', \mathcal{S}), \exists \sigma \in \Phi$ such that $\sigma \prec_A \rho [\text{vars}(t) \cup \text{vars}(t')]$.

Our unification algorithm computes a complete set of \mathcal{S} -solutions; however, we do not enforce that this set is a minimal set of unifiers.

We give below the set of inference rules used for the unification algorithm. They operate on 3-tuples of the form $\langle V; E; \sigma \rangle$ and on 4-tuples of the form $[V; C; \sigma; \theta]$. A 3-tuple $\langle V; E; \sigma \rangle$ consists of a set of variables V , a set of equations E , and a substitution σ . A 4-tuple $[V; C; \sigma; \theta]$ consists of a set of variables V , a set of membership constraints C , and substitutions σ and θ . The rules operating on the first kind of tuples correspond to the first phase of the process, which is quite similar to syntactic unification. The main differences are in the rules **Check** and **Eliminate**, in which the sort information is used to try to quickly discard failure. In the second phase the constraints on the solutions are checked.

We assume a well-founded order \succ on equations, and we denote by $s \cap s'$ the set of maximal lower bounds of sorts s and s' .

Deletion of trivial equations

$$\frac{\langle V; \{t \stackrel{?}{=} t, E\}; \sigma \rangle}{\langle V; E; \sigma \rangle}$$

Decomposition

$$\frac{\langle V; \{f(t_1, \dots, t_n) \stackrel{?}{=} f(t'_1, \dots, t'_n), E\}; \sigma \rangle}{\langle V; \{t_1 \stackrel{?}{=} t'_1, \dots, t_n \stackrel{?}{=} t'_n, E\}; \sigma \rangle}$$

if $n \neq 2$ or f noncommutative

$$\frac{\langle V; \{f(t_1, t_2) \stackrel{?}{=} f(t'_1, t'_2), E\}; \sigma \rangle}{\langle V; \{t_1 \stackrel{?}{=} t'_1, t_2 \stackrel{?}{=} t'_2, E\}; \sigma \rangle \vee \langle V; \{t_1 \stackrel{?}{=} t'_2, t_2 \stackrel{?}{=} t'_1, E\}; \sigma \rangle}$$

if f commutative

Clash of symbols

$$\frac{\langle V; \{f(t_1, \dots, t_n) \stackrel{?}{=} g(t'_1, \dots, t'_m), E\}; \sigma \rangle}{\text{failure}}$$

if $n \neq m$ or $f \neq g$

Merging

$$\frac{\langle \{x : s, V\}; \{x \stackrel{?}{=} t, x \stackrel{?}{=} t', E\}; \sigma \rangle}{\langle \{x : s, V\}; \{x \stackrel{?}{=} t, t \stackrel{?}{=} t', E\}; \sigma \rangle}$$

if $x = t \succ t = t'$

Check

$$\frac{\langle \{x : s, V\}; \{x \stackrel{?}{=} t, E\}; \sigma \rangle}{\text{failure}}$$

if $x \neq t$ and $(x \text{ occurs in } t \text{ or } s \cap LS(t) = \emptyset)$

Eliminate

$$\frac{\langle \{x : s, V\}; \{x \stackrel{?}{=} t, E\}; \sigma \rangle}{\langle \{x : s, V\}; E\theta; \{\sigma\theta, \theta\} \rangle}$$

with $\theta = \{x \leftarrow t\}$

if x does not occur in t and $s \cap LS(t) \neq \emptyset$

Transition

$$\frac{\langle V; \emptyset; \sigma \rangle}{[V; \emptyset; \sigma]}$$

Solving ($x \leftarrow t$)

$$\frac{[\{x : s, V\}; C; \{x \leftarrow t, \sigma\}; \theta]}{[V; \{(t : s), C\}; \sigma; \theta]}$$

Solving ($x : s$)

$$\frac{[\{x : s, V\}; \{x : s', C\}; \emptyset; \theta]}{\bigvee_{s'' \in s \cap s'} [\{x : s'', V\}; C; \emptyset; \theta]}$$

Solving $(f(t_1, \dots, t_n) : s)$

$$\frac{[V; \{f(t_1, \dots, t_n) : s, C\}; \emptyset; \theta]}{\bigvee_{\substack{f:s_1 \dots s_n \rightarrow s' \\ s' \leq s \\ s_1 \dots s_n \text{ maximal}}} [V; \{t_1 : s_1, \dots, t_n : s_n, C\}; \emptyset; \theta]}$$

These inference rules will be mapped into Maude specifications almost without modification.

One of the components of both kinds of tuples is a set of variables, which is used in several rules to obtain the sort of a variable and to keep the restrictions on their sorts. Although we do not use such declarations to get such sorts, since the metarepresentation of variables already includes their sorts, we get the variables from the terms at the beginning of the process, so that they can be used later when considering the restrictions on them in the second stage of the process.

The module UNIFICATION below uses several auxiliary operations defined in the following UNIFICATION-AUX-OPS module, in which we define some basic functions for the manipulation of terms, attributes, etc.

```
fmod UNIFICATION-AUX-OPS is
pr META-LEVEL .
pr EXT-BOOL .
pr INT .
```

These are the variables used in the equations in this module:

```
vars T T' : Term .
var TL : TermList .
vars Tp Tp' Tp'' Tp''' : Type .
vars TpL TpL' : TypeList .
var M : Module .
var At : Attr .
var AtS : AttrSet .
vars L F G : Qid .
var ODS : OpDeclSet .
vars V V' : Variable .
var C : Constant .
var Subst : Substitution .
```

We define subsort-overloaded versions of the predefined functions `sortLeq`, `leastSort`, and `sameKind` to handle lists of types and terms (see Section 14.5.8).

```
op sortLeq : Module TypeList TypeList ~> Bool [ditto] .
eq sortLeq(M, (Tp Tp' TpL), (Tp'' Tp''' TpL'))
  = sortLeq(M, Tp, Tp'') and-then sortLeq(M, Tp' TpL, Tp''' TpL') .
eq sortLeq(M, nil, nil) = true .
eq sortLeq(M, TpL, TpL') = false [owise] .
```

```

op leastSort : Module TermList ~> TypeList [ditto] .
eq leastSort(M, (T, T', TL))
  = leastSort(M, T) leastSort(M, (T', TL)) .
eq leastSort(M, empty) = nil .

op sameKind : Module TypeList TypeList ~> Bool [ditto] .
eq sameKind(M, (Tp Tp' TpL), (Tp'' Tp''' TpL'))
  = sameKind(M, Tp, Tp'')
  and-then sameKind(M, Tp' TpL, Tp''' TpL') .
eq sameKind(M, nil, nil) = true .
eq sameKind(M, TpL, TpL') = false [owise] .

```

The `occurs` predicate checks whether a variable name occurs in a term or not.

```

op occurs : Variable Term -> Bool .
op occurs : Variable TermList -> Bool .
eq occurs(V, V') = V == V' .
eq occurs(V, C) = false .
eq occurs(V, F[TL]) = occurs(V, TL) .
eq occurs(V, (T, TL)) = occurs(V, T) or-else occurs(V, TL) .

```

The `length` function returns the length of a term list.

```

op length : TermList -> Nat .
eq length((T, TL)) = 1 + length(TL) .
eq length(empty) = 0 .

```

The following functions check whether an attribute belongs to a set of attributes, and return the value of a label attribute in a set of attributes, respectively.

```

op _in_ : Attr AttrSet -> Bool .
eq At in At AtS = true .
eq At in AtS = false [owise] .

op getLabel : AttrSet -> Qid .
eq getLabel(label(L) AtS) = L .
eq getLabel(AtS) = 'no-label [owise] .

```

The `hasAttr` function checks whether there is in the given module an operator with the specified name and arity having the given attribute.

```

op hasAttr : Module Qid TypeList Attr -> Bool .
op hasAttr : Module OpDeclSet Qid TypeList Attr -> Bool .
eq hasAttr(M, G, TpL, At) = hasAttr(M, getOps(M), G, TpL, At) .
eq hasAttr(M, op F : TpL -> Tp [AtS] . ODS, G, TpL', At)
  = if (F == G) and-then sameKind(M, TpL, TpL')
    then At in AtS
    else hasAttr(M, ODS, G, TpL', At)
  fi .
eq hasAttr(M, none, G, TpL, At) = false .

```

The function `substitute` takes a term t and a substitution σ and returns the term $t\sigma$.

```

op substitute : Term Substitution -> Term .
op substitute : TermList Substitution -> TermList .

eq substitute(T, none) = T .
eq substitute(V, ((V' <- T) ; Subst))
  = if V == V' then T else substitute(V, Subst) fi .
eq substitute(C, ((V' <- T); Subst)) = C .
eq substitute(F[TL], Subst) = F[substitute(TL, Subst)] .
eq substitute((T, TL), Subst)
  = (substitute(T, Subst), substitute(TL, Subst)) .
endfm

```

We present now the UNIFICATION module, which reflects quite closely the inference rules described above. But before presenting such rules, the module introduces some declarations for variable declarations, sets of substitutions, unification tuples, and commutative equations, plus some additional auxiliary functions.

```

fmod UNIFICATION is
pr UNIFICATION-AUX-OPS .

vars QI F G X Y : Qid .
vars S S' : Sort .
vars Tp Tp' : Type .
vars TpL TpL' : TypeList .
var TpS : TypeSet .
vars TpLS TpLS' : TypeListSet .
var M : Module .
vars T T' T'' T''' T1 T1' : Term .
vars TL TL' : TermList .
var CEqS : CommEqSet .
var D : Disjunction .
var UT : UnifTuple .
var VDS : VarDeclSet .
vars Subst Subst' : Substitution .
var SubstS : SubstitutionSet .
var MAS : MembAxSet .
vars N I : Nat .
var AtS : AttrSet .
var IL : ImportList .
var SS : SortSet .
var SSDS : SubsortDeclSet .
var ODS : OpDeclSet .
var EqS : EquationSet .
var RLS : RuleSet .
vars V W : Variable .
vars C C' : Constant .

```

```

var EqC : EqCondition .
var Cd : Condition .

```

First, we give declarations for manipulating variables as in the unification tuples in the inference rules. In particular, we declare a sort `VarDecl` of variable declarations of the form `var V : T .`, and a sort `VarDeclSet` for sets of such declarations. The variable declaration corresponding to a variable `N` of sort `Nat`, which is metarepresented as `'N:Nat`, is `var 'N:Nat : 'Nat .`

```

sorts VarDecl VarDeclSet .
subsort VarDecl < VarDeclSet .
op var_:_ : Variable Type -> VarDecl .
op none : -> VarDeclSet .
op __ : VarDeclSet VarDeclSet -> VarDeclSet [assoc comm id: none] .
eq VD:VarDecl VD:VarDecl = VD:VarDecl .

```

The following overloaded versions of the `varDecls` function return the set of variable declarations for the variables in a term, in a term list, and in a set of commutative equations, respectively. Note that we can get the type of the metarepresentation of a variable with the `getType` function. Commutative equations are built with syntax `_=?c_` as terms of sort `CommEq`; then, the `CommEqSet` sort defines sets of commutative equations (see below).

```

op varDecls : Term -> VarDeclSet .
op varDecls : TermList -> VarDeclSet .
op varDecls : CommEqSet -> VarDeclSet .

eq varDecls(V) = (var V : getType(V) .) .
eq varDecls(C) = none .
eq varDecls(F[TL]) = varDecls(TL) .
eq varDecls(empty) = none .
eq varDecls((T, TL)) = varDecls(T) varDecls(TL) .

eq varDecls((T =?c T') CEqS)
  = varDecls(T) varDecls(T') varDecls(CEqS) .
eq varDecls((none).CommEqSet) = none .

```

The solution of a unification problem will be given as a set of substitutions.

```

sort SubstitutionSet .
subsort Substitution < SubstitutionSet .
op emptySubstitutionSet : -> SubstitutionSet .
op substitutionSet :
  SubstitutionSet SubstitutionSet -> SubstitutionSet
  [assoc comm id: emptySubstitutionSet] .

```

Sorts `UnifTuple` and `Disjunction` define unification tuples and disjunctions of unification tuples; the syntax of the operators defining terms of these sorts mirror the syntax used in the inference rules given above.

Since the module in which the variables and unification problems are defined will be needed for accomplishing some of the operations on variables and terms, we need to have access to it. We define an operator `unifPair` of sort `UnifPair` to carry such module together with the disjunction of unification tuples.

```

sorts UnifPair UnifTuple Disjunction .
subsort UnifTuple < Disjunction .

op <_;<_> : VarDeclSet CommEqSet Substitution -> UnifTuple .
op [_;<_;<_] :
  VarDeclSet MembAxSet Substitution Substitution -> UnifTuple .

op failure : -> Disjunction .
op _\/_ : Disjunction Disjunction -> Disjunction
  [assoc comm id: failure] .

op unifPair : Module Disjunction -> UnifPair .

```

Commutative equations are built with syntax `_=?c_` as terms of sort `CommEq`.

```

sorts CommEq CommEqSet .
subsort CommEq < CommEqSet .
op _=?c_ : Term Term -> CommEq [comm] .
op none : -> CommEqSet .
op __ : CommEqSet CommEqSet -> CommEqSet [assoc comm id: none] .

```

The disjunction of tuples that must be created for the *Solving* $(x : s)$ rule is generated by the following `unifTuplesVar` function.

```

op unifTuplesVar :
  Module Variable Type Type UnifTuple -> Disjunction .
op unifTuplesVarAux : TypeSet Qid UnifTuple -> Disjunction .
eq unifTuplesVar(M, V, Tp, Tp', UT)
  = unifTuplesVarAux(glbSorts(M, Tp, Tp'), V, UT) .

eq unifTuplesVarAux((Tp ; TpS), V, [VDS ; MAS ; Subst ; Subst'])
  = ([var V : Tp . VDS ; MAS ; Subst ; Subst']
     \
     unifTuplesVarAux(TpS, V, [VDS ; MAS ; Subst ; Subst'])) .
eq unifTuplesVarAux(none, V, UT) = failure .

```

Similarly, the following `unifTuplesNonVar` function generates the disjunction of tuples that must be created for the *Solving* $(f(t_1, \dots, t_n) : s)$ rule.

```

op unifTuplesNonVar : Module MembAx UnifTuple -> Disjunction .
op unifTuplesNonVarAux :
  TypeListSet TermList UnifTuple -> Disjunction .
op unifTuplesNonVarAux2 : TypeList TermList UnifTuple -> UnifTuple .
eq unifTuplesNonVar(M, (mb F[TL] : S [AtS].), UT)

```

```

    = unifTuplesNonVarAux(maximalAritySet(M, F, leastSort(M, TL), S),
      TL, UT) .
eq unifTuplesNonVarAux(TpL ; TpLS, TL, UT)
  = (unifTuplesNonVarAux2(TpL, TL, UT)
    \∨
    unifTuplesNonVarAux(TpLS, TL, UT)) .
eq unifTuplesNonVarAux(none, TL, UT) = failure .
eq unifTuplesNonVarAux2((Tp TpL), (T, TL),
  [VDS ; MAS ; Subst ; Subst'])
  = unifTuplesNonVarAux2(TpL, TL,
    [VDS ; mb T : Tp [none] . MAS ; Subst ; Subst']) .
eq unifTuplesNonVarAux2(nil, empty, UT) = UT .

```

The following `greaterCommEq` predicate defines a well-founded order on equations, based on the size of terms, defined in turn as the number of operator symbols in them.

```

op greaterCommEq : CommEq CommEq -> Bool .
op size : TermList -> Nat .
op size : Term -> Nat .

eq greaterCommEq((T =?c T'), (T1 =?c T1'))
  = (max(size(T), size(T')) > max(size(T1), size(T1')))
  or-else
  ((max(size(T), size(T')) == max(size(T1), size(T1')))
  and-then
  (sd(max(size(T), size(T')), min(size(T), size(T')))
    > sd(max(size(T1), size(T1')), min(size(T1), size(T1'))))) .

eq size(V) = 0 .
eq size(C) = 1 .
eq size(F[TL]) = 1 + size(TL) .
eq size((T, TL)) = size(T) + size(TL) .

```

We are now ready to give the equations corresponding to the inference rules given above. The first one corresponds to the rule *Deletion of Trivial Equations*.

```

eq unifPair(M, (< VDS ; (T =?c T) CEqS ; Subst > \∨ D))
  = unifPair(M, (< VDS ; CEqS ; Subst > \∨ D)) .

```

Since we assume that all subsort-overloaded operators have the same attributes, the *Decomposition* rule can be written as follows:

```

eq unifPair(M, (< VDS ; (F[TL] =?c G[TL']) CEqS ; Subst > \∨ D))
  = if (F =/= G)
    or-else (length(TL) =/= length(TL'))
    then unifPair(M, D)
    else if (length(TL) == 2)
      and-then hasAttr(M, F, leastSort(M, TL), comm)
      then unifPair(M,

```

```

        commUnifTupleSet(VDS, F, TL, TL', CEqS, Subst) \ / D)
    else unifPair(M,
        < VDS ; commEqBreak(TL, TL') CEqS ; Subst > \ / D)
    fi
fi .

op commEqBreak : TermList TermList -> CommEqSet .
eq commEqBreak(T, T') = (T =?c T') .
eq commEqBreak((T, TL), (T', TL'))
    = ((T =?c T') commEqBreak(TL, TL')) .

```

where the disjunction of unification tuples corresponding to the commutative operators is created by the following function:

```

op commUnifTupleSet : VarDeclSet Qid TermList TermList CommEqSet
    Substitution -> Disjunction .
eq commUnifTupleSet(VDS, F, (T, T'), (T'', T'''), CEqS, Subst)
    = (< VDS ; (T =?c T'') (T' =?c T''') CEqS ; Subst >
        \ /
        < VDS ; (T =?c T''') (T' =?c T'') CEqS ; Subst >) .

```

The *Clash of Symbols* rule is specified as follows:

```

ceq unifPair(M, (< VDS ; (C =?c C') CEqS ; Subst > \ / D))
    = unifPair(M, D)
    if getName(C) /= getName(C')
        or not sameKind(M, getType(C), getType(C')) .
eq unifPair(M, (< VDS ; (C =?c F[TL]) CEqS ; Subst > \ / D))
    = unifPair(M, D) .

```

Given the well-founded order defined by the function `greaterCommEq` above, the *Merging* rule has the following representation:

```

ceq unifPair(M, (< VDS ; (V =?c T) (V =?c T') CEqS ; Subst > \ / D))
    = unifPair(M, (< VDS ; (V =?c T) (T =?c T') CEqS ; Subst > \ / D))
    if greaterCommEq((V =?c T'), (T =?c T')) .

```

Given functions `substCommEqs` and `substSubst` that apply a substitution, respectively, on a set of (commutative) equations and on a substitution, the *Check* and *Eliminate* rules can be specified as follows:

```

ceq unifPair(M, (< VDS ; (V =?c T) CEqS ; Subst > \ / D))
    = if occurs(V, T)
        then unifPair(M, D)
        else if glbSorts(M, leastSort(M, T), getType(V)) == none
            then unifPair(M, D)
            else unifPair(M,
                < VDS ;
                substCommEqs(CEqS, V <- T) ;
                (substSubst(Subst, V <- T) ; V <- T) >
                \ / D)

```

```

        fi
    fi
    if V /= T .

op substCommEqs : CommEqSet Substitution -> CommEqSet .
eq substCommEqs(none, Subst) = none .
eq substCommEqs((T =?c T') CEqS, Subst)
  = ((substitute(T, Subst) =?c substitute(T', Subst))
     substCommEqs(CEqS, Subst)) .

op substSubst : Substitution Substitution -> Substitution .
eq substSubst(none, Subst) = none .
eq substSubst((V <- T); Subst'), Subst)
  = ((V <- substitute(T, Subst));
     substSubst(Subst', Subst)) .

```

The following rule specifies the transition from a 3-unification tuple to a 4-unification tuple as expressed by the *Transition* inference rule.

```

eq unifPair(M, (< VDS ; none ; Subst > \ / D))
  = unifPair(M, ([VDS ; none ; Subst ; Subst] \ / D)) .

```

The *solving* ($V \leftarrow T$) rule is easily specified:

```

eq unifPair(M,
  [var V : S . VDS ; MAS ; (V <- T ; Subst) ; Subst'] \ / D)
  = unifPair(M,
  [VDS ; mb T : S [none] . MAS ; Subst ; Subst'] \ / D) .

```

Given the functions `unifTuplesVar` and `unifTuplesNonVar` presented above, the *solving* ($X : S$) and *solving* ($f(t_1, \dots, t_n) : S$) rules are specified as follows.

```

eq unifPair(M,
  [var V : S . VDS ; mb V : S' [none] . MAS ; none ; Subst]
  \ / D)
  = unifPair(M,
  unifTuplesVar(M, V, S, S', [VDS ; MAS ; none ; Subst])
  \ / D) .

eq unifPair(M, [VDS ; mb C : S [none] . MAS ; none ; Subst] \ / D)
  = if sortLeq(M, getType(C), S)
    then unifPair(M, [VDS ; MAS ; none ; Subst] \ / D)
    else unifPair(M, D)
    fi .

eq unifPair(M, [VDS ; mb F[TL] : S[AtS]. MAS ; none ; Subst] \ / D)
  = unifPair(M,
  unifTuplesNonVar(M, (mb F[TL] : S [AtS] .),
  [VDS ; MAS ; none ; Subst])
  \ / D) .

```

The main function of the UNIFICATION module is given by the `metaUnify` operator, which builds the unification solution using the `getUnifSolution` from the result of the inference process defined by the above rules.

```

op metaUnify : Module CommEqSet -> SubstitutionSet .
eq metaUnify(M, CEqS)
  = getUnifSolution(unifPair(M, < varDecls(CEqS) ; CEqS ; none >)) .

op getUnifSolution : UnifPair -> SubstitutionSet .
eq getUnifSolution(unifPair(M, [VDS ; none ; none ; Subst] \ / D))
  = substitutionSet(
    (substSubst(Subst, substMembAxSet(M, VDS)) ;
     substMembAxSet(M, VDS)),
    getUnifSolution(unifPair(M, D))) .
eq getUnifSolution(unifPair(M, failure)) = emptySubstitutionSet .

op substMembAxSet : Module VarDeclSet -> Substitution .
eq substMembAxSet(M, (var V : S . VDS))
  = ((V <- qid(string(getName(V)) + "@" + string(S))) ;
     substMembAxSet(M, VDS)) .
eq substMembAxSet(M, none) = none .
endfm

```

We finish this section illustrating the use of the `metaUnify` function with several examples on the PEANO-NAT module from Section 4.10.

```

Maude> red in UNIFICATION :
  metaUnify(upModule('PEANO-NAT, false),
    '_+['_X:NzNat, '_*['_0.Zero, 'Y:NzNat]]
    =?c '_+['_W:Nat, 's['_Z:Nat]]) .
result Substitution:
  'W:Nat <- '_*['_0.Zero, 'Y@:NzNat] ;
  'X:NzNat <- 's['_Z@:Nat] ;
  'Y:NzNat <- 'Y@:NzNat ;
  'Z:Nat <- 'Z@:Nat

Maude> red in UNIFICATION :
  metaUnify(upModule('PEANO-NAT, false),
    '_+['_X:NzNat, 's['_*['_Y:Nat, 'W:Nat]]]
    =?c '_+['_s['_V:Nat], 'Z:Nat]) .
result SubstitutionSet:
  substitutionSet(
    'V:Nat <- 'V@:Nat ;
    'W:Nat <- 'W@:Nat ;
    'X:NzNat <- 's['_V@:Nat] ;
    'Y:Nat <- 'Y@:Nat ;
    'Z:Nat <- 's['_*['_Y@:Nat, 'W@:Nat]],
    'V:Nat <- '_*['_Y@:Nat, 'W@:Nat] ;
    'W:Nat <- 'W@:Nat ;
    'X:NzNat <- 'Z@:NzNat ;
  )

```

```

'Y:Nat <- 'Y@:Nat ;
'Z:Nat <- 'Z@:NzNat)

Maude> red in UNIFICATION :
  metaUnify(upModule('PEANO-NAT, false),
    ('_+_'s_['X:Nat], '_*_'s_['X:Nat, 'Y:Nat]]
      =?c '_+_'s_['Z:NzNat, 's_['s_['0.Zero]]])
    ('Y:Nat =?c 's_['s_['W:NzNat]])
    ('s_['V:Nat] =?c 's_['s_['s_['s_['0.Zero]]]])
    ('Z:NzNat =?c '_*_'s_['V:Nat, 's_['0.Zero]])) .
result Substitution:
'V:Nat <- 's_['s_['s_['s_['0.Zero]]]] ;
'W:NzNat <- 's_['s_['0.Zero]] ;
'X:Nat <- 's_['0.Zero] ;
'Y:Nat <- 's_['s_['s_['s_['0.Zero]]]] ;
'Z:NzNat <- '_*_'s_['s_['0.Zero], 's_['s_['s_['s_['0.Zero]]]]]

```

16.2 Rule instrumentation

In the context of software applications, instrumentation is understood as the addition of mechanisms to some application for the purpose of gathering data to be utilized by tools such as monitoring agents, profilers, etc. Such changes should be purely additive, meaning that these tools should not modify the application's state or behavior.

The instrumentation that we propose here is very simple, although it may very well suggest further possibilities. We are interested in collecting a history of the rules being applied on a configuration of objects and messages. In addition to adding the appropriate extra definitions, our construction transforms a given specification—only the specified module, not its submodules—so that each rule

$$(c)r1 [L] : T \Rightarrow T' \text{ (if cond) .}$$

with T a term of sort `Configuration` (or some other sort in the same kind) is transformed into a rule

$$(c)r1 [L] : \{T, LL\} \Rightarrow \{T', LL L\} \text{ (if cond) .}$$

Note that this transformation will instrument properly object-oriented modules whose rules rewrite terms of kind `[Configuration]`. A more general transformation could be defined for arbitrary system modules by generalizing the ideas presented here; but in the general case we would need to deal with the fact that rewrites could happen at any depth in subterms.

The `INSTRUMENTATION-INFRASTRUCTURE` module contains the basic definitions needed for our instrumentation. In particular, it defines a sort `InstrConfig` of instrumented configurations, whose terms are pairs of the form $\{T, LL\}$ with T a configuration—a term of sort `Configuration`—and LL a list of labels—a term of sort `QidList`.

```

mod INSTRUMENTATION-INFRASTRUCTURE is
  pr CONFIGURATION .
  pr QID-LIST .

  sorts InstrConfig .

  var C : Configuration .
  var QL : QidList .

  op {_,_} : Configuration QidList -> InstrConfig .
  op getConfig : InstrConfig -> Configuration .
  op getLabels : InstrConfig -> QidList .
  eq getConfig({C, QL}) = C .
  eq getLabels({C, QL}) = QL .
endm

```

The ad-hoc overloaded `instrument` functions in the `INSTRUMENTATION` module below perform the transformation. The main `instrument` function takes as argument a quoted identifier, which corresponds to the name of the module that we want to instrument. This function gets the metarepresentation of such a module using the `upModule` function with the second argument set to `false`, thus getting the metarepresentation of the top module without expanding its submodules. This function calls the third `instrument` function with the rules in the module, and is in charge of transforming them. Notice that, in addition to transforming the rules, these functions add an importation declaration including `INSTRUMENTATION-INFRASTRUCTURE` into the module being instrumented. We use the auxiliary operations `getLabels`, `setRls`, and `addImports` in order to, respectively, get a label from a set of attributes, replace the rules in a module by a given set of rules, and add a list of importations to the imports of a module.

```

fmod INSTRUMENTATION is
  pr META-LEVEL .

  op instrument : Qid -> Module .
  op instrument : Module -> Module .
  op instrument : Module RuleSet -> RuleSet .

  var M : Module .
  var L : Qid .
  var AtS : AttrSet .
  vars T T' : Term .
  var Cd : Condition .
  var H : Header .
  vars IL IL' : ImportList .
  var SS : SortSet .
  var SSDS : SubsortDeclSet .
  var OPDS : OpDeclSet .

```

```

var MAS : MembAxSet .
var EqS : EquationSet .
vars RLS RLS' : RuleSet .

op getLabel : AttrSet -> Qid .
eq getLabel(label(L) AtS) = L .

op setRls : Module RuleSet -> Module .
eq setRls(
  mod H is IL sorts SS . SSDS OPDS MAS EqS RLS endm, RLS')
  = mod H is IL sorts SS . SSDS OPDS MAS EqS RLS' endm .
eq setRls(fmod H is IL sorts SS . SSDS OPDS MAS EqS endfm, RLS)
  = if RLS == none
    then fmod H is IL sorts SS . SSDS OPDS MAS EqS endfm
    else mod H is IL sorts SS . SSDS OPDS MAS EqS RLS endm
    fi .

op addImports : Module ImportList -> Module .
eq addImports(
  mod H is IL sorts SS . SSDS OPDS MAS EqS RLS endm, IL')
  = mod H is IL IL' sorts SS . SSDS OPDS MAS EqS RLS endm .
eq addImports(fmod H is IL sorts SS . SSDS OPDS MAS EqS endfm,
  IL')
  = fmod H is IL IL' sorts SS . SSDS OPDS MAS EqS endfm .

eq instrument(H) = instrument(upModule(H, false)) .

eq instrument(M)
  = setRls(
    addImports(M, (including 'INSTRUMENTATION-INFRASTRUCTURE .)),
    instrument(M, getRls(M))) .

eq instrument(M, rl T => T' [AtS] . RLS)
  = if sameKind(M, leastSort(M, T), 'Configuration)
    then (rl '{_','_}'[_][T, 'C@:Configuration], 'QL@:QidList]
      => '{_','_}'[_][T', 'C@:Configuration],
        '[_]'QL@:QidList,
        qid("'" + string(getLabel(AtS)) + ".Qid"')]
      [AtS] .)
    else (rl T => T' [AtS] .)
    fi
  instrument(M, RLS) .

eq instrument(M, crl T => T' if Cd [AtS] . RLS)
  = if sameKind(M, leastSort(M, T), 'Configuration)
    then (crl '{_','_}'[_][T, 'C@:Configuration],
      'QL@:QidList]
      => '{_','_}'[_][T', 'C@:Configuration],
        '[_]'QL@:QidList,

```

```

                                qid("'" + string(getLabel(AtS)) + ".Qid"')]
      if Cd
        [AtS] .)
      else (crl T => T' if Cd [AtS] .)
      fi
      instrument(M, RLS) .
    eq instrument(M, none) = none .
  endfm

```

We illustrate the use of these instrumentation functions on the example specifying bank accounts presented in Section 11.1. We will observe that for the instrumented module we obtain the same results for the executions using `rewrite` and `search` as those for the original module in such a section. Notice, however, that object-message rules (see Section 11.2) are also transformed in our instrumentation, and therefore the results for `frewrite` will most probably be different.

To execute our examples, we define a module `INSTRUMENTATION-TEST` which includes the modules `INSTRUMENTATION`, `INSTRUMENTATION-INFRASTRUCTURE`, and `BANK-ACCOUNT`; the last two are included to be able to use the `upTerm` and `downTerm` functions (see Section 14.5.1).

```

mod INSTRUMENTATION-TEST is
  protecting INSTRUMENTATION .
  protecting INSTRUMENTATION-INFRASTRUCTURE .
  protecting BANK-ACCOUNT .

ops A-001 A-002 A-003 : -> Oid .

var H : Header .
var IL : ImportList .
var SS : SortSet .
var SSDS : SubsortDeclSet .
vars OPDS OPDS' : OpDeclSet .
var MAS : MembAxSet .
var EqS : EquationSet .
var RLS : RuleSet .

op addOps : Module OpDeclSet -> Module .
eq addOps(mod H is IL sorts SS . SSDS OPDS MAS EqS RLS endm, OPDS')
  = mod H is IL sorts SS . SSDS OPDS OPDS' MAS EqS RLS endm .
eq addOps(fmod H is IL sorts SS . SSDS OPDS MAS EqS endfm, OPDS')
  = fmod H is IL sorts SS . SSDS OPDS OPDS' MAS EqS endfm .
endm

```

We can now perform some of the rewrites in Section 11.1.

```

Maude> red in INSTRUMENTATION-TEST :
      downTerm(
        getTerm(

```

```

metaRewrite(
  addOps(instrument('BANK-ACCOUNT),
    op 'A-001 : nil -> 'Oid [ctor] .
    op 'A-002 : nil -> 'Oid [ctor] .
    op 'A-003 : nil -> 'Oid [ctor] .),
  '{_','_'}[upTerm(< A-001 : Account | bal : 300 >
    debit(A-001, 200)
    debit(A-001, 150)
    < A-002 : Account | bal : 250 >
    debit(A-002, 400)
    < A-003 : Account | bal : 1250 >
    (from A-003 to A-002 transfer 300)),
    'nil.QidList],
  unbounded)),
{(none).Configuration, (nil).QidList}) .
result InstrConfig:
{ debit(A-001, 200)
  < A-001 : Account | bal : 150 >
  < A-002 : Account | bal : 150 >
  < A-003 : Account | bal : 950 >,
  'debit 'transfer 'debit }

```

Notice that the resulting term consists of the final configuration plus a list of the labels of the rules used in the rewrite, that is, we know the sequence of steps taken. We can also use the instrumented module for doing search.

```

Maude> red in INSTRUMENTATION-TEST :
metaSearch(
  addOps(instrument('BANK-ACCOUNT),
    op 'A-001 : nil -> 'Oid [ctor] .
    op 'A-002 : nil -> 'Oid [ctor] .
    op 'A-003 : nil -> 'Oid [ctor] .),
  '{_','_'}[
  upTerm(< A-001 : Account | bal : 300 >
    debit(A-001, 200)
    debit(A-001, 150)
    < A-002 : Account | bal : 250 >
    debit(A-002, 400)
    < A-003 : Account | bal : 1250 >
    (from A-003 to A-002 transfer 300)),
    'nil.QidList],
  '{_','_'}[
  upTerm(C:Configuration debit(A-001, 150)),
  'QL:QidList],
  nil, '!, unbounded, 1) .
result ResultTriple:
{'{'_','_'}[
  '[_]'debit['A-001.Oid,'s_~150['0.Zero]],
  '<:_|_>['A-001.Oid,'Account.Cid,'bal':_['s_~100['0.Zero]]],

```

```

    '<:_|_>['A-002.Oid,'Account.Cid,'bal':_['s_~150['0.Zero]]],
    '<:_|_>['A-003.Oid,'Account.Cid,'bal':_['s_~950['0.Zero]]],
    '[_['transfer.Qid,'debit.Qid, 'debit.Qid]],
  'InstrConfig,
  'C:Configuration <-
    '[_['<:_|_>['A-001.Oid,'Account.Cid,'bal':_['s_~100['0.Zero]]],
      '<:_|_>['A-002.Oid,'Account.Cid,'bal':_['s_~150['0.Zero]]],
      '<:_|_>['A-003.Oid,'Account.Cid,'bal':_['s_~950['0.Zero]]] ;
  'QL:QidList <- '[_['transfer.Qid, 'debit.Qid, 'debit.Qid]}

```

We can use the `getLabels` function in the `INSTRUMENTATION-INFRASTRUCTURE` module to extract the path to the found state.

```

Maude> red in INSTRUMENTATION-TEST :
  getLabels(
    downTerm(
      getTerm(
        metaSearch(
          addOps(instrument('BANK-ACCOUNT),
            op 'A-001 : nil -> 'Oid [ctor] .
            op 'A-002 : nil -> 'Oid [ctor] .
            op 'A-003 : nil -> 'Oid [ctor] .),
          '{_','_'}[
            upTerm(< A-001 : Account | bal : 300 >
              debit(A-001, 200)
              debit(A-001, 150)
              < A-002 : Account | bal : 250 >
              debit(A-002, 400)
              < A-003 : Account | bal : 1250 >
              (from A-003 to A-002 transfer 300)),
            'nil.QidList],
          '{_','_'}[
            upTerm(C:Configuration debit(A-001, 150)),
            'QL:QidList],
            nil, '!, unbounded, 1)),
          {(none).Configuration, (nil).QidList})) .
  result NeTypeList: 'transfer 'debit 'debit

```

This sequence is actually quite similar to the result that we get using the `show path labels` command at the object level.

The `upModule` function allows us to move a module up to the metalevel, where it can be manipulated; then, the instrumented metamodule obtained from the transformation is used as argument of the `metaRewrite` and `metaSearch` descent functions. However, it is worth remarking that the instrumented module is at the metalevel, and that it cannot be moved down to the object level to be used there. We will see in Section 19.7.2 how, in the context of Full Maude, we can add functionality for making modules generated at the metamodule available at the object level.

16.3 A deadlock-freedom transformation

One of the technical requirements for using the abstraction techniques presented in Section 13.4 for LTL model checking a theory is that such a theory should be *deadlock free*, at least for the states reachable from an initial state. As pointed out in such section, we can always transform a rewrite theory \mathcal{R} (with no rules with rewrites in their conditions) into another theory bisimilar to it that is deadlock free. Specifically, we can always associate to such an executable rewrite theory \mathcal{R} a semantically equivalent (from the LTL point of view) theory \mathcal{R}_{df} which is both deadlock free and executable. We present in this section a Maude specification of the transformation proposed in [256, 294]. In the rest of this section, we assume that the rewrite theories considered do not have conditional rewrite rules with rewrites in their conditions.

The reason why deadlock freedom is imposed on rewrite theories is because deadlocks can pose a problem, due to a technical point in the Kripke structure semantics of LTL. The transition relation of a Kripke structure is *total* (see Section 13.2), and this requirement is also imposed on the Kripke structures arising from rewrite theories.

One simple way to deal with this difficulty is to just add idle transitions for each of the states in the resulting specification by means of a rule of the form $X \Rightarrow X$. The resulting system, in addition to all the rules that the minimal system should contain, may in fact have some extra “junk” transitions that are not part of it. Therefore, we would end up with a system that can be soundly used to infer properties of the original system (it is immediate to see that for any equational abstraction we have a simulation map) but that in general would be coarser than the minimal system.

A better way of addressing the problem is to characterize the set of deadlock states. For this, given a rewrite theory \mathcal{R} , we introduce a new operator *enabled* : $k \rightarrow [Bool]$ for each kind k in \mathcal{R} that will be *true* for a term iff there is a rule that can be applied to it.

Given a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, we define an extension (Σ', E') of its equational part by adding:

1. for each kind k in Σ , a new operator *enabled* : $k \rightarrow [Bool]$ in Σ' ;
2. for each rule $l \rightarrow r$ **if** C in R , an equation *enabled*(l) = *true if* C in E' , and
3. for each operator $f : k_1 \dots k_n \rightarrow k$ in Σ and for each i with $1 \leq i \leq n$ such that i is not a frozen argument position, the equation

$$\textit{enabled}(f(x_1, \dots, x_n)) = \textit{true if } \textit{enabled}(x_i) = \textit{true}.$$

The *enabled* predicate and its properties are the key ingredients for the transformation of an executable rewrite theory into a semantically equivalent one that is both deadlock free and executable. Given an executable rewrite theory $\mathcal{R} = (\Sigma, E \cup A, R)$ and a chosen kind of states k , we construct an executable theory $\mathcal{R} \subseteq \mathcal{R}_{df}^k = (\Sigma', E' \cup A, R')$ by extending the equational

theory (Σ, E) in \mathcal{R} with an *enabled* predicate as explained above, and by adding a new kind k' , a new operator $\{-\} : k \rightarrow k'$, and the rule

$$\{x\} \rightarrow \{x\} \text{ if } \text{enabled}(x) \neq \text{true}$$

to R .

Such transformation satisfies the following properties (see [256, 294] for the corresponding proofs):

- \mathcal{R}_{df}^k is k' -deadlock free and k' -encapsulated for a certain kind k' ;
- there is a function $h : T_{\Sigma', k'} \rightarrow T_{\Sigma, k}$ inducing a bijection

$$h : T_{\Sigma'/E' \cup A, k'} \rightarrow T_{\Sigma/E \cup A, k}$$

such that for each $t, t' \in T_{\Sigma', k'}$ we have

$$h(t)(\rightarrow_{\mathcal{R}, k}^1) \bullet h(t') \iff t \rightarrow_{\mathcal{R}_{df}^k, k'}^1 t'.$$

Furthermore, if Π are state predicates for \mathcal{R} and k defined by equations D , then we can define state predicates Π for \mathcal{R}_{df}^k and k' by equations D' such that the above map h becomes a bijective AP_{Π} -bisimulation

$$h : \mathcal{K}(\mathcal{R}_{df}^k, k')_{\Pi} \rightarrow \mathcal{K}(\mathcal{R}, k)_{\Pi}.$$

The `deadlock-free` function in the `DEADLOCK-FREEDOM` module below takes (the metarepresentation of) a module and a sort, and gives a new module resulting from applying the transformation described above to the given module, taking the kind of the given sort as the kind of states.

```
fmod DEADLOCK-FREEDOM is
pr META-LEVEL .
pr CONVERSION .
pr (SET * (op empty to empty*,
           op _',_ to *_*,
           op insert to insert*,
           op delete to delete*,
           op _in_ to _in*_,
           op |_| to |_|*,
           op $card to $card*,
           op union to union*,
           op intersection to intersection*,
           op $intersect to $intersect*,
           op _\_ to _\*_,
           op $diff to $diff*,
           op _subset_ to _subset*_,
           op _psubset_ to _psubset*_)) {TypeList} .

op deadlock-free : Module Sort -> Module .

var F : Qid .
var H : Header .
var M : Module .
```

```

var N : Nat .
vars T T' : Term .
var TL : TermList .
vars IL IL' : ImportList .
var S : Sort .
vars SS SS' : SortSet .
vars SSDS SSDS' : SubsortDeclSet .
var OPD : OpDecl .
vars OPDS OPDS' : OpDeclSet .
var MAS : MembAxSet .
vars EqS EqS' : EquationSet .
vars RLS RLS' : RuleSet .
vars Tp Tp' : Type .
vars TpL TpL' : TypeList .
var TpLS : Set{TypeList} .
var TpS : TypeSet .
var AtS : AttrSet .
var Cd : Condition .
vars NL NL' : NatList .

```

The auxiliary functions `addSorts`, `addImports`, `addOps`, `addEqs`, and `addEqs` allow us to add the corresponding declarations to the module given as argument.

```

op addSorts : Module SortSet ~> Module .
eq addSorts(mod H is IL sorts SS . SSDS OPDS MAS EqS RLS endm, SS')
  = mod H is IL sorts SS ; SS' . SSDS OPDS MAS EqS RLS endm .
eq addSorts(fmod H is IL sorts SS . SSDS OPDS MAS EqS endfm, SS')
  = fmod H is IL sorts SS ; SS' . SSDS OPDS MAS EqS endfm .

op addImports : Module ImportList ~> Module .
eq addImports(mod H is IL sorts SS . SSDS OPDS MAS EqS RLS endm, IL')
  = mod H is IL IL' sorts SS . SSDS OPDS MAS EqS RLS endm .
eq addImports(fmod H is IL sorts SS . SSDS OPDS MAS EqS endfm, IL')
  = fmod H is IL IL' sorts SS . SSDS OPDS MAS EqS endfm .

op addOps : Module OpDeclSet ~> Module .
eq addOps(mod H is IL sorts SS . SSDS OPDS MAS EqS RLS endm, OPDS')
  = mod H is IL sorts SS . SSDS OPDS OPDS' MAS EqS RLS endm .
eq addOps(fmod H is IL sorts SS . SSDS OPDS MAS EqS endfm, OPDS')
  = fmod H is IL sorts SS . SSDS OPDS OPDS' MAS EqS endfm .

op addEqs : Module EquationSet ~> Module .
eq addEqs(mod H is IL sorts SS . SSDS OPDS MAS EqS RLS endm, EqS')
  = mod H is IL sorts SS . SSDS OPDS MAS (EqS EqS') RLS endm .
eq addEqs(fmod H is IL sorts SS . SSDS OPDS MAS EqS endfm, EqS')
  = fmod H is IL sorts SS . SSDS OPDS MAS (EqS EqS') endfm .

op addRLs : Module RuleSet ~> Module .

```

```

---- It makes a functional module to become a system module
eq addRls(mod H is IL sorts SS . SSDS OPDS MAS EqS R1S endm, R1S')
  = mod H is IL sorts SS . SSDS OPDS MAS EqS EqS R1S R1S' endm .
eq addRls(fmod H is IL sorts SS . SSDS OPDS MAS EqS endfm, R1S)
  = mod H is IL sorts SS . SSDS OPDS MAS EqS R1S endm .

```

The following `frozen` function checks whether the specified argument position of the operator given as argument is declared frozen or not. The `id` function returns the identity element of the given operator declaration; notice that this operator is declared at the kind level, so that an operator declaration with no identity element will produce an error term.

```

op frozen : OpDecl Nat -> Bool .
eq frozen(op F : TpL -> Tp [frozen(NL N NL') AtS] ., N)
  ---- we could assume that all non-ctor operators are frozen
  = true .
eq frozen(OPD, N) = false [owise] .

```

```

op id : OpDecl ~> Term .
eq id(op F : TpL -> Tp [id(T) AtS] .) = T .

```

The `enabled-eqs` function creates an equation $enabled(l) = true$ if C for each rule $l \rightarrow r$ if C in the set of rules given as argument.

```

op enabled-eqs : RuleSet -> EquationSet .
eq enabled-eqs(r1 T => T' [AtS] . R1S)
  = (eq 'enabled[T] = 'true.Bool [none] . enabled-eqs(R1S)) .
eq enabled-eqs(crl T => T' if Cd [AtS] . R1S)
  = (ceq 'enabled[T] = 'true.Bool if Cd [none] .
      enabled-eqs(R1S)) .
eq enabled-eqs((none).RuleSet) = none .

```

Given an operator $f : k_1 \dots k_n \rightarrow k$, the following `enabled-eqs` function creates an equation $enabled(f(x_1, \dots, x_n)) = true$ if $enabled(x_i) = true$ for each i with $1 \leq i \leq n$. Notice that the conditional equation is only introduced for unfrozen argument positions. To avoid nonterminating equations when an operator has an identity axiom, the equations generated for such an operator include conditions checking for arguments different to the corresponding identity elements.

```

op enabled-eqs : RuleSet -> EquationSet .
op enabled-eqs : OpDeclSet TypeSet -> EquationSet .
op enabled-eqs1 : OpDecl Set{TypeList} -> EquationSet .
op enabled-eqs2 : OpDecl TypeList TermList Nat -> EquationSet .
op unfoldUniversal : OpDecl TypeSet -> Set{TypeList} .
op unfoldUniversal1 : TypeList TypeSet -> Set{TypeList} .
op unfoldUniversal2 : TypeList Type -> Set{TypeList} .
op enabled-args : TypeList Nat -> TermList .
op make-cond : TermList Term -> EqCondition .

```

```

eq enabled-eqs(r1 T => T' [AtS] . R1S)
  = (eq 'enabled[T] = 'true.Bool [none] . enabled-eqs(R1S)) .
eq enabled-eqs(crl T => T' if Cd [AtS] . R1S)
  = (ceq 'enabled[T] = 'true.Bool if Cd [none] .
      enabled-eqs(R1S)) .
eq enabled-eqs((none).RuleSet) = none .

eq enabled-eqs(OPD OPDS, TpS)
  = enabled-eqs1(OPD, unfoldUniversal(OPD, TpS))
    enabled-eqs(OPDS, TpS) .
eq enabled-eqs((none).OpDeclSet, TpS) = none .

eq unfoldUniversal(op F : TpL -> Tp [poly(NL) AtS] ., TpS)
  = unfoldUniversal1(TpL, TpS) .
eq unfoldUniversal(op F : TpL -> Tp [AtS] ., TpS) = TpL [owise] .

eq unfoldUniversal1(TpL, Tp ; TpS)
  = unfoldUniversal2(TpL, Tp) * unfoldUniversal1(TpL, TpS) .
eq unfoldUniversal1(TpL, none) = empty* .

eq unfoldUniversal2('Universal TpL, Tp) = Tp unfoldUniversal2(TpL, Tp) .
eq unfoldUniversal2(Tp TpL, Tp') = Tp unfoldUniversal2(TpL, Tp') [owise] .
eq unfoldUniversal2(nil, Tp) = nil .

eq enabled-eqs1(op F : TpL -> Tp [AtS] ., TpL' * TpLS)
  = enabled-eqs2(op F : TpL -> Tp [AtS] ., TpL', enabled-args(TpL', 0), 0)
    enabled-eqs1(op F : TpL -> Tp [AtS] ., TpLS) .
eq enabled-eqs1(op F : TpL -> Tp [AtS] ., empty*) = none .

eq enabled-eqs2(op F : TpL -> Tp [AtS] ., Tp' TpL', TL, N)
  --- id-left and id-right should also be considered
  = (if frozen(op F : TpL -> Tp [AtS] ., N)
      then none
      else if id(op F : TpL -> Tp [AtS] .) :: Term
          then ceq 'enabled[F[TL]] = 'true.Bool
                if make-cond(TL, id(op F : TpL -> Tp [AtS] .))
                /\ 'enabled[qid("V" + string(N, 10)
                    + ":" + string(Tp'))] = 'true.Bool
                [none] .
          else ceq 'enabled[F[TL]] = 'true.Bool
                if 'enabled[qid("V" + string(N, 10)
                    + ":" + string(Tp'))] = 'true.Bool
                [none] .
      fi
      fi
      enabled-eqs2(op F : TpL -> Tp [AtS] ., TpL', TL, s N)) .
eq enabled-eqs2(OPD, nil, TL, N) = none .

```

```

eq enabled-args(Tp TpL, N)
  = (qid("V" + string(N, 10) + ":" + string(Tp)),
     enabled-args(TpL, s N)) .
eq enabled-args(nil, N) = empty .

eq make-cond((T, TL), T')
  = ('_=/=_[T, T'] = 'true.Bool) /\ make-cond(TL, T') .
eq make-cond(empty, T) = nil .

```

Finally, the following equation defines the main function `deadlock-free`, adding the operator declarations and equations necessary to make the module given as argument `deadlock-free`.

```

eq deadlock-free(M, S)
  = addRls(
    addSorts(
      addOps(
        addEqs(M,
          (enabled-eqs(getRls(M))
            enabled-eqs(getOps(M), getKinds(M)))),
          (op 'enabled : 'Universal -> 'Bool [poly(1)] .
            op '{_}' : S -> 'EConfig [none] .)),
        'EConfig ; 'Universal),
    (crl '{_}'[qid("V:" + string(S))]
      => '{_}'[qid("V:" + string(S))]
      if '_=/=_['enabled[qid("V:" + string(S))], 'true.Bool]
        = 'true.Bool
      [none] .)) .
endfm

```

We illustrate the use of the `deadlock-free` operator on the bakery example presented in Section 13.4. We can transform the `ABSTRACT-BAKERY` module introduced in such section by using the following command.

```

Maude> reduce in DEADLOCK-FREEDOM :
  deadlock-free(upModule('ABSTRACT-BAKERY, true), 'BState) .

```

We will see in Section 19.7.2 how to use the resulting module for model checking properties, for example.

Mobile Maude

with Adrián Riesco
and Alberto Verdejo

The popularity of the Internet has brought much attention to the world of distributed applications development. Now, more than ever, the network is being viewed as a platform for the development of cost-effective, mission-critical applications. Mobile code and mobile agents [218, 213] are emerging technologies that promise to make much easier the design, implementation, and maintenance of distributed systems. Mobile agents may reduce the network traffic, provide an effective means of overcoming network latency, and, perhaps more importantly, help us to construct more robust and fault-tolerant systems, thanks to their ability to operate asynchronously and autonomously.

Mobile Maude is a mobile agent language extending Maude and supporting mobile computation. Mobile Maude uses reflection to obtain a simple and general declarative mobile language design and makes possible strong assurances about mobile agent behavior. The formal semantics of Mobile Maude is given by a rewrite theory in rewriting logic. Since this specification is executable, it can be used as a prototype of the language, in which mobile agent systems can be simulated. The two key notions are *processes* and *mobile objects*. Processes are located computational environments where mobile objects can reside. Mobile objects have their own code, can move between different processes in different locations, and can communicate asynchronously with each other by means of messages. An overall Mobile Maude configuration is a “soup” of processes, possibly with some mobile objects and messages “in transit” from one process to another. The code of a mobile object is given by (the metarepresentation of) an object-based module—a rewrite theory—and its data is given by (the metarepresentation of) a configuration of objects and messages (internal to the mobile object itself) that represent its state. Such a configuration is a valid term in the object’s code module. The mobile object changes its state by executing its own code at the metalevel. Mobile Maude’s key characteristics include:

- reflection as a way of endowing mobile objects with “higher-order” capabilities;

- object-orientation and asynchronous message passing; and
- a simple semantics without any loss in the expressive power of application code.

We first introduced Mobile Maude in [113], where we presented a “simulator” of Mobile Maude, an executable Maude specification on top of Maude 1.0.5, in which the system code was written entirely in Maude, and thus locations and processes were encoded as Maude terms. In the same paper, we also sketched a development plan including two further development efforts: a first step in which a single-host executable would be implemented, and a second implementation effort focussing on true distributed execution.

The release of Maude 2.0 allowed us to take the first step. This implementation effort was completed in a very short time, using the built-in object system for object/message fairness, just by simplifying and extending the previous specification. This new version was developed by Durán and Verdejo, and was used in several examples, one of which was reported in [131].

The built-in string handling and internet socket module available since Maude 2.2 has allowed us to build a truly distributed implementation, thus advancing the second development effort. The Maude 2.2 socket modules support non-blocking client and server TCP sockets (see Section 11.4.1). In this implementation effort, a Mobile Maude server runs on top of a Maude interpreter and performs the following tasks:

1. keeps track of the current locations of mobile objects created on a host,
2. handles change of location messages,
3. reroutes messages to mobile objects, and
4. runs the code of mobile objects by invoking the metalevel.

In fact, we have introduced quite a significant number of changes into Mobile Maude. Processes and locations are no longer part of the Mobile Maude specification. We now talk about Maude processes—not terms, but OS processes, which may be running on different machines—and IP addresses. We have also introduced the notion of *root objects* as managers of the configurations of mobile objects in the different processes.

We explain below the design of processes and mobile objects and their rewriting semantics, based on a formal specification of Mobile Maude written in Maude. For presentation purposes, some of the declarations and rules given here have been simplified. The complete code for Mobile Maude and of the corresponding examples is available from <http://maude.cs.uiuc.edu> and also in the companion cd-rom.

17.1 Processes and mobile objects

17.1.1 Processes and root objects

The key entities in Mobile Maude are *processes* and *mobile objects*.

Mobile objects are modeled as distributed objects in the class `MobileObject`. A *distributed configuration* is made up of located configurations. Each located configuration is executed in a Maude process. Such processes can therefore be seen as *located* computational environments *inside which* mobile objects can reside, execute, and send and receive messages to and from other mobile objects located in different processes. We assume that each located configuration has exactly one *root object*, of class `RootObject`, which keeps information about the location of the process, the mobile objects in such a configuration, and the whereabouts of the mobile objects created in it, which may have moved to other processes. The *names* of root objects range over the sort `Loc`, and have the form `l(IP, N)` with `IP` the IP address of the machine in which the process is being executed and `N` a number. We assume uniqueness of root object names in a distributed configuration.

The class `RootObject` of root objects is declared as follows:¹

```
class RootObject |
  cnt : Nat,                *** counter to generate names
  guests : Set{Oid},         *** objects in the location
  forward : Map{Nat, Tuple{Loc, Nat}},
                            *** forwarding information
  state : RootObjectState,  *** idle, waiting-connection, active
  neighbors : Map{Loc, Oid}, *** each location has a socket to be
                            *** used to sent messages to it
  defNeighbor : Maybe{Oid} . *** default socket
```

The root object of each process keeps information about the mobile objects currently in it in the `guests` attribute. Mobile objects are named with identifiers of the form `o(L, N)`. The root object's `cnt` attribute stores a counter to generate unique names for such new mobile objects.

Finally, a root object may be in state `idle`, `waiting-connection`, or `active`. The attribute `state` will take one of these values. Root objects are only idle when they are created, their first action being their activation either as a client or as a server socket. They stay in `waiting-connection` state until they get the confirmation from the server socket, passing then to `active` mode, the state in which they will develop their normal activity.

¹ We use here the Full Maude notation for defining classes (see Section 21.1.2). The corresponding declarations in Core Maude for the class `RootObject` are:

```
sort RootObject .
subsort RootObject < Cid .
op RootObject : -> RootObject [ctor] .
op cnt :_ : Nat -> Attribute [ctor gather(&)] .
op guests :_ : Set{Oid} -> Attribute [ctor gather(&)] .
op forward :_ : Map{Nat, Tuple{Loc, Nat}} -> Attribute
  [ctor gather(&)] .
op state :_ : RootObjectState -> Attribute [ctor gather(&)] .
op neighbors :_ : Map{Loc, Oid} -> Attribute [ctor gather(&)] .
op defNeighbor :_ : Maybe{Oid} -> Attribute [ctor gather(&)] .
```

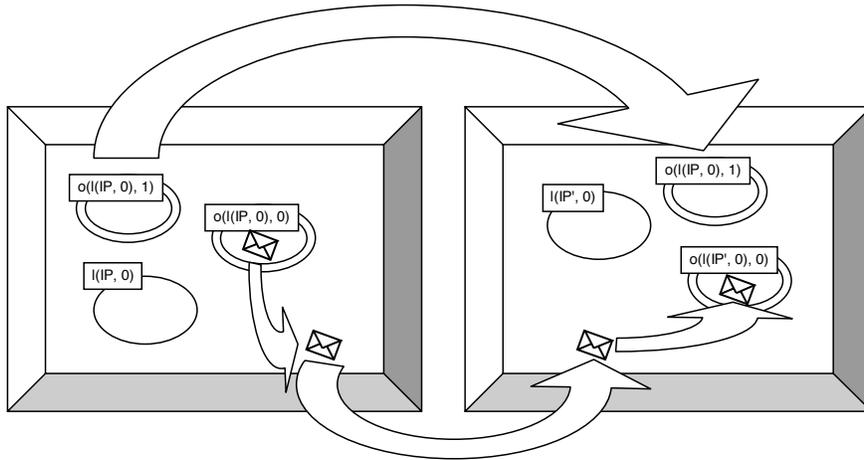


Fig. 17.1. Object and message mobility

17.1.2 Mobile objects

Mobile objects carry their own internal state and code (an object-based system module) with them, can move from one process to another, and can communicate with each other by asynchronous message passing. The names of mobile objects range over the sort `Mid` and have the form $o(L, N)$, with L the name of the root object of the process in which it was created and N a number. Figure 17.1 shows several mobile objects in two processes, with (mobile) object $o(l(IP, 0), 1)$ moving from the process with root object $l(IP, 0)$ to the process of root object $l(IP', 0)$, and with object $o(l(IP, 0), 0)$ sending a message to $o(l(IP', 0), 0)$.

Mobile objects are specified as objects of the class `MobileObject`:²

```
class MobileObject |
  mod : Module,          *** rewrite rules of the mobile object
  s : Term,              *** current state
  gas : Nat,             *** bound on resources
  hops : Nat .          *** number of hops
```

² We use here again the Full Maude notation for defining classes. The corresponding declarations in Core Maude for the class `MobileObject` are:

```
sort MobileObject .
subsort MobileObject < Cid .
op MobileObject : -> MobileObject [ctor] .
op mod :_ : Module -> Attribute [ctor gather(&)] .
op s :_ : Term -> Attribute [ctor gather(&)] .
op gas :_ : Nat -> Attribute [ctor gather(&)] .
op hops :_ : Nat -> Attribute [ctor gather(&)] .
```

The sorts `Module` and `Term`, associated with the attributes `mod` and `s`, respectively, are sorts in the module `META-LEVEL`. The value of a mobile object's `mod` attribute is the metarepresentation of an object-based system module. The mobile object's `state s` must be the metarepresentation of a pair of configurations meaningful for the module in `mod` and having the form `C & C'`, where `C` is a configuration of objects and messages representing unprocessed incoming messages and inter-inner-objects messages, and `C'` is a multiset of messages representing the *outgoing* messages tray. One of the objects in the configuration of objects and messages is supposed to have the same identifier as the mobile object it is in. We sometimes refer to this object as the *main* one, which in most cases will be the only one. Therefore, we can think of a mobile object as a *wrapper* that encapsulates the state and code of its inner object and mediates its communication with other objects. For this reason, Figure 17.1 depicts mobile objects by two concentric circles, with the inner object and its incoming and outgoing messages contained in the inner circle.

A `MobileObject` includes the attribute `hops`, which stores the number of “hops” a mobile object has performed while moving from one process to another. This information is necessary for the forwarding process, which is discussed later in this section. To guarantee that all mobile objects eventually have some activity, and to impose a bound on the resources they can consume, they have a `gas` attribute.

17.1.3 Message forwarding

Since mobile objects may move from one process to another, reaching them by messages is nontrivial. The solution adopted in Mobile Maude is that, when a message's addressee is not in the current process, the message is forwarded to the addressee's parent process (the process it was created at). Each root object stores forwarding information about the whereabouts of its children in its `forward` attribute, a partial function in `Map{Nat, Tuple{Loc, Nat}}` that maps child number n to a pair consisting of the name of the located process in which the object currently resides, and the number of “hops” to different processes that the mobile object has taken so far. The number of hops is important in disambiguating situations when old messages (containing old location information) arrive after newer ones containing the current location. The most recent location is that associated with the largest number of hops. Whenever a mobile object moves to a new process, the object's parent process is always notified. Note that this system does not guarantee message delivery in the case that objects move more rapidly than messages.

In the previous versions of Mobile Maude [113, 131], all the processes were in the same configuration, and reaching a particular process was represented by one single rule. However, the current version uses TCP sockets to connect processes. Therefore, when a mobile object moves to a different location, or a message is sent to a mobile object in a different location, we need to know

which socket must be used to send the information. The root object in the process is in charge of sending the message or the mobile object through the appropriate socket.³ Assuming that all processes are directly connected to each other is not realistic, would severely limit the number of processes we could connect, and would make the task of connecting a new process really expensive. Fortunately, connectivity between two nodes does not necessarily imply a direct connection between them. An indirect connectivity may be achieved among a set of cooperating nodes. Nevertheless, just because a set of hosts are directly or indirectly connected to each other does not mean that we have succeeded in providing host-to-host connectivity. When a source node wants the network to deliver a message to a certain destination node, it specifies the address of the destination node. If the sending and receiving nodes are not directly connected, then the nodes of the network between them—switchers and routers—use this address to decide how to forward the message toward the destination. The process of determining systematically how to forward messages toward the destination node based on its address—which is usually called *routing*—is nontrivial.⁴ Here, we assume a very simple, although quite general, approach consisting in having a routing table in each root object. Such a table specifies the socket through which a message must be sent if one wants to reach a particular location. If there is a socket between the source and the target of the message, then it reaches its destination in a single step; otherwise, the forwarding has to be repeated several times. The `neighbors` attribute maintains such a routing table as a map, associating socket object identifiers to location identifiers. That is, the attribute `neighbors` holds a partial function `Map{Loc, Oid}` providing information on the sockets (identified by an `Oid`) through which data should be sent to reach a particular location.

In case there is no socket associated with a particular location in the `neighbors` map, a default socket may be specified in the `defNeighbor` attribute. Since the value of the `defNeighbor` attribute may also be undefined, we import the module `MAYBE` from Section 8.3.3 instantiated with `Oid` and renaming the constant `maybe` to `null`; then, we declare the attribute `defNeighbor` of sort `Maybe{Oid}`, so that it can take as value either an object identifier or the constant `null`.

If there is no socket associated with a particular location and a default one has not been specified then the data is not delivered. Note that this model allows us to represent many different network architectures, since the routing information may be updated and used in a very flexible way (although we do

³ As we will see in the coming sections, root objects send messages through buffered sockets. We discuss the use of sockets and buffered sockets in Section 17.4.

⁴ We only consider the case of a source node wanting to send a message to a single destination node (*unicast*). The cases of *multicasting*—the source node wants to send a message to some subset of the nodes on the network—and *broadcasting*—the source node wants to send a message to all the nodes on the network—could similarly be specified.

not care here about this). We explain how to build a very simple architecture in Section 17.4.

17.2 Mobile Maude additional definitions

Mobile Maude *system code* is specified by a relatively small number of rules for root objects, mobile objects, mobility, and message passing. Such rules work in an *application-independent* way. Application code, on the other hand, can be written as Maude object-oriented modules with great freedom, except for being aware that, as explained in Section 17.1.2, the top level of the internal state of a mobile object has to be a pair of configurations, with the second component, called *outgoing tray*, containing outgoing messages and the first, called the *inner configuration*, containing the inner object(s) and incoming messages. Such a pair is built with the constructor

```
sort MobObjState .
op &_amp;_ : Configuration Configuration -> MobObjState [ctor] .
```

The messages sent or received by a mobile object must be of the form

1. `go(L)`,
2. `go-find(O, L)`,
3. `newo(Mod, Conf, O)`,
4. `to O : C`, or
5. `kill`,

for `L` a location (of sort `Loc`), `O` a mobile object identifier (of sort `Mid`), `C` a term of sort `Contents`, `Mod` a term of sort `Module`, and `Conf` a term of sort `Configuration`.

Such messages may in fact be understood as commands that the main object—or one of the other objects—in the internal state of a mobile object gives to its wrapper object. Thus, an object may

1. request to move from its current location to a given location `L` with the `go(L)` message;
2. request going to the location in which the mobile object `O` resides, which is possibly `L`, with the message `go-find(O, L)`;
3. request creating a new mobile object with module `Mod`, initial state `Conf`, and temporal identifier of the main object in such a configuration `O`, with the message `newo(Mod, Conf, O)`;
4. send a message with contents `C` to the object `O` with the message `to O : C`; and
5. request the destruction of the mobile object it resides into with the message `kill`.

The commands/messages available in Mobile Maude are defined in the module `MOBILE-OBJECT-INTERFACE`, which is assumed to be imported by the user in all Mobile Maude applications. Among others, such module includes the declarations of the previous messages.

```

sort Contents .

op go : Loc -> Msg [ctor message] .
op go-find : Mid Loc -> Msg [ctor message] .
op newo : Module Configuration Oid -> Msg [ctor message] .
op to_:_ : Mid Contents -> Msg
  [ctor message format(!m s s o) gather (& &)] .
op kill : -> Msg [ctor message] .

```

Note that messages being sent to other mobile objects must be of the form `to_:_`, with the addressee of the message as first argument and a term of sort `Contents` as second argument. The definition of the `Contents` sort is left to each particular application, which in fact gives the user the freedom to define any kind of message, with the only requirement of having the identifier of the addressee as first argument.

The `newo` message takes a module (a term of sort `Module` metarepresenting a module), a term of sort `Configuration` (which will be the initial configuration in the belly of the mobile object to be created, so it should make sense in the module given as first argument), and the provisional identifier of the main object in the configuration given as second argument. As we shall see in Section 17.3.4, the first action accomplished by a mobile object when it detects the `newo` command is creating a new mobile object with the metarepresentation of the configuration given as second argument to the `newo` message, and then sending a `start-up` message to the main object with its new name, so that it coincides with the name of the mobile object it is in. Let us recall that the name of a mobile object depends on the root object in its process, and on the number of mobile objects already created in it. Therefore, such a name cannot be known when the creation is requested. Thus, the main object in the configuration will be created with a provisional identifier—usually `tmp-id`—that will be changed by its mobile object once it is created. We can also find in the module `MOBILE-OBJECT-INTERFACE` the following declarations.

```

op tmp-id : -> Mid [ctor] .
op start-up : Mid -> Contents [ctor] .

```

17.3 Mobile Maude's rewriting semantics

The semantics of Mobile Maude is specified by an object-oriented rewrite theory containing the definitions of the classes `RootObject` and `MobileObject` and rewrite rules that describe the behavior of the different primitives: object

mobility, message passing, and object and process creation. This specification is the *system code* of Mobile Maude, which works in an application-independent way as a prototype on which to execute Mobile Maude applications. Such applications need of course to satisfy certain requirements, as being object-oriented, using the `_&_` constructor for sending messages out of the mobile objects, and using the primitive messages for moving to other processes.

17.3.1 Letting mobile objects do something

The rules discussed in the next subsections specify the way in which the different Mobile Maude commands are handled. To allow the mobile objects to evolve, and therefore to allow the application code, which “lives” inside the mobile objects, to invoke such commands, the state of the mobile objects must be rewritten. In the `do-something` rule below, the internal state of a mobile object is rewritten using the rules of the module in its `mod` attribute. Instead of using the deterministic functions `metaRewrite` or `metaFrewrite`, the `metaSearch` function is used in the equations for the operation `getPossibleTerms` to obtain the set of terms that can be reached in one rewriting step. From all these possible rewrites one is chosen in the condition of the `do-something` rule; this allows us to explore all possible executions as we discuss in Section 17.6.

As a fairness condition, or, more concretely, to make sure that no mobile object consumes all the resources—to avoid, for example, that when rewriting the state of a mobile object we get into an infinite computation—and to try to balance such consumption, we establish a bound on the number of rewrites for each of the mobile objects. Such a bound is given in their `gas` attribute. Each time the `do-something` rule is applied, the mobile object’s `gas` value is decremented. Note that the `gas` attribute gives the number of rewrites a mobile object can perform. If no rewriting step can be taken, the `do-something` rule cannot be applied.

```

vars TL TL' : TermList .
var  RST? : ResultTriple? .

crl [do-something] :
  < 0 : V@MobileObject | mod : MOD, s : T, gas : s(N), AtS >
=> < 0 : V@MobileObject | mod : MOD, s : T', gas : N, AtS >
  if (TL, T', TL') := getPossibleTerms(MOD, T) .

op getPossibleTerms : Module Term -> TermList .
op getPossibleTerms : Module Term Nat -> TermList .

eq getPossibleTerms(MOD, T) = getPossibleTerms(MOD, T, 0) .
ceq getPossibleTerms(MOD, T, N)
  = (getTerm(RST?), getPossibleTerms(MOD, T, N + 1))

```

```

if RST? := metaSearch(MOD, T, 'M:MobObjState, nil, '+, 1, N)
  /\ RST? /= (failure).ResultTriple? .
eq getPossibleTerms(MOD, T, N) = empty [owise] .

```

Here and in what follows, variables (O , $V@MobileObject$, MOD , T , N , AtS , etc.) are written in capitals, but their sort declarations are omitted. MOD , for example, stands for a term of sort `Module`, and AtS stands for a set of attributes (of sort `AttributeSet`).

17.3.2 Object communication

There are three kinds of communication between objects. Objects inside the same mobile object can communicate with each other by means of messages with any format, and such communication may be synchronous or asynchronous. Objects in different mobile objects may communicate when such mobile objects are in the same process and when they are in different processes; in these cases, the actual kind of communication is transparent to the mobile objects, but such communication must be asynchronous, and messages must be of the form `to:_`, where the first argument is the identifier of the addressee object, and the second argument is the message contents, a value of sort `Contents` built with free user-defined syntax (see, for example, Section 17.5). That is, the minimum information needed to dispatch a message is the receiver's identity; if the sender wants to communicate its identifier, it has to include it in the message contents. If the addressee is an object in a different mobile object, then the message must be put by the sender object in the second component of its state (the outgoing messages tray). The system code will then send the message to the addressee object.

An important issue when managing messages is that the rewriting rules and state of mobile objects are metarepresented, that is, the system code of Mobile Maude is at the metalevel of the application code. Therefore, before dealing with such messages, they must be moved up, or, as we say, they must be *pulled out*. The internal state of a mobile object will have the form `'_&_[T, T']`, with T and T' the terms metarepresenting, respectively, the inner configuration and the outgoing messages. In the case of messages of the form `to:_` we will have a term of the form `'to:_[T, T']`, which may be alone or with more messages in the outgoing tray. Since we must leave the tray in a valid state we need to include the following three rules.⁵

```

r1 [message-out-to] :
  < O : V@MobileObject |
    mod : MOD, s : '_&_[T, 'to:_[T', T''']], AtS >
=> < O : V@MobileObject | mod : MOD,

```

⁵ Although in general two cases are enough to deal with associative lists (one element and more than one element), at the metalevel, since the engine is giving the list in flattened form and expects it in flattened form, we must make sure that when we have more than one element the top operator is `__`.

```

      s : ' &_[T, 'none.Configuration], AtS >
      (to downTerm(T', o(1("null", 0), 0)) { T'' }) .

rl [message-out-to] :
  < O : V@MobileObject | mod : MOD,
    s : ' &_[T, '[_[to:_[T', T'']], T'']] , AtS >
=> < O : V@MobileObject | mod : MOD, s : ' &_[T, T'']] , AtS >
      (to downTerm(T', o(1("null", 0), 0)) { T'' }) .

crl [message-out-to] :
  < O : V@MobileObject | mod : MOD,
    s : ' &_[T, '[_[to:_[T', T'']], T'']] , TL]] , AtS >
=> < O : V@MobileObject | mod : MOD, s : ' &_[T, '[_[T'']] , TL]] ,
      AtS >
      (to downTerm(T', o(1("null", 0), 0)) { T'' })
      if TL /= empty .

```

Notice that, although the contents of messages are left at the metalevel, i.e., as found, the identifier of the addressee object is moved down to the object level, so that the message can be correctly delivered. The rules pull out a message to $O : C$, which is metarepresented as $'to_:_[O, \bar{C}]$, into a message to $O \{ \bar{C} \}$. We will find similar pull-out rules for each of the commands.

Once the message is out of the mobile object, it can be appropriately delivered. The `msg-send` rules below are in charge of redirecting messages addressed to mobile objects in different locations. Notice the use of the message

```
op Send : Oid Oid Msg -> Msg [ctor msg format (b o)] .
```

to send messages to the appropriate locations. The first and second arguments of the `Send` message are, respectively, the addressee and sender of the message, and the third argument is the message being sent. We will see in Section 17.4 how the `Send` messages will be used to send the corresponding data through the appropriate sockets.

```

crl [msg-send] :
  to o(L, N) { T }
  < L : V@RootObject |
    state : active, guests : OS, forward : F, AtS >
=> < L : V@RootObject |
    state : active, guests : OS, forward : F, AtS >
    Send(p1(F[N]), L, to o(L, N) hops p2(F[N]) in p1(F[N]) { T })
    if (p1(F[N]) /= L) /\ (not o(L, N) in OS) .

crl [msg-send] :
  to o(L, N) { T }
  < O : V@RootObject |
    state : active, guests : OS, forward : F, AtS >
=> < O : V@RootObject |
    state : active, guests : OS, forward : F, AtS >

```

```

      Send(L, 0, to o(L, N) hops null in L { T })
    if (0 /= L) /\ (not o(L, N) in OS) .

```

Note that the conditions of the rules make sure not only that the object is not in the current process (`not o(L, N) in OS`), but also that the forwarding info does not point to the location itself (`p1(F[N]) /= L`), which would mean that the mobile object has not arrived to its destination yet. If the location in which the message is generated is the parent location of the mobile object the message is addressed to, then the message is forwarded to the location indicated by the forwarding info with the corresponding number of hops; otherwise, the message is forwarded to the parent location with the number of hops set to `null`. We will see below how the hops information is used in the `msg-arrive-to-proc` rules to avoid unnecessary forwarding of messages when the destination object is in transit.

The arrival of an inter-object message to a location is handled by the following five rules. We explain the case handled by each of the rules separately.

If the object is at the location, then the message is just left at the location so the object can get it.

```

r1 [msg-arrive-to-loc] :
  to o(L, N) hops H in L' { T' }
  < L' : V@RootObject | state : active, guests : (o(L, N), OS), AtS >
=> < L' : V@RootObject |
      state : active, guests : (o(L, N), OS), AtS >
  to o(L, N) { T' } .

```

If the object is not at the location and the number of hops is `null`, then the message is being sent to the mobile object's parent location. If the forwarding information is pointing to its home location, then the object is in transit, and the forwarding information has not been updated with its new location, and therefore the message is not handled; otherwise, the message is sent to the location indicated by the forwarding information with the corresponding number of hops.

```

cr1 [msg-arrive-to-loc] :
  to o(L, N) hops null in L { T }
  < L : V@RootObject |
      state : active, guests : OS, forward : F, AtS >
=> < L : V@RootObject |
      state : active, guests : OS, forward : F, AtS >
  Send(p1(F[N]), L, to o(L, N) hops p2(F[N]) in p1(F[N]) { T })
  if (not o(L, N) in OS) /\ (p1(F[N]) /= L) .

```

If the object is not at the location and the location is not its home location, then the message is forwarded back to the parent location with the same hops number. Note that, since it is not its home location, the number of hops is not `null`, that is, it is a natural number. Note also that, since the forwarding information is updated once the object has arrived to a location, it cannot be

the case that the message has arrived before the object. If the object to which the message is addressed is not at the location registered in the forwarding information, it is because the object has already left the location and the message must be returned to its home location.

```

crl [msg-arrive-to-loc] :
  to o(L, N) hops N' in L' { T }
  < L' : V@RootObject | state : active, guests : OS, AtS >
  => < L' : V@RootObject | state : active, guests : OS, AtS >
      Send(L, L', to o(L, N) hops N' in L { T } )
  if (not o(L, N) in OS) /\ (L /= L') .

```

Finally, if the message is being returned from a location to which the message was forwarded from its home location because the object already left it, then the message will be forwarded again by its home location only if its forwarding information has been updated since the message was forwarded the first time, that is, if the number of hops in the message is smaller than the number of hops in the forwarding information in its home location. Note that we do not check whether the forwarding information points to the parent location itself anymore, since in this case the hops would have been appropriately incremented.

```

crl [msg-arrive-to-loc] :
  to o(L, N) hops N' in L { T }
  < L : V@RootObject |
      state : active, guests : OS, forward : F, AtS >
  => < L : V@RootObject |
      state : active, guests : OS, forward : F, AtS >
      Send(p1(F[N]), L, to o(L, N) hops p2(F[N]) in p1(F[N]) { T })
  if (not o(L, N) in OS) /\ (N' < p2(F[N])) .

crl [msg-arrive-to-loc] :
  to 0 hops H in L' { T }
  < L : V@RootObject | AtS >
  => < L : V@RootObject | AtS >
      Send(L', L, to 0 hops H in L' { T })
  if L /= L' .

```

Once the message reaches its addressee object, the message must be inserted in—*push into*—the internal state of such a mobile object. To make sure that the mobile object will remain in a valid state, we check that the metarepresentation of the corresponding message is a valid message in the module of the object. We can assume that, since the previous state was a valid one, adding a valid message will result in a new valid state.⁶

⁶ In a previous version, we checked that the result of introducing the current message to the configuration representing the current state of the mobile object was a valid configuration.

```

rl [msg-in] :
  to 0 { T }
  < 0 : V@MobileObject | mod : MOD, s : '[_&_[T', T']', AtS >
  => if sortLeq(MOD, leastSort(MOD, 'to:_[_upTerm(0), T]'), 'Msg)
      or
      sortLeq(MOD, 'Msg, leastSort(MOD, 'to:_[_upTerm(0), T]'))
  then < 0 : V@MobileObject | mod : MOD,
        s : '[_&_[']_['to:_[_upTerm(0), T], T'], T']', AtS >
  else < 0 : V@MobileObject | mod : MOD, s : '[_&_[T', T']', AtS >
  fi .

```

17.3.3 Object mobility

We explain in this section the rules that govern object mobility. Such mobility is initiated by the mobile object's inner object, which puts the `go` or `go-find` messages in the second component (i.e., as an outgoing message) of the state. The rules for both cases are quite similar; the main difference is that a `go-find` message tries to reach a particular object that can be itself on the move; that is, we may reach the tentative location and not find the object there, in which case we must go on looking for it in a different location.

The go message

When a mobile object wants to move to another process it puts in its outgoing messages tray a `go(L)` message, where `L` is the target location. When a mobile object has an outgoing `go` message, a new *inter-mobile-objects* `go` message is sent, with the mobile object as one of its arguments, after removing the outgoing message. Since the `go` message is declared to be frozen (see Section 4.4.9), the mobile object is inactive while on the move.

If the message's sender and addressee are at different locations, then this message must be sent to the desired location, being sent through the appropriate socket by the root object of the location. When the message reaches the destination location, the root object in its home location is informed, so it can update its forwarding information; if the object has reached its home location, such information is directly updated.

If there is a `go` message in the outgoing message tray, we observe that the state of the corresponding mobile object has the form `'[_&_[T, 'go[T']]`, where the term `T'` metarepresents the name of the location where the object wants to go. Notice that in this case it must be the only message in the tray, it is assumed that any other message has already been handled. The rule `message-out-go` indicates how such a name is decoded by the `downTerm` function, and shows in its righthand side the mobile object ready to go—which is indicated by being enclosed inside a `go` operator.

```

rl [message-out-go] :
  < 0 : V@MobileObject | s : '[_&_[T, 'go[T']]', AtS >
  => go(downTerm(T', 1("null", 0))),

```

```

< 0 : V@MobileObject |
  s : ' &_[T, 'none.Configuration], AtS > ) .

```

Root objects are in charge of handling go messages and sending them to the appropriate locations. To reduce the number of cases, we first check whether the mobile object should stay in the current location, or should move to a different one, removing it from the guests list if it was in it. How they are actually sent through the appropriate sockets will be explained in Section 17.4.

```

rl [go-loc] :
  < L : V@RootObject | state : active, guests : (0, OS), AtS >
  go(L, < 0 : V@MobileObject | AtS' >)
=> < L : V@RootObject | state : active, guests : (0, OS), AtS >
  < 0 : V@MobileObject | AtS' > .

crl [go-loc] :
  < L : V@RootObject | state : active, guests : (0, OS), AtS >
  go(L', < 0 : V@MobileObject | AtS' >)
=> < L : V@RootObject | state : active, guests : OS, AtS >
  Send(L', L, go(L', < 0 : V@MobileObject | AtS' >))
  if L /= L' .

crl [go-loc] :
  < L : V@RootObject | state : active, guests : OS, AtS >
  go(L', < 0 : V@MobileObject | AtS' >)
=> < L : V@RootObject | state : active, guests : OS, AtS >
  Send(L', L, go(L', < 0 : V@MobileObject | AtS' >))
  if L /= L' /\ not 0 in OS .

```

When a go message reaches the location it is addressed to, the mobile object that it carries as an argument is put into the configuration. Depending on whether the location is the home location of such a mobile object or not, the forwarding information is updated or a message to_@_{_} is sent to its home location so that the root object in it can update its forwarding information.

```

rl [arrive-loc] :
  go(L, < o(L',N) : V@MobileObject | hops : N', AtS' >)
  < L : V@RootObject | guests : OS, forward : F, AtS >
=> < o(L', N) : V@MobileObject | hops : N' + 1, AtS' >
  if L == L'
  then < L : V@RootObject | guests : (o(L', N), OS),
        forward : insert(N, (L, N' + 1), F), AtS >
  else < L : V@RootObject | guests : (o(L', N), OS),
        forward : F, AtS >
  Send(L', L, to L' @ (L, N' + 1) { N })
  fi .

```

The following rules specify the update of a mobile object's forwarding information in the root object of its home location upon the reception of a to_@_{_} message. Note that, since the message to update the forwarding

information is sent when the object arrives to its destination location, the forwarding information is not valid during the transit of the mobile objects. However, thanks to the guests lists we still have enough information to guide messages appropriately. Notice also that the forwarding information on a mobile object may be `undefined` upon the reception of a `to_@_{_}` message if the corresponding mobile object was destroyed and the message communicating its destruction arrives before a message communicating a previous move.

```

r1 [forwarding-update] :
  to L @ (L', N') { N }
  < L : V@RootObject | forward : F, AtS >
  => if F [ N ] == undefined
    then < L : V@RootObject | forward : F, AtS >
    else if p2(F [ N ]) < N'
      then < L : V@RootObject |
        forward : insert(N, (L', N'), F), AtS >
      else < L : V@RootObject | forward : F, AtS >
    fi
  fi .

cr1 [forwarding-update] :          *** message in transit
  to L @ (L', N') { N }
  < L'' : V@RootObject | AtS >
  => < L'' : V@RootObject | AtS >
    Send(L, L'', to L @ (L', N') { N })
  if L /= L'' .

```

The go-find message

In the `go` message, the mobile object indicates the location it wants to go to. However, sometimes, a mobile object wants to reach another object, but it only knows the identifier of the object it wants to catch up with, not the location it is at. In this case, the `go-find` message can be used, which takes as arguments the identifier of the mobile object to be reached, and the identifier of a tentative location, where it may be.

The rules for the `go-find` messages are very similar to those for the `go` messages discussed in Section 17.3.3. However, in this case we do not only want to reach a location, but also to find a mobile object, which may move from one place to another. Although the message includes a tentative location for the object, such information may be incorrect, or obsolete.

When a mobile object has a `go-find` message in its state it is pulled out with the following rule.

```

r1 [message-out-go-find] :
  < O : V@MobileObject | s : '_&_[T, 'go-find[T', T'']] , AtS >
  => go-find(downTerm(T', o(1("null", 0), 0)),
            downTerm(T'', 1("null", 0)),
            < O : V@MobileObject |
              s : '_&_[T, 'none.Configuration], AtS > ) .

```

Then, depending on whether the object is at the same location, the location is the home location of the object to be reached, etc., we have rules dealing with the different cases.

The simplest case is the one in which the object to be reached is at the same location.

```
rl [go-find-loc] :
  go-find(O, L', < O' : V@MobileObject | AtS' >)
  < L : V@RootObject | state : active, guests : (O, OS), AtS >
  => < L : V@RootObject | state : active, guests : (O, OS), AtS >
    < O' : V@MobileObject | AtS' > .
```

The following rule deals with the case in which, although the location is the home of the object it is trying to reach, such an object is currently not at it, and therefore the message must be forwarded appropriately, assuming that the forwarding information in the home location is better than the tentative one—the forwarding can take place only if there is an updated forwarding information.

```
crl [go-find-loc] :
  go-find(o(L, N), L', < O : V@MobileObject | AtS' >)
  < L : V@RootObject |
    state : active, guests : (O, OS), forward : F, AtS >
  => < L : V@RootObject |
    state : active, guests : OS, forward : F, AtS >
    Send(p1(F[N]), L,
      go-find(o(L, N), p1(F[N]), p2(F[N]),
        < O : V@MobileObject | AtS' >))
    if not o(L, N) in (O, OS) /\ p1(F[N]) /= L .
```

In case the tentative location is the location where the `go-find` message is generated, if the object to be reached is not at such a location and it is not the home location for such an object, then the `go-find` message is sent to its home location.

```
crl [go-find-loc] :
  go-find(o(L', N), L, < O : V@MobileObject | AtS' >)
  < L : V@RootObject | state : active, guests : (O, OS), AtS >
  => < L : V@RootObject | state : active, guests : OS, AtS >
    Send(L', L,
      go-find(o(L', N), L', (null).Maybe{Nat},
        < O : V@MobileObject | AtS' >))
    if not o(L', N) in (O, OS) /\ L /= L' .
```

If the object to be reached is not at the current location and it is not the home location nor the tentative one, then the `go-find` message is sent to the tentative location.

```
crl [go-find-loc] :
  go-find(o(L', N), L'', < O : V@MobileObject | AtS' >)
```

```

< L : V@RootObject | state : active, guests : (0, OS), AtS >
=> < L : V@RootObject | state : active, guests : OS, AtS >
    Send(L'', L,
        go-find(o(L', N), L'', (null).Maybe{Nat},
            < 0 : V@MobileObject | AtS' >))
if not o(L', N) in (0, OS) /\ L /= L' /\ L /= L'' .

```

When a `go-find` message reaches the tentative location it was addressed to, depending on whether the object the message is trying to find is at such a location or not, the mobile object will be put into the configuration or forwarded. As in the `arrive-loc` rule (see above), the forwarding information is then updated. In addition, if the message requires to be forwarded, this will be done towards the location the mobile object is at according to the forwarding information in its home location, or to such a home location depending on whether it is at its home location or not.

```

rl [arrive-find-loc] :
  *** the object has been reached in its home location
  go-find(o(L, N), L', H,
    < o(L', N') : V@MobileObject | hops : N'', AtS' >)
  < L' : V@RootObject |
    state : active, guests : (o(L, N), OS), forward : F, AtS >
=> < L' : V@RootObject |
    state : active, guests : (o(L, N), o(L', N'), OS),
    forward : insert(N', (L', N'' + 1), F), AtS >
    < o(L', N') : V@MobileObject | hops : N'' + 1, AtS' > .

crl [arrive-find-loc] :
  *** the object has been reached in the tentative location,
  *** which is not its home location
  go-find(o(L, N), L', H,
    < o(L'', N') : V@MobileObject | hops : N'', AtS >)
  < L' : V@RootObject |
    state : active, guests : (o(L, N), OS), forward : F, AtS' >
=> < L' : V@RootObject |
    state : active, guests : (o(L, N), o(L'', N'), OS),
    forward : F, AtS' >
    < o(L'', N') : V@MobileObject | hops : N'' + 1, AtS >
    Send(L'', L', to L'' @ (L', N'' + 1) { N' })
if L' /= L'' .

crl [arrive-find-loc] :
  go-find(o(L, N), L, (null).Maybe{Nat},
    < o(L'', N') : V@MobileObject | AtS >)
  < L : V@RootObject |
    state : active, guests : OS, forward : F, AtS' >
=> < L : V@RootObject |
    state : active, guests : OS, forward : F, AtS' >
    Send(pl(F[N]), L,

```

```

      go-find(o(L, N), p1(F[N]), p2(F[N]),
        < o(L'', N') : V@MobileObject | AtS >))
    if not(o(L, N) in OS) .

crl [arrive-find-loc] :
  go-find(o(L, N), L, N'', < o(L'', N') : V@MobileObject | AtS' >)
  < L : V@RootObject |
    state : active, guests : OS, forward : F, AtS >
=> < L : V@RootObject |
  state : active, guests : OS, forward : F, AtS >
  Send(p1(F[N]), L,
    go-find(o(L, N), p1(F[N]), p2(F[N]),
      < o(L'', N') : V@MobileObject | AtS' >))
  if not(o(L, N) in OS) /\ p2(F[N]) > N'' .

crl [arrive-find-proc] :
  go-find(o(L, N), L', H, < o(L'', N') : V@MobileObject | AtS >)
  < L' : V@RootObject | state : active, guests : OS, AtS' >
=> < L' : V@RootObject | state : active, guests : OS, AtS' >
  Send(L, L',
    go-find(o(L, N), L, (null).Maybe{Nat},
      < o(L'', N') : V@MobileObject | AtS >))
  if not(o(L, N) in OS) /\ L /= L' .

crl [arrive-find-proc] :
  go-find(O, L, H, < O' : V@MobileObject | AtS >)
  < L' : V@RootObject | state : active, AtS' >
=> < L' : V@RootObject | state : active, AtS' >
  Send(L, L', go-find(O, L, H, < O' : V@MobileObject | AtS >))
  if L /= L' .

```

17.3.4 The creation of mobile objects

When an object (in the inner configuration of a mobile object, as part of the application code) wants to create a new mobile object, it sends a **newo** message to the system (by putting it in the second component, the outgoing tray, of its state). The **newo** message takes as arguments (the metarepresentation of) a module M , a configuration C (which will be the initial configuration to be put in the belly of the mobile object to be created, and which should be a valid term in the module M), and the provisional identifier of the main object in the configuration C . The first action accomplished by the system when it detects the **newo** message is to create a new mobile object with the configuration C as its state and the module M as its code, and then to send a **start-up** message to the main object in C with its new name, so we guarantee that its name coincides with the name of the mobile object it is in. Let us first recall that **newo** is defined in the `MOBILE-OBJECT-INTERFACE` module as follows:

```
op newo : Module Configuration Oid -> Msg [ctor msg].
```

First, as for the other Mobile Maude commands, we need to provide rules for pulling out `newo` commands. As for the `to_:_` message in Section 17.3.2, we need three rules to cover the different cases.

```

rl [message-out-newo] :
  < O : V@MobileObject | s : ' &_amp;_[T, 'newo[T', T'', T''']] , AtS >
=> < O : V@MobileObject | s : ' &_amp;_[T, 'none.Configuration], AtS >
    newo(downTerm(T', errorModule), T'', T''') .

rl [message-out-newo] :
  < O : V@MobileObject |
    s : ' &_amp;_[T, '[_[newo[T', T'', T''']] , T''']] , AtS >
=> < O : V@MobileObject | s : ' &_amp;_[T, T''']] , AtS >
    newo(downTerm(T', errorModule), T'', T''') .

crl [message-out-newo] :
  < O : V@MobileObject |
    s : ' &_amp;_[T, '[_[newo[T', T'', T''']] , T''']] , TL]] , AtS >
=> < O : V@MobileObject | s : ' &_amp;_[T, '[_[T''']] , TL]] , AtS >
    newo(downTerm(T', errorModule), T'', T''')
  if TL /= empty .

```

Before creating the mobile object, we check that the initial state given to the `newo` command as second argument together with the `start-up` message is a valid configuration.

When a mobile object is created, its number of hops is set to zero, and the forwarding information in the root object at its parent location is initialized as expected—with its home location as the location it is at and zero as its number of hops. Note that the value initially given to the `gas` attribute of the new mobile object is 100, and that its identifier is included in the set of guests of its root object.

```

rl [create-object] :
  newo(MOD, T, T')
  < L : V@RootObject |
    cnt : N, guests : OS, forward : F, state : active, AtS >
=> if sortLeq(MOD,
    leastSort(MOD,
      '[_[T, 'to_:_[T', 'start-up[upTerm(o(L, N))]]]),
      'Configuration)
  or
  sortLeq(MOD,
    'Configuration,
    leastSort(MOD,
      '[_[T, 'to_:_[T', 'start-up[upTerm(o(L, N))]]])
  then < L : V@RootObject | cnt : N + 1,
    guests : (o(L, N), OS),
    forward : insert(N, (L, 0), F),
    state : active, AtS >

```

```

< o(L, N) : MobileObject | mod : MOD,
  s : ' &_[' __[T,
      'to:_[T', 'start-up[upTerm(o(L, N))]],
      'none.Configuration],
  gas : 100,
  hops : 0 >
else < L : V@RootObject | cnt : N, guests : OS, forward : F,
  state : active, AtS >
fi .

```

17.3.5 Mobile object destruction

After it has completed its task, a mobile object's inner object may request the death of its container mobile object. The rule `message-out-kill` directly destroys the mobile object with the kill message in its outgoing messages tray (it must be the only one, so that all other messages have been previously handled). However, its home location must be informed, so that the forwarding information is appropriately updated. The first two `mobile-object-dead` rules involve the root object of the process the killed mobile object was at. If the object is destroyed at its home location, the information can be directly updated; if the process is not its home location, the second `mobile-object-dead` rule just removes the object's identifier from its guests list and sends a `to_dead{ }` message to inform its home location. The third `mobile-object-dead` rule handles the message in intermediate locations, just forwarding it, and the last one updates the `forward` attribute of its home location.

```

r1 [message-out-kill] :
  < o(L, N) : V@MobileObject | s : ' &_[T, 'kill.Msg], AtS >
  => to L dead { N } .

r1 [mobile-object-dead] :
  to L dead { N }
  < L : V@RootObject |
    guests : (o(L, N), OS), forward : ((N |-> (L, N')), F), AtS >
  => < L : V@RootObject | guests : OS, forward : F, AtS > .

crl [mobile-object-dead] :
  to L dead { N }
  < L' : V@RootObject | guests : OS, AtS >
  => < L' : V@RootObject | guests : OS, AtS >
  Send(L, L', to L dead { N })
  if L' /= L /\ not o(L, N) in OS .

crl [mobile-object-dead] :
  to L' dead { N }
  < L : V@RootObject | guests : (o(L', N), OS), AtS >
  => < L : V@RootObject | guests : OS, AtS >
  Send(L', L, to L' dead { N })
  if L /= L' .

```

```

crl [mobile-object-dead] :
  to L dead { N }
  < L : V@RootObject |
    guests : OS, forward : ((N |-> (L', N')), F), AtS >
  => < L : V@RootObject | guests : OS, forward : F, AtS >
  if not o(L, N) in OS .

```

17.4 Mobile Maude Architecture

Once the semantics of Mobile Maude has been presented, in this section we show how locations are connected by means of *buffered sockets*, the external objects explained in Section 11.4.2.

The specification of Mobile Maude presented in the previous sections allows different configurations of processes. As an example, we present here a very simple client/server architecture. We distinguish clients and servers by declaring two subclasses of `RootObject`, namely, `ServerRootObject` and `ClientRootObject`, with no additional attributes, although with different behavior.⁷

```

classes ClientRootObject ServerRootObject .
subclasses ClientRootObject ServerRootObject < RootObject .

```

The architecture we present here consists in a process with a server root object, and several processes with client root objects. The server is connected to all clients, and each client is connected only to the server. If a mobile object residing in a client process—a process with a client root object in it—wants to move to (or send a message to a mobile object in) another client process, then it will be sent to the server process, and from there to its final destination. That is, we have a very simple star network, with a server root object in the middle redirecting all messages.

The server root object plays the server role, and offers its services on a port `port`. It creates a server socket with the following `connect` rule.

```

r1 [connect] :
  < 0 : V@ServerRootObject | state : idle, AtS >
  => < 0 : V@ServerRootObject | state : waiting-connection, AtS >
    CreateServerTcpSocket(socketManager, 0, port, 5) .

```

Note that it goes from state `idle` to `waiting-connection`, so this rule is applied only once. The response is handled by the rule `connected` below.

⁷ We use here the Full Maude notation for defining subclasses (see Section 21.1.3); the corresponding declarations in Core Maude are:

```

sorts ClientRootObject ServerRootObject .
subsorts ClientRootObject ServerRootObject < RootObject .
op ClientRootObject : -> ClientRootObject [ctor] .
op ServerRootObject : -> ServerRootObject [ctor] .

```

Once it receives the `CreatedSocket` message, it becomes `active` and is ready to accept clients. It could also result in a `socketError` message.

```

rl [connected] :
  CreatedSocket(0, SOCKET-MANAGER, SOCKET)
  < 0 : V@ServerRootObject | state : waiting-connection, AtS >
  => < 0 : V@ServerRootObject | state : active, AtS >
      AcceptClient(SOCKET, 0) .

```

There are two things one can do with a server socket: accepting a client or closing it.

```

rl [acceptedClient] :
  AcceptedClient(1(IP, N), SOCKET, IP', NEW-SOCKET)
  < 1(IP, N) : V@ServerRootObject | state : active, AtS >
  => < 1(IP, N) : V@ServerRootObject | state : active, AtS >
      AcceptClient(SOCKET, 1(IP, N))
      Receive(NEW-SOCKET, 1(IP, N))
      Send(NEW-SOCKET, 1(IP, N), msg2string(new-socket(1(IP, N)))) .

```

In the rule `acceptedClient`, in addition to sending `AcceptClient` and `Receive` messages indicating, respectively, that it is ready to accept new clients through the server socket and messages through the new socket, the server root object that gets the `AcceptedClient` message sends a start-up message `new-socket` communicating its identifier. Notice that, as we will see below, the client knows the address and port of the server root object, but not its identity. In this first message the server sends its name to its client, allowing the client to establish the association between the socket and the identity of the object in it.

At any time, a server root object can receive a message communicating the closing of a socket.

```

crl [closedSocket] :
  ClosedSocket(SOCKET, SOCKET, DATA)
  < 0 : V@ServerRootObject |
      neighbors : LSPF, defNeighbor : 0', AtS >
  => < 0 : V@ServerRootObject |
      neighbors : remove(SOCKET, LSPF), defNeighbor : 0', AtS >
      if SOCKET /= 0' .

crl [closedSocket] :
  ClosedSocket(SOCKET, SOCKET, DATA)
  < 0 : V@ServerRootObject |
      neighbors : LSPF, defNeighbor : SOCKET, AtS >
  => < 0 : V@ServerRootObject | neighbors : remove(SOCKET, LSPF),
      defNeighbor : null, AtS > .

```

where `remove` is a function that removes all entries indexed by the socket identifier given as first argument from the map given as second argument.

Given the address of a server root object—a constant `server-address`—and a port `port`, the first thing a client root object does is to request a socket connection.

```
rl [connect] :
  < 0 : V@ClientRootObject | state : idle, AtS >
  => < 0 : V@ClientRootObject | state : waiting-connection, AtS >
      CreateClientTcpSocket(socketManager, 0, server-address, port) .
```

As done by server root objects, clients go to the `waiting-connection` state as a result of the application of this rule. The response to a client root object's socket connection request is handled by the following rule `connected`, where a client also sends a `new-socket` message right after the socket is created.

```
rl [connected] :
  CreatedSocket(0, SOCKET-MANAGER, SOCKET)
  < 1(IP, N) : V@ClientRootObject |
      state : waiting-connection, AtS >
  => < 1(IP, N) : V@ClientRootObject | state : active, AtS >
      Receive(SOCKET, 1(IP, N))
      Send(SOCKET, 1(IP, N), msg2string(new-socket(1(IP, N)))) .
```

The attributes `neighbors` and `defNeighbor` are key for sending messages through the appropriate sockets. To initialize these attributes, as explained above, the first message sent through a socket after its creation is the message `new-socket`. When it is received, depending on whether the receiver is a client or a server, and whether there is already a default neighbor or not, one action or another is taken.

To avoid unintended loops in the delivering of messages, we assume that server root objects do not have default neighbors. For clients, the first connection is made the default one.

```
crl [received] :
  ---- the first connection is made the default one
  Received(0, SOCKET, DATA)
  < 0 : V@RootObject |
      state : active, neighbors : empty, defNeighbor : null, AtS >
  => < 0 : V@RootObject | state : active,
      neighbors : insert(L, SOCKET, empty),
      defNeighbor : if V@RootObject :: ServerRootObject
                    then null
                    else SOCKET
                    fi,
      AtS >
      Receive(SOCKET, 0)
  if new-socket(L) := string2msg(DATA) .

crl [received] :
  Received(0, SOCKET, DATA)
```

```

< 0 : V@RootObject | state : active, neighbors : LSPF, AtS >
=> < 0 : V@RootObject | state : active,
    neighbors : insert(L, SOCKET, LSPF), AtS >
    Receive(SOCKET, 0)
if LSPF /= empty /\ new-socket(L) := string2msg(DATA) .

```

If it is not a `new-socket` message, then the message is just left in the configuration.

```

crl [received] :
  Received(0, SOCKET, DATA)
  < 0 : V@RootObject | state : active, AtS >
=> < 0 : V@RootObject | state : active, AtS >
    string2msg(DATA)
    Receive(SOCKET, 0)
if not new-socket(DATA) .

op new-socket : String -> Bool .
ceq new-socket(DATA) = true if new-socket(L) := string2msg(DATA) .
eq new-socket(DATA) = false [owise] .

```

Note the way we define the `new-socket` operator to check that the message is not a `new-socket` message.

Sending of messages through the appropriate sockets is responsibility of the root objects, which use the information in their `neighbors` and `defNeighbor` attributes to do it. Since the `Send` message takes a `String` as argument, messages need to be converted into strings. The auxiliary functions `msg2string` and `string2msg`, whose specification is not shown here, do the conversions in both ways.

```

crl [changeSend] :
  Send(L, 0'', MSG)
  < 0 : V@RootObject |
    state : active, neighbors : LSPF, defNeighbor : 0', AtS >
=> < 0 : V@RootObject |
    state : active, neighbors : LSPF, defNeighbor : 0', AtS >
    Send(0', 0'', msg2string(MSG))
if LSPF[L] == undefined /\ 0' /= null .

crl [changeSend] :
  Send(L, 0', MSG)
  < 0 : V@RootObject | state : active, neighbors : LSPF, AtS >
=> < 0 : V@RootObject | state : active, neighbors : LSPF, AtS >
    Send(LSPF[L], 0', msg2string(MSG))
if LSPF[L] /= undefined .

rl [messageSent] :
  Sent(0, 0')
  < 0 : V@RootObject | AtS >
=> < 0 : V@RootObject | AtS > .

```

17.5 A buying printers example

In this section we present a simple application to illustrate how mobile *application code* can be written in Maude and can be wrapped in mobile objects. In this example we have printers, buyers, and sellers; a buyer agent visits several printer sellers, who provide him information on their printers. The buyer looks for the cheapest printer, and once he has visited all the sellers, he goes back to the location of the seller offering the best price.

From the previous description, we can identify different actors, which may move freely from one process to another, and therefore could be represented as mobile objects. In the Mobile Maude approach the specification of the system consists of objects embedded inside mobile objects, which communicate with each other via messages. In addition to the term representing its state, each mobile object carries the *code* managing the behavior of the configuration of objects and messages representing such a state. The main difference with the usual specification of systems in Maude is that these objects must be aware of the fact that they are inside mobile objects, and that in order to communicate with (objects in) other mobile objects or to use some of the system messages available, they must follow the appropriate protocol.

In our sample application we have two different classes of mobile objects: sellers and buyers. Although in the simple example modeled here sellers do not move, they should be mobile objects, because they communicate with other mobile objects, and therefore have to be recognized as mobile objects by the Mobile Maude system. A buyer visits several sellers. The buyer asks each seller he visits for the description of the seller's printer, represented here only by its price. The seller sends back this information, which the buyer keeps if it corresponds to a better (cheaper) printer. Otherwise he discards it. Once the buyer has visited all the sellers he knows, he goes back to the location of the best offer.

We represent sellers and buyers as objects of respective classes `Seller` and `Buyer`. Such objects in the application code will then be embedded as *inner objects* of their corresponding mobile objects.

The class `Seller` has an attribute `description` with the printer price. We use here the Full Maude notation for defining classes. The corresponding declarations in Core Maude are given in the `SELLER` module below.

```
class Seller | description : Nat .
```

Sellers receive messages of the form `get-printer-price(B)`, with `B` the identifier of the buyer mobile object sending the message. A seller can send messages of the form `printer-price(N)`, with `N` a natural number representing the printer's price. Both are defined with sort `Contents`, declared in the module `MOBILE-OBJECT-INTERFACE` (see Section 17.2). A seller's behavior is represented by the `get-des` rewrite rule: When a seller receives a description (price) request, it sends the description back to the seller. The whole module defining the sellers in Maude is as follows:

```

mod SELLER is
  including MOBILE-OBJECT-INTERFACE .

  sort Seller .
  subsort Seller < Cid .
  op Seller : -> Seller .
  var V@Seller : Seller .

  op description :_ : Nat -> Attribute [ctor] .

  op get-printer-price : Mid -> Contents [ctor] .
  op printer-price : Nat -> Contents [ctor] .

  vars S B : Oid .
  var N : Nat .
  var AtS : AttributeSet .
  var Conf : Configuration .

  rl [get-des] :
    Conf (to S : get-printer-price(B))
    < S : V@Seller | description : N, AtS > & none
    => Conf < S : V@Seller | description : N, AtS >
      & (to B : printer-price(N)) .
endm

```

Note the use of the `_&_` constructor. Since the printer description is sent to an object outside the mobile object in which the `Seller` object is located, the message is placed in the righthand outgoing tray. The rule `get-des` is applied only if the outgoing messages tray is empty, making sure in this way that any previous outgoing message has been handled. The `_&_` operator is the top operator of the term representing the state of the mobile object. Therefore, since there may be other objects and messages in the configuration in its lefthand side component, we include a variable `Conf` of sort `Configuration` to match the rest. Note also how an object may communicate with objects in other mobile objects, which may be in different processes, in a completely transparent way.

A buyer has an attribute `sellers` with a list of the identifiers of the known seller mobile objects. It also has an attribute `status` with its current state: `onArrival`, `asking`, `done`, or `buying`. Finally, the buyer keeps information about the printer with the best price in the attributes `price` and `bestSeller` of sorts, respectively, `Maybe{Nat}` and `Maybe{Oid}`. Initially, these two last attributes are `null`.

```

class Buyer | sellers : List{Mid},
             status : Status,
             price : Maybe{Nat},
             bestSeller : Maybe{Oid} .

```

The `BUYER` module that follows describes the buyers class.

```

mod BUYER is
  inc MOBILE-OBJECT-INTERFACE .
  pr LIST{Mid} * (op __ to __., op nil to no-id) .
  pr MAYBE{Nat} * (op maybe to null) .
  pr MAYBE{Oid} * (op maybe to null) .

  sort Status .
  ops onArrival asking done buying : -> Status [ctor] .

  sort Buyer .
  subsort Buyer < Cid .
  op Buyer : -> Buyer .
  var V@Buyer : Buyer .

  op sellers :_ : List{Mid} -> Attribute [ctor gather(&)] .
  op status :_ : Status -> Attribute [ctor] .
  op price :_ : Maybe{Nat} -> Attribute [ctor] .
  op bestSeller :_ : Maybe{Oid} -> Attribute [ctor] .

  op get-printer-price : Mid -> Contents [ctor] .
  op printer-price : Nat -> Contents [ctor] .

  var S S' B : Oid .
  var OS : List{Mid} .
  var AtS : AttributeSet .
  vars N N' : Nat .
  var L : Loc .

```

The first rewrite rule, `move`, handles the travels of the buyer to request information on printers: if it is not in the middle of a request (its status is `done`) and there is at least one seller name in the `sellers` attribute, it asks the system to take it to the host where the next seller is located.

```

r1 [move] :
  < B : V@Buyer | sellers : o(L,N) . OS, status : done, AtS > Conf
  & none
  => < B : V@Buyer |
      sellers : o(L, N) . OS, status : onArrival, AtS > Conf
      & go-find(o(L, N), L) .

```

Since Mobile Maude guarantees that mobile objects moving from one process to another are frozen (see Section 17.3.3), we know that, once the `go-find` command is given in the `move` rule, the buyer object will not be able to do anything until the mobile object in which it is embedded has reached the seller's process. Therefore, since there is no rule taking a `Buyer` object in `onArrival` state and a non-empty outgoing messages tray, this object will not do anything until it reaches its destination.

On arrival, the buyer asks the seller for the printer description.

```

r1 [onArrival] :
  < B : V@Buyer | sellers : S . OS, status : onArrival, AtS > Conf
  & none
  => < B : V@Buyer | sellers : S . OS, status : asking, AtS > Conf
      & (to S : get-printer-price(B)) .

```

When the printer price arrives, if it corresponds to the first time the buyer is asking for a price (the attribute `price` is `null`) the buyer keeps it as the best known price; otherwise, it compares it with the best known printer and updates its information if needed. Notice that the first identifier in the list of known sellers gives us the identifier of the seller it is currently interacting with.

```

r1 [new-des] :
  (to B : printer-price(N))
  < B : V@Buyer | sellers : S . OS,
    price : null, status : asking, bestSeller : null, AtS >
  => < B : V@Buyer | sellers : OS,
    price : N, status : done, bestSeller : S, AtS > .

r1 [new-des] :
  (to B : printer-price(N))
  < B : V@Buyer | sellers : S . OS,
    price : N', status : asking, bestSeller : S', AtS >
  => if (N < N') then
    < B : V@Buyer | sellers : OS,
      price : N, status : done, bestSeller : S, AtS >
  else < B : V@Buyer | sellers : OS,
    price : N', status : done, bestSeller : S', AtS >
  fi .

```

Notice that since these last rules do not imply the sending of any message out of the mobile object, we do not need to use the `_&_` operator and the variable `Conf` to encompass the whole state.

Finally, when the list of remaining sellers is empty, the buyer travels to find the best buyer and reaches the `buying` status.

```

r1 [buy-it] :
  < B : V@Buyer | sellers : no-id,
    status : done, bestSeller : o(L, N), AtS >
  Conf & none
  => < B : V@Buyer | sellers : no-id,
    status : buying, bestSeller : o(L, N), AtS >
    Conf & go-find(o(L,N), L) .
endm

```

Let us see an example of a distributed configuration, and how we can rewrite it by using the `erewrite` command. Our sample buyers/sellers configuration, shown in Figure 17.2, is constituted by three located configurations,

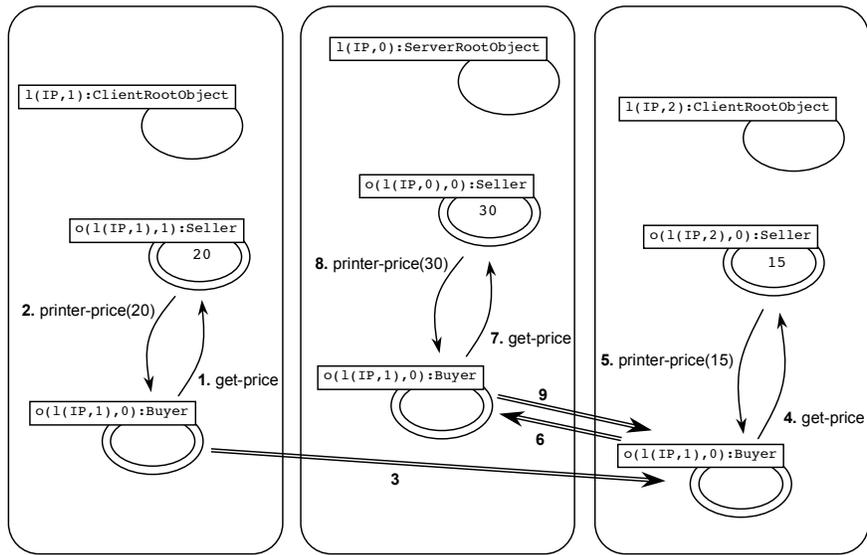


Fig. 17.2. Buyers and sellers configuration

each one to be executed in a Maude process—in this case the three processes run on the same machine, with IP address IP. The first located configuration (shown in the middle of the figure) contains a `ServerRootObject`, with identifier `l(IP, 0)`, and a mobile object with identifier `o(l(IP, 0), 0)` with a `Seller` in its belly. The Maude command to introduce the initial state of this configuration is as follows:

```

erew <>
  < l(IP, 0) : ServerRootObject |
    cnt : 1,
    guests : o(l(IP, 0), 0),
    forward : 0 |-> (l(IP, 0), 0),
    neighbors : empty,
    state : idle,
    defNeighbor : null >
  < o(l(IP, 0), 0) : MobileObject |
    mod : upModule('SELLER, false),
    s : upTerm(< o(l(IP, 0), 0) : Seller | description : 30 >
      & none),
    gas : 200,
    hops : 0 > .

```

Note how the function `upModule` is used to obtain the metarepresentation of the module `SELLER`, and how the function `upTerm` is used to metarepresent the initial state of the inner object.

This configuration must be executed before the other two ones, because it contains the object `ServerRootObject`, which is in the central process of the star network.

The second located configuration (on the left in the figure) contains a `ClientRootObject`, a `Buyer` and a `Seller` with cheaper printers. The Maude command, introduced in a different Maude process, is the following one:

```

erew <>
  < l(IP, 1) : ClientRootObject |
    cnt : 2,
    guests : o(l(IP, 1), 0) . o(l(IP, 1), 1),
    forward : 0 |-> (l(IP, 1), 0),
              1 |-> (l(IP, 1), 0),
    neighbors : empty,
    state : idle,
    defNeighbor : null >
  < o(l(IP, 1), 0) : MobileObject |
    mod : upModule('BUYER, false),
    s : upTerm(< o(l(IP, 1), 0) : Buyer |
              price : null,
              status : done,
              bestSeller : null,
              sellers : o(l(IP, 1), 1) .
                       o(l(IP, 0), 0) .
                       o(l(IP, 2), 0) >
              & none),
    gas : 200,
    hops : 0 >
  < o(l(IP, 1), 1) : MobileObject |
    mod : upModule('SELLER, false),
    s : upTerm(< o(l(IP, 1), 1) : Seller | description : 20 >
              & none),
    gas : 200,
    hops : 0 > .

```

Finally, the third located configuration (on the right) contains another `ClientRootObject` and a `Seller` with the cheapest printers.

```

erew <>
  < l(IP, 2) : ClientRootObject |
    cnt : 1,
    guests : o(l(IP, 2), 0),
    forward : 0 |-> (l(IP, 2), 0),
    neighbors : empty,
    state : idle,
    defNeighbor : null >
  < o(l(IP, 2), 0) : MobileObject |
    mod : upModule('SELLER, false),
    s : upTerm(< o(l(IP, 2), 0) : Seller |

```

```

        description : 15 >
        & none),
    l : l(IP, 2),
    gas : 200,
    hops : 0 > .

```

Figure 17.2 shows the order in which the different actions occur. First, the buyer asks the seller at his same location (price 20). Then, the buyer travels to the location on the right (through the location with the `ServerRootObject`) and asks the seller who sells printers costing 15. After that, the buyer travels to the middle location and asks the seller there (price 30). Finally, the buyer travels to the right location to find the seller with the best offer.

The execution of these three commands in three different Maude processes does not terminate. This is due to the blocking behavior of socket messages like `receive`. An execution of a Mobile Maude application is not intended to terminate, since the located configurations are always waiting for messages or mobile objects to come in from other configurations. Due to this fact, it is recommended to execute these applications with the trace on. In this way, we can see what is happening in each Maude process. When the execution of a concrete example seems to be finished, because we do not see evolution in any of the involved processes, we can finish them by typing `^C`.

In the first Maude process we obtain the following configuration:

```

result Configuration:
<>
receive(socket(4), b(socket(4)))
Receive(b(socket(5)), l(IP, 0))
Receive(b(socket(6)), l(IP, 0))
< b(socket(4)) : BufferedSocket |
  read : "",
  bState : active >
< b(socket(5)) : BufferedSocket |
  read : "",
  bState : active >
< b(socket(6)) : BufferedSocket |
  read : "",
  bState : active >
< l(IP, 0) : ServerRootObject |
  cnt : 1,
  guests : o(l(IP, 0), 0),
  forward : 0 |-> (l(IP, 0),0),
  state : active,
  neighbors : (l(IP, 1) |-> b(socket(5)),
              l(IP, 2) |-> b(socket(6))),
  defNeighbor : null >
< o(l(IP, 0), 0) : MobileObject |
  mod : mod_is_sorts_._____endm(...),
  s : ('_&_['<_:_|_>['o['l['IP.String, '0.Zero], '0.Zero],

```

```

        'Seller.Seller,
        'description':_['s'^30['0.Zero]],
        'none.Configuration]),
    gas : 199,
    hops : 0 >

```

In the second Maude process we obtain:

```

result Configuration:
<>
Receive(b(socket(4)), l(IP, 1))
< b(socket(4)) : BufferedSocket |
  read : "",
  bState : active >
< l(IP, 1) : ClientRootObject |
  cnt : 2,
  guests : o(l(IP, 1), 1),
  forward : (0 |-> (l(IP, 2),2), 1 |-> (l(IP, 1),0)),
  state : active,
  neighbors : l(IP, 0) |-> b(socket(4)),
  defNeighbor : b(socket(4)) >
< o(l(IP, 1), 1) : MobileObject |
  mod : mod_is_sorts_._____endm(...),
  s : ('_&_['<_:_|_>[
        'o['1['IP.String, 's_['0.Zero]], 's_['0.Zero]],
        'Seller.Seller,
        'description':_['s'^20['0.Zero]],
        'none.Configuration]),
    gas : 199,
    hops : 0 >

```

And in the third Maude process we obtain:

```

result Configuration:
<>
Receive(b(socket(4)), l(IP, 2))
< b(socket(4)) : BufferedSocket |
  read : "",
  bState : active >
< l(IP, 2) : ClientRootObject |
  cnt : 1,
  guests : (o(l(IP, 1), 0), o(l(IP, 2), 0)),
  forward : 0 |-> (l(IP, 2),0),
  state : active,
  neighbors : l(IP, 0) |-> b(socket(4)),
  defNeighbor : b(socket(4)) >
< o(l(IP, 1), 0) : MobileObject |
  mod : mod_is_sorts_._____endm(...),
  s : ('_&_['<_:_|_>['o['1['IP.String, 's_['0.Zero]], '0.Zero],
        'Buyer.Buyer,'_','_>

```

```

      'bestSeller':_ [
        'o[1[IP.String, 's^2[0.Zero], '0.Zero]],
        'price':_ ['s^15[0.Zero]],
        'sellers':_ ['no-id.List{Mid}],
        'status':_ ['buying.Status]],
      'none.Configuration]),
    gas : 193,
    hops : 2 >
< o(1(IP, 2), 0) : MobileObject |
  mod : mod_is_sorts_._____endm(...),
  s : ('&_['<:_|_>[
    'o[1[IP.String, 's^2[0.Zero], '0.Zero],
    'Seller.Seller,
    'description':_ ['s^15[0.Zero]],
    'none.Configuration]),
  gas : 199,
  hops : 0 >

```

Note that the buyer has finished his travel at the same location as that of the best seller. We can observe in this last state how the buyer has visited (the processes of) all the sellers, and has the identifier and price of the seller offering the best price.

17.6 Model checking Mobile Maude applications

Maude's model checker (see Chapter 13) allows us to prove properties on Maude specifications when the set of states reachable from an initial state in such a Maude system module is finite. This is supported in Maude by its `MODEL-CHECKER` module and other related modules, which can be found in the file `model-checker.maude` distributed with Maude.

The properties to be checked are described by using a specific property specification logic, namely linear temporal logic (LTL) (see [226, 64] and Chapter 13), which allows specification of properties such as safety properties (ensuring that something bad never happens) and liveness properties (ensuring that something good eventually happens). Then, the model checker can be used to check whether a given initial state, represented by a Maude term, fulfills a given property.

Using the model checker on Mobile Maude is not easy, however. Mobile Maude configurations are distributed among several hosts, and therefore the model checker cannot be used directly to prove properties about these global configurations. Moreover, we would like to check properties on the application code, which is metarepresented in the belly of mobile objects. We show in the following sections how we have addressed both issues. The former problem has been solved by considering an algebraic specification of the sockets provided by Maude. The later one has been solved by considering two-level properties, stating different properties on each of the reflection levels.

17.6.1 Redefinition of the SOCKET module

To avoid having a distributed configuration made up of processes communicating via sockets, we provide an algebraic specification of sockets. This allows us to simulate the distributed configurations in one single Maude process. Note that the specification of Mobile Maude does not change, we just provide an algebraic specification of sockets to be used instead of the built-in one. As we will see in the next section this implies that the initial configurations will have to have this into account, but neither the Mobile Maude specification nor the application code have to be changed.

We have redefined the `SOCKET` module (see Section 11.4.1), simulating the behavior of sockets on local configurations. This specification expresses processes—each Maude process represents a local configuration—as terms of a class `Process` with a single attribute `conf` which stores the corresponding local configuration. We just make Maude processes and sockets be part of the specification. Thus, terms of sort `Process` in this new specification work as hosts in the distributed original version, and message passing is then defined between processes instead of between hosts.

Thus, we have specified sockets, socket managers, and server sockets to deal with processes:

- The socket manager is now an instance of a class `Manager`, with a `counter` attribute to name the new sockets.
- Sockets are instances of class `Socket` with attributes `source` (the source `Process`), `target` (the target `Process`), and `socketState` (the socket state). Notice that although we talk about source and target, sockets are bidirectional.
- Server sockets are instances of class `ServerSocket` with the attributes `address` (the server address), `port` (the server port), and `backlog` (the number of queue requests for connection that the server will allow). When one object wants to create a server, we create one server socket at process level and the object receives a `createdSocket` message with the server socket identifier.

Note that there is no need for a client sockets class, they are only processes, so to create a client socket we create a socket with target the server and source the process.

The class `Process` allows to represent in a single term a whole distributed configuration. The rest of the above mentioned classes and the rewrite rules defined in the new module `SOCKET` allow to use the specification of Mobile Maude with no more changes. So in order to prove a property about a distributed configuration we have to prove it on the corresponding “local” configuration by using `Processes`.

17.6.2 Two-level atomic propositions for the buying printers example

To use the model checker we just need to make explicit two things: the intended sort of states, `Configuration`, and the relevant *state predicates*, that is, the relevant LTL atomic propositions (see Section 13.1).

To be able to model check Mobile Maude application code, we propose defining these predicates at two different levels: the processes level and the inner objects level. At the processes level we look for inner objects which have some properties; at the inner objects level we check such properties.

Let us see an example about the buying printers case study. Suppose we want to prove that the buyer always finds the best price, and that, when he has visited all sellers, he finishes in the process of the seller who has the best price. If `bestPrice&Seller` represents the state predicate asserting that the buyer is in the process of the seller with the best offer, then the LTL formula we want to check is $\langle \rangle [] \text{bestPrice\&Seller}$, that is, it is always possible to reach an state where the property `bestPrice&Seller` is fulfilled and from that state the property remains invariant.

The following `PRINTERS-PREDS` module includes the definition of the `exSeller` and `exBuyer` predicates at the level of the application code.

```

mod PRINTERS-PREDS is
  pr PRINTERS .
  including SATISFACTION .

  subsort Configuration < State .

  var C : Configuration .
  vars N N' : Nat .
  vars S B : Oid .
  var AtS : AttributeSet .

  op exSeller : Nat -> Prop .
  op exBuyer : Nat -> Prop .

  eq (< S : Seller | description : N, AtS > C) |= exSeller(N)
    = true .
  eq C |= exSeller(N) = false [owise] .

  eq (< B : Buyer | price : N, status : buying, AtS > C)
    |= exBuyer(N)
    = true .
  eq C |= exBuyer(N) = false [owise] .

```

The `minPrice` auxiliary operator, declared as frozen, returns the minimum price offered by the sellers in the given configuration. The function is not defined on configurations with no seller object.

```

op minPrice : Configuration ~> Nat [frozen] .
op minPrice : Configuration Nat -> Nat [frozen] .

eq minPrice(< S : Seller | description : N, AtS > C)
  = minPrice(C, N) .
eq minPrice(< S : Seller | description : N, AtS > C, N')
  = minPrice(C, min(N, N')) .
eq minPrice(C, N) = N [owise] .
endm

```

The `bestPrice&Seller` predicate is defined in the `MOBILE-PRINTERS-PREDS` module.

```

mod MOBILE-PRINTERS-PREDS is
  pr MOBILE-MAUDE .
  pr SATISFACTION .
  pr MAYBE{Nat} * (op maybe to null) .

  vars C C' : Configuration .
  var N : Nat .
  vars O PID : Oid .
  vars T T' : Term .
  var AtS : AttributeSet .

  subsort Configuration < State .

```

The `bestPrice&Seller` predicate is defined using an auxiliary predicate with the same name but with an argument, (the metarepresentation of) the best price, obtained by means of the auxiliary function `minPrice`, which is introduced below.

```

op bestPrice&Seller : -> Prop .
op bestPrice&Seller : Nat -> Prop .
eq C |= bestPrice&Seller = C |= bestPrice&Seller(minPrice(C)) .

```

The definition of `bestPrice&Seller(N)` recursively traverses all the processes, going inside each configuration and looking for a seller with the given price and a buyer who has it as the best price.

```

eq ((C < PID : Process | conf : C' >) |= bestPrice&Seller(N))
  = (C |= bestPrice&Seller(N)) or
    ((C' |= existsSeller(N)) and (C' |= existsBuyer(N))) .
eq C |= bestPrice&Seller(N) = false [owise] .

```

The definitions of the `existsSeller(N)` and `existsBuyer(N)` propositions use the `exSeller` and `exBuyer` predicates defined in the `PRINTERS-PREDS` module. Notice that the `MOBILE-PRINTERS-PREDS` module is at the metalevel of `PRINTERS-PREDS`.

```

op existsSeller : Nat -> Prop .
op existsBuyer : Nat -> Prop .

eq (<< O : MobileObject | s : ('_&_[T, T']), AtS > C)
  |= existsSeller(N)
  = (getTerm(metaReduce(upModule('PRINTERS-PREDS, false),
    '_|=_[T, 'exSeller[upTerm(N)]])) == 'true.Bool)
    or (C |= existsSeller(N)) .
eq C |= existsSeller(N) = false [owise] .

eq (< O : MobileObject | s : ('_&_[T, T']), AtS > C)
  |= existsBuyer(N)
  = (getTerm(metaReduce(upModule('PRINTERS-PREDS, false),
    '_|=_[T, 'exBuyer[upTerm(N)]])) == 'true.Bool)
    or (C |= existsBuyer(N)) .
eq (C |= existsBuyer(N) = false [owise] .

```

The `minPrice` auxiliary operator, also declared as frozen, returns the minimum price offered by the sellers in the configurations in the mobile objects of the given configuration. The function returns `null` if there is no mobile object with a seller.

```

op min : Maybe{Nat} Maybe{Nat} -> Maybe{Nat} [ditto] .

eq min(null, N) = N .
eq min(N, null) = N .
eq min(null, null) = null .

op minPrice : Configuration -> Maybe{Nat} [frozen] .
eq minPrice(C < PID : Process | conf : C' >)
  = min(minPrice(C), minPrice(C')) .
eq minPrice(C < O : MobileObject | s : ('_&_[T, T']), AtS >)
  = min(minPrice(C),
    downTerm(
      getTerm(metaReduce(upModule('PRINTERS-PREDS, false),
        'minPrice[T])),
      null)) .
eq minPrice(C) = null [owise] .
endm

```

Once entered these modules, the Maude command to use the model checker for examining whether an initial configuration `initial` fulfills the formula `<> [] bestPrice&Seller` is as follows:

```

Maude> red modelCheck(initial, <> [] bestPrice&Seller) .
result Bool: true

```

User Interfaces and Metalanguage Applications

This chapter explains how to use the facilities provided by the predefined modules `META-LEVEL` and `LOOP-MODE` for constructing user interfaces and metalanguage applications, in which Maude is used not only to define a domain-specific language or tool, but also to build an environment for the given language or tool. In such applications, the `LOOP-MODE` module can be used to handle the input/output and to maintain the persistent state of the language environment or tool. This chapter also describes an approach based on actors to endow Maude with interactive capabilities.

18.1 The `LOOP-MODE` module

Using object-oriented concepts, we specify in Maude a general input/output facility provided by the `LOOP-MODE` module shown below, which extends the module `QID-LIST` (see Section 9.10), into a generic read-eval-print loop.

```
mod LOOP-MODE is
  protecting QID-LIST .
  sorts State System .
  op [_,_,_] : QidList State QidList -> System
    [ctor special (...)] .
endm
```

The operator `[_,_,_]` can be seen as an object—that we call the *loop object*—with an input stream (the first argument), an output stream (the third argument), and a state (given by its second argument). This read-eval-print loop provided by `LOOP-MODE` is a simple mechanism that may not be maintained in future versions, because the support for communication with external objects (see Section 11.4) makes it possible to develop more general and flexible solutions for dealing with input/output in future releases.

Since in the current release only one input stream is supported (the current terminal), the way to distinguish the input passed to the loop object from

the input passed to the Maude system—either modules or commands—is by enclosing them in parentheses. When something enclosed in parentheses is written after the Maude prompt, it is converted into a list of quoted identifiers. This is done by the system by first breaking the input stream into a sequence of Maude identifiers (see Section 3.1) and then converting each of these identifiers into a quoted identifier by putting a quote in front of it, and appending the results into a list of quoted identifiers, which is then placed in the first slot of the loop object. The output is handled in the reverse way, that is, the list of quoted identifiers placed in the third slot of the loop object is displayed on the terminal after applying the inverse process of “unquoting” each of the identifiers in the list. However, the output stream is not cleared at the time when the output is printed; it is instead cleared when the next input is entered. We can think of the input and output events as *implicit rewrites* that transfer—in a slightly modified, quoted or unquoted form—the input and output data between two objects, namely the loop object and the “user” or “terminal” object.

Besides having input and output streams, terms of sort **System** give us the possibility of maintaining a state in their second component. This state has been declared in a completely generic way. In fact, the sort **State** in **LOOP-MODE** does not have any constructors. This gives complete flexibility for defining the terms we want to have for representing the state of the loop in each particular application. In this way, we can use this input/output facility not only for building user interfaces for applications written in Maude, but also for uses of Maude as a *metalanguage*, where the object language being implemented may be completely different from Maude. For each such tool or language the nature of the state of the system may be completely different. We can tailor the **State** sort to any such application by importing **LOOP-MODE** in a module in which we define the state structure and the rewrite rules for changing the state and interacting with the loop.

18.2 User interfaces

In order to generate in Maude an *interface* for an application \mathcal{P} , the first thing we need to do is to define the language for interaction. This can be done by defining a data type $Sign_{\mathcal{P}}$ for commands and other constructs.

As a running example for this chapter, we will specify a basic interface for the vending machine introduced in Section 6.1. First, we define in the module **VENDING-MACHINE-GRAMMAR** a language for interacting with the vending machine. The signature of this module extends the signature of **VENDING-MACHINE-SIGNATURE** with operators to represent the valid actions, namely: **\$** and **q** for inserting a dollar or a quarter in the machine; **showBasket** and **showCredit** for showing the items already bought or the remaining credit; **buy1Apple** and **buy1Cake** for buying an apple or a cake; and **buy_:_** for buying a number of pieces of the same item.

```

fmod VENDING-MACHINE-GRAMMAR is
  protecting VENDING-MACHINE-SIGNATURE .
  protecting NAT .
  sort Action .
  op $ : -> Action .
  op q : -> Action .
  op showBasket : -> Action .
  op showCredit : -> Action .
  op buy1Cake : -> Action .
  op buy1Apple : -> Action .
  op buy_:_ : Item Nat -> Action .
endfm

```

Next, we define in an extension of the LOOP-MODE module the terms of sort `State` representing the state of the loop for this application. In the module `VENDING-MACHINE-INTERFACE` below, we define this state as a triple: its first component is the next action requested by the client (inserting a coin, showing information about the remaining credit or the items already bought, or buying one or more items); its second component is the current state of the machine (the *marking* of the vending machine, that is, the remaining credit plus the items already bought); and its third component represents the response of the machine to the last action requested by the client. The response of the vending machine will have the form of a message, which can be represented as a list of quoted identifiers. The constant `init` denotes the initial state of the whole system, including the empty input and output streams and the “empty” initial state of the vending machine.

```

mod VENDING-MACHINE-INTERFACE is
  including LOOP-MODE .
  including VENDING-MACHINE-GRAMMAR .
  protecting BUYING-STRATS .
  protecting CONVERSION .

  op <_;;_> : Action Marking QidList -> State .
  op init : -> System .
  op idle : -> Action .
  eq init = [nil, < idle ; null ; nil >, nil] .

  var A : Action .
  var I : Item .
  var C : Coin .
  var M : Marking .
  vars QIL QIL' QIL'' : QidList .
  var N : Nat .

```

Now we define in this module, using rewrite rules, the interaction of the state of the vending machine with the loop—and, consequently, with the client—and the changes produced in the state of the vending machine by the actions requested by the client. As explained before, the client will request an

action by enclosing the text in parentheses, which will then be converted into a list of quoted identifiers and placed in the first slot of the loop object. The rule `in` detects when a valid request has been introduced by the user and, if the vending machine is idle, passes it as the next action to be attempted. To define the interaction of the state of the vending machine with the client, we can use the strategies introduced in the `BUYING-STRATS` module described in Section 14.6. Recall that `BUYING-STRATS` includes the `META-LEVEL` module.

In the rule `in` below, the operation `metaParse` checks whether the input stream corresponds to a term of sort `Action`. If this is the case, `metaParse` returns the metarepresentation of that term, which is then “moved down” using the `META-LEVEL` function `downTerm` (see Section 14.5.1), and is placed in the action slot of the `State` triple; otherwise, an error message is placed in its output.

```

cr1 [in] :
  [QIL, < idle ; M ; nil >, QIL']
  => if T:ResultPair? :: ResultPair
    then [nil,
          < downTerm(getTerm(T:ResultPair?), idle) ; M ; nil >,
          QIL']
    else [nil, < idle ; M ; nil >, 'ERROR QIL]
    fi
  if QIL' /= nil
    /\ T:ResultPair?
      := metaParse(upModule('VENDING-MACHINE-GRAMMAR, false),
                  QIL, 'Action) .

```

For the other direction of the interaction, the rule `out` detects when the vending machine has a response to be output and, in that case, it places it in the output slot of the loop object.

```

cr1 [out] :
  [QIL, < A ; M ; QIL' >, QIL''']
  => [QIL, < A ; M ; nil >, QIL'' QIL']
  if QIL' /= nil .

```

Next, we define the effects of the different actions on the state of the vending machine. The rules `showBasket` and `showCredit` extract the information about the remaining credit or the items already bought, and place it in the output slot of the state; the rule `out` will then take care of moving it to the output slot of the loop object. In the definitions of the auxiliary functions `showBasket` and `showCredit`, the operation `metaPrettyPrint` takes the metarepresentation of a coin or an item, and returns the list of quoted identifiers that encode the list of tokens produced by pretty-printing the coin or the item in the module `VENDING-MACHINE-SIGNATURE`. Coins and items, and, more generally, markings of a vending machine are metarepresented using the `META-LEVEL` function `upTerm` (see Section 14.5.1).

```

op showBasket : Marking -> QidList .
eq showBasket(I M)
  = metaPrettyPrint(upModule('VENDING-MACHINE-SIGNATURE, false),
    upTerm(I))
    showBasket(M) .
eq showBasket(C M) = showBasket(M) .
eq showBasket(null) = nil .

op showCredit : Marking -> QidList .
eq showCredit(C M)
  = metaPrettyPrint(upModule('VENDING-MACHINE-SIGNATURE, false),
    upTerm(C))
    showCredit(M) .
eq showCredit(I M) = showCredit(M) .
eq showCredit(null) = nil .

rl [showBasket] :
  < showBasket ; M ; nil >
  => < idle ; M ; ('\u 'basket: '\o showBasket(M) '\n) > .

rl [showCredit] :
  < showCredit ; M ; nil >
  => < idle ; M ; ('\u 'credit: '\o showCredit(M) '\n) > .

```

The rules labeled `insertCoin` implement the actions of inserting a dollar or a quarter in the vending machine. The strategy `insertCoin` defined in the module `BUYING-STRATS` (see Section 14.6) is used to produce the corresponding change in the current marking of the vending machine. Since strategies are applied at the metalevel, both the marking of the vending machine and the coin to be inserted must be first metarepresented using again the `META-LEVEL` function `upTerm`.

```

rl [insertCoin] :
  < q ; M ; nil >
  => < idle ;
    downTerm(insertCoin('add-q, upTerm(M)), null) ;
    nil > .

rl [insertCoin] :
  < $ ; M ; nil >
  => < idle ;
    downTerm(insertCoin('add-$, upTerm(M)), null) ;
    nil > .

```

The rules `buy1Cake`, `buy1Apple`, and `buyNItems` implement the actions of buying one or more items. The strategy `onlyNItems` defined in the module `BUYING-STRATS` (see Section 14.6) is used to produce the corresponding change in the current marking of the vending machine. Again, since strategies are applied at the metalevel, the marking of the vending machine must be first metarepresented.

```

r1 [buy1Cake]:
  < buy1Cake ; M ; nil >
  => < buy c : 1 ; M ; nil > .

r1 [buy1Apple]:
  < buy1Apple ; M ; nil >
  => < buy a : 1 ; M ; nil > .

r1 [buyNitems]:
  < buy c : N ; M ; nil >
  => < idle ;
      downTerm(onlyNitems(upTerm(M), 'buy-c, N), null) ;
      nil > .

r1 [buyNitems]:
  < buy a : N ; M ; nil >
  => < idle ;
      downTerm(onlyNitems(upTerm(M), 'buy-a, N), null) ;
      nil > .

endm

```

18.3 Using the loop

We illustrate the basic ideas of using the loop with a sample session with the interface for the vending machine. Once the `VENDING-MACHINE-INTERFACE` module has been entered, we must first initialize the loop by setting its initial state by means of the `loop` command.

```
Maude> loop init .
```

We can inspect the state of the loop with the `continue` command, abbreviated `cont`, as follows:

```
Maude> cont .
result System: [nil, < idle ; null ; nil >, nil]
```

Once the loop has been initialized, we can input any data by writing it after the prompt enclosed in parentheses. For example,

```
Maude> ($)

Maude> (showCredit)
credit: $

Maude> ($)

Maude> cont .
result System: [nil, < idle ; $ $ ; nil >, nil]

Maude> (q)

```

```

Maude> (buy1Apple)

Maude> (showBasket)
basket: a

Maude> (showCredit)
credit: $ q q

Maude> ($)

Maude> (buy a : 3)

Maude> (showBasket)
basket: a a a a

Maude> (showCredit)
credit: q

Maude> cont .
result System:
  [nil,
   < idle ; q a a a a ; nil >,
   '\u 'credit: '\o '\r '\! 'q '\o '\n]

```

Note that, as already mentioned, the data in the output stream remains there after being printed; it is removed at the time of the next input event.

18.4 Metalanguage applications: tokens, bubbles, and metaparsing

The example presented in the previous sections is a toy example to illustrate the basic features of the `LOOP-MODE` module. However, the most interesting applications of this module are *metalanguage* applications, in which Maude is used to define the syntax, parse, execute, and pretty print the execution results of a given object language or tool. In such applications, most of the hard work is done by the `META-LEVEL` module, but handling the input/output and maintaining the persistent state of the object language interpreter or tool is done by `LOOP-MODE`. Full Maude (see Chapter 19) is entirely implemented in Maude using the methodology explained in this section.

In order to generate in Maude an *environment* for a language \mathcal{L} , including the case of a language with user-definable syntax, the first thing we need to do is to define the syntax for \mathcal{L} -modules. This can be done by defining a data type $Sign_{\mathcal{L}}$ for \mathcal{L} -modules, and with auxiliary declarations for commands and other constructs. Maude provides great flexibility to do this, thanks to its mixfix front-end and to the use of *bubbles* (any non-empty list of Maude identifiers). The intuition behind bubbles is that they correspond to pieces of a module in a language that can only be parsed once the grammar introduced by the signature of the module is available.

The idea is that, for a language that allows modules with user-definable syntax—as is the case for Maude itself—it is natural to see its syntax as a combined syntax at two different levels: what we may call the *top-level* syntax of the language, and the user-definable syntax introduced in each module. The bubble data type allows us to reflect this duality of levels in the syntax definition by encapsulating portions of (as yet unparsed) text in the user-definable syntax. Similar ideas have been exploited using ASF+SDF [103, 104].

To illustrate this concept, suppose that we want to define the syntax of Maude in Maude. Consider the following Maude module:

```
fmod NAT3 is
  sort Nat3 .
  op s_ : Nat3 -> Nat3 .
  op 0 : -> Nat3 .
  eq s s s 0 = 0 .
endfm
```

Notice that the lists of characters inside the boxes are not part of the top level syntax of Maude and therefore should be treated as bubbles until they are parsed. In fact, they can only be parsed with the grammar associated with the signature of the module `NAT3`. In this sense, we say that the syntax for Maude modules is a combination of two levels of syntax. The term `s s s 0`, for example, has to be parsed in the grammar associated with the signature of `NAT3`. The definition of the syntax of Maude in Maude must reflect this duality of syntax levels.

So far, we have talked about bubbles in a generic way. In fact, there can be many different kinds of bubbles. In Maude we can define different types of bubbles as built-in data types by parameterizing their definition. Thus, for example, a bubble of length one, which we call a *token*, can be defined as follows:

```
sort Token .
op token : Qid -> Token
  [special (id-hook Bubble (1 1)
    op-hook qidSymbol (<Qids> : ~> Qid))] .
```

Any name can be used to define a bubble sort. It is the `special` attribute

```
id-hook Bubble (1 1)
```

in its constructor declaration that makes the sort `Token` a bubble sort. The second argument of the `id-hook` special attribute indicates the minimum and maximum length of such bubbles as lists of identifiers. Therefore, `Token` has only bubbles of size 1. To specify a bubble of any length we would use the pair of values 1 and -1. The operator used in the declaration of the bubble, in this case the operator `token`, is a bubble constructor that represents tokens

in terms of their quoted form. For example, the token `abc123` is represented as `token('abc123)`.

We can define bubbles of any length, that is, non-empty sequences of Maude identifiers, with the following declarations.

```
op bubble : QidList -> Bubble
  [special
   (id-hook Bubble          (1 -1)
    op-hook qidListSymbol  (__ : QidList QidList ~> QidList)
    op-hook qidSymbol      (<Qids> : ~> Qid))] .
```

In this case, the system will represent the bubble as a list of quoted identifiers under the constructor `bubble`. For example, the bubble `ab cd ef` is represented as `bubble('ab 'cd 'ef)`.

Different types of bubbles can be defined using the `id-hook` special attribute `Exclude`, which takes as parameter a list of identifiers to be excluded from the given bubble, that is, the bubble being defined cannot contain such identifiers. We can, for example, declare the sort `NeTokenList` with constructor `neTokenList` as a list of identifiers, of any length greater than one, excluding the identifier `'.` with the following declarations.

```
op neTokenList : QidList -> NeTokenList
  [special
   (id-hook Bubble          (1 -1)
    op-hook qidListSymbol  (__ : QidList QidList ~> QidList)
    op-hook qidSymbol      (<Qids> : ~> Qid)
    id-hook Exclude       ( . ))] .
```

In general, the syntax `Exclude (I1 I2 ... Ik)` is used to exclude identifiers `I1, I2, ..., Ik` inside tokens.

We are now ready to give the signature to parse modules such as `NAT3` above. The following module `MINI-MAUDE-SYNTAX` uses the above definitions of sorts `Token`, `Bubble` and `NeTokenList` to define the syntax of a sublanguage of Maude, namely, many-sorted, unconditional, functional modules, in which the declarations of sorts and operators have to be done one at a time, no attributes are supported for operators, and variables must be declared on-the-fly.

```
fmod MINI-MAUDE-SYNTAX is
  protecting QID-LIST .
  sorts Token Bubble NeTokenList .
  op token : Qid -> Token
    [special
     (id-hook Bubble          (1 1)
      op-hook qidSymbol      (<Qids> : ~> Qid))] .

  op bubble : QidList -> Bubble
    [special
     (id-hook Bubble          (1 -1)
```

```

    op-hook qidListSymbol (__ : QidList QidList ~> QidList)
    op-hook qidSymbol      (<Qids> : ~> Qid)] .

op neTokenList : QidList -> NeTokenList
  [special
   (id-hook Bubble      (1 -1)
    op-hook qidListSymbol (__ : QidList QidList ~> QidList)
    op-hook qidSymbol    (<Qids> : ~> Qid)
    id-hook Exclude     ( . ))] .

sorts Decl DeclList PreModule .
subsort Decl < DeclList .

--- sort declaration
op sort_.. : Token -> Decl .

--- operator declaration
op op_ : ->_.. : Token Token -> Decl .
op op_ : _->_.. : Token NeTokenList Token -> Decl .

--- equation declaration
op eq_=_.. : Bubble Bubble -> Decl .

--- functional module
op fmod_is_endfm : Token DeclList -> PreModule .
op __ : DeclList DeclList -> DeclList [assoc gather(e E)] .
endfm

```

Notice how we explicitly declare operators that correspond to the top-level syntax of Maude, and how we represent as terms of sort `Bubble` those pieces of the module—namely, terms in equations—that can only be parsed afterwards with the user-defined syntax. The name of the sort `PreModule` reflects the fact that not all terms of this sort do actually represent Maude modules. In particular, for a term of sort `PreModule` to represent a Maude module all the bubbles must be correctly parsed as terms in the module's user-defined syntax.

As an example, we can call the operation `metaParse`, from module `META-LEVEL`, with the metarepresentation of the module `MINI-MAUDE-SYNTAX` and the previous module `NAT3` transformed into a list of quoted identifiers.

```

Maude> red in META-LEVEL :
  metaParse(upModule('MINI-MAUDE-SYNTAX, false),
    'fmod 'NAT3 'is
    'sort 'Nat3 '.
    'op 's_ ': 'Nat3 '-> 'Nat3 '.
    'op '0 ': '-> 'Nat3 '.
    'eq 's 's 's '0 '= '0 '.
    'endfm,
    'PreModule) .

```

We get the following term of sort `ResultPair` as a result:

```
result ResultPair:
  { 'fmod_is_endfm' [ 'Nat3' ],
    '__ [ 'sort..' [ 'Nat3' ] ],
    '__ [ 'op_:->..' [ 's_', 'Nat3', 'Nat3' ] ],
    '__ [ 'op_:'->..' [ '0', 'Nat3' ] ],
    'eq_=_.. [ 's 's 's '0', '0' ] ] ], 'PreModule }
```

Of course, Maude does not return these boxes. Instead, the system returns the bubbles using their constructor form as specified in their corresponding declarations. For example, the bubbles `'Nat3` and `'s 's 's '0` are represented, respectively, as `token('Nat3)` and `bubble('s 's 's '0)`. Maude returns them metarepresented. The result given by Maude is therefore the following.

```
result ResultPair: {
  'fmod_is_endfm [ 'token [ ' 'Nat3.Qid ],
  '__ [ 'sort..' [ 'token [ ' 'Nat3.Qid ] ],
  '__ [ 'op_:->..' [ 'token [ ' 's_.Qid ],
    'neTokenList [ ' 'Nat3.Qid ],
    'token [ ' 'Nat3.Qid ] ],
  '__ [ 'op_:'->..' [ 'token [ ' '0.Qid ], 'token [ ' 'Nat3.Qid ] ],
  'eq_=_.. [ 'bubble [ '__ [ ' 's.Qid, ' 's.Qid, ' 's.Qid, ' '0.Qid ] ],
    'bubble [ ' '0.Qid ] ] ] ] ],
  'PreModule }
```

The first component of the result pair is a metaterm of sort `Term`. To convert this term into a term of sort `FModule` is now straightforward. As already mentioned, we first have to extract from the term the module's signature. For this, we can use an equationally defined function

```
op extractSignature : Term ~> FModule .
```

that goes along the term metarepresenting the premodule looking for sort and operator declarations; these are obtained by means of auxiliary operations `extractSorts` and `extractOpDecls`, respectively. Notice that the operation `extractSignature` is partial, because it is not well defined for metaterms of sort `Term` that do not metarepresent terms of sort `PreModule` in `MINI-MAUDE-SYNTAX`.

Once we have extracted the signature of the module—expressed as a functional module with no equations and no membership axioms—we can then build terms of sort `EquationSet` with an equationally defined operation `solveBubbles` (also partial) that recursively replaces each bubble in an equation with the result of calling `metaParse` with the already extracted signature and with the quoted identifier form of the bubble.

```
op solveBubbles : Term FModule ~> FModule .
```

Finally, the partial operation `processPreModule` takes a term and, if it metarepresents a term of sort `PreModule` in `MINI-MAUDE-SYNTAX`, and, furthermore, the `solveBubbles` function succeeds in parsing the bubbles in equations as terms, then it returns a term of sort `FModule`.

The complete specification of these operations is as follows:

```
fmod MINI-MAUDE is
  protecting META-LEVEL .

  vars T T1 T2 T3 : Term .
  vars TL TL' : TermList .
  var  QI : Qid .
  var  QIL : QidList .
  var  F : Qid .
  var  M : Module .

  op processPreModule : Term ~> FModule .
  eq processPreModule(T) = solveBubbles(T, extractSignature(T)) .

  ---- extractSignature
  op extractSignature : Term ~> FModule .
  op extractSorts : Term ~> SortSet .
  op extractOpDecls : Term ~> OpDeclSet .

  eq extractSignature('fmod_is_endfm['token[QI], T])
    = (fmod downTerm(QI, 'error) is
       nil
       sorts extractSorts(T) .
       none
       extractOpDecls(T)
       none
       none
       endfm) .

  eq extractSorts('sort_.'[token[T]]) = downTerm(T, 'error) .
  eq extractSorts('__[sort_.'[token[T1]], T2])
    = downTerm(T1, 'error) ; extractSorts(T2) .
  ceq extractSorts(F[TL]) = none if F /= '__ /\ F /= 'sort_.. .
  ceq extractSorts('__[F[TL], T])
    = extractSorts(T)
    if F /= 'sort_.. .

  eq extractOpDecls(
    'op_:_->_.'[token[T1], 'neTokenList[TL], 'token[T2]])
    = (op downTerm(T1, 'error) : downTerm(TL, nil)
       -> downTerm(T2, 'error) [none] .) .
  eq extractOpDecls('op_:'->_.'[token[T1], 'token[T2]])
    = (op downTerm(T1, 'error) : nil -> downTerm(T2, 'error)
       [none] .) .
```

```

ceq extractOpDecls(F[TL])
  = none
  if F /= '___ /\ F /= 'op_:_->_. /\ F /= 'op_:'->_. .

eq extractOpDecls(
  '___[ 'op_:_->_. ['token[T1], 'neTokenList[TL], 'token[T2]], T3])
  = (op downTerm(T1, 'error) : downTerm(TL, nil)
    -> downTerm(T2, 'error) [none] .
    extractOpDecls(T3)) .
eq extractOpDecls('___[ 'op_:'->_. ['token[T1], 'token[T2]], T3])
  = (op downTerm(T1, 'error) : nil -> downTerm(T2, 'error)
    [none] .
    extractOpDecls(T3)) .
ceq extractOpDecls('___[F[TL], T2]) = extractOpDecls(T2)
  if F /= 'op_:_->_. /\ F /= 'op_:'->_. .

---- solveBubbles
op solveBubbles : Term FModule ~> FModule .
op solveBubblesAux : Term FModule ~> EquationSet .
op addEqs : FModule EquationSet -> FModule .

eq solveBubbles('fmod_is_endfm['token[QI], T], M)
  = addEqs(M, solveBubblesAux(T, M)) .

eq solveBubblesAux('eq=_.[ 'bubble[T1], 'bubble[T2]], M)
  = (eq getTerm(metaParse(M, downTerm(T1, nil), anyType))
    = getTerm(metaParse(M, downTerm(T2, nil), anyType))
    [none] .) .
eq solveBubblesAux('___[ 'eq=_.[ 'bubble[T1], 'bubble[T2]], T3], M)
  = (eq getTerm(metaParse(M, downTerm(T1, nil), anyType))
    = getTerm(metaParse(M, downTerm(T2, nil), anyType))
    [none] .
    solveBubblesAux(T3, M)) .
ceq solveBubblesAux('___[F[TL], T2], M)
  = solveBubblesAux(T2, M)
  if F /= 'eq=_ . .
ceq solveBubblesAux(F[TL], M)
  = none
  if F /= '___ /\ F /= 'eq=_ . .

eq addEqs(fmod QI is
  IL:ImportList
  sorts SS:SortSet .
  SubSorts:SubsortDeclSet
  OpDecls:OpDeclSet
  MembAxs:MembAxSet
  Eqs:EquationSet
  endfm,
  Eqs':EquationSet)

```

```

= fmod QI is
  IL:ImportList
  sorts SS:SortSet .
  SubSorts:SubsortDeclSet
  OpDecls:OpDeclSet
  MembAxs:MembAxSet
  (Eqs:EquationSet Eqs':EquationSet)
endfm .
endfm

```

We have then the following reductions:

```

Maude> red in MINI-MAUDE :
  extractSignature(
    getTerm(metaParse(upModule('MINI-MAUDE-SYNTAX, false),
      'fmod 'NAT3 'is
        'op 's_ ': 'Nat3 '-> 'Nat3 '.
        'sort 'Nat3 '.
        'op '0 ': '-> 'Nat3 '.
        'eq 's 's 's '0 '= '0 '.
      'endfm,
      'PreModule))) .
result FModule: fmod 'NAT3 is
  nil
  sorts 'Nat3 .
  none
  op '0 : nil -> 'Nat3 [none] .
  op 's_ : 'Nat3 -> 'Nat3 [none] .
  none
  none
endfm

Maude> red in MINI-MAUDE :
  processPreModule(
    getTerm(metaParse(upModule('MINI-MAUDE-SYNTAX, false),
      'fmod 'NAT3 'is
        'sort 'Nat3 '.
        'op 's_ ': 'Nat3 '-> 'Nat3 '.
        'op '0 ': '-> 'Nat3 '.
        'eq 's 's 's '0 '= '0 '.
      'endfm,
      'PreModule))) .
result FModule: fmod 'NAT3 is
  nil
  sorts 'Nat3 .
  none
  op '0 : nil -> 'Nat3 [none] .
  op 's_ : 'Nat3 -> 'Nat3 [none] .
  none
  eq 's_['s_['s_['0.Nat3]]] = '0.Nat3 [none] .

```

```

endfm

Maude> red in MINI-MAUDE :
      processPreModule(
        getTerm(metaParse(upModule('MINI-MAUDE-SYNTAX, false),
          'fmod 'NAT3 'is
            'sort 'Nat3 '.
            'op 's_ ': 'Nat3 '-> 'Nat3 '.
            'op '0 ': '-> 'Nat3 '.
            'eq 's 's 's 'N:Nat3 '= 'N:Nat3 '.
          'endfm,
          'PreModule))) .
result FModule: fmod 'NAT3 is
  nil
  sorts 'Nat3 .
  none
  op '0 : nil -> 'Nat3 [none] .
  op 's_ : 'Nat3 -> 'Nat3 [none] .
  none
  eq 's_['s_['s_['N:Nat3]]] = 'N:Nat3 [none] .
endfm

```

18.5 Interactive Maude

In order for formal tools to be more generally useful it is important that they can interact with one another via simple, well defined, semantically meaningful communication interfaces. In addition, it is important for a formal tool to provide an interactive, graphical means of exploring and analyzing formal models.

This section describes the IOP+IMaude approach to developing an interactive extension of Maude. It is based on the notion of actors [15, 1] communicating via asynchronous message passing. In this case the idea is to make Maude an actor, rather than using Maude to specify a system of actors as illustrated in Chapters 11, 17, and 21. There are two aspects to making Maude an actor. One aspect is the communication infrastructure that enables exchanging messages with other actors. This is provided by IOP (InterOperability Platform). The other aspect is specifying the actor's behavior, that is, how messages are handled. IMaude is a set of Maude modules that provide a general framework for defining Maude actors.

We begin with a description of the design and implementation of IMaude in some detail, and provide a simple interactive application, the rewrite actor, built on top of IMaude. This IMaude actor can run as a standalone actor, interacting with the user through the LOOP-MODE interface. We then give a brief description of IOP, and show how to extend the rewrite actor to include interactions with the filemanager actor, provided in the IOP distribution, to read and write files, and to save and restore portions of the IMaude state.

We conclude by showing how the IOP+IMaude combination can be used to develop an interactive graphical representation of a Maude specification.

IMaude is available from <http://www.csl.sri.com/~clt/IMaudeWeb/>. It comes with several library modules, developed to facilitate interaction with other actors provided with the IOP distribution. It also includes documentation of the code and use scenarios, including the examples discussed below. IOP binaries and documentation are available from the IOP website <http://mcs.une.edu.au/~iop>.

Several substantial applications have been based on IOP+IMaude, including: a formal model of goal-based autonomous systems, instantiated to a simple rover (<http://www.csl.sri.com/users/denker/remoteAgents/>); an executable model for Strand Space protocol specifications (<http://www.csl.sri.com/users/clt/StrandWeb/>); and the Pathway Logic Assistant (<http://pl.csl.sri.com/>). The latter is the most substantial application and has been a key driving force in the development of both IOP and IMaude. It is briefly described in Section 23.2.8.

18.5.1 IMaude

IMaude is a collection of Maude modules that support writing interactive Maude applications. Although IMaude can be used in Maude alone for simple command line interaction, the intended use is as the basis for specifying the behavior of Maude actors within the IOP framework. In this setting, a Maude actor can interact not only with the user, but also with other actors, including actors providing file and socket management services, other Maude actors, actors providing graphical display services, and other formal tools such as model checkers, theorem provers, and so on. Applications are developed by extending the core IMaude system with data structures and rules describing Maude actor behavior specific to the application.

Here we discuss requirements for specifying actors and give an overview of the IMaude design. Then we describe the key data structures used to represent the Maude actor state. IMaude extends the data structure modules with two modules for processing input: `IMAUDE-STATE` and `SCHEDULER`. The module `IMAUDE-STATE` defines the rules for managing, initializing, examining, and resetting the state. The rules for scheduling interactions are defined in the `SCHEDULER` module. Finally, we describe the `REWRITE` module, treating this as an example of developing a simple Maude actor for interactive rewriting.

Requirements and overview

Implementing an actor behavior requires:

1. an interactive loop that maintains state between interactions; and
2. managing asynchronous interactions with other actors.

Requirement 1 is the most problematic, as the Maude interpreter is stateless by design. IMaude uses the `LOOP-MODE` module (see Section 18.1) provided by

Maude specifically to support writing user interfaces. An alternative would be the use of socket external objects (see Section 11.4). Although sockets are somewhat cleaner and more elegant than `LOOP-MODE`, there are two reasons why they are not used in the current version of IMAude: sockets were not available when IMAude was first developed, and, crucially, `LOOP-MODE` provides support for parsing and printing Maude terms that is not yet available using sockets. The need to support asynchronous interaction with multiple actors suggests a need for task management. Therefore, a Maude actor's state should include a representation of the processing state (current and pending tasks).

The objective of supporting very general behavior entails additional requirements, including:

3. extensible data structures for representing state along with functions for managing state (initialization, update, retrieval, reset), and
4. support for multiple ongoing asynchronous interactions.

IMaude provides two key data types (sorts) to support representation of an actor's state: an extensible sort `Val` to represent data values, and a sort `Request` to represent tasks. An IMAude state has the following components: control, requests, wait4s, environment, and log. The *control* component contains a description of the request currently being processed or the constant `ready`, indicating that no task is currently being processed. Pending tasks are partitioned into two classes: queued requests, stored in the *requests* component; and tasks waiting to handle incoming messages, stored in the *wait4s* component. Wait4 tasks are a kind of continuation; they play the role of either call-backs or service listeners. The *environment* component contains a set of entries mapping descriptors to annotated elements of `Val`. The *log* component is a list of log items; it supports debugging by allowing events and status to be recorded as requests are processed, without interrupting the processing. Additional support for debugging is provided by commands for browsing and resetting state components.

There are two forms of interaction with IMAude: commands and requests. Commands are typically submitted directly by the user, handled upon receipt, and generally result in a reply to the user (printed on the terminal or IOP's output window). Requests may be submitted by the user, sent in messages by other actors, or generated by IMAude in the process of handling some other request. Requests are queued and processed when enabled, possibly resulting in messages being sent to other actors or additional requests being queued.

IMaude provides rules for interpreting commands, rules for dispatching incoming messages, as well as rules for selecting requests from the request queue to process. By queuing requests as they arrive, either directly or as the result of matching a `wait4`, and processing them when cycles are available, incoming messages are not lost and deadlocks resulting from messages that are not immediately processable are avoided. Thus IMAude provides support for responsive asynchronous interaction with other actors.

IMaude data types

As mentioned above, the key sorts used to represent IMAude's state are `Val` and `Request`. `Val` is essentially a tagged union of sorts, thus making it easy to extend. Injection operators (constructors) are used to form a tagged union rather than simply making sorts such as `QidList` subsorts of `Val` to avoid confusion in, and possible collapse of, the sort hierarchy. In addition, there is a Boolean function, `compat`, on `Val`, that can be used to check whether two values belong to the same subsort. IMAude defines three `Val` subsorts: `QVal`, `SVal`, and `TVal`. Elements of `QVal` are of the form `ql(toks)`, where `toks` is a qid list (i.e., a list of quoted identifiers) and `ql` is the injection operation, so that `QVal` is the image of the sort `QidList` under `ql`. Elements of `SVal` are of the form `sv(s)`, where `s` is a string and `sv` is the injection operation. Elements of `TVal` are of the form `tm(modname, term)`, where `modname` is a qid naming a module, `term` is the metarepresentation of a term in that module, and `tm` is the injection operation.

In addition to specifying the injection operation, to specify a subsort of `Val` it is necessary to give axioms extending the `compat` function. It is also convenient to define selectors that extract the injected elements from their tagged form. For example, the subsort `QVal` is defined in the functional module `QVAL` as follows.

```
fmod QVAL is
  protecting QID-LIST .
  extending VAL .
  sort QVal .
  subsort QVal < Val .

  op ql : QidList -> QVal .
  eq compat(x:QVal,y:QVal) = true .
  var qidl : QidList .
  op qvalQidList : QVal -> QidList .
  eq qvalQidList(ql(qidl)) = qidl .
endfm
```

An element of the sort `Request` has either the form

```
req(reqid, val, reqQ)
```

or

```
creq(reqid, qids, val, reqQ)
```

where `reqid` is a qid identifying the request, `val` is the parameter, an element of `Val`, `qids` is a qid list, and `reqQ` is a list of requests, possibly empty, to be used in determining how to continue when the request is processed. We say that a request is *servicing as a continuation* if it appears in a `wait4` or in the request list of another request. The `creq` form is used when it is necessary to

separate parameters supplied to a request serving as a continuation at continuation time (its `qids` parameter), from the parameter supplied at request creation time (its `val` parameter).

The function `supplyPars(reqid, toks)` adds the qid list `toks` to the second parameter of the request `reqid`; in the case of a request of the form `req(reqid, val, reqQ)`, this is only defined if `val` has the form `ql(qids)`.

An IMAude state has the form

```
st(control, wait4s, requests, environment, log)
```

In the following we describe each of the five components.

The *control* component (sort `Control`) of an IMAude state reflects what IMAude is currently doing. An element of sort `Control` is either the constant `ready` or a term of the form

```
processing(req)
```

indicating that a request `req` is currently being processed.

The *wait4s* component is a set (sort `Wait4Set`) of elements of sort `Wait4`. An element of sort `Wait4` has the form

```
wait4(aname, toks, reqQ)
```

where `aname` is the name of an actor from whom IMAude is listening for a message, and `reqQ` is a list of requests that specifies what IMAude should do when such an expected message arrives. The qid list `toks` indicates the reason for waiting and is currently just used for debugging purposes. When a message arrives from the named actor, the message tokens are added to the qid list parameter of each request in `reqQ`, using the function `supplyPars`, and the instantiated requests are queued for processing. As mentioned above, an element of the *wait4s* component can play the role of a call-back, a standard technique used in many programming languages for asynchronous communication, or the role of a server listening for requests.

The *requests* component is a list of requests (sort `RequestQ`) waiting to be scheduled.

The *environment* component is a set (sort `ESet`) of entries (sort `Entry`) used to store values for later use. An entry has the form

```
e(etype, ids, notes, val)
```

where `etype`, a quoted identifier, is the entry type (typically corresponding to the value subsort), and `ids`, a qid list, uniquely identifies the entry amongst those in the entry set with the same type. The parameter `notes` of sort `Notes` is an annotation of the stored value `val`. The sort `Notes` consists of finite maps from `String` to `Val`. It can be used to associate many kinds of information to a stored value, including: the source, user-defined annotations, and links between entries. The default value for `notes` is the empty note set, `mt`. This

is formalized by defining an operator $e(\text{etype}, \text{ids}, \text{val})$ that rewrites to $e(\text{etype}, \text{ids}, \text{mt}, \text{val})$.

There are several functions for manipulating entry sets, including:

- $\text{getEntry}(\text{es}, \text{etype}, \text{ids})$ yields either the entry identified by the pair $(\text{etype}, \text{ids})$ in the entry set es , or the empty entry set if there is no such entry;
- $\text{removeEntry}(\text{es}, \text{etype}, \text{ids})$ yields the entry set obtained by removing the entry in es identified by the pair $(\text{etype}, \text{ids})$, if any;
- $\text{addEntry}(\text{es}, \text{etype}, \text{ids}, \text{notes}, \text{val})$ yields the entry set obtained by first removing any entry identified by $(\text{etype}, \text{ids})$ from es , and then adding the entry $e(\text{etype}, \text{ids}, \text{notes}, \text{val})$;
- $\text{getVal}(\text{es}, \text{etype}, \text{ids}, \text{val})$ returns the value identified by $(\text{etype}, \text{ids})$ if one exists that is compatible with val , and otherwise returns val as the default.

In the above, es has sort ESet , etype has sort Qid , ids has sort QidList , notes has sort Notes , and val has sort Val .

Finally, the *log* component is a list (sort Log) of log items (sort LogItem). Each log item has the form

```
log(id, toks, val)
```

One use of the log component is to record status of interactions with other actors; another use is to record a trace of interactions.

We note that the collection sorts RequestQ and Log are specified by instantiating the parameterized list module provided in the Maude's prelude (see Section 9.12.1). The instantiations use views mapping the element sort of the parameterized module to Request and LogItem , respectively. Similarly, the sorts Wait4Set and ESet are instantiations of a multiset module defined as part of the IMAude utility library. We axiomatize these sorts as multisets, but maintain the invariant that elements occur at most once, rather than incur the overhead of always trying to remove duplicates. The sort Notes is an instantiation of the parameterized array module provided in Maude's prelude (see Section 9.13.2). The use of parameterized modules greatly simplifies the specifications.

The IMAUDE-STATE module provides rules defining commands to initialize, examine, and reset components of the state. The rule *ini* initializes the control to *ready* and leaves the remaining components empty.

```
op init : -> System .
***          inQ      ctl wait4s reqQ es log outQ
rl [ini] : init => [nil, st(ready, mt, nil, mt, nil), nil] .
```

There are commands to display, via a message to the user, the current value of a state component, as well as commands to reset to their initial value each of the different state components. For example,

- `(show control)` prints the control component,
- `(reset control)` resets it to `ready`,
- `(show eset)` prints the environment,
- `(reset eset)` resets it to `mt`.

There are also commands to show or remove specific entries.

IMaude behavior

The `SCHEDULER` module provides rules to control the processing of input other than commands. The first `qid` of the input queue is used to classify the input type as a command, a request, or a message expected from a known actor. The Boolean function `isReq` is used to determine if a `qid` is a request identifier. When a new request is defined, an axiom for `isReq` must be added so that it will be properly handled. The rule `read.input` handles the case in which the first token is a request identifier. In this case, a request is constructed and added to the end of the requests component of the state. The token is used as the first argument to the request constructor and the remaining `qids` in the input queue are used as the second argument, tagged as a `QVal`.

```

cr1 [read.input] :
  [token InQ, st(ready, wait4s, reqQ, es, log), OutQ]
=> [nil,
    st(ready, wait4s, (reqQ req(token, ql(InQ), nil)), es, log),
    OutQ]
  if isReq(token) .

```

If the first `qid` of the input is the name of an actor with a `wait4` entry, then the `wait4` entry is removed, and its `requests` parameter is appended to the request queue after supplying the input `qids` to each request. This is handled by the rule `schedule.wait4`.

```

r1 [schedule.wait4] :
  [aname InQ,
   st(ready, (wait4(aname, toks, reqQ') ! wait4s), reqQ, es, log),
   OutQ]
=> [nil,
    st(ready, wait4s,
      (reqQ, supplyPars(reqQ', aname InQ)), es, log),
    OutQ] .

```

If there is no pending input, and there is an enabled request in the request queue, the first such request can be scheduled. This is specified by the rule `schedule.request`. The Boolean function `enabled` is used to determine if a request is enabled in the context of a `wait4` set. For example, interactions with each known actor can be sequentialized by disabling a request that might result in sending a message to an actor for whom there is already a `wait4` entry. The next request to schedule is determined by evaluating

```

findEnabled(wait4s, reqQ, nil)

```

which returns a pair `?req @ reqQ'` consisting of the first enabled request `?req` in `reqQ`, and the rest of the requests, `reqQ'`. If there is no enabled request the pair `nil @ reqQ` is returned. Because `?req` is declared to be of sort `Request`, the rule condition will be false in the latter case.

```

crl [schedule.request] :
  [nil, st(ready, wait4s, reqQ, es, log), OutQ]
=> [nil, st(processing(?req), wait4s, reqQ', es, log), OutQ]
  if (?req @ reqQ') := findEnabled(wait4s, reqQ, nil) .

```

Interactive rewriting

The `REWRITE` module is an IMAude application that defines requests for querying modules loaded into Maude. Terms can be reduced and rewritten using the default interpreter or by specifying a list of rules to apply. The results are saved in the environment for further processing. In addition, functions from the object module can be applied to arguments stored in the environment. In all cases the request continuation, with no additional arguments provided, is queued once the environment is updated. These requests can be entered directly by the user, in which case the continuation will be `nil`. They can also be invoked as part of the processing of another request. The `REWRITE` module makes essential use of the Maude `META-LEVEL` (see Chapter 14).

Naming things in the environment

The `setqc`, `letc`, and `applyc` requests provide a means for storing qid lists and terms in the environment.

- `(setqc vname qids)` adds an entry `e('qval, vname, ql(qids))` to the environment. (Recall that the default `notes` argument for an entry is `mt`.)
- `(letc vname modname sort <exp>)` attempts to parse the qid list that results from reading and tokenizing `<exp>` in the module named by `modname` as an element of sort `sort`.¹ If successful, the resulting term is reduced to canonical form, `res`, and an entry

```
e('tval, vname, tm(modname, res))
```

of type `tval` with identifier `vname` and value `tm(modname, res)` is added to the environment.

- `(applyc modname vname fname arg-1 ... arg-n)` applies the function named `fname` to arguments stored (as `tvals`) in `arg-1 ... arg-n` in the module named by `modname`, reducing the application to canonical form, `res`. The result is saved in `vname`, i.e., an entry of type `tval` with identifier `vname` and value `tm(modname, res)` is added to the environment.

¹ From the user's point of view, `<exp>` appears as the term would if typed as part of a Maude command in the context of the named module, while from the IMAude point of view, it appears in the input component of a `LOOP-MODE` system as a qid list.

As an example, we show the code for the `letc` request. It begins with equations specifying that `letc` is a request identifier, and that `letc` requests with sufficiently many arguments—they must have at least a value name, a module name, and a sort—are always enabled. Requests that are ill-formed because they have too few tokens just remain in the request queue.

```
eq isReq('letc) = true .
eq enabled(wait4s, req('letc, ql(vname modname sort toks), reqQ'))
  = true .
```

The rule `letc` specifies how to process a `letc` request.

```
crl [letc] :
  [nil,
   st(processing(req('letc, ql(vname modname sort toks), reqQ')),
     wait4s, reqQ, es, log),
   OutQ]
=> [nil,
    st(ready, wait4s, (reqQ reqQ'), es', log'),
    OutQ]
if res?? := metaParse([modname], toks, sort)
/\  es' :=
  (if (res?? :: ResultPair)
    then addEntry(es, 'tval, vname,
                  tm(modname,
                    getTerm(metaReduce([modname],
                                        getTerm(res??))))))
    else es fi )
/\  log' :=
  (if (res?? :: ResultPair)
    then log
    else (log log('noParse,
                  'letc vname modname sort toks,
                  dummy))
  fi) .
```

First, the `qid` list `toks` is parsed using the descent function `metaParse` (see Section 14.5.7), and bound to the variable `res??` of kind `[ResultPair?]`. The term `[modname]` refers to the module loaded into Maude with the name `modname`. The sort `ResultPair?` includes terms that indicate parsing errors as well as type-term pairs resulting from successful parsing. Its associated kind `[ResultPair?]` contains additional terms that cannot be reduced, for example if the `qid` `modname` does not name a known module. The term `(res?? :: ResultPair)` is true if `res??` has sort `ResultPair`, indicating a successful parse. In this case the term component of the pair, `getTerm(res??)`, is reduced using the descent function `metaReduce` (see Section 14.5.2), the reduced term is extracted again using the `getTerm` selector, and the result is added to the environment using the `addEntry` function. If parsing fails, then the

environment is left unchanged, and a log item is added to the log reporting the parse failure.

Rewriting terms in the environment

The `rewritesc` and `applyrulesc` requests provide a means for rewriting a term stored in the environment and saving the result.

- (`rewritesc nat vname rname`) rewrites the term stored with entry type `tval` and identifier `vname`, using at most `nat` rewrites (rule applications). If `rname` is present, the result of rewriting is stored as an entry of type `tval` and identifier `rname`, otherwise it is stored back in `vname`.
- (`applyrulesc vname rname q`) tries to apply each rule named in the `qid` list stored under identifier `q` to the term stored in `vname`. The result is stored in `rname`. Rules that don't apply are simply skipped.
- (`listrulesc vname rname`) lists the names of rules that apply to the term stored in `vname`, each followed by the number of application instances. The result is stored in `rname`.

The `rewritesc` request uses Maude's default rewrite strategy, also used by the `rewrite` command. There is also a `frewritesc` request that uses the same position-fair rewrite strategy used by the `frewrite` command. (See Section 6.4 for explanations and examples.)

Using the rewrite IMAude application

We conclude the discussion of the `REWRITE` module with a small scenario illustrating its use. For this purpose we recall the `VENDING-MACHINE` module defined in Section 6.1.

```
mod VENDING-MACHINE is
  including VENDING-MACHINE-SIGNATURE .
  var M : Marking .
  rl [add-q] : M => M q .
  rl [add-$] : M => M $ .
  rl [buy-c] : $ => c .
  rl [buy-a] : $ => a q .
  rl [change]: q q q q => $ .
endm
```

We extend the `VENDING-MACHINE` module with definitions of operators that return either the coins or the items in a marking.

```
mod VENDING-MACHINE-QUERY is
  including VENDING-MACHINE .
  var M : Marking .
  var I : Item .
  var C : Coin .
  ops getItems getCoins : Marking -> Marking .
```

```

eq getItems(I M) = I getItems(M) .
eq getItems(M) = null [owise] .
eq getCoins(C M) = C getCoins(M) .
eq getCoins(M) = null [owise] .
endm

```

After loading the above module, `IMaude`, and the `REWRITE` module into Maude, and initializing the loop state (using the command `loop init .`, as described in Section 18.3), we can use the rewrite requests to query the `VENDING-MACHINE` module.

To begin we use the `letc` request to define an initial marking and store it in `vm0`.

```
(letc vm0 VENDING-MACHINE-QUERY Marking $ $ $)
```

This results in an entry

```
e('letc, 'vm0, tm('VENDING-MACHINE-QUERY, vm0T))
```

being added to the entry set component of the state, where `vm0T` is the metarepresentation of the marking `$ $ $` (for the curious reader, it is the term `'_['$.Coin,'$.Coin,'$.Coin]`).

Now we can rewrite this marking, say for five steps, and save the result as `vm1`.

```
(rewritec 5 vm0 vm1)
(show entry tval vm1)
```

Using the `show entry` command we see that the result is `$ $ $ $ q q q`, as the rule-fair rewrite strategy first applies the rule `add-q`, then `add-$`, then `add-q`, and so on (see Section 6.4 for discussion).

In contrast, if the position-fair rewriting command `frewrite` is used,

```
(frewritec 10 1 vm0 vm2)
```

the result is `$ $ q q a c`. Having saved the results of rewriting, we can extract different parts of the state by applying appropriate functions. For example, the coins and items resulting from the `frewrite` command can be obtained by

```
(applyc VENDING-MACHINE-QUERY coins getCoins vm2)
(applyc VENDING-MACHINE-QUERY items getItems vm2)
```

Now the `coins` entry contains (the metarepresentation of) `$ $ q q q` and the `items` entry contains `a c`. The `applyc` request is especially useful if the term you are rewriting is large and you want to rewrite for a few steps, examine a small part of the result, and then continue rewriting.

The `applyrulesc` request corresponds to a simple internal strategy (see Section 14.6) for rewriting, namely applying a specified sequence of rules. For example, defining `rls` to be the list of rule names `buy-c buy-a buy-c` we can

rewrite the initial marking according to the associated strategy. The result, `q a c c`, is stored in `vm3`.

```
(setqc rls buy-c buy-a buy-c)
(applyrulesc vm0 vm3 rls)
```

Finally, using the request `litrulesc` we can find out how many ways each rule can apply to a given state. For example, the following request stores, in `vm2rules`, information about the number of ways each rule can apply to the `frewrite` result (stored in `vm2`).

```
(litrulesc vm2 vm2rules)
```

Showing this entry we get the `qid` list

```
add-$ 47 add-q 47 buy-a 1 buy-c 1 change 0
```

The interpretation is that there are forty seven ways to apply each of the `add-$` and `add-q` rules, and one way to apply each of the `buy-a` and `buy-c` rules. The `change` rule does not apply to the marking in `vm2`.

Comparing interface approaches

The IMAude `REWRITE` extension provides a simple, general user interface that can be used to interact with any Maude module. In contrast, the `VENDING-MACHINE` user interface (UI) described in Section 18.2 is specific to the `VENDING-MACHINE`, although the approach is quite general. The UI approach, being specific, can be more “user friendly,” while the IMAude extension can be used with any newly developed module without additional effort (beyond defining functions to analyze rewriting results). A technical distinction between the two approaches is the way in which reflection and the metalevel are used. In the IMAude approach, terms are metarepresented in the state, descent functions are used to implement commands/requests, and the metalevel parsing and printing functions are used for user interaction. In the special purpose UI approach, the module being interfaced is combined with the metalevel, reflection (up, down) is used to move between representations, and special purpose parsing and printing functions are developed.

18.5.2 IOP

The IOP system provides infrastructure for allowing tools to interact and interoperate. It was motivated by the specific aim of making it possible for Maude to communicate with other tools, including other instances of itself, web resources, visualization tools, and theorem provers.

An executing IOP system consists of a pool of actors. There is always a system actor. Other actors are created by requests to the system actor, either at startup or during execution, according to application needs. IOP comes with a basic set of actors including: a system actor, wrappers that encapsulate

Maude and other formal tools as actors, a `graphics2d` actor, communication actors that support sockets, file system access, and a GUI interface to the system that allows the user to communicate as an IOP actor. Additional actors can be added quite easily. The `graphics2d` actor supports specification of interactive graphical representation of models via the `JLambda` language [306]. `JLambda` is a Scheme-like language that provides full access to standard Java classes and defines classes with dynamically extensible fields and methods that simplify developing interactive graphical representations. The combination `IOP+IMaude` provides the Maude programmer with a much richer modeling environment, with support for developing visualization and animation of Maude specifications in interesting ways, for exporting Maude modules to other tools (based on other formalisms) to perform alternative analyses and visualizations, and for developing notions of session state that can be saved and resumed. Using the communication actors as a go-between, the Maude actor or any other tool adapted to become an IOP actor, can talk to any tool that is capable of interacting via an internet socket connection or the file system.

An actor in IOP is typically a UNIX-style process that has been registered with the system according to a simple procedure. Part of this registration process involves allocating three FIFOs, or UNIX-style named pipes, and redirecting the actor's `stdin`, `stdout`, and `stderr` file descriptors to these special files [358]. There is no restriction on the language used to write an actor's script or executable. Some are written in C, some are written in Java, some are written in Perl. One simply chooses the appropriate language for the desired task or function that the actor is supposed to perform. Actors can be single-threaded or multi-threaded, each according to its needs. They can even consist of several processes written in different languages. The process by which new actors can be incorporated into the system is described in the IOP manual [241].

For example, the Maude actor consists of two processes, one running the Maude executable, while the other, called the *wrapper*, acts as an intermediary between Maude and the registry. Any error messages Maude emits are, like all other actor's error messages, redirected to the error and output text area of the GUI front end. Maude's output is interpreted by the wrapper, and is then translated to a format acceptable to the underlying *inter-actor* communication system. The process of interpretation consists of replacing symbolic control characters such as `\n`, `\r`, `\t`, `\"`, and `\\` by the appropriate control sequences themselves.

Apart from the dynamic pool of actors in the system, IOP consists of three independent processes that interact: the `main` process that creates and configures the system; the registry; and a GUI front end. Invoking IOP from the command line results in the following startup procedure taking place. The first process, being the `main` of IOP, parses the command line arguments, and creates the registry or system actor, the GUI actor, and any other actors that have been requested. A typical IOP process configuration is shown in

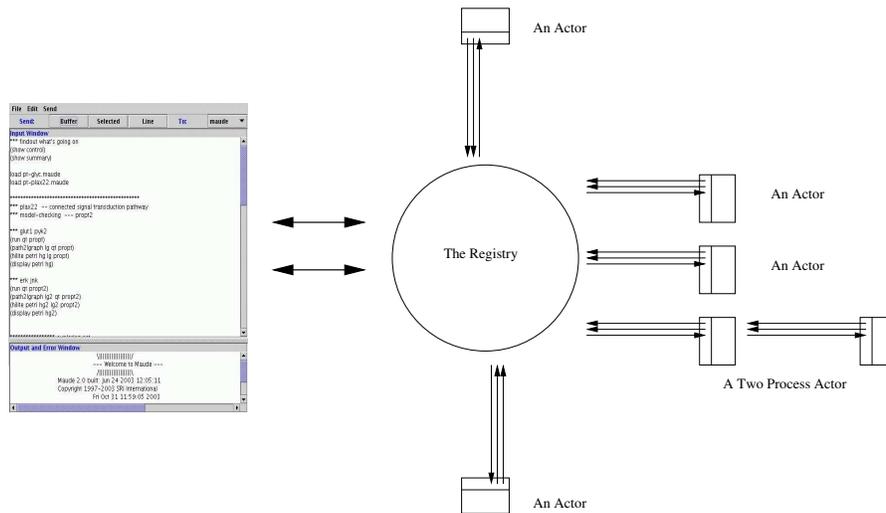


Fig. 18.1. An IOP process configuration

Figure 18.1. After startup, the `main` process acts mainly as a signal handler, ensuring clean and graceful shutdown. The registry keeps track of the current actors and maintains the lines of communication between these actors, making sure that messages sent by an actor are delivered to the specified target. The GUI front end, pictured in Figure 18.1 on the left, provides the user with an easy means of becoming an actor and sending messages to other actors in the system. The upper part can be used to compose messages to be sent to any of the IOP actors. The lower part displays any output from the actors that is not inter-actor communication (errors or messages to the user).

Inter-actor communication is purely ASCII text. A message consists of the name of the target actor, the name of the sending actor, followed by the body of the message, each on a new line. For example, a message from the Maude actor to the `graphics2d` actor consisting of an expression to interpret might look like:

```
graphics2d
maude
(involve graph "redisplay")
```

Interacting with the IOP filemanager actor

The IMAude `FILEMANAGER` module specifies rules for requests to read and write files and to save and restore components of the IMAude state by communicating with IOP’s filemanager actor. To support saving and restoring state, the IMAude utilities library provides `show<sort>` and `parse<sort>` functions for the `Val` and `Notes` sorts. IMAude extends the list of sorts to cover all IMAude state data types. Like the Maude functions `metaPrettyPrint` and `metaParse`,

the functions `show<sort>` and `parse<sort>` generate and consume qid lists. These qid lists are tokenizations of a textual representation inspired by the S-expression notation. Symbols are grouped by enclosing them in balanced pairs of parentheses, `(tag ...)`, where the first symbol of a group is a tag that specifies the sort of entity represented. These functions are also used by the `show` commands of the `IMAUDE-STATE` module described in Section 18.5.1. For example, in the vending machine example above, after executing

```
(letc vm0 VENDING-MACHINE-QUERY Marking $ $ $)
```

showing the entry produces the external representation

```
(entry tval (qidlist vm0) (notes) (tval VENDING-MACHINE-QUERY $ $ $))
```

Here, the first `tval` is the type argument to the entry constructor, while the second `tval` is a type tag.

The main file management requests are summarized in the following.

- `(filewrite <fname> <mode> <toks>)` is a request to write to a file. It is enabled whenever IMAude is not waiting for the filemanager. When the request is processed, a `write` message is sent to the filemanager actor, specifying the mode `<mode>`, the file name `<fname>`, and the qid list `<toks>` (the file contents). The write mode will be *append* if `<mode>` is 'A, and otherwise a new file will be created, overwriting any existing file of the same name. A `wait4` task is added to the `wait4s` state component, with a continuation that logs the filemanager's reply, followed by the continuation requests from the `filewrite` request. The filemanager's reply indicates whether or not the file write succeeded.
- `(fileread <fname> <vname>)` is a request to read from a file. It is enabled whenever IMAude is not waiting for the filemanager. When the request is processed, a `read` message is sent to the filemanager actor, specifying the file name `<fname>`, and a `wait4` is added to the `wait4s` state component to handle the filemanager's reply. The `wait4` continuation request queue contains a request to store the result in a `qval` with identifier `<vname>` followed by the continuation requests from the `fileread` request.
- `(save <fname> <mode> <stype> <toks>)` is a request to save state. It is always enabled. When a `save` request is processed, a `filewrite` request with parameters `<fname>`, `<mode>`, and a qid list obtained by "showing" the specified part of the IMAude state. The qid `<stype>` specifies the state component to be saved: `control`, `requests`, `wait4s`, `eset`, `entry`, or `log`. Except for the `entry` case, the full named state component is shown, by applying the appropriate `show<sort>` function to that state component, and `<toks>` is ignored. In the `entry` case, the `showEntry` function is applied to the entry identified by `<toks>`.
- `(restore <fname> <vname> <stype>)` is a request to restore some state component from a file. It is always enabled. When processed, a `fileread` request is queued with parameters `<fname>`, `<vname>`, and a continuation

that will parse the qid list stored in <vname> as a component saved using component type <stype>, and restore the corresponding state component.

As an example we show the axioms and rule specifying a `fileread` request.

```

eq isReq('fileread) = true .
eq enabled(wait4s, req('fileread, ql(fname vname toks), reqQ0))
  = not(waiting4(wait4s, 'filemanager)) .

rl [fileread] :
  [nil,
   st(processing(req('fileread, ql(fname vname toks), reqQ0)),
      wait4s, reqQ, es, log),
   OutQ]
=>
  [nil,
   st(ready,
      wait4s !
      wait4('filemanager, 'read fname,
            creq('setqval, nil, ql(vname), reqQ0)),
      reqQ, es, log),
   OutQ 'filemanager '\n 'maude '\n 'read '\n fname] .

```

Note that we use the `creq` request constructor in the `wait4` continuation, to separate the request time parameter `vname` from the qid list supplied when the `filemanager`'s reply is received. As indicated above, the `setqval` request stores the supplied qid list as a `qval` with identifier `vname` and then queues its continuation `reqQ0`. In the case of a restore request `reqQ0` will begin with a request to parse the contents of `vname` and update the `IMaude` state.

Using the filemanager

To illustrate interaction with the `filemanager`, imagine that we start `IOP`, telling `IOP` to create `Maude` and `filemanager` actors, and telling `Maude` to load the vending machine specification (from Section 18.5.1) and `IMaude` extended with the `REWRITE` and `FILEMANAGER` modules. Now we can send commands to `IMaude` via the `IOP GUI`. Suppose we have sent

```
(letc vm0 VENDING-MACHINE-QUERY Marking $ $ $)
```

Then we can save this entry in the file named `vm0.txt` using the following request:

```
(save vm0.txt C entry tval vm0 )
```

Now the file `vm0.txt` contains the external representation of the entry

```
(entry tval (qidlist vm0) (notes) (tval VENDING-MACHINE-QUERY $ $ $))
```

Next we remove the `vm0` entry and confirm it is gone by asking `IMaude` to show the entry.

```
(remove entry tval vm0)
(show entry tval vm0)
```

Finally, we restore the entry from the file `vm0.txt` and show it to confirm the entry has indeed been restored.

```
(restore vm0.txt tmp entry)
(show entry tval vm0)
```

An interactive graphical vending machine

The main goal of the IOP+IMaude work is to enable interactive visual representations of executable Maude specifications. The main steps to develop such a representation are:

- specifying the Maude representation of the system;
- defining the set of interactions—what operations can be performed on the system; and
- designing a visual representation of the system state and defining the interactive elements.

The `graphics2d` actor provided by IOP is used for visualization. `JLambda` code is written to define visual objects. Action listeners associated to interactive elements respond to user gestures by sending requests for the corresponding operation to the Maude actor. IMaude is extended with an application-specific module defining rules to initialize system state and handle the requests for operations. In addition to carrying out the operations locally, the rules must provide for sending updates to the `graphics2d` actor. These messages should contain `JLambda` expressions whose evaluation will update the visual object and redisplay it.

To illustrate the main ideas, we describe a very simple visual interface to our vending machine example. On the Maude side, the representation of the vending machine state is a marking. Its metarepresentation is stored as a `tval` whose identifier is the vending machine name. The possible interactions are to insert a coin, make change, or buy an item.

Figure 18.2 shows screen shots of the vending machine. The text areas display the machine state. The line labeled “Deficit:” keeps track of the total money inserted, as an added feature. The interactive elements are menus: one (labeled “Insert Money”) with menu items for inserting a dollar, inserting a quarter, and making change ; and one (labeled “Buy Item”) with menu items for buying an apple or buying a cake. When a menu item is selected, the associated action listener sends a message to the Maude actor requesting the corresponding operation. For example, if the user selects the insert quarter menu item, the following `JLambda` expression is executed.

```
(sinvoke "g2d.util.ActorMsg" "send" "maude" "vm" "vend add-q")
```

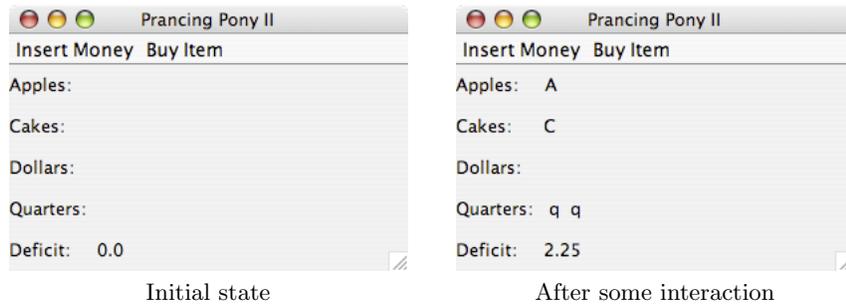


Fig. 18.2. A simple vending machine interface

Here "vm" is the name used by Maude and the graphics2d actor to refer to the vending machine object. The keyword `sinvoke` invokes a static method, in this case the `send` method in the `g2d.util.ActorMsg` class. This causes the third argument to be sent to the actor named by the first argument, in this case the Maude actor, giving the second argument as the sender. The result is that the Maude actor receives the message

```
(vm vend add-q)
```

This is received by a listener for messages from `vm`, in the `wait4` set. Recall from Section 18.5.1 that each `wait4` has the form `wait4(aname,ql(toks),reqQ)`. When a message of the form `(aname requestid args)` arrives, the full message is supplied to each of the requests in `reqQ` and the resulting requests are queued for processing. In the case of the vending machine listener, `reqQ` is a single request. Processing this request restarts the vending machine listener, and queues the request `(vend vm <cmd>)`, where in the above example `<cmd>` is `add-q`. The Maude rule for handling a `vend` request uses the `applyrulesc` request to apply the corresponding vending machine rule. This request is queued with a continuation that sends a state update to the vending machine object.

The vending machine application is started by starting IOP telling it to create a `graphics2d` actor and a Maude actor, telling Maude to load the vending machine assistant loader file. This file loads the vending machine specification, `IMaude`, and the `VEND-ASSISTANT` module that defines the interaction rules. It also contains the following sequence of requests to initialize the Maude actor state and the vending machine object.

```
(letc vm VENDING-MACHINE-QUERY Marking null)
(setqc add-$ add-$)
(setqc add-q add-q)
(setqc buy-c buy-c)
(setqc buy-a buy-a)
(setqc change change)
(loadg2dlib graphics2d vend.lsp)
```

```
(startListener vm vendreq graphics2d)
```

The vending machine marking is initialized to `null` (first line). The next five lines define rule lists to be used with the `applyrulesc` request. The `loadg2dlib` request sends a request to the `graphics2d` actor to load the file `vend.lsp` which contains the `JLambda` code to define and display a vending machine object. Finally a vending machine listener is started. The lefthand side of Figure 18.2 shows the initial vending machine state. The righthand side shows the state after inserting two dollars and a quarter, and buying an apple and a cake.

Part II

Full Maude

Full Maude: Extending Core Maude

During the development of the Maude system we have put special emphasis on the creation of metaprogramming facilities to allow the generation of execution environments for a wide variety of languages and logics. The first most obvious area where Maude can be used as a metalanguage is in building language extensions for Maude itself. Our experience in this regard—first reported in [121], and further documented in [122, 111, 112, 123]—is very encouraging.

We have been able to define in Core Maude a language, that we call Full Maude, with all the features of Maude plus notation for object-oriented programming, parameterized views, module expressions specifying tuples of any size, etc. Although the Maude distribution has included the specification/implementation of Full Maude since it was first distributed in 1999, Core Maude and Full Maude are now closer than ever before. Many of the features now available in Core Maude, like parameterized modules, views, and module expressions like summation, renaming and instantiation, were available in Full Maude long before they were available in Core Maude [121]. In fact, Full Maude has not only been a complement to Core Maude, but also a vehicle to experiment with new language features. Once these features have been mature enough to be implemented in the core language, we have made the effort to do so. Similarly, it is very likely that those features in Full Maude which are not yet available in Core Maude will become part of it sooner or later, and that new features will be added to Full Maude for purposes of language design and experimentation. This applies not only to Full Maude, but also to further language extensions based on Full Maude such as the strategy language proposed in [234], whose Core Maude implementation is currently underway.

Full Maude implements a complete user interface for the extended language. Using the `META-LEVEL` and `LOOP-MODE` modules, we have been able to define in Core Maude all the additional functionality required for parsing, evaluating, and pretty-printing modules in the extended language, and also for input/output interaction, as already discussed in Chapter 18. Thanks to the efficient implementation of the rewrite engine, the parser, and the `META-LEVEL` module, such a language extension executes with reasonable efficiency.

Full Maude contains Core Maude as a sublanguage, so that Core Maude modules can also be entered at the Full Maude level. However, currently there are a few syntactic restrictions that have to be satisfied by modules and commands in order to be acceptable inputs at the Full Maude level, including the fact that Full Maude inputs, for both modules and commands, must be enclosed in parentheses. These syntactic restrictions are explained in Section 19.6.

The structure of this chapter is as follows. Section 19.1 gives instructions on how to load and use Full Maude, how to enter modules, reduce terms, trace executions, etc. Section 19.2 explains how modules in Core Maude's database may be used in Full Maude. Section 19.3 introduces the additional module operations that are available in Full Maude. Section 19.4 explains how to move terms and modules up and down reflection levels. Section 19.6 enumerates the main differences between Full Maude and Core Maude. Finally, Section 19.7 illustrates how to add new commands to Full Maude.

19.1 Running Full Maude

Since the execution environment for Full Maude has been implemented in Core Maude, to initialize the system so that we can start using it the first thing we have to do is to load the `FULL-MAUDE` module in the system. Assuming that the file `full-maude.maude`, which contains the executable specification of Full Maude, is located in the current directory (or in a place where Maude can locate it, see Section 2.2), we just need to type the corresponding `in` or `load` command in the Maude prompt:

```
Maude> load full-maude.maude
```

```
Full Maude 2.7 March 10th 2015
```

The Full Maude system is then loaded, and we can use it as any other module.

Since Maude can take file names as arguments when started, assuming one is working on a Linux platform, one may also run Maude as follows:

```
~/maude-linux/bin$ ./maude.linux full-maude.maude
\|/
--- Welcome to Maude ---
/|/
Maude 2.7 built: Mar 3 2014 18:07:27
Copyright 1997-2014 SRI International
Tue Jul 15 22:57:15 2014
```

```
Full Maude 2.7 March 10th 2015
```

At the end of this file `full-maude.maude` there is the command

```
loop init .
```

which initializes the system just after loading the specification. This command starts the read-eval-print loop (see Section 18.1) to allow the interaction with the user by entering modules, theories, views, and commands, and to maintain a database in which to store all the modules, theories and views being introduced. The term `init` is a constant of sort `System`, in the specification of Full Maude, standing for the initial state of the Full Maude database.

Typing control-C may result in the loop being broken, and with it the current execution of Full Maude. Maude may try to recover the loop by itself, but if it is not successful we must reinitialize it with the `loop` command. That is, we need to type

```
Maude> loop init .
```

This command will be successful only if the `full-maude.maude` file is loaded and the `FULL-MAUDE` module is the default one. If it is not the default one, we may select it with the `select` command (see Section 25.13):

```
Maude> select FULL-MAUDE .
Maude> loop init .
```

The `loop init` command may be omitted here: Maude will try to restart the loop, using the last `loop` command, if something is written in parentheses henceforth.

Let us recall from Section 18.1 that to get something into the `LOOP-MODE` system the text *has to be enclosed in parentheses*. This means that any module, theory, view, or command intended for Full Maude has to be enclosed in parentheses. Since Core Maude is running underneath Full Maude—indeed, it now provides what might be called the *system programming* level—it will handle any input not enclosed in parentheses. This allows the possibility of using both systems at the same time. Thanks to this, we may use many Core Maude commands when interacting with Full Maude. For example, we may use Core Maude trace or profile facilities on Full Maude specifications, may load files, etc. However, this may lead to some confusion, and we should take care of putting parentheses around those pieces of text intended for Full Maude.

A Core Maude module, such as those presented in previous sections, can be entered in Full Maude by enclosing it in parentheses. For example, a module `PATH`¹ can be entered to Full Maude as follows:

```
Maude> (fmod PATH is
      sorts Node Edge .
      ops source target : Edge -> Node .

      sort Path .
      subsort Edge < Path .
      op _;_ : [Path] [Path] -> [Path] .
```

¹ Some fragments of this module have been discussed in Sections 3.5 and 4.3.

```

var E : Edge .
vars P Q R S : Path .
cmb E ; P : Path if target(E) = source(P) .
ceq (P ; Q) ; R
  = P ; (Q ; R)
  if target(P) = source(Q) /\ target(Q) = source(R) .

ops source target : Path -> Node .
ceq source(P) = source(E) if E ; S := P .
ceq target(P) = target(S) if E ; S := P .

protecting NAT .

ops n1 n2 n3 n4 n5 : -> Node .
ops a b c d e f : -> Edge .
op length : Path -> Nat .

eq length(E) = 1 .
ceq length(E ; P) = 1 + length(P) if E ; P : Path .

eq source(a) = n1 .
eq target(a) = n2 .
eq source(b) = n1 .
eq target(b) = n3 .
eq source(c) = n3 .
eq target(c) = n4 .
eq source(d) = n4 .
eq target(d) = n2 .
eq source(e) = n2 .
eq target(e) = n5 .
eq source(f) = n2 .
eq target(f) = n1 .
endfm)

rewrites: 5438 in 10ms cpu (157ms real) (543800 rews/sec)
Introduced module PATH

```

As in Core Maude, we can enter any module or command by writing it directly after the prompt, or by having it in a file and then using the `in` or `load` commands. Also as in Core Maude, we can write several Full Maude modules or commands in a file and then enter all of them with a single `in` or `load` command (without parentheses), but each of the modules or commands has to be enclosed in parentheses.

When entering a module, as above, Maude gives us information about the rewrites executed to handle such a module. This is the number of rewrites done by Full Maude to evaluate the module being entered. In the same way, every time we enter a command, although in most cases it finally makes a

call to Core Maude, Full Maude needs to perform some additional rewrites. Thus, as we will see below, the number of rewrites given by the system for Full Maude commands includes the reductions due to the evaluation of the command and those due to the execution of the command itself.

We can perform reduction or rewriting using a syntax for commands such as that of Core Maude.

```
Maude> (red in PATH : b ; (c ; d) .)
rewrites: 893 in 30ms cpu (21ms real) (29766 rewrites/second)
reduce in PATH :
  b ;(c ; d)
result Path :
  b ;(c ; d)

Maude> (red length(b ; (c ; d)) .)
rewrites: 474 in 10ms cpu (2ms real) (47400 rewrites/second)
reduce in PATH :
  length(b ;(c ; d))
result NzNat :
  3

Maude> (red a ; (b ; c) .)
rewrites: 587 in 0ms cpu (2ms real) (~ rewrites/second)
reduce in PATH :
  a ;(b ; c)
result [Path] :
  a ;(b ; c)

Maude> (red source(a ; (b ; c)) .)
rewrites: 616 in 0ms cpu (2ms real) (~ rewrites/second)
reduce in PATH :
  source(a ;(b ; c))
result [Node] :
  source(a ;(b ; c))

rewrites: 622 in 0ms cpu (2ms real) (~ rewrites/second)
reduce in PATH :
  target((a ; b); c)
result [Node] :
  target((a ; b); c)

Maude> (red length(a ; (b ; c)) .)
rewrites: 579 in 0ms cpu (2ms real) (~ rewrites/second)
reduce in PATH :
  length(a ;(b ; c))
result [Nat] :
  length(a ;(b ; c))
```

Note the number of rewrites. These figures include, as said above, the rewrites accomplished by Full Maude in the processing of the inputs and

outputs, plus the number of rewrites of the reduction itself. For example, the first two reductions above in Core Maude would produce the following output:

```
Maude> red in PATH : b ; (c ; d) .
reduce in PATH : b ; (c ; d) .
rewrites: 7 in 0ms cpu (23ms real) (~ rews/sec)
result Path: b ; (c ; d)

Maude> red length(b ; (c ; d)) .
reduce in PATH : length(b ; (c ; d)) .
rewrites: 12 in 0ms cpu (0ms real) (~ rews/sec)
result NzNat: 3
```

Tracing, debugging, profiling, and the other facilities available in Core Maude (see Section 24.1) are also available in Full Maude. Since these facilities are provided by Core Maude, the corresponding commands for managing them must be written without parentheses. For example, we can do the following:

```
Maude> set trace on .
Maude> set trace mb off .
Maude> set trace condition off .
Maude> set trace substitution off .
Maude> (red length(b ; c) .)
***** trial #1
ceq length(E:Edge ; P:Path) = length(P:Path) + 1
  if E:Edge ; P:Path : Path .
***** solving condition fragment
E:Edge ; P:Path : Path
***** success for condition fragment
E:Edge ; P:Path : Path
***** success #1
***** equation
ceq length(E:Edge ; P:Path) = length(P:Path) + 1
  if E:Edge ; P:Path : Path .
length(b ; c)
--->
length(c) + 1
***** equation
eq length(E:Edge) = 1 .
length(c)
--->
1
***** equation
(built-in equation for symbol _+_)
1 + 1
--->
2
rewrites: 444 in 0ms cpu (7ms real) (~ rewrites/second)
reduce in PATH :
```

```

length(b ; c)
result NzNat :
  2

```

One should always bear in mind that Full Maude is part of the specification being run. The specification of Full Maude is loaded in the system, and as said above, some of the rewrites taking place are the result of applying equations or rules in these modules. In the case of tracing, the rewrites done by Full Maude are not shown thanks to one of the trace commands available, namely `trace exclude` (see Sections 24.1.1 and 25.6). With such a command we may exclude particular modules from being traced. In particular, the `full-maude.maude` file includes the command `trace exclude FULL-MAUDE`, where `FULL-MAUDE` is the top module of the specification of Full Maude.

19.2 Using Core Maude modules in Full Maude

Full Maude maintains a module database independent from the one used by Core Maude to store the modules entered into it. In fact, this module database is a Maude term stored as part of the state in the `LOOP-MODE` input/output object. Therefore, a module entered into Core Maude can only import modules previously entered into Core Maude. However, Full Maude modules can import modules previously entered either into Full Maude or into Core Maude. Basically, if Full Maude cannot find a module in its own database, it looks into Core Maude's module database to find it.

When metaprogramming, the system behaves differently. In Core Maude, a metamodule (that is, the metarepresentation of a module) can include a module at the object level. In Full Maude, however, metamodules cannot import modules entered into Full Maude and can only import modules entered into Core Maude. Note that Full Maude is implemented using reflection, and that in the end all modules are handled by Core Maude, which is not aware of Full Maude's database.

Notice also that loading a Core Maude module once Full Maude is running will break the read-eval-print loop (see Section 18.1). Therefore, one should enter such modules *before* starting Full Maude. Assuming there is a file `path.maude` containing the Core Maude module `PATH`, we will have the following behavior if we enter it into Full Maude.

```

Maude> load path.maude

Maude> (red in PATH : b ; (c ; d) .)
Warning: no loop state.
Advisory: attempting to reinitialize loop.
Warning: "full-maude.maude", line 13692: bad token init
Warning: "full-maude.maude", line 13692: no parse for term.
Advisory: unable to reinitialize loop.

```

As said above, when the loop gets broken, as in this case, we must select the FULL-MAUDE module and restart the loop. We may now do the following:

```
Maude> select FULL-MAUDE .
Maude> loop init .

Full Maude 2.7 March 10th 2015

Maude> (red in PATH : b ; (c ; d) .)
reduce in PATH :
  b ;(c ; d)
result Path :
  b ;(c ; d)
```

Notice that with a `loop init` command Full Maude is restarted with an empty database. That is, any Full Maude module entered before the reinitialization will have to be reentered again. In this case, `PATH` is a Core Maude module, which is being executed *in* Full Maude. Since it is not in Full Maude's database, Full Maude looks into Core Maude's database and then executes the command in it. This functionality is useful for using any of the predefined modules, but also other modules which are not part of Maude's prelude. For example, for using inside Full Maude the model checker, which although predefined is not part of the `prelude.maude` file, we just need to load the `model-checker.maude` file *before* starting the loop. For example, we can do the following:

```
~/maude-linux/bin$ ./maude.linux model-checker.maude full-maude.maude
\|||||/
--- Welcome to Maude ---
/|||||/
Maude 2.7 built: Mar 3 2014 18:07:27
Copyright 1997-2014 SRI International
Tue Jul 15 22:57:15 2014

Full Maude 2.7 March 10th 2015

Maude> (mod CHECK-RESP is
  protecting MODEL-CHECKER .
  ...
endm)

Maude> (red p(0) |= (<> Qstate) .)
```

See Section 21.8 for a concrete example of the use of Maude's model checker with Full Maude modules.

19.3 Additional module operations in Full Maude

As for Core Maude, in Full Maude we can use the keywords `protecting`, `extending`, and `including` (or `pr`, `ex`, and `inc` in abbreviated form) to define structured specifications, as well as summation, renaming, and instantiation operations on parameterized modules (see Chapter 8). All the predefined modules introduced in Chapter 9, plus the module `META-LEVEL` and its submodules, described in Chapter 14, are also available in Full Maude.²

In addition to the module operations available in Core Maude, Full Maude supports the following extensions:

- *Tuple* and *power expressions* which, given any nonzero natural number, generate parameterized modules specifying *tuples* and *powers* of the corresponding size.

```
TUPLE[⟨NonzeroNaturalNumber⟩]{⟨ViewExpression⟩}
POWER[⟨NonzeroNaturalNumber⟩]{⟨ViewExpression⟩}
```

See Section 19.3.1.

- *Parameterized views*, and the instantiation of parameterized modules with instantiations of views. See Section 19.3.2.
- *Object-oriented modules*, extending all the module operations available in Core Maude to this new type of modules. Thus, in Full Maude we may rename object-oriented modules, with renamings of classes, attributes, and messages, or use object-oriented modules in the summation of modules. Full Maude also supports object-oriented theories, views from object-oriented theories to object-oriented modules, and object-oriented parameterized modules, as well as the instantiation of such object-oriented parameterized modules. We devote Chapter 21 to the study of object-oriented modules.

As in Core Maude, a module or theory importing some combination of modules or theories, given by module expressions, can be seen as a structured module with more or less complex relationships among its component submodules. For execution purposes, however, we typically want to convert this structured module into an equivalent unstructured module, that is, into a “flattened” module without submodules; this flattened module will then be compiled into the Maude rewrite engine. By systematically using the metaprogramming capabilities of Maude, we can both evaluate module expressions into structured module hierarchies and flatten such hierarchies into unstructured modules for execution. All such module operations are defined by equations that operate on the metalevel term representations of modules. This is essentially the idea behind the implementation of Full Maude in Maude.

In Full Maude, the use of module expressions is somewhat more general than in Core Maude. A Full Maude module expression can be used in any

² The predefined module `LOOP-MODE` described in Section 18.1 is not supported in Full Maude.

place where a module name is expected. Thus, as in Core Maude, in Full Maude, module expressions can be used as:

- arguments of a `protecting`, `extending`, or `including` importation,
- the source or target of a view, or
- the parameter of a module, provided the top level is a theory.

Furthermore, in Full Maude, they can also be used, e.g., to express the module in which a `red` or `rew` command will be executed,

```
Maude> (red in BOOL * (op true to T, op false to F) : T or F .)
result Bool :
  T
```

or as argument of any other command requiring a module name,

```
Maude> (show ops LIST{Nat} .)
  op $reverse : List{Nat}List{Nat}-> List{Nat}.
  op $size : List{Nat}Nat -> Nat .
  op append : List{Nat}List{Nat}-> List{Nat}.
  op append : List{Nat}NeList{Nat}-> NeList{Nat}.
  op append : NeList{Nat}List{Nat}-> NeList{Nat}.
  ...
```

Of course, this works with any module, and not only with predefined modules. For example, let us do the same with the instantiation of the `SET-MAX` module presented in Section 8.3.4 (which we assume is in file `set-max.maude`) with the view `IntAsToset` described in Section 8.3.2. Although we can use Core Maude modules in Full Maude, we do not have access to user-defined Core Maude views from Full Maude. Any such view must be entered into Full Maude before it is used in a module instantiation. Note that although Core Maude modules are implicitly entered into Full Maude's database, they are recompiled, and therefore, any view required for recompiling the corresponding module must also be entered. The evaluation of the module expression `SET-MAX{IntAsToset}` requires views `TOSET` and `IntAsToset`.

```
Maude> load set-max.maude
Maude> select FULL-MAUDE .
Maude> loop init .
```

Full Maude 2.7 March 10th 2015

```
Maude> (view TOSET from TRIV to TOSET is
  sort Elt to Elt .
  endv)
Introduced view TOSET
```

```
Maude> (view IntAsToset from TOSET to INT is
  sort Elt to Int .
  endv)
```

Introduced view `IntAsToset`

```
Maude> (red in SET-MAX{IntAsToset} : max((5, 4, 8, 4, 6, 5)) .)
result NzNat :
  8
```

Similarly, after entering the Full Maude version of the `RingToRat` view, we can reduce the same expression we reduced in Section 8.3.4 as follows:

```
Maude> (red in RAT-POLY{Qid} :
  (((2 / 3) ((X ^ 2) (Y ^ 3)))
   ++ ((7 / 5) ((Y ^ 2) (Z ^ 5))))
  (((1 / 7) (U ^ 2)) ++ (1 / 2)) .)
result Poly{RingToRat,Qid} :
  (1/3(X ^ 2)Y ^ 3)
  ++ (1/5(U ^ 2)(Y ^ 2)Z ^ 5)
  ++ (2/21(U ^ 2)(X ^ 2)Y ^ 3)
  ++ (7/10(Y ^ 2)Z ^ 5)
```

As we will see below, a module expression can also be used as the parameter of a view, provided the top level is a theory.

19.3.1 The tuple and power module expressions

The evaluation of an n -tuple module expression consists in the generation of a parameterized functional module with the number of `TRIV` parameters specified by the argument n . A sort for tuples of such size, and the corresponding constructor `(_, ..., _)` and selector operators `p1_`, ..., `pn_`, are also defined. For example, the module expression `TUPLE[2]` automatically generates as result the following module (notice the backquotes in the declaration of the tuple constructor).

```
(fmod TUPLE[2]{C1 :: TRIV, C2 :: TRIV} is
  sorts Tuple{C1, C2} .
  op ‘(‘, ‘) : C1$Elt C2$Elt -> Tuple{C1, C2} [ctor].
  op p1_ : Tuple{C1, C2} -> C1$Elt .
  op p2_ : Tuple{C1, C2} -> C2$Elt .
  var E1 : C1$Elt .
  var E2 : C2$Elt .
  eq p1(E1, E2) = E1 .
  eq p2(E1, E2) = E2 .
endfm)
```

In the `Clear` [42] and `OBJ` [174] family of languages, module operations take theories, modules, and views, and return new theories and modules (see Chapter 8); on the other hand, the `TUPLE[_]` operation takes a nonzero natural number n and returns a parameterized `TUPLE[n]` module; this is impossible to achieve with the `Clear/OBJ` repertoire of module operations. Even though

an n -tuple module expression is in principle of a completely different nature from the usual Clear/OBJ module operations, the way Full Maude handles it is the same as the way it handles any other module expression. Its evaluation produces a new unit, a parameterized functional module in this case, with the module expression as its name.

Suppose that we want to specify a library in which we have the information on the books in a record structure with the title, author, year of publication, publisher, and number of copies available. We may use a specification beginning as follows:

```
(fmod LIBRARY is
  pr TUPLE[5]{String, String, Nat, String, Nat}
    * (op p1_ to title,
       op p2_ to author,
       op p3_ to year,
       op p4_ to publisher,
       op p5_ to copies) .
  ---- ...
endfm)
```

The particular case of a tuple in which all component sorts are equal is provided by the n -power module expression. For example, the module expression `POWER[5]` automatically generates as result the following module:

```
(fmod POWER[5]{X :: TRIV} is
  protecting TUPLE[5]{X, X, X, X, X}
    * (sort Tuple{X, X, X, X, X} to Power{X}) .
endfm)
```

We can use the power module expression in any place where a module name is expected, like in a reduction

```
Maude> (red in POWER[10]{Nat} : p5 (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) .)
result NzNat :
  4
```

or in an importation to build other modules:

```
(fmod PERSON-RECORD is
  pr POWER[3]{String}
    * (sort Tuple{String, String, String} to PersonRecord,
       op p1_ to firstname,
       op p2_ to lastname,
       op p3_ to address) .
  op fullName : PersonRecord -> String .
  vars F L A : String .
  eq fullName((F, L, A)) = F + " " + L .
endfm)
```

```
Maude> (red fullName(("John", "Smith", "Maude Ave")) .)
result String :
  "John Smith"
```

19.3.2 Parameterized views

Suppose we have defined modules `LIST{X :: TRIV}` and `SET{X :: TRIV}`, specifying, respectively, lists and sets, and suppose that we need, e.g., the data type of lists of sets of natural numbers. Typically, we would first instantiate the module `SET` with a view, say `Nat`, from `TRIV` to the module `NAT` mapping the sort `Elt` to the sort `Nat`, thus getting the module `SET{Nat}` of sets of natural numbers. Then, we would instantiate the module specifying lists with a view, say `NatSet`, from `TRIV` to `SET{Nat}`, obtaining the module `LIST{NatSet}`. But, what if we need now the data type of lists of sets of Booleans? Should we repeat the whole process again? One possibility is to define a combined module `SET-LIST{X :: TRIV}`. But what if we later want stacks of sets instead of lists of sets?

We can greatly improve the reusability of specifications by using *parameterized views*. Let us consider the following parameterized view `Set` from `TRIV` to `SET`, which maps the sort `Elt` to the sort `Set{X}`.

```
(view Set{X :: TRIV} from TRIV to SET{X} is
  sort Elt to Set{X} .
endv)
```

With this kind of views we can keep the parameter part of the target module still as a parameter. We can now have lists of sets, stacks of sets, and so on, for any instance of `TRIV`, by instantiating the appropriate parameterized module with the appropriate view. For example, given the view `Nat` above, we can have the module `LIST{Set{Nat}}` of lists of sets of natural numbers, or lists of sets of Booleans with `LIST{Set{Bool}}`, given a view `Bool` from `TRIV` to the predefined module `BOOL`. Similarly, we can have `STACK{Set{Nat}}` or `STACK{Set{Bool}}`.

We can also link the parameter of a module like `LIST{Set{X}}` to the parameter of the module in which it is being included. That is, we can, for example, declare a module of the form

```
(fmod GENERIC-SET-LIST{X :: TRIV} is
  protecting LIST{Set{X}} .
endfm)
```

Then, instantiating the parameterized module `GENERIC-SET-LIST` with a view `V` from `TRIV` to another module or theory results in a module with name `GENERIC-SET-LIST{V}`, which includes the module `LIST{Set{V}}`. Note that even with parameterized views we still follow conventions for module interfaces and for sort names (see Section 8.3). The only difference is that now, instead of having simple view names, we must consider names of views which are parameterized.

The use of parameterized views in the instantiation of parameterized modules allows very reusable specifications. For example, a very simple way of specifying (finite) partial functions is to see a partial function as a set of

input-result pairs. Of course, for such a set to represent a function there cannot be two pairs associating different results with the same input value. We show later in this section (in the module `PFUN` below) how this property can be specified by means of appropriate membership axioms. Note, however, that since membership axioms cannot be given on associative operators over sorts (see Section 24.2.8), we cannot use either the specification of sets described in Section 8.3.3 or the predefined module in Section 9.12.2. Let us consider instead the following module:³

```
(fmod SET-KIND{X} :: TRIV} is
  sorts NeKSet{X} KSet{X} .
  subsort X$Elt < NeKSet{X} < KSet{X} .
  op empty : -> KSet{X} [ctor] .
  op _',_ : KSet{X} KSet{X} ~> KSet{X} [ctor assoc comm id: empty] .
  mb NS:NeKSet'{X'}, NS':NeKSet'{X'} : NeKSet{X} .

  var E : X$Elt .

  *** idempotency
  eq E, E = E .
endfm)
```

Here the operator `_,_` is declared at the kind level (notice the different form of the arrow in its declaration) together with a membership axiom, that is logically equivalent to the declaration

```
op _',_ : NeKSet{X} NeKSet{X} -> NeKSet{X} .
```

at the sort level.

We can then specify sets of pairs by instantiating this `SET-KIND` module with a parameterized view from `TRIV` to the parameterized module `TUPLE[2]{X, Y}` defining pairs of elements. The appropriate parameterized view can be defined as follows:

```
(view Tuple{X} :: TRIV, Y :: TRIV} from TRIV to TUPLE[2]{X, Y} is
  sort Elt to Tuple{X, Y} .
endv)
```

A partial function can be lifted to a total function by adding a special value to its codomain, to be used as the result for the input elements for which the function is not defined. For this we make good use of the parameterized module `MAYBE`, introduced in Section 8.3.3, which adds a supersort and a new element `maybe` to this supersort; in this application, the constant `maybe` is renamed to `undefined`.

³ Note the use of the equivalent single-identifier-form for on-the-fly declarations of variables; as we will see in Section 19.6, this is one of the syntactic restrictions of Full Maude.

We are now ready to give the specification of partial functions. The sets representing the domain and codomain of the function are given by TRIV parameters, and then the set of tuples is provided by the imported module expression SET-KIND{Tuple{X, Y}} with sorts KSet{Tuple{X, Y}} and NeKSet{Tuple{X, Y}}. We define operations dom and im returning, respectively, the domain and image of a set of pairs. The dom operation will be used for checking whether there is already a pair in a set of pairs with a given input value. With these declarations we can define the sort PFun{X, Y} as a subsort of KSet{Tuple{X, Y}}, by adding the appropriate membership axioms specifying those sets that satisfy the required property. Finally, we define operators [_] and [_->_] to evaluate a function at a particular element, and to add or redefine an input-result pair, respectively. We use the Core Maude predefined module SET (see Section 9.12.2) for representing the sets of elements in the domain and image of a partial function.

```
(fmod PFun{X :: TRIV, Y :: TRIV} is
  pr SET-KIND{Tuple{X, Y}} .
  pr SET{X} + SET{Y} .
  pr MAYBE{Y} * (op maybe to undefined) .

  sort PFun{X, Y} .
  subsorts Tuple{X, Y} < PFun{X, Y} < KSet{Tuple{X, Y}} .

  vars A D : X$Elt .
  vars B C : Y$Elt .
  var F : PFun{X, Y} .
  var S : KSet{Tuple{X, Y}} .

  op dom : KSet{Tuple{X, Y}} -> Set{X} .          *** domain
  eq dom(empty) = empty .
  eq dom((A, B), S) = A, dom(S) .
  op im : KSet{Tuple{X, Y}} -> Set{Y} .          *** image
  eq im(empty) = empty .
  eq im((A, B), S) = B, im(S) .

  op empty : -> PFun{X, Y} [ctor] .
  cmb (A, B), (D, C), F : PFun{X, Y}
    if (D, C), F : PFun{X, Y} /\ not(A in dom((D, C), F)) .

  op _'[_'] : PFun{X, Y} X$Elt -> Maybe{Y} .
  op _'[_->_'] : PFun{X, Y} X$Elt Y$Elt -> PFun{X, Y} .
  ceq ((A, B), F)[ A ] = B if ((A, B), F) : PFun{X, Y} .
  eq F [ A ] = undefined [owise] .
  ceq ((A, B), F)[ A -> C ] = (A, C), F
    if ((A, B), F) : PFun{X, Y} .
  eq F [ A -> C ] = (A, C), F [owise] .
endfm)
```

Now, we can instantiate the PFUN module with, for example, the view `Nat`, in order to get the finite partial functions from natural numbers to natural numbers by means of the module expression `PFUN{Nat, Nat}`.

19.3.3 Example: leftist trees

Among the many different data structures implementing priority queues (Section 10.3) the most efficient are *heaps*, which can be defined as binary trees (Section 10.7) satisfying the additional constraints that the value in each node is (in the case of *min heaps*) smaller than (or equal to) the values in its children and, moreover, that the tree is complete. If we forget the latter requirement and instead assign to each node a *rank* (also known as *minimum depth*) defined as the length of the rightmost path to a leaf, and require that the root of each left child has a rank bigger than or equal to the rank of the corresponding right child (that can be empty), we get the trees known as *leftist trees*. These trees implement priority queues with the same efficiency as standard heaps, and have the additional property that two leftist trees can be merged to obtain a leftist tree containing all their elements in logarithmic time with respect to the total number of nodes.

Remember that priority queues, and thus also leftist trees, are parameterized with respect to the theory TOSET for totally ordered sets, specifying together both the strict `_<_` and the non-strict `_<=_` order relations (see Sections 8.3.1 and 10.3).

As we did for search trees in Section 10.9, the sort of leftist trees can also be defined as a subsort of binary trees by means of appropriate membership assertions. Moreover, in order to compare quickly the ranks of two nodes, we need to save in each node its rank in the same way that we saved the depth in each node of the AVL trees defined in Section 10.10.

Since we are importing binary trees, which are parameterized with respect to the theory TRIV, we first define in the module TREE-NODE the construction of pairs formed by a natural number (representing the rank) and an element (identified with its priority).

```
(fmod TREE-NODE{T :: TOSET} is
  protecting NAT .
  sort Node{T} .
  op n : Nat T$Elt -> Node{T} [ctor] .
endfm)
```

Now we instantiate the module BIN-TREE of binary trees with the *parameterized view* `Node{T}`. The view is parameterized because it still keeps as a parameter, as expected, the sort of the elements on top of which we are building the leftist trees.

```
(view Node{T :: TOSET} from TRIV to TREE-NODE{T} is
  sort Elt to Node{T} .
endv)
```

```

(fmod LEFTIST-TREES{T :: TOSET} is
  protecting BIN-TREE{Node{T}} .
  sorts NeLTree{T} LTree{T} .
  subsorts NeLTree{T} < LTree{T} < BinTree{Node{T}} .
  subsorts NeLTree{T} < NeBinTree{Node{T}} .

  op rank : BinTree{Node{T}} -> Nat .
  op rankL : LTree{T} -> Nat .
  op findMin : NeLTree{T} -> T$Elt .

  vars NeTL NeTR : NeLTree{T} .
  vars M N N1 N2 : Nat .
  vars T TL TR TL1 TR1 TL2 TR2 : LTree{T} .
  vars X X1 X2 : T$Elt .

```

The following memberships, defining sorts `LTree{T}` for leftist trees and `NeLTree{T}` for non-empty leftist trees, faithfully represent in Maude the informal definition given above of this data structure. The operations `rank` and `findMin` calculate, respectively, the rank and the minimum element in the root of the tree.

```

mb empty : LTree{T} .
mb empty [n(1, X)] empty : NeLTree{T} .
cmb NeTL [n(1, X)] empty : NeLTree{T} if X < findMin(NeTL) .
cmb NeTL [n(N, X)] NeTR : NeLTree{T}
  if rank(NeTL) >= rank(NeTR) /\ X < findMin(NeTL) /\
     X < findMin(NeTR) /\ N = 1 + rank(NeTR) .

eq rank(empty) = 0 .
eq rank(TL [n(N, X)] TR) = 1 + rank(TR) .
eq rankL(empty) = 0 .
eq rankL(TL [n(N,X)] TR) = N .
eq findMin(TL [n(N,X)] TR) = X .

```

The most important operation on leftist trees is `merge`, because both `insert` and `deleteMin` are easily defined from it. The operation `merge` is specified by structural induction on its arguments with the help, in the recursive case (when both arguments are non-empty), of an auxiliary operation `make` that takes care of putting the tree with bigger rank at its root to the left of the tree being built.

```

op merge : LTree{T} LTree{T} -> LTree{T} .
op insert : T$Elt LTree{T} -> NeLTree{T} .
op deleteMin : NeLTree{T} -> LTree{T} .
op make : T$Elt LTree{T} LTree{T} -> LTree{T} .

eq merge(empty, T) = T .
eq merge(T, empty) = T .
eq merge(TL1 [n(N1, X1)] TR1, TL2 [n(N2, X2)] TR2) =

```

```

      if X1 < X2 then make(X1, TL1, merge(TR1, TL2 [n(N2,X2)] TR2))
        else make(X2, TL2, merge(TL1 [n(N1,X1)] TR1,TR2))
      fi .

eq make(X, TL, TR) = if rankL(TL) >= rankL(TR)
  then TL [n(rankL(TR) + 1,X)] TR
  else TR [n(rankL(TL) + 1,X)] TL fi .

eq insert(X,T) = merge(empty [n(1,X)] empty, T) .
eq deleteMin(TL [n(N,X)] TR) = merge(TL,TR) .
endfm)

```

We show the result of some reductions after instantiating the parameterized module LEFTIST-TREES with the following view:

```

(view IntAsToset from TOSET to INT is
  sort Elt to Int .
endv)

(fmod LEFTIST-TREES-TEST is
  protecting LEFTIST-TREES{IntAsToset} .
endfm)

Maude> (red in LEFTIST-TREES-TEST :
  insert(5, insert(4, empty)) .)
result NeLTree{IntAsToset} : (empty [n(1, 5)] empty) [n(1, 4)] empty

Maude> (red in LEFTIST-TREES-TEST :
  findMin(
    insert(5, deleteMin(insert(3, insert(-10, empty)))) .)
result NzNat : 3

Maude> (reduce in LEFTIST-TREES-TEST :
  deleteMin(insert(900, insert(-901, insert(902, insert(-903,
    insert(904, insert(-905, insert(906, insert(-907,
    insert(908, insert(-909, insert(910, insert(-910,
    insert(911, insert(-912, insert(913, insert(-914,
    empty)))))))))))))) .)
result NeLTree{IntAsToset} :
(((empty [n(1, 908)] empty)
 [n(2, -907)]
 (empty [n(1, 906)] empty))
 [n(3, -909)]
 (((empty [n(1, 911)] empty)
 [n(2, -901)]
 ((empty [n(1, 904)] empty)
 [n(1, 900)]
 empty))
 [n(2, -905)]
 ((empty [n(1, 913)] empty)

```

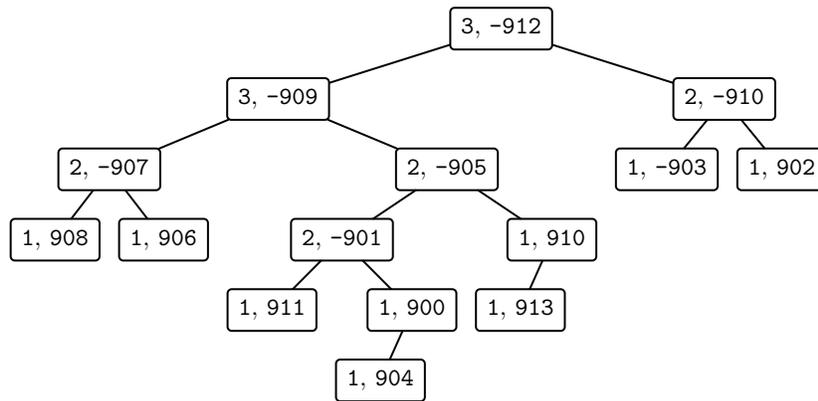


Fig. 19.1. An example of a leftist tree

```

[n(1, 910)]
  empty)))
[n(3, -912)]
  ((empty [n(1, -903)] empty)
   [n(2, -910)]
   (empty [n(1, 902)] empty))

```

The leftist tree obtained as result in the last reduction is displayed in Figure 19.1.

We can also instantiate the parameterized module `LEFTIST-TREES` with elements that are pairs, as we did at the end of Section 10.3.

```

(view IntStringAsToset from TOSET to
  PRIORITY-PAIR{IntAsToset, String} is
  sort Elt to Priority-Pair{IntAsToset, String} .
endv)

(fmod LEFTIST-TREES-TEST-PAIR is
  protecting LEFTIST-TREES{IntStringAsToset} .
endfm)

Maude> (red in LEFTIST-TREES-TEST-PAIR :
  findMin(insert(< 4, "d" >, insert(< 8, "h" >,
    insert(< 1, "a" >, empty)))) .)
result Priority-Pair{IntAsToset, String} :
  < 1, "a" >

Maude> (red in LEFTIST-TREES-TEST-PAIR :
  findMin(insert(< 5, "e" >, deleteMin(insert(< 3, "c" >,
    insert(< -10, "zzz" >, empty)))))) .)
result Priority-Pair{IntAsToset, String} :
  < 3, "c" >

```

19.4 Moving up and down between reflection levels

The functions provided by Core Maude for moving up and down reflection levels (see Section 14.5.1) are not very useful in Full Maude. Although they are available as part of the module `META-LEVEL`, they take as one of their arguments the name of a module entered into Core Maude. Since the databases of modules are different, these functions work in Full Maude only for Core Maude predefined modules. Full Maude provides its own functions `upTerm` and `upModule` for moving, respectively, terms and modules up reflection levels, and an additional `down` command which allows moving terms down reflection levels.

Let us consider the following module for the examples in the coming sections.

```
(fmod NAT-PLUS is
  sort Nat .
  op 0 : -> Nat .
  op s_ : Nat -> Nat .
  op _+_ : Nat Nat -> Nat [assoc comm id: 0] .
  vars N M : Nat .
  eq s N + s M = s s (N + M) .
endfm)
```

In what follows we will use the notation \bar{t} and \overline{M} to refer to the metarepresentations of a term t and a module M , respectively. For example, we will write the metarepresentation of $0 + s\ 0$ as $\overline{0 + s\ 0}$ instead of

```
'_+_['0.Nat, 's_['0.Nat]]
```

19.4.1 Up

As in Core Maude, in Full Maude we can use the `upModule` and `upTerm` functions to avoid the cumbersome task of explicitly writing, respectively, the metarepresentation of a module or the metarepresentation of a term in a given module. The Full Maude `upModule` function takes as a single argument the name of a module and returns its metarepresentation;⁴ `upTerm` takes two arguments, the name of a module and a term in such a module, and returns the corresponding metarepresentation of the term.

Therefore, by evaluating in any module importing the module `META-LEVEL` the `upModule` function with the name of any module in the system—either in Core Maude or in Full Maude—as argument, we obtain the metarepresentation of such a module. For example, assuming that the previous module `NAT-PLUS` has been entered into Full Maude, and therefore it is in its database, we can get its metarepresentation, which we denoted by $\overline{\text{NAT-PLUS}}$, as follows:

⁴ The Core Maude `upModule` function takes as second argument a Boolean value (see Section 14.5.1).

```
Maude> (red in META-LEVEL : upModule(NAT-PLUS) .)
result FModule :
  fmod 'NAT-PLUS is
  nil
  sorts 'Bool ; 'Nat .
  none
  op '0 : nil -> 'Nat [none] .
  op '+'_ : 'Nat 'Nat -> 'Nat [assoc comm id('0.Nat)] .
  ...
```

We can use the metarepresentation obtained in this way in any place where a term of sort `Module` is expected. For example, we can apply the function `getOps` in `META-LEVEL` (see Section 14.3) to `upModule(NAT-PLUS)` as follows:

```
Maude> (red in META-LEVEL : getOps(upModule(NAT-PLUS)) .)
result OpDeclSet :
  op '0 : nil -> 'Nat [none] .
  op '+'_ : 'Nat 'Nat -> 'Nat [assoc comm id('0.Nat)] .
  op '_=/_ : 'Universal 'Universal -> 'Bool
  [poly(1 2)
   prec(51)
   special(
     id-hook('EqualitySymbol,nil)
     term-hook('equalTerm,'false.Bool)
     term-hook('notEqualTerm,'true.Bool))] .
  ...
```

Similarly, we can use it with descent functions as discussed in Section 14.5.

```
Maude> (red in META-LEVEL :
  metaReduce(upModule(NAT-PLUS), '+'_['0.Nat, 's_['0.Nat]]) .)
result ResultPair :
  {'s_['0.Nat], 'Nat}
```

But, instead of explicitly writing the metarepresentation $\overline{0 + s\ 0}$ in the above reduction we can also make good use of the `upTerm` function, that allows us to get the metarepresentation of a term in a given module.

```
Maude> (red in META-LEVEL :
  metaReduce(upModule(NAT-PLUS), upTerm(NAT-PLUS, 0 + s 0)) .)
result ResultPair :
  {'s_['0.Nat], 'Nat}
```

As another example, to obtain the metarepresentation of the term `s 0` in the module `NAT-PLUS` above, which we denoted by $\overline{s\ 0}$, we can write

```
Maude> (red in META-LEVEL : upTerm(NAT-PLUS, s 0) .)
result GroundTerm :
  's_['0.Nat]
```

The module name is the first argument of the `upTerm` function, with the term of that module to be metarepresented as the second argument. Since the same term can be parsed in different ways in different modules, and therefore can have different metarepresentations depending on the module in which it is considered, the module to which the term belongs has to be used to obtain the correct metarepresentation. Note also that the above reduction only makes sense at the metalevel, that is, either in the `META-LEVEL` module itself or in a module importing it.

19.4.2 Down

The result of a metalevel computation, that may use several levels of reflection, can be a term or a module metarepresented one or more times, which may be hard to read. Therefore, to display the output in a more readable form we can use the `down` command, which is in a sense inverse to `upTerm`, since it gives us back the original term from its metarepresentation. Notice that `down` is not a function, but a command instead, because it is more general, taking other commands as arguments, as we are going to explain.

The `down` command takes two arguments. The first argument is the name of the module to which the term to be returned belongs. The metarepresentation of the desired output term should be the result of the command given as second argument. The syntax of the `down` command is as follows:

```
down <ModuleExpression> : <Command>
```

Thus, we can give the following command.

```
Maude> (down NAT-PLUS :
      red in META-LEVEL :
        getTerm(
          metaReduce(upModule(NAT-PLUS),
                    upTerm(NAT-PLUS, 0 + s 0))) .)
result Nat :
  s 0
```

Notice that this is equivalent to what we may write using the overline notation as:

```
Maude> red getTerm(metaReduce( $\overline{\text{NAT-PLUS}}$ ,  $\overline{0 + s 0}$ )) .
result Term:  $\overline{s 0}$ 
```

The use of `upTerm` and `down` can be iterated with as many levels of reflection as we wish. For example, we can give the command

```
Maude> (red in META-LEVEL :
      getTerm(
        metaReduce(upModule(META-LEVEL),
                  upTerm(META-LEVEL,
```

```

getTerm(
  metaReduce(upModule(NAT-PLUS),
    upTerm(NAT-PLUS, 0 + s 0)))) .)
result GroundTerm :
  '[_['_[']'s_.Sort, ''0.Nat.Constant]

```

This is equivalent to what we would have written using the overline notation as

```

Maude> red getTerm(metaReduce(META-LEVEL,
  metaReduce(NAT-PLUS, 0 + s 0))) .
result Term: s 0

```

We can write expressions involving simultaneously `down`, `upModule`, and `upTerm`:

```

Maude> (down NAT-PLUS :
  down META-LEVEL :
  red in META-LEVEL :
  getTerm(
    metaReduce(upModule(META-LEVEL),
      upTerm(META-LEVEL,
        getTerm(
          metaReduce(upModule(NAT-PLUS),
            upTerm(NAT-PLUS, 0 + s 0)))))) .)
result Nat :
  s 0

```

The metalevel function `downTerm` can also be used, but it is a Core Maude function, and therefore can only be used on Core Maude modules.

```

Maude> (down NAT-PLUS :
  red in META-LEVEL :
  downTerm(
    getTerm(
      metaReduce(upModule(META-LEVEL),
        upTerm(META-LEVEL,
          getTerm(
            metaReduce(upModule(NAT-PLUS),
              upTerm(NAT-PLUS, 0 + s 0)))))),
    'T:Term) .)
result Nat :
  s 0

```

19.5 *Ax*-coherence completion

As pointed out in Section 15.9.1, the equations used for variant generation in an admissible equational theory must be confluent, terminating,

sort-decreasing, and *explicitly* Ax -coherent. The confluence, termination, and sort-decreasingness of equational Maude specifications are the typical executability requirements for equational specifications, and can be checked using Maude's Church-Rosser Checker (CRC) [126, 128] and Termination Checker (MTT) [118].

Regarding the Ax -coherence requirement, the situation is different for different purposes. Ax -coherence is not currently required for rewrite theories written in Maude because specifications are implicitly completed for Ax -coherence within Maude (see Section 4.8). It is not required either for tools like the CRC or the Coherence Checker (ChC), or for narrowing (see Chapter 20) which also automatically complete the specifications provided. However, rewrite theories are assumed to be explicitly Ax -coherent for variant generation and variant-based unification (Section 15.10), the reason being that there is a different treatment of extension variables in rewriting and narrowing (see [373] for more details). Indeed, we require coherence rather than ground coherence, since the latter is weaker and sufficient for rewriting but insufficient for narrowing or critical pair analysis.

For theories Ax that are combinations of associativity, commutativity, and identity axioms, we can make any specification explicitly Ax -coherent by using a procedure which adds Ax -extensions and always terminates (see [297], and Section 4.8 for a more informal explanation). The procedure followed to automatically complete for Ax -coherence for rewriting and for narrowing, or in the CRC or ChC tools is carried on differently. Whilst for rewriting the completion is performed by (Core) Maude, in the other cases the completion is performed by Maude code provided by Full Maude.

Full Maude provides the following `axCohComplete` operation in its module `AX-COHERENCE-COMPLETION`.

```
op axCohComplete : Module -> Module .
```

This operation takes (the metarepresentation of) a module which defines an order-sorted specification (i.e., no memberships are allowed) and returns (the metarepresentation of) another module whose equations and rules are modified to complete them modulo associativity and commutativity (AC), associativity, commutativity, and identity (ACU), associativity and identity (AU), associativity, commutativity, and left identity ($ACUl$), associativity, commutativity, and right identity ($ACUr$), and associativity (A).

More specifically, for each operator $f : S \ S \rightarrow S$ and equation/rule $f(t_1, \dots, t_n) \rightarrow r$ if C in the module,

if f is *AC* add $f(t_1, \dots, t_n, x:[S]) \rightarrow f(r, x:[S])$ if C
 if f is *ACU* replace by $f(t_1, \dots, t_n, x:[S]) \rightarrow f(r, x:[S])$ if C
 if f is *AU* replace by $f(x:[S], t_1, \dots, t_n, y:[S]) \rightarrow f(x:[S], r, y:[S])$ if C
 if f is *ACU_l* replace by $f(x:[S], t_1, \dots, t_n, y:[S]) \rightarrow f(x:[S], r, y:[S])$ if C
 add $f(x:[S], t_1, \dots, t_n) \rightarrow f(x:[S], r)$ if C
 if f is *ACU_r* replace by $f(x:[S], t_1, \dots, t_n, y:[S]) \rightarrow f(x:[S], r, y:[S])$ if C
 add $f(t_1, \dots, t_n, y:[S]) \rightarrow f(r, y:[S])$ if C
 if f is *A* add $f(x:[S], t_1, \dots, t_n, y:[S]) \rightarrow f(x:[S], r, y:[S])$ if C
 add $f(x:[S], t_1, \dots, t_n) \rightarrow f(x:[S], r)$ if C
 add $f(t_1, \dots, t_n, y:[S]) \rightarrow f(r, y:[S])$ if C

Ax-completion is available in Full Maude through the following command:

```
(ax coherence completion [ ModuleExpression ] .)
```

where *ModuleExpression* is any module expression. As usual, if no module expression is given the default current module is completed.

For example, let us consider the following non-coherent version of the equational theory for exclusive or (see Section 15.9).

```

mod EXCLUSIVE-OR-NOT-COHERENT is
  sorts Nat NatSet .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .

  subsort Nat < NatSet .
  op mt : -> NatSet .
  op *_ : NatSet NatSet -> NatSet [ctor assoc comm] .

  vars X Y Z : [NatSet] .
  eq [idem] : X * X = mt [variant] .
  eq [id] : X * mt = X [variant] .
endm

```

The module EXCLUSIVE-OR-NOT-COHERENT is not *AC*-coherent. The command

```
Maude> (ax coherence completion EXCLUSIVE-OR-NOT-COHERENT .)
```

shows the completed version of the EXCLUSIVE-OR-NOT-COHERENT module. Specifically, the following equation is *added*:

```

eq [idem] : X:[NatSet] * X:[NatSet] * X@@:[NatSet]
  = mt * X@@:[NatSet] [variant] .

```

Notice the new variable $X@@:[\text{NatSet}]$. The resulting module is equivalent to the module EXCLUSIVE-OR in Section 15.9.

As another example, the *Ax*-completion of the VENDING-MACHINE module in Section 6.1 results in a module with the same definitions but with the change rule *replaced* by the rule

```
rl [change] : q q q q X@@@:[Marking] => $ X@@@:[Marking] .
```

In general, of course, a module can have operators with any combination of associativity, commutativity, and identity, with associated equations and rules. In that case the `ax coherence completion` command will add or replace the corresponding completion equations and rules.

The narrowing command (see Section 20.3) and the CRC and ChC tools (see Sections 23.1.3 and 23.1.4) use the metalevel operation `axCohComplete` to complete the modules before using them. As any other function in Full Maude, it is available for its direct use in other tools or commands. We can illustrate the use of this metalevel function with the coherence completion of the `EXCLUSIVE-OR-NOT-COHERENT` module:

```
Maude> reduce in FULL-MAUDE :
      axCohComplete(upModule('EXCLUSIVE-OR-NOT-COHERENT, false)) .

result SModule :
  mod 'EXCLUSIVE-OR-NOT-COHERENT is
    including 'BOOL .
    sorts 'Nat ; 'NatSet .
    subsort 'Nat < 'NatSet .
    op '0 : nil -> 'Nat [ctor] .
    op 'mt : nil -> 'NatSet [ctor] .
    op '*_ : 'NatSet 'NatSet -> 'NatSet [assoc comm ctor] .
    op 's : 'Nat -> 'Nat [ctor] .
    none
    eq '*_['X:'[NatSet'], 'mt.NatSet]
      = 'X:'[NatSet'] [label('id) variant] .
    eq '*_['X:'[NatSet'], 'X:'[NatSet']]
      = 'mt.NatSet [label('idem) variant] .
    eq '*_['X:'[NatSet'], 'X:'[NatSet'], 'X@@@:'[NatSet']]
      = '*_['mt.NatSet, 'X@@@:'[NatSet']] [label('idem) variant] .
    none
  endm
```

19.6 Differences between Full Maude and Core Maude

Apart from those features available in Full Maude that are not supported in Core Maude (discussed above in Sections 19.3-19.5 and later in Chapters 20 and 21), we find a number of differences between Full Maude and Core Maude. There are some obvious ones, like the fact that any input enclosed in parentheses is handled by Full Maude. Thus, Full Maude modules, theories, views, and commands can refer to modules, theories, and views in both Core Maude and Full Maude's databases. There are also differences in pretty-printing, tracing, debugging, etc. Moreover, there are also other differences that impose certain limitations on the specifications themselves:

1. Operator and message names have to be given in their equivalent *single-identifier form* when they are declared (see below), but they can later be written in the usual way in statements and in terms for evaluation.
2. Sort names used in term qualifications, membership assertions, and on-the-fly declarations of variables have to be in their equivalent *single-identifier form*.
3. The `continue`, `show component`, `show path`, and `show search graph` commands are not supported in Full Maude.
4. Full Maude does not support external objects.

In the rest of the section we explain the first two restrictions in some detail and give some hints on how to avoid them.

An operator name has to be given as a single identifier; multi-identifier operators have to be declared in their single-identifier form, that is, each identifier in a multi-identifier name has to be preceded by a backquote. For example, to define an operator with name `_less than or equal_`, we have to declare it in its single identifier form `_less'than'or'equal_`. Except for having to use the single-identifier form in the operator name, the declaration of operators is exactly as in Core Maude. For example, the declaration of this operator on sort, say, `Int` is as follows.

```
op _less'than'or'equal_ : Int Int -> Bool .
```

Remember that not only blank spaces, but also the special characters `{`, `}`, `(`, `)`, `[`, `]` and `,` break the identifiers. Therefore, to declare in Full Maude an operator such as `{_}` taking an element of sort, say, `Int` and with value sort `Set`, we should add appropriate backquotes, as follows:

```
op '{_' : Int -> Set .
```

As in Core Maude, several operators with the same arity and coarity can be defined in the same declaration using the keyword `ops`, but, again, each operator name has to be given in its single-identifier form. We could have for example the following declaration.

```
ops _'{_' _',_' : Foo Bar -> Baz .
```

Since each operator name is a single identifier, parentheses are not needed to indicate the boundaries between the syntactic forms of the different operators.

As for operator names, message names can be mixfix, but they have to be declared in single-identifier form. Thus, to define a message `credit` in an object-oriented module (see Chapter 21) with syntax, say, `(_)credit_` the declaration has to be given as follows.

```
msg '(_)'credit_ : Oid Nat -> Msg .
```

And the same applies to declarations of multiple message names:

```
msgs '(_)'credit_ '(_)'debit_ : Oid Nat -> Msg .
```

The second problem mentioned at the beginning of this section affects the qualification of terms by sort names, on-the-fly declarations of variables, and membership assertions. In these three situations, the user must use the names of parameterized sorts, not as he or she has defined them, but in their equivalent single-identifier form. Thus, if we have, for example, a sort `List{Nat}` and a constant `nil` in it, if necessary, it should be qualified as `(nil).List' {Nat'}`. A variable `L` of sort `List{Nat}` being declared on the fly should be written `L>List' {Nat'}`. Similarly, to check whether a term `T` has sort `List{Nat}` we have to write `T : List' {Nat'}` or `T :: List' {Nat'}`, depending on the kind of sort check we wish to perform.

19.7 Adding new features to Full Maude

Full Maude and its execution environment are implemented using the reflective capabilities of Maude, which yield very good flexibility, maintainability, and extensibility. Several interesting extensions have already been done, either by adding new commands, like in the works by Durán, Escobar, and Lucas [116, 114, 115], or by adding new types of statements or declarations, like in Real-Time Maude [288, 291] or MSOS [55].

We illustrate in the following sections the addition of new commands and new module expressions to Full Maude. In particular, in Section 19.7.1 we illustrate such a process by adding a new command to provide unification modulo the commutativity of certain operators, making it available inside the Full Maude programming environment like any other of its commands. In Section 19.7.2, we extend Full Maude with a new module expression which makes available the transformation presented in Section 16.3 making a system specification deadlock free. As we will see, the addition of the new command and module expression that we present here mirror the Full Maude declarations for the currently available commands and module expressions, and in fact some of the infrastructure available in Full Maude is used, thus greatly simplifying the implementation task.

19.7.1 A `unifyc` unification command

The unification algorithm presented in Section 16.1 was implemented as a meta-level function `metaUnify`, which takes a module and a set of commutative equations to be unified. To simplify the example, we assume that we always unify on the selected module—an alternative command taking a module expression, as for other commands in Full Maude, could also be given; anyway, we can always use the `select` command (see Section 25.13) to change modules. Then, the syntax for our new command is the following:

$$\text{unifyc } \langle pattern \rangle = \langle pattern \rangle (\wedge \langle pattern \rangle = \langle pattern \rangle)^* .$$

That is, we try to unify a set of equations. The result of this command is a complete set of unifiers for the given unification problem.

The specification/implementation of the behavior of commands like `reduce` and `rewrite` performs some parsing of the arguments and pretty printing of the outputs; but the key functionality is provided by the corresponding descent function, in these cases by `metaReduce` and `metaRewrite`, respectively. Given that we already have a function `metaUnify`, which was defined in Section 16.1, we follow a pattern very similar to that for these previous commands.

As we saw in Section 18.1, in the current version of Maude, input/output is accomplished by the predefined `LOOP-MODE` module, which provides a generic read-eval-print loop. In the case of Full Maude, the persistent state of the loop is given by a single object of class `DatabaseClass` which maintains the database of the system. Objects of this class have an attribute `db`, of sort `Database`, to keep the actual database in which all the modules being entered are stored (as a set of records), an attribute `default`, to keep the identifier of the current module by default, and attributes `input` and `output`, to simplify the communication of the read-eval-print loop given by the `LOOP-MODE` module with the database object. Using the notation for classes⁵ we can declare such a class as follows:

```
class DatabaseClass | db : Database, default : ModName,
                    input : TermList, output : QidList .
```

The state of the read-eval-print loop is then given by an object of class `DatabaseClass`. In the case of Full Maude, the rules handling the read-eval-print loop are given in the modules `DATABASE-HANDLING` and `FULL-MAUDE`. In particular, the module `FULL-MAUDE` includes the rules to initialize the loop (`init` rule), and to specify the communication between the loop—the input/output of the system—and the database object (`in` and `out` rules). Depending on the kind of input that the database receives, its state will be changed, or some output will be generated, or both.

To parse some input using the built-in function `metaParse`, Full Maude needs the metarepresentation of the signature in which the input is going to be parsed. In Full Maude, such a grammar is provided by the `FULL-MAUDE-SIGN` module, in which we can find the appropriate declarations, so that any valid input, namely modules, theories, views, and commands, can be parsed. In particular we find in such a module sorts `@Bubble@` and `@Command@`, of bubbles (see Section 18.4) and commands, respectively. Thus, we declare the module `UNIFICATION-SIGN` extending the module `FULL-MAUDE-SIGN` with syntax for the `unifyc` command as follows.

```
fmod UNIFICATION-SIGN is
  extending FULL-MAUDE-SIGN .
```

⁵ Although we will introduce the notation for class declarations in the next chapter, it is quite intuitive and may help us to visualize the structure of the objects in the `DatabaseClass` class at this point.

```

    op unifyc_ . : @Bubble@ -> @Command@ .
endfm

```

This module must be entered after deactivating the default inclusion of the `BOOL` module by using the `set include BOOL off` command (see Section 25.13). Note that, since a bubble can be anything, trying to define here syntax for the conjunction of equations would lead to unavoidable ambiguous parses. Instead, we will see below how we make a second parse of the bubble considering the module in which the unification takes place, that is, in which the terms being unified are defined, extended with declarations for equations and sets of equations.

As we will see below, this module is used as a grammar for parsing the inputs using `metaParse`, and therefore it must be used at the metalevel. Since we need additional declarations for parsing tokens and bubbles (see Section 18.4), we cannot use the `upModule` operator. Instead, since these additional declarations are available in a metamodule `GRAMMAR`, we get the intended metamodule as follows:

```

fmod UNIFICATION-META-SIGN is
  pr META-LEVEL .
  pr META-FULL-MAUDE-SIGN .
  pr UNIT .
  op UNIF-GRAMMAR : -> FModule .
  eq UNIF-GRAMMAR
    = addImports((including 'UNIFICATION-SIGN .), GRAMMAR) .
endfm

```

Given this module, we will use the constant `UNIF-GRAMMAR` for parsing the inputs.

Another important module in Full Maude is `COMMAND-PROCESSING`, which includes definitions of functions in charge of processing the different commands. In particular, there is a function `procCommand`, which takes a term—as given by the function `metaParse`—corresponding to a command, the name of the current module, and the current database. Since Maude delays the compilation of the entered modules until the moment when they are going to be used, before being able to use a module, we must make sure that it is compiled; if it is not, it is compiled at this time by using the `evalModExp` function. Among others, the `COMMAND-PROCESSING` module and its submodules provide functions that are used here, namely, `compiledModule`, `evalModExp`, and `getFlatModule`. The `compiledModule` predicate checks the database to see whether the given module expression has already been compiled or not; the `evalModExp` function evaluates a given module expression in the context of the given database and returns as result a term of sort `Tuple{Database, ModuleExpression}`, for which there exist projection functions `database` and `modExp`; finally, the `getFlatModule` function returns the flat version of the specified module in the given database term.

The UNIFICATION-COMMAND-PROCESSING module below extends the Full Maude module COMMAND-PROCESSING by providing an equation dealing with the new command. In such an equation we call an auxiliary function `procUnify` which is in charge of calling `metaUnify` with the appropriate arguments. In particular, before calling `metaUnify`, the bubble must be transformed into the corresponding conjunction of equations, if possible. To do that we use an auxiliary function `solveBubblesCommEqSet`; to avoid the previously mentioned potential ambiguity, we do the parsing in the module in which the unification takes place extended with the declarations in the following module:⁶

```
fmod EQ-CONDITION-SYNTAX is
  sort @CommEqSet@ .
  op _/\_ : @CommEqSet@ @CommEqSet@ -> @CommEqSet@
    [ctor assoc prec 73] .
  op _=_ : Universal Universal -> @CommEqSet@
    [ctor poly(1 2) prec 71] .
endfm
```

Note the use of the `poly` attribute to declare a polymorphic operator `_=_`. To extend a module with these declarations we use the Full Maude function `addDecls`, which returns the module resulting from adding all the declarations in the module passed as second argument to the module passed as first argument. The function `metaParse` is then called with the module resulting from the addition of these declarations to the module passed to `solveBubblesCommEqSet`. The term resulting from the parsing is transformed into a term of sort `CommEqSet` by the `getCommEqSet` function. Notice that these functions are declared as partial, returning a term at the kind level in case an error occurs, which allows us to give the corresponding message (see the `procUnify` function below). Note also that both `procCommand` and `procUnify` return a list of quoted identifiers, that will be given as output to the loop.

```
fmod UNIFICATION-COMMAND-PROCESSING is
  pr COMMAND-PROCESSING .
  pr UNIFICATION .

  vars T T' : Term .
  var ME : ModuleExpression .
  var DB : Database .
  var M : Module .
  var Sb : Substitution .
  var SbS : SubstitutionSet .
  var N : Nat .

  eq procCommand('unifyc_.[T], ME, DB)
    = if compiledModule(ME, DB)
```

⁶ As for other modules defining syntax, we must make sure that the `BOOL` module is not imported.

```

    then << DB ; procUnify(getFlatModule(ME, DB), T) >>
    else << DB ; procUnify(
      getFlatModule(modExp(evalModExp(ME, DB)),
        database(evalModExp(ME, DB))),
      T) >>
    fi .

op getCommEqSet : Term ~> CommEqSet .
eq getCommEqSet('_/\_[T, T']) = getCommEqSet(T) getCommEqSet(T') .
eq getCommEqSet('=_[T, T']) = T =?c T' .

op solveBubblesCommEqSet : Term Module -> CommEqSet .
eq solveBubblesCommEqSet('bubble[T], M)
  = getCommEqSet(
    getTerm(
      metaParse(
        addDecls(M, upModule('EQ-CONDITION-SYNTAX, true)),
        downQidList(T),
        '@Condition@))) .

op procUnify : Module Term -> QidList .
op procUnify : Module SubstitutionSet Nat -> QidList .
eq procUnify(M, T)
  = if metaUnify(M, solveBubblesCommEqSet(T, M)) :: SubstitutionSet
    then procUnify(M, metaUnify(M, solveBubblesCommEqSet(T, M)), 1)
    else ('\r 'Error: '\o 'Incorrect 'unifyc 'command. '\n)
    fi .
eq procUnify(M, emptySubstitutionSet, N)
  = ('\n '\n 'No 'more 'solutions.) .
eq procUnify(M, substitutionSet(Sb, SbS), N)
  = ('\n 'Solution qid(string(N, 10)) '\s
    '\n eMetaPrettyPrint(M, Sb)
    procUnify(M, SbS, s N)) .
endfm

```

The module `DATABASE-HANDLING` defines the behavior of the database upon new entries. The behavior associated with commands is managed by rules describing transitions which call the function `procCommand`. The following module `UNIFICATION-DATABASE-HANDLING` extends `DATABASE-HANDLING` by adding a rule handling the new `unifyc` command.

```

mod UNIFICATION-DATABASE-HANDLING is
  inc DATABASE-HANDLING .
  pr UNIFICATION-COMMAND-PROCESSING .

  sort Unifier .
  subsort Unifier < DatabaseClass .
  op Unifier : -> Unifier .

```

```

var Atts : AttributeSet .
var X@Unifier : Unifier .
var O : Oid .
var ME : ModuleExpression .
var DB : Database .
vars T T' : Term .
var QIL : QidList .

rl [Unifier] :
  < O : X@Unifier | db : DB, input : ('unifyc_.[T]),
    output : QIL, default : ME, Atts >
=> < O : X@Unifier | db : DB, input : nilTermList,
    output : procCommand('unifyc_.[T], ME, DB),
    default : ME, Atts > .
endm

```

Note that, given the syntax defined above, when a `unifyc` command is entered, the function `metaParse` returns a term of the form `'unifyc_.[T]`, where `T` is variable of sort `Term` representing a bubble. As we will see in the redefinition of the module `FULL-MAUDE` below, the result of parsing the input is placed in the `input` attribute of the database object of class `DatabaseClass`.

Before presenting the redefinition of the `FULL-MAUDE` module, we need a last module defining a constant that will be used to show a banner at start-up time.

```

fmod UNIFICATION-BANNER is
pr STRING .
op unification-banner : -> String .
eq unification-banner
  = "Unification command available October 29th 2008" .
endfm

```

Finally, to make the new command available in Full Maude, we need to redefine the `FULL-MAUDE` module. Notice that now the `init` rule creates an instance of the class `Unifier`, the `in` and `out` rules take `Unifier` objects, and parsing for them is done using `UNIF-GRAMMAR` instead of `GRAMMAR` inputs.

```

mod FULL-MAUDE is
pr UNIFICATION-META-SIGN .
inc UNIFICATION-DATABASE-HANDLING .
inc LOOP-MODE .
pr BANNER .
pr UNIFICATION-BANNER .

subsort Object < State .

op o : -> Oid .
op init : -> System .

```

```

var Atts : AttributeSet .
var X@Unifier : DatabaseClass .
var O : Oid .
var DB : Database .
var ME : Header .
var QI : Qid .
vars QIL QIL' QIL'' : QidList .
var TL : TermList .
var N : Nat .
vars RP RP' : ResultPair .

rl [init] :
  init
  => [nil,
    < o : Unifier |
      db : initialDatabase,
      input : nilTermList, output : nil,
      default : 'CONVERSION >,
      ('\t string2qidList(unification-banner) '\n)] .

rl [in] :
  [QI QIL,
    < O : X@Unifier | db : DB, input : nilTermList,
      output : nil, default : ME, Atts >,
    QIL']
  => if metaParse(UNIF-GRAMMAR, QI QIL, '@Input@') :: ResultPair
    then [nil,
      < O : X@Unifier | db : DB,
        input : getTerm(
          metaParse(UNIF-GRAMMAR, QI QIL, '@Input@')),
        output : nil, default : ME, Atts >,
      QIL']
    else [nil,
      < O : X@Unifier | db : DB, input : nilTermList,
        output :
          ('\r 'Warning:
            printSyntaxError(
              metaParse(UNIF-GRAMMAR, QI QIL, '@Input@'),
              QI QIL)
            '\n '\r 'Error: '\o 'No 'parse 'for 'input. '\n),
        default : ME, Atts >,
      QIL']
    fi .

rl [out] :
  [QIL,
    < O : X@Unifier | db : DB, input : TL,
      output : (QI QIL'), default : ME, Atts >,
    QIL'']

```

```

=> [QIL,
    < 0 : X@Unifier | db : DB, input : TL,
      output : nil, default : ME, Atts >,
    (QI QIL' QIL'')] .
endm

```

Loading Full Maude with the file `unification-command.maude`, containing the modules presented in this section together with a previous command to load the file `unification.maude`, which in turn contains the modules presented in Section 16.1, we get the following:

```

$ maude.linux full-maude.maude unification-command.maude
\|||||/
--- Welcome to Maude ---
/|||||/
Maude 2.7 built: Mar 3 2014 18:07:27
Copyright 1997-2014 SRI International
Tue Jul 15 22:57:15 2014

Full Maude 2.7 March 10th 2015

Unification command available October 29th 2008

```

Maude>

We can now solve the unification problems presented in Section 16.1 at the object level by means of the `unifyc` command, after making sure that we have loaded first the `PEANO-NAT` module (from Section 4.10 and assuming it is in the file `peano-nat.fm`) on which the unification will take place.

```

Maude> load peano-nat.fm
Introduced module PEANO-NAT

```

```

Maude> (unifyc X:NzNat + (0 * Y:NzNat) = W:Nat + s Z:Nat .)

```

Solution 1

```

W:Nat --> 0 * Y@:NzNat ;
X:NzNat --> s Z@:Nat ;
Y:NzNat --> Y@:NzNat ;
Z:Nat --> Z@:Nat

```

No more solutions.

```

Maude> (unifyc X:NzNat + s (Y:Nat * W:Nat) = s V:Nat + Z:Nat .)

```

Solution 1

```

V:Nat --> V@:Nat ;
W:Nat --> W@:Nat ;
X:NzNat --> s V@:Nat ;

```

```

Y:Nat --> Y@:Nat ;
Z:Nat --> s(Y@:Nat * W@:Nat)
Solution 2
V:Nat --> Y@:Nat * W@:Nat ;
W:Nat --> W@:Nat ;
X:NzNat --> Z@:NzNat ;
Y:Nat --> Y@:Nat ;
Z:Nat --> Z@:NzNat

```

No more solutions.

```

Maude> (unifyc s X:Nat + (X:Nat * Y:Nat) = Z:NzNat + s s 0
      /\ Y:Nat = s s W:NzNat
      /\ s V:Nat = s s s s s 0
      /\ Z:NzNat = V:Nat * s 0 .)

```

```

Solution 1
V:Nat --> s s s s 0 ;
W:NzNat --> s s 0 ;
X:Nat --> s 0 ;
Y:Nat --> s s s s 0 ;
Z:NzNat --> s 0 * s s s s 0

```

No more solutions.

19.7.2 A new module expression: DEADLOCK-FREE

In this section we describe some guidelines for making available in Full Maude new module expressions. We have seen in Section 19.3.1 that very powerful module expressions, like `TUPLE[_]` and `POWER[_]`, are available in Full Maude. A module expression like `TUPLE[_]` is very useful in practice, but it also helps as an example to envision many other uses of module expressions. The `TUPLE[_]` operation takes a nonzero natural number n and returns a parameterized `TUPLE[n]` module, which is impossible to achieve with the Clear/OBJ repertoire of module operations [42, 174]. Even though an n -tuple module expression is in principle of a completely different nature from the usual Clear/OBJ module operations, the way Full Maude handles it is the same as the way it handles any other module expression. Its evaluation produces a new unit, a parameterized functional module in this case, with the module expression as its name. The semantics of the module expression is then defined by the way such module is generated.

In Section 16.3 we defined a function `deadlock-free` that takes the metarepresentations of a module and a sort, and returns the metarepresentation of the module resulting from transforming it so that the rewrite theory is deadlock free. However, since the `deadlock-free` function works at the metalevel and returns the metarepresentation of a module, it is not easy to use its result at the object level. Here we present a module expression that will use

such `deadlock-free` function to generate a new module containing the result of the transformation that can later be used to perform other computations such as, for example, model-checking analysis for equational abstractions (see Section 13.4).

As for the unification command, we first extend the grammar of Full Maude with the corresponding declarations. We declare a constant `DEADLOCK-GRAMMAR` in a module `DEADLOCK-FREE-META-SIGN` as we did for the unification command in Section 19.7.1.

```
fmod DEADLOCK-FREE-SIGN is
  ex FULL-MAUDE-SIGN .
  op DEADLOCK-FREE['_','_'] : @ModExp@ @Sort@ -> @ModExp@ .
endfm

fmod DEADLOCK-FREE-META-SIGN is
  pr META-LEVEL .
  pr META-FULL-MAUDE-SIGN .
  pr UNIT .

  op DEADLOCK-GRAMMAR : -> FModule .

  eq DEADLOCK-GRAMMAR
    = addImports((including 'DEADLOCK-FREE-SIGN .), GRAMMAR) .
endfm
```

The `DEADLOCK-FREE-EXPR` module below declares a `DEADLOCK-FREE[_ , _]` operator, of sort `ModuleExpression`, and equations completing the definition of functions on it in Full Maude. The main of these functions is `evalModExp`, which evaluates a given module expression in the context of a given database and returns as result a term of sort `Tuple{Database, ModuleExpression}`. The `evalModExp` function generates and introduces in the database the modules required for giving semantics to the given module expression. The modified database is returned as first component of the resulting tuple; in this case, the new module is generated using the `deadlock-free` function presented in Section 16.3. Since some module expressions are modified during their evaluation, the new module expression is also returned as second component of the resulting tuple. Furthermore, other functions must be completed:

- given a term, resulting from a call to the `metaParse` function with a module expression, the `parseModExp` function returns the corresponding term of sort `ModuleExpression`;
- before calling the `parseModExp` function, all occurrences of the `up` function are removed by a `solveUps` function, which needs to be defined on this new type of module expression;
- related to the handling of linking parameters, the `labelInModExp` function checks whether a given label occurs in the module expression given as second parameter;

- the `header2Qid` function returns the module expression given as parameter represented as a single quoted identifier;
- the `prepModExp` is needed for the instantiation of module expressions; it is in charge of appropriately handling the repeated use of view expressions;
- `setUpModExpDeps` establishes the dependencies between the modules in the database, so that if a module is redefined all modules depending on it are recompiled before being used.

```
fmod DEADLOCK-FREE-EXPR is
  inc MOD-EXPR .
  pr INST-EXPR-EVALUATION .
  pr EVALUATION .
  pr DEADLOCK-FREEDOM .
  pr MOD-EXP-PARSING .

  vars N N' : NzNat .
  var PDL : ParameterDeclList .
  vars DB DB' : Database .
  vars T T' T'' : Term .
  var IL : ImportList .
  var VEPS : Set<Tuple<ViewExp|ViewExp>> .
  var X : Qid .
  var S : Sort .
  vars ME ME' : ModuleExpression .
  var DT : Default{Term} .
  vars U U' M DM : Module .
  vars MNS MNS' MNS'' MNS3 MNS4 : Set{ModuleName} .
  vars VES VES' : Set{ViewExp} .
  var MIS : Set{ModuleInfo} .
  var VIS : Set{ViewInfo} .
  var QIL : QidList .
  var VDS : OpDeclSet .

  op DEADLOCK-FREE[_,_] : ModuleExpression Sort -> ModuleExpression .

  ceq evalModExp(DEADLOCK-FREE[ME, S], PDL, DB)
    = if unitInDb(DEADLOCK-FREE[ME', S], DB')
      then < DB' ; DEADLOCK-FREE[ME', S] >
      else < evalModule(
        setName(
          deadlock-free(getFlatModule(ME', DB'), S),
          DEADLOCK-FREE[ME', S]),
          none, DB') ;
        DEADLOCK-FREE[ME', S] >
      fi
  if ME' := modExp(evalModExp(ME, PDL, DB))
    /\ DB' := database(evalModExp(ME, PDL, DB)) .

  eq parseModExp('DEADLOCK-FREE'[_','_][T, T'])
```

```

= DEADLOCK-FREE[parseModExp(T), parseType(T')] .

eq solveUps('upModule['DEADLOCK-FREE'[_',_'][T, T']], DB)
= solveUpsModExp('upModule['DEADLOCK-FREE'[_',_'][T, T']], DB) .
eq solveUps(
  'upTerm['DEADLOCK-FREE'[_',_'][T, T'], 'bubble[T']'], DB)
= solveUpsModExp(
  'upTerm['DEADLOCK-FREE'[_',_'][T, T'], 'bubble[T']'], DB) .

eq labelInModExp(X, DEADLOCK-FREE[ME, S]) = false .

eq header2Qid(DEADLOCK-FREE[ME, S])
= qid("DEADLOCK-FREE[" + string(header2Qid(ME))
  + ", " + string(S) + "]") .
eq header2QidList(DEADLOCK-FREE[ME, S])
= ('DEADLOCK-FREE ''[ header2QidList(ME) ''', S '''] .

eq prepModExp(DEADLOCK-FREE[ME, S], VEPS)
= DEADLOCK-FREE[prepModExp(ME, VEPS), S] .

eq setUpModExpDeps(DEADLOCK-FREE[ME, S],
  db(< ME ; DT ; U ; U' ; M ; VDS ; MNS ; VES > MIS,
    MNS', VIS, VES', MNS'', MNS3, MNS4, QIL))
= db(< ME ; DT ; U ; U' ; M ; VDS ;
  MNS . DEADLOCK-FREE[ME, S] ; VES >
  MIS, MNS', VIS, VES', MNS'', MNS3, MNS4, QIL) .
eq setUpModExpDeps(DEADLOCK-FREE[ME, S],
  db(< ME ; DM ; U ; U' ; M ; VDS ; MNS ; VES > MIS,
    MNS', VIS, VES', MNS'', MNS3, MNS4, QIL))
= db(< ME ; DM ; U ; U' ; M ; VDS ;
  MNS . DEADLOCK-FREE[ME, S] ; VES > MIS,
  MNS', VIS, VES', MNS'', MNS3, MNS4, QIL) .
eq setUpModExpDeps(DEADLOCK-FREE[ME, S], DB)
= warning(DB, '\r 'Error: '\o 'Module header2QidList(ME)
  'not 'in 'database. '\n)
[owise] .
endfm

```

As we did in Section 19.7.1 for the extension adding the unification command, the FULL-MAUDE module must be redefined. In this case, the grammar defined by the DEADLOCK-GRAMMAR module is used for parsing in the in rules. The DEADLOCK-FREE-BANNER module is used to show a banner at start-up.

```

fmod DEADLOCK-FREE-BANNER is
pr STRING .
op deadlock-free-banner : -> String .
eq deadlock-free-banner
= "DEADLOCK-FREE mod. expr. available Oct. 28th 2008" .
endfm

```

```

mod FULL-MAUDE is
  pr DATABASE-HANDLING .
  inc LOOP-MODE .
  pr BANNER .
  pr DEADLOCK-FREE-META-SIGN .
  pr DEADLOCK-FREE-BANNER .
  pr DEADLOCK-FREE-EXPR .

  subsort Object < State .
  sort DeadlockFree .
  subsort DeadlockFree < DatabaseClass .
  op DeadlockFree : -> DeadlockFree .
  op o : -> Oid .
  op init : -> System .

  var Atts : AttributeSet .
  var X@DeadlockFree : DatabaseClass .
  var O : Oid .
  var DB : Database .
  var ME : Header .
  var QI : Qid .
  vars QIL QIL' QIL'' : QidList .
  var TL : TermList .
  var N : Nat .
  vars RP RP' : ResultPair .

  rl [init] :
    init
    => [nil,
      < o : DeadlockFree |
        db : initialDatabase,
        input : nilTermList, output : nil,
        default : 'CONVERSION >,
        ('\t string2qidList(deadlock-free-banner) '\n)] .

  rl [in] :
    [QI QIL,
      < O : X@DeadlockFree | db : DB, input : nilTermList,
        output : nil, default : ME, Atts >,
      QIL']
    => if metaParse(DEADLOCK-GRAMMAR, QI QIL, '@Input@') :: ResultPair
      then [nil,
        < O : X@DeadlockFree | db : DB,
          input :
            getTerm(
              metaParse(DEADLOCK-GRAMMAR, QI QIL, '@Input@')),
          output : nil, default : ME, Atts >,
        QIL']
      else [nil,

```

```

    < 0 : X@DeadlockFree | db : DB, input : nilTermList,
      output :
        ('\r 'Warning:
          printSyntaxError(
            metaParse(DEADLOCK-GRAMMAR, QI QIL, '@Input@),
            QI QIL)
          '\n '\r 'Error: '\o 'No 'parse 'for 'input. '\n),
        default : ME, Atts >,
    QIL']
  fi .

rl [out] :
  [QIL,
    < 0 : X@DeadlockFree |
      db : DB, input : TL, output : (QI QIL'), default : ME, Atts >,
    QIL'' ]
=> [QIL,
    < 0 : X@DeadlockFree |
      db : DB, input : TL, output : nil, default : ME, Atts >,
    (QI QIL' QIL'')] .
endm

```

Now, after loading the above modules (which we assume are in the file `deadlock-free-mod-expr.maude`, together with a command to load the modules from Section 16.3), we may use the `DEADLOCK-FREE` module expression to transform a system module before model checking some properties on it. For example, if the file `bakery.maude` contains the `BAKERY` module introduced in Section 13.4, we may proceed as follows:

```

$ maude.linux model-checker.maude bakery.maude full-maude.maude
deadlock-free-mod-expr.maude
      \|||||/
      --- Welcome to Maude ---
      /|||||\
      Maude 2.7 built: Mar  3 2014 18:07:27
      Copyright 1997-2014 SRI International
      Tue Jul 15 22:57:15 2014

      Full Maude 2.7 March 10th 2015

      DEADLOCK-FREE mod. expr. available Oct. 28th 2008

Maude> (mod DF-BAKERY is
        protecting DEADLOCK-FREE[BAKERY, BState] .
        endm)
Introduced module DF-BAKERY

```

The `DF-BAKERY` module is now ready for proceeding as shown in Section 13.4 to model check the mutual exclusion and liveness properties. One

can also use it for other purposes, as any other module; for example, we can request its set of rules as follows:

```
Maude> (show rls DF-BAKERY .)

crl {V:BState}
=> {V:BState}
  if enabled(V:BState) /= true = true .
rl < P:Mode, 0, wait, Y:Nat >
=> < P:Mode, 0, crit, Y:Nat > [label p2_wait] .
rl < P:Mode, X:Nat, crit, Y:Nat >
=> < P:Mode, X:Nat, sleep, 0 > [label p2_crit] .
rl < P:Mode, X:Nat, sleep, Y:Nat >
=> < P:Mode, X:Nat, wait, s X:Nat > [label p2_sleep] .
rl < crit, X:Nat, Q:Mode, Y:Nat >
=> < sleep, 0, Q:Mode, Y:Nat > [label p1_crit] .
rl < sleep, X:Nat, Q:Mode, Y:Nat >
=> < wait, s Y:Nat, Q:Mode, Y:Nat > [label p1_sleep] .
rl < wait, X:Nat, Q:Mode, 0 >
=> < crit, X:Nat, Q:Mode, 0 > [label p1_wait] .
crl < P:Mode, X:Nat, wait, Y:Nat >
=> < P:Mode, X:Nat, crit, Y:Nat >
  if Y:Nat < X:Nat = true [label p2_wait] .
crl < wait, X:Nat, Q:Mode, Y:Nat >
=> < crit, X:Nat, Q:Mode, Y:Nat >
  if not Y:Nat < X:Nat = true [label p1_wait] .
```

Narrowing

20.1 Introduction

Narrowing is a generalization of term rewriting that allows free variables in terms (as in logic programming) and replaces pattern matching by unification in order to (non-deterministically) reduce these terms. Narrowing was originally introduced as a mechanism for solving equational unification problems [157]. It was later generalized to solve the more general problem of symbolic reachability [266]. The narrowing mechanism has a number of important applications, including automated proofs of termination [12], execution of functional-logic programming languages [172, 101, 179, 312, 244], partial evaluation [10], verification of cryptographic protocols [266], and equational unification [196], to mention just a few.

At each rewriting step one must choose which subterm of the subject term and which rule of the specification are going to be considered. Similarly, at each narrowing step one must choose which subterm of the subject term, which rule of the specification, and which instantiation on the variables of the subject term and the rule's lefthand side are going to be considered. The narrowing relation is formally defined as follows. Let $\mathcal{R} = (\Sigma, E \cup Ax, R)$ be an order-sorted rewrite theory where R is a set of *unconditional* rewrite rules specified with the `r1` keyword, E is a set of *unconditional* equations specified with the `eq` keyword, and Ax is a set of commonly occurring axioms declared in Maude as equational attributes (see Section 4.4.1) such that an $E \cup Ax$ -unification procedure is available in Maude.¹ Let $CSU_{E \cup Ax}(u = u')$ provide² a finitary and complete set of unifiers for any pair of terms u, u' with the same top sort. The $R, E \cup Ax$ -narrowing relation on $T_\Sigma(X)$ is defined as

¹ $E \cup Ax$ -unification is available in Maude 2.7 via the *variant*-based equational unification of Section 15.10. Ax -unification for order-sorted signatures with free, C , AC , or ACU function symbols is also available, see Section 15.4. Note that $E \cup Ax$ -unification reduces to Ax -unification if the theory has no variant equation.

² In the current implementation we are not interested in a minimal set of unifiers, but only in a finite and complete set. Minimality is easily achieved in syntactic

$t \rightsquigarrow_{\sigma,p,R,E \cup Ax} t'$ (or $\rightsquigarrow_{\sigma}$ when $p, R, E \cup Ax$ are understood) if there is a non-variable position p of t , a (possibly renamed) rule $l \rightarrow r$ in R , and a unifier $\sigma \in CSU_{E \cup Ax}(t|_p = l)$ such that $t' = \sigma(t[r]_p)$. We denote by $t \rightsquigarrow_{\sigma,R,E \cup Ax}^+ t'$ (resp. $t \rightsquigarrow_{\sigma,R,E \cup Ax}^* t'$) the transitive (resp. reflexive-transitive) closure of the narrowing relation, where σ is obtained as the composition of the substitutions for each narrowing step in the sequence.

The difference between a rewriting step and a narrowing step is that in both cases we use a rewrite rule $l \rightarrow r$ to rewrite t at a position p in t , but narrowing *unifies* the lefthand side l and the chosen subject term $t|_p$ before actually performing the rewriting step. Also, narrowing is usually³ restricted to non-variable positions of t , whereas rewriting does not require such a restriction.

Consider the following system module defining the addition function $_+_$ on natural numbers built from 0 and \mathbf{s} :

```

mod NAT-NARROWING is
  sort Nat .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat .
  vars X Y : Nat .
  rl [base] : 0 + Y => Y .
  rl [ind] : s(X) + Y => s(X + Y) .
endm

```

Consider the term $X + \mathbf{s}(0)$ and the two rules **base** and **ind**. Narrowing will instantiate variable X with 0 and $\mathbf{s}(X')$ respectively in order to be able to apply each of these rules, i.e., the following two narrowing steps are generated:

$$\begin{aligned}
X + \mathbf{s}(0) &\rightsquigarrow_{\{X \mapsto 0\}, \text{base}} \mathbf{s}(0) \\
X + \mathbf{s}(0) &\rightsquigarrow_{\{X \mapsto \mathbf{s}(\#1:\text{Nat})\}, \text{ind}} \mathbf{s}(\#1:\text{Nat} + \mathbf{s}(0))
\end{aligned}$$

Note that, for simplicity, we show only the bindings of the unifier that affect the input term. There are infinitely many narrowing derivations starting at the input expression $X + \mathbf{s}(0)$ (at each step the reduced subterm is underlined):

1. $\underline{X + \mathbf{s}(0)} \rightsquigarrow_{\{X \mapsto 0\}, \text{base}} \mathbf{s}(0)$
2. $\underline{X + \mathbf{s}(0)} \rightsquigarrow_{\{X \mapsto \mathbf{s}(\#1:\text{Nat})\}, \text{ind}} \mathbf{s}(\#1:\text{Nat} + \mathbf{s}(0))$
 $\rightsquigarrow_{\{\#1:\text{Nat} \mapsto 0\}, \text{base}} \mathbf{s}(\mathbf{s}(0))$

unification [227], but it is very costly in Ax -unification or $E \cup Ax$ -unification, e.g., the ACU -unification considered in Section 15.4 does not always provide a minimal set of unifiers (but see Section 15.4.3 for a method to compute a minimal set of ACU -unifiers).

³ The *paramodulation* inference rule used in paramodulation-based theorem proving [277] is similar to narrowing and does not require non-variable positions; see Section 20.3.3.

$$\begin{aligned}
3. \quad \underline{X + s(0)} &\rightsquigarrow_{\{X \mapsto s(\#1:\text{Nat})\}, \text{ind}} s(\#1:\text{Nat} + s(0)) \\
&\rightsquigarrow_{\{\#1:\text{Nat} \mapsto s(\#2:\text{Nat})\}, \text{ind}} s(s(\#2:\text{Nat} + s(0))) \\
&\rightsquigarrow_{\{\#2:\text{Nat} \mapsto 0\}, \text{base}} s(s(s(0)))
\end{aligned}$$

and so on.

The following infinite narrowing derivation resulting from applying rule **ind** infinitely many times can also be shown:

$$\begin{aligned}
\underline{X + s(0)} &\rightsquigarrow_{\{X \mapsto s(\#1:\text{Nat})\}, \text{ind}} s(\#1:\text{Nat} + s(0)) \\
&\rightsquigarrow_{\{\#1:\text{Nat} \mapsto s(\#2:\text{Nat})\}, \text{ind}} s(s(\#2:\text{Nat} + s(0))) \\
&\rightsquigarrow_{\{\#2:\text{Nat} \mapsto s(\#3:\text{Nat})\}, \text{ind}} s(s(s(\#3:\text{Nat} + s(0)))) \\
&\dots
\end{aligned}$$

20.1.1 Applications

The classical application of narrowing modulo an equational theory is to perform $E \cup Ax$ -unification by **E**, Ax -narrowing when the equations E are oriented into sort-decreasing, confluent, terminating, and coherent modulo Ax rules **E**. Indeed the variant-based equational order-sorted unification algorithm of Section 15.10 is based on an E , Ax -narrowing strategy, called *folding variant narrowing* [151], that terminates when $E \cup Ax$ has the finite variant property [82], even though full E , Ax -narrowing typically does not terminate when Ax contains AC axioms [82, 151].

Instead, when the rules R are understood as *transition rules*, a completely different application of R , $E \cup Ax$ -narrowing is that of *symbolic reachability analysis* [266] (see Section 15.6.2). Specifically, we will consider the case of transition systems specified by order-sorted rewrite theories of the form $\mathcal{R} = (\Sigma, E \cup Ax, R)$ where: (i) $E \cup Ax$ has a finite and complete $E \cup Ax$ -unification algorithm (see the requirements of Section 15.9), and (ii) the transition rules R are $E \cup Ax$ -coherent and *topmost* (so that rewriting is always done at the top of the term). Then, narrowing is a *complete* deductive method [266] for symbolic reachability analysis, that is, for solving existential queries of the form $\exists \bar{x} t(\bar{x}) \rightarrow^* t'(\bar{x})$ in the sense that the formula holds for R iff there is a narrowing sequence $t \rightsquigarrow_{R, E \cup Ax}^* u$ such that u and t' have an $E \cup Ax$ -unifier.

Furthermore, in symbolic reachability analysis we may be interested in verifying properties more general than existential questions of the form $\exists X t \rightarrow^* t'$. One can also generalize the above reachability question to questions of the form $\mathcal{R}, t \models \varphi$, with φ a temporal logic formula. The papers [147, 14] show how narrowing can be used (again, both at the level of transitions with rules R and at the level of equations E) to perform *logical model checking* to verify such temporal logic formulas; this is a kind of *symbolic model checking* not in the binary decision diagram sense of “symbolic,” which still remains finite-state, but in a much more general sense in which possibly infinite sets of states are finitely described by patterns with logical variables. Two distinctive features are: (i) the term t does not describe a single initial

state, but a possibly infinite set of instances of t (i.e., a possibly infinite set of initial states); and (ii) the set of reachable states does not have to be finite. Therefore, standard model-checking techniques may not be usable, because of a possible double infinity: in the number of initial states, and in the number of states reachable for each of those initial states.

20.1.2 Completeness of narrowing

Due to nontermination, narrowing behaves as a semi-decision procedure for equational unification and for reachability analysis in a wide variety of theories. However, for some particular subject terms narrowing may terminate, providing a complete set of solutions. For instance, in the Maude module `NAT-NARROWING` above, narrowing computes the solution $\{X \mapsto \mathbf{s}(Y)\}$ for the reachability problem $\exists X, Y \ 0 + X \rightarrow^* \mathbf{s}(Y)$ and it terminates with no more solutions. Instead, for the reachability problem $\exists X, Z \ X + \mathbf{s}(0) \rightarrow^* \mathbf{s}(Z)$, narrowing computes the solution $\{X \mapsto \mathbf{s}(0), Z \mapsto 0\}$ but it cannot terminate because of the above infinite narrowing sequence using `ind`. Moreover, narrowing cannot prove that the reachability problem $\exists X \ X + \mathbf{s}(0) \rightarrow^* 0$ does not have a solution, again because of the above infinite narrowing sequence using `ind`.

Note that for any narrowing sequence $t \rightsquigarrow_{\sigma, R, E \cup Ax}^* s$, we have a corresponding rewrite sequence $\sigma(t) \rightarrow_{R, E \cup Ax}^* s$. However, only under appropriate conditions is narrowing *complete* as an equational unification algorithm [196, 200], or as a procedure to solve reachability problems [266]. That is, given a reachability problem⁴ $\exists X_1, \dots, X_k \ s \rightarrow^* t$ completeness means that for each possible substitution ρ that binds some variables of X_1, \dots, X_k in such a way that $\rho(s) \rightarrow_{R, E \cup Ax}^* \rho(t)$, and for all the substitutions $\sigma_1, \dots, \sigma_n, \dots$ provided by narrowing from s , there is an index i and two substitutions θ, τ such that $\rho(s) \rightarrow_{R, E \cup Ax}^* \theta(s)$, $\rho(t) \rightarrow_{R, E \cup Ax}^* \theta(t)$, $\theta(s) \rightarrow_{R, E \cup Ax}^* \theta(t)$, and $\theta =_{E \cup Ax} \sigma_i; \tau$, where $\sigma_i; \tau$ denotes substitution composition in diagrammatic order, i.e., $(\sigma_i; \tau)(X) = \tau(\sigma_i(X))$. Essentially, completeness holds either

1. for $(R, E \cup Ax)$ -normalized substitutions ρ above [266] (a stronger condition is $(R \cup E, Ax)$ -normalized substitutions);
2. for *topmost rewrite theories*⁵;

⁴ Equational unification can be represented in terms of a reachability problem [200] (see Section 15.6.1) and, therefore, we do not consider completeness results for equational unification here; see [11] for a recent survey on termination of narrowing and completeness results for equational unification and reachability problems in the free case.

⁵ That is, theories where every rewrite step is performed only at the top position of the term. In such theories completeness can be simplified as follows: given $\exists X_1, \dots, X_k \ s \rightarrow^* t$, for each ρ such that $\rho(s) \rightarrow_{R, Ax}^* \rho(t)$, and for all the substitutions $\sigma_1, \dots, \sigma_n, \dots$ provided by narrowing from s , there is an index i and a substitution τ such that $\rho =_{Ax} \sigma_i; \tau$.

3. for right-linear theories and linear reachability goals⁶; and
4. in particular for theories that are confluent, terminating, and coherent modulo axioms Ax , as the equational theories in Maude functional modules with such properties restricted to unconditional equations.

20.1.3 Narrowing with simplification

We have implemented a version of narrowing with simplification, which is slightly different from the one formally defined above. Let $\mathcal{R} = (\Sigma, G \cup E \cup Ax, R)$ be an order-sorted rewrite theory where R , E , and Ax are defined as above and G are the remaining equations. Note that when an equational theory $(\Sigma, G \cup E \cup Ax, R)$ is provided to Full Maude, each equation in E (used for *variant* computation) must include the `variant` attribute. Note that equations in G do not contain the `variant` attribute and do not have any restriction, i.e., they can be conditional equations, with the `owise` attribute, etc. Each narrowing step of the form $t \rightsquigarrow_{\sigma, p, R, E \cup Ax} t'$ is followed by simplification using the relation $\rightarrow_{G \cup E, Ax}^!$, i.e., the combined relation $(\rightsquigarrow_{\sigma, p, R, E \cup Ax}; \rightarrow_{G \cup E, Ax}^!)$ is defined as $t \rightsquigarrow_{\sigma, p, R, E \cup Ax}; \rightarrow_{G \cup E, Ax}^! t''$ iff $t \rightsquigarrow_{\sigma, p, R, E \cup Ax} t'$, $t' \rightarrow_{G \cup E, Ax}^* t''$, and t'' is $G \cup E, Ax$ -irreducible. Note that this combined relation $(\rightsquigarrow_{\sigma, p, R, E \cup Ax}; \rightarrow_{G \cup E, Ax}^!)$ may be incomplete, i.e., given a reachability problem of the form $t \rightarrow^* t'$ and a solution σ (i.e., $\sigma(t) \rightarrow_{R, G \cup E \cup Ax}^* \sigma(t')$), the relation $\rightsquigarrow_{\sigma, p, R, E \cup Ax}; \rightarrow_{G \cup E, Ax}^!$ may not be able to find a more general solution. The reason is that the equations G should also be executed by narrowing instead of rewriting to ensure completeness under appropriate conditions (see [266] and Section 20.1.2). However, the combination of narrowing using rules, equations, and axioms with simplification using additional equations can be quite helpful to allow built-in Maude functions such as addition or multiplication, which cannot be executed by narrowing in their predefined form. It can also be useful in other applications where specific combinations of narrowing and simplification are needed.

20.2 Theories supported for narrowing reachability

The narrowing relation is defined on top of the order-sorted variant-based unification procedure described in Section 15.10.

Let `mod` $(\Sigma, G \cup E \cup Ax, R)$ `endm` be an order-sorted system module where R is a set of rewrite rules specified with the `rl` or `cr1` keywords, Ax is a set of commonly occurring axioms (declared in Maude as equational attributes, see Section 4.4.1), E is a set of equations specified with the `eq` keyword and the attribute `variant` such that $(\Sigma, E \cup Ax)$ satisfies the restrictions mentioned

⁶ A reachability goal of the form $\exists X_1, \dots, X_k s \rightarrow^* t$ is linear if s is linear and s and t do not have variables in common.

in Section 15.10, and G are the remaining equations specified with the `eq` or `ceq` keywords. Furthermore, the rules R must satisfy the following extra conditions:

- Conditional rules specified with the `cr1` keyword are not taken into account, i.e., there may be conditional rewrite rules in the system module but they will not be used for narrowing.
- A rule's lefthand side cannot be a variable. If a variable is needed, one should instead specify a new kind with an extra unary symbol grabbing the whole system's state (which would before have been matched by a single variable lefthand side). In this way, the problem of having a variable lefthand side can often be solved.
- The rules must be explicitly Ax -coherent (see Section 6.3). Moreover, Ax -coherence of equations and rules is enforced automatically by calling the coherence completion algorithm of Section 19.5. If the rules are already Ax -coherent, no extra rules will be added.

We recall again that the equations G in the system module are disregarded for narrowing purposes. However, they are applied for simplification after each narrowing step (see Section 20.1.3), as it is performed in Maude for rewriting. Recall, again, that this combination of one narrowing step followed by equational simplification is not complete. A full treatment of rules, equations, and axioms for narrowing is outside the scope of the present implementation and is left for future work.

Furthermore, *frozen arguments* (see Section 4.4.9) are allowed for narrowing, as for rewriting. They are given the standard meaning of not allowing any narrowing step below such frozen arguments, just as in the *context-sensitive narrowing* of [222].

Finally, we do not consider any narrowing strategy at all for solving reachability problems, i.e., all positions in a term with an admissible $R, E \cup Ax$ -narrowing step are explored.

20.3 The narrowing search command

Given a system module $\langle ModId \rangle$, the user can give to Full Maude a search command of the form:

```
(search [  $n$ ,  $m$  ] in  $\langle ModId \rangle$  :  $\langle Term-1 \rangle$   $\langle SearchArrow \rangle$   $\langle Term-2 \rangle$  .)
```

where:

- n is an optional argument providing a bound on the number of desired solutions;
- m is another optional argument stating the maximum depth of the search;
- the system module $\langle ModId \rangle$ where the search takes place can be omitted if it is the current one;

- $\langle Term-1 \rangle$ is the starting *non-variable term*, which may contain variables;
- $\langle Term-2 \rangle$ is the term specifying the pattern that has to be reached, with variables possibly shared with $\langle Term-1 \rangle$;
- $\langle SearchArrow \rangle$ is an arrow indicating the form of the narrowing proof from $\langle Term-1 \rangle$ until $\langle Term-2 \rangle$:
 - $\sim>1$ means a narrowing proof consisting of exactly one step,
 - $\sim>+$ means a narrowing proof consisting of one or more steps,
 - $\sim>*$ means a narrowing proof consisting of none, one, or more steps, and
 - $\sim>!$ indicates that only *strongly irreducible* final states are allowed, i.e., states that cannot be further narrowed; note that this is stronger than requiring states that cannot be rewritten.

The one step arrow $\sim>1$ is an abbreviation for the one-or-more steps arrow $\sim>+$ with the depth bound m set to 1.

Consider, for example, the following new version of the vending machine to buy apples (a) or cakes (c) with dollars (\$) and/or quarters (q). The reader can check that the only difference with the VARIANT-VENDING-MACHINE module in Section 15.9.2 is the change equation, which is ACU-coherent in the former version but it is not in this new version, because it is made ACU-coherent internally.

```

mod NARROWING-VENDING-MACHINE is
  sorts Coin Item Marking Money State .
  subsort Coin < Money .
  op empty : -> Money .
  op _ : Money Money -> Money [assoc comm id: empty] .
  subsort Money Item < Marking .
  op _ : Marking Marking -> Marking [assoc comm id: empty] .
  op <_> : Marking -> State .
  ops $ q : -> Coin .
  ops a c : -> Item .
  var M : Marking .
  rl [buy-c] : < M $ > => < M c > .
  rl [buy-a] : < M $ > => < M a q > .
  eq [change] : q q q q = $ [variant] .
endm

```

We can use the narrowing search command to answer the question:

Is there any combination of one or more coins that returns exactly an apple and a cake?

This can be done by searching for states that are reachable from a term $\langle M:Money \rangle$ and match the desired pattern at the end.

```

Maude> (search [1] in NARROWING-VENDING-MACHINE :
  < M:Money >  $\sim>*$  < a c > .)

```

```
Solution 1
M:Money --> $ q q q
```

```
No more solutions.
```

Note that `_` is an *ACU* symbol and that such an *ACU* symbol appears in the equation `change`, disallowing the basic narrowing strategy [196] to be used for equational unification and requiring the folding variant narrowing [151] to be used for equational unification.

Note that we have restricted the previous reachability problem to just one solution. Narrowing does not terminate for this reachability problem even though the above solution is indeed the *only* solution. The problem is that narrowing follows a breadth-first exploration and does not stop until the whole narrowing tree demanded by the search command is created, even though this infinite search may not yield any further solutions. The very same problem happens for the standard `search` command (see Section 6.4.3). If we increase the depth of the narrowing tree, we can experimentally observe that no more solutions are found.

```
Maude> (search [,5] in NARROWING-VENDING-MACHINE :
      < M:Money > ~>* < a c > .)
```

```
Solution 1
M:Money --> $ q q q
```

```
No more solutions.
```

As with the `search` command of Full Maude (see Section 19.6), the narrowing version does not provide paths to solutions but there is a metalevel command that does provide paths, described in Section 20.3.2.

In the previous reachability problem, we can change the arrow `~>*` for reachability in zero or more steps by the arrow `~>!` for reachability in zero or more steps including only states that cannot be narrowed any more.

```
Maude> (search [,5] in NARROWING-VENDING-MACHINE :
      < M:Money > ~>! < a c > .)
```

```
No more solutions.
```

And surprisingly we do not find the previous solution. The reason is that the transition rules of the vending machine are not terminating for narrowing and for rewriting, so it is impossible to find a state that cannot be narrowed any more. However, if we replace the variable `M:Money` by variables of sort `Coin`, we are able to find appropriate solutions. That is, we formulate the following interesting question:

Is there any combination of four coins that returns an apple and a cake and is such that some extra money is left but that extra money cannot be used to buy anything else?

The fact that some money is left is characterized by including a variable of sort `Money` in the final state, and the fact that nothing else can be bought is characterized by using the `~>!` arrow instead of `~>*`.

```
Maude> (search [,5] in NARROWING-VENDING-MACHINE :
      < C1:Coin C2:Coin C3:Coin C4:Coin > ~>! < M:Money a c > .)

Solution 1
C1:Coin --> $ ; C2:Coin --> q ; C3:Coin --> q ; C4:Coin --> q ; M:Money --> empty

Solution 2
C1:Coin --> q ; C2:Coin --> $ ; C3:Coin --> q ; C4:Coin --> q ; M:Money --> empty

Solution 3
C1:Coin --> q ; C2:Coin --> q ; C3:Coin --> $ ; C4:Coin --> q ; M:Money --> empty

Solution 4
C1:Coin --> q ; C2:Coin --> q ; C3:Coin --> q ; C4:Coin --> $ ; M:Money --> empty

No more solutions.
```

Another point of interest is the occurrence of variables of the form `#n:Sort` or `%n:Sort`, which are called *fresh* and are described in Chapter 15. Unification modulo axioms usually introduces fresh variables; furthermore, narrowing introduces many fresh variables because the rule applied at each narrowing step is appropriately *renamed* so that no variable is shared by it and the current term. Indeed, the standard solution used in logic and functional-logic programming language implementations is to use a counter along each narrowing derivation to ensure that fresh variables have never been used previously in that narrowing derivation. This method is called *standardized apart*.

20.3.1 Narrowing with extra variables in righthand sides of rules

Although rewriting does not allow extra variables in the righthand side of rules⁷, extra variables in righthand sides pose no problem for narrowing. Since rules having extra variables in the righthand side are not allowed in Maude for rewriting purposes, the attribute `nonexec` (see Section 4.5.3) must be added to such rules if one wants to use them for narrowing. The `nonexec` attribute is not taken into account by narrowing: *all* unconditional rules, regardless of whether or not they include the `nonexec` attribute, are used by narrowing. Extra variables in the righthand side are a common feature of programs using narrowing as the operational evaluation mechanism, as in logic programming

⁷ Except for the case of *conditional* rules, where such extra variables may be introduced in matching or rewrite conditions (see Chapter 4). But conditional rules have been excluded from the present narrowing implementation, as explained in Section 20.2.

or functional-logic programming [179]. For further details on how to write functional-logic programs in Maude using symbolic reachability, see [141]. Let us motivate this feature with an example. Consider the following program defining the concatenation of two lists and the function `last`:

```
(mod LAST-APPEND is
  sort Success .
  op success : -> Success .
  sort Nat .
  op 0 : -> Nat .
  op s : Nat -> Nat .
  sort NatList .
  op nil : -> NatList .
  op _:_ : Nat NatList -> NatList .

  vars XS YS : NatList .
  vars N M X Y : Nat .

  op append : NatList NatList -> [NatList] .
  rl append(nil, YS) => YS .
  rl append(N : XS, YS) => N : append(XS, YS) .

  op last : NatList -> [Nat] .
  rl last(XS) => append(YS, N : nil) ::= XS >> N [nonexec] .

  op _>>_ : [Success] [Nat] -> [Nat] [frozen (2) strat (1 0)] .
  eq success >> X:[Nat] = X:[Nat] .

  op _:=_ : Nat Nat -> [Success] [comm] .
  rl N := N => success .

  op _:=_ : NatList NatList -> [Success] [comm] .
  rl XS := XS => success .
endm)
```

In the rule

```
rl last(XS) => append(YS, N : nil) ::= XS >> N [nonexec] .
```

we have used an extra variable `N` to denote the last element of the list and used a constraint

```
append(YS, N : nil) ::= XS
```

that narrowing will solve by instantiating `N` in the proper way. Furthermore, note the use of kinds and the sort `Success` in order to describe what a successful solution is; this follows a logic programming approach (as in Prolog) to success and failure, see [141]. The following reachability problem is solved

by narrowing but cannot be solved by rewriting due to the extra variable in the `last` rule.⁸

```
Maude> (search [1] in LAST-APPEND : last(0 : s(0) : nil) ~>! Z:Nat .)

Solution 1
Z:Nat --> s(0)

No more solutions.
```

Another interesting example of narrowing with extra variables is the function `member`:

```
(mod MEMBERSHIP is
  protecting LAST-APPEND .
  vars XS YS ZS : NatList .
  vars N M X Y : Nat .
  op member : Nat NatList -> [Success] .
  rl member(N, XS) => append(YS, N : ZS) ::= XS [nonexec] .
endm)
```

The following reachability problem is solved by narrowing but cannot be solved by rewriting due to the extra variable in the rule defining the `member` function.

```
Maude> (search [1] in MEMBERSHIP :
  member(s(0), 0 : s(0) : nil) ~>! success .)

Solution 1
empty substitution

No more solutions.
```

But the interesting application is to enumerate all the elements of a list by computing different substitutions, as in logic programming.

```
Maude> (search [,5] in MEMBERSHIP :
  member(N:Nat, 0 : s(0) : nil) ~>! success .)

Solution 1
N:Nat --> 0

Solution 2
```

⁸ Of course, one could specify the `last` function in a way not requiring the `nonexec` rule, for example by the equations `last(nil) = error`, `last(N : nil) = N`, and `last(N : (M : XS)) = last (M : XS)`. The point here is to illustrate with a simple example the use of narrowing with rules having extra variables in their righthand sides, which is typical of logic programming and functional-logic programming.

```
N:Nat --> s(0)
```

```
No more solutions.
```

Note that we have to restrict the depth of the narrowing tree to five because there exists an infinite number of narrowing sequences from the term `member(N:Nat,0 : s(0): nil)` even if only two solutions exist.

20.3.2 Reachability at the metalevel: `metaNarrowSearch` and `metaNarrowSearchPath`

Narrowing-based reachability analysis is also available at the metalevel by using the function `metaNarrowSearch`, provided in the `META-NARROWING-SEARCH` module with the following declaration:

```
op metaNarrowSearch :
  Module Term Term Substitution Qid Bound Bound Bound -> ResultTripleSet .
```

Recall that the sort `ResultTriple` contains triples formed by a term, its type, and a substitution:

```
op {_,_,_} : Term Type Substitution -> ResultTriple .
```

The sort `ResultTripleSet` contains sets of such result triples and is defined as follows:

```
sort ResultTripleSet .
subsort ResultTriple < ResultTripleSet .
op empty : -> ResultTripleSet [ctor] .
op _|_ : ResultTripleSet ResultTripleSet -> ResultTripleSet
  [ctor assoc comm id: empty prec 65 format (d d n d)] .
eq X:ResultTriple | X:ResultTriple = X:ResultTriple .
```

If a non-identity substitution is provided in the fourth argument of `metaNarrowSearch`, then any computed substitution must be an instance of the provided one, i.e., we can restrict the computed narrowing sequences to some concrete shape. The `Qid` argument metarepresents the appropriate search arrow, similar to the `metaSearch` command (see Section 14.5.6). For the bounds, the first one is the maximum length of the narrowing sequences, the second one is the number of computed solutions, i.e., the depth of the narrowing tree, and the third one is the chosen solution (in order to provide all solutions in a sequential way, as many metalevel commands in Maude do).

For the `NARROWING-VENDING-MACHINE` system module introduced at the beginning of Section 20.3, the following `search` command considered above

```
Maude> (search [1] in NARROWING-VENDING-MACHINE :
  < M:Money > ~>* < a c > .)
```

can be specified at the metalevel as follows, where `'<_>[M:Money]` is the metarepresentation of the state `< M:Money >`, `'<_>['__[a.Item, c.Item]]` is the metarepresentation of the state `< a c >`, and we use the coherence completion of the `NARROWING-VENDING-MACHINE` module given above.

```
Maude> reduce in FULL-MAUDE :
  metaNarrowSearch(
    axCohComplete(
      upModule('NARROWING-VENDING-MACHINE, false)),
    '<_>[M:Money],
    '<_>[ '__[a.Item, c.Item]],
    none, '*', unbounded, 1, unbounded) .
result ResultTriple:
{ '<_>[ '__[a.Item, c.Item, empty.Money]],
  'State,
  '#1:Marking <- ' __[q.Coin, q.Coin, q.Coin, empty.Money] ;
  '#4:Marking <- ' __[a.Item, empty.Money] ;
  '#6:Money <- 'empty.Money ;
  '%3:Money <- ' __[q.Coin, q.Coin, q.Coin, empty.Money] ;
  'M:Money <- ' __['$.Coin, ' __[q.Coin, q.Coin, q.Coin, empty.Money]] }
```

Note that we obtain the very same solution, where the output contains one term of type `ResultTriple`, which contains the actual output term, its type, and the computed substitution, and

```
'M:Money <- ' __['$.Coin, ' __[q.Coin, q.Coin, q.Coin]]
```

is the metarepresentation of the substitution `M:Money --> $ q q q`. Moreover, the substitutions computed in this way also contain all the temporary bindings that narrowing has computed.

Moreover, we can also obtain the narrowing sequence associated to a narrowing-based reachability command with the function `metaNarrowSearchPath` provided in the same module `META-NARROWING-SEARCH` with the following declaration:

```
op metaNarrowSearchPath :
  Module Term Term Substitution Qid Bound Bound
  Bound -> TraceNarrowSet .
```

It works in exactly the same way as `metaNarrowSearch` but providing as a result a more detailed data structure. If we redo the previous `metaNarrowSearch` computation but using this time the `metaNarrowSearchPath` function, we obtain:

```
Maude> reduce in FULL-MAUDE :
  metaNarrowSearchPath(
    axCohComplete(upModule('NARROWING-VENDING-MACHINE, false)),
    '<_>[M:Money],
    '<_>[ '__[a.Item, c.Item]],
    none, '*', unbounded, 1, unbounded) .
```

```

result TraceNarrow:
{ '<_>[ '__[%3:Money, '__[ 'q.Coin, 'a.Item]]],
  '#1:Marking <- '%3:Money ;
  'M:Money <- '__[ '$.Coin, '%3:Money],
  'State,
  rl '<_>[ '__[ '$.Coin, '#1:Marking]]
  => '<_>[ '__[ '#1:Marking, '__[ 'q.Coin, 'a.Item]]]
  [label('buy-a)] . }
{ '<_>[ '__[ 'c.Item, '__[ 'a.Item, '#6:Money]]],
  '#4:Marking <- '__[ 'a.Item, '#6:Money] ;
  '%3:Money <- '__[ 'q.Coin, 'q.Coin, 'q.Coin, '#6:Money],
  'State,
  rl '<_>[ '__[ '$.Coin, '#4:Marking]]
  => '<_>[ '__[ 'c.Item, '#4:Marking]]
  [label('buy-c)] .}

```

The data structure `TraceNarrow`, which is the basic element of `TraceNarrowSet`, is very similar to the data structure `ResultTriple` but it contains a sequence of narrowing results instead of only the final result, each one together with the rule that has been used in that narrowing step.

20.3.3 Paramodulation

The difference between a rewriting step and a narrowing step is that in both cases we use a rewrite rule $l \rightarrow r$ to rewrite t at a position p in t , but narrowing unifies the lefthand side l and the chosen subject term $t|_p$ before actually performing the rewriting step. Rewriting does not impose any restriction on the chosen subterm $t|_p$ nor the lefthand side l but narrowing imposes that both cannot be a variable, i.e.:

- the chosen subterm $t|_p$ of a narrowing step *cannot be a variable*, and
- the lefthand side l *cannot be a variable*.

However, there are interesting cases where these two restrictions are not desirable, for example the *paramodulation* inference rule used in paramodulation-based theorem proving [277] does not require them (see [363] for an example of use of this paramodulation feature for theorem proving).

The function `metaParamodulationSearch` provides an extension at the metalevel of the narrowing-based search command without these two restrictions. It is declared again in the same module `META-NARROWING-SEARCH` and has the same parameters as `metaNarrowSearch`:

```

op metaParamodulationSearch :
  Module Term Term Substitution Qid Bound Bound Bound -> ResultTripleSet .

```

Let us consider yet another version of the `NARROWING-VENDING-MACHINE` system module where we do not use the symbol `<_>` for restricting rewriting:

```

mod NARROWING-VM-NOTOP is
  sorts Coin Item Marking Money State .
  subsort Coin < Money .
  op empty : -> Money .
  op _ : Money Money -> Money [assoc comm id: empty] .
  subsort Money Item < Marking .
  op _ : Marking Marking -> Marking [assoc comm id: empty] .
  subsort Marking < State .
  ops $ q : -> Coin . ops a c : -> Item .
  var M : Marking .
  rl [buy-c] : $ => c .
  rl [buy-a] : $ => a q .
  eq [change]: q q q q M = $ M [variant] .
endm

```

Then we can perform the following search command that provides a variable as the starting term, without using the `<_>` symbol anymore:

```

Maude> red in FULL-MAUDE :
  metaParamodulationSearch(
    axCohComplete(upModule('NARROWING-VM-NOTOP, false)),
    'M:Money,
    '_[ 'a.Item, 'c.Item],
    none, '*', 1, unbounded, unbounded) .

```

Note that we obtain the same solution that we computed by means of `metaNarrowSearch`.

Object-Oriented Modules

In Full Maude, concurrent object-oriented systems can be defined by means of *object-oriented modules*—introduced by the keyword `omod . . . endom`—using a syntax more convenient than that of system modules, because it assumes acquaintance with the basic entities, such as objects, messages and configurations, and supports linguistic distinctions appropriate for the object-oriented case.

As in Core Maude, we may have specifications of object-oriented systems in system modules; for example, we could enter into Full Maude the system modules describing object-based systems discussed in Chapter 11 by enclosing them in parentheses. However, although Maude’s system modules are sufficient for specifying object-oriented systems, there are important conceptual advantages provided by Full Maude’s syntax for object-oriented modules. Such syntax allows the user to think and express his/her thoughts in object-oriented terms whenever such a viewpoint seems best suited for the problem at hand. Those conceptual advantages would be partially lost if only system modules at the Core Maude level were provided.

Object-oriented modules are however just syntactic sugar: they are internally transformed into system modules for execution purposes (Section 21.9). All object-oriented modules implicitly include the `CONFIGURATION` module (see Section 11.1), and thus assume the latter’s syntax. Recall that the module `CONFIGURATION` defines the basic concepts of concurrent object systems; among others, and besides the `Configuration` sort, it includes the declarations of sorts

- `Objid` of object identifiers,
- `Cid` of class identifiers,
- `Object` for objects, and
- `Msg` for messages.

21.1 Object-oriented systems

Some object-oriented concepts were introduced in Chapter 11. Here we recall some of them and then focus on the notions of class and inheritance, and on the additional syntactic facilities provided by Full Maude to support object-oriented programming.

21.1.1 Objects and messages

As in Core Maude, an *object* in a given state is represented as a term of the form

$$\langle 0 : C \mid \langle att-1 \rangle, \dots, \langle att-n \rangle \rangle$$

but Full Maude supports and enforces a specific choice for the syntax of attributes. Each attribute of sort `Attribute` consists of a *name* (attribute identifier), followed by a colon ‘:’ (with white space both before and after), followed by its *value*, which must have a given sort. Therefore, the Full Maude syntax for objects is

$$\langle 0 : C \mid a_1 : v_1, \dots, a_n : v_n \rangle$$

where `0` is the object’s name or identifier, `C` is its class identifier, the `ai`’s are the names of the object’s *attribute identifiers*, and the `vi`’s are the corresponding *values*, for $i = 1 \dots n$. In particular, an object with no attributes can be represented as

$$\langle 0 : C \mid \rangle$$

Messages do not have a fixed syntactic form; their syntax is instead defined by the user for each application. The only assumption made by the system is that the first argument of a message is the identifier of its destination object. Messages satisfying this requirement should be declared using the `msg` keyword. It is still possible to declare messages not following this requirement as operators of sort `Msg`; but, if declared as operators, no `message` attribute will be provided for them (see Sections 11.1 and 11.2). For example, the following declarations of messages are possible.

```
msg credit : Oid Nat -> Msg .
op left : -> Msg .
```

The concurrent state of an object-oriented system is then a multiset of objects and messages, of sort `Configuration`, with multiset union described with empty syntax `__`, and with `assoc`, `comm`, and `id: none` as operator attributes.

21.1.2 Classes

Classes are defined with the keyword `class`, followed by the name of the class, followed by a bar ‘|’, followed by a list of attribute declarations separated by commas. Each attribute declaration has the form $a : S$, where a is an attribute identifier and S is the sort in which the values of the attribute identifier range. That is, class declarations have the form

```
class C | a1 : ⟨Sort-1⟩, ... , an : ⟨Sort-n⟩ .
```

In particular, we can declare classes without attributes using syntax

```
class C .
```

Class names have the same form as sort names. In particular, class names may be parameterized in a way completely similar to parameterized sort names (see Sections 8.3.3, 21.4, and 21.5).

As an example of class declaration, the class `Account` of bank account objects with a `balance` attribute, introduced in Section 11.1, is now declared as follows:

```
class Account | bal : Int .
```

As another example, a class `Person`, with a name, an age, and a bank account as attributes can then be declared as follows:

```
class Person | name : String, age : Nat, account : Oid .
```

In this case, a person has a reference to his/her account in an `account` attribute of sort `Oid`.

All Full Maude object-oriented modules have an operation `class` that takes an object as argument and returns its actual class. Thus, for example,

```
class(< A-002 : Account | bal : 1000 >)
```

returns the class identifier `Account`. This operation will be particularly useful when combined with the inheritance relation (see the use of the `class` operation in the example in Section 21.6).

The syntax for message declarations is similar to the syntax for the declaration of operators, but using `msg` and `msgs` instead of `op` and `ops`, and having as result sort `Msg` or a subsort of it. Thus, `msg` is used to declare a single message, and `msgs` may be used for declaring multiple messages. The user can introduce subsorts of the predefined sort `Msg`, so that it is possible to declare messages of different types. This may be useful for restricting the kind of messages that could be received by a particular type of object. As in the case of operators, messages can be overloaded and can be declared with operator attributes.

In the account example, the three kinds of messages involving accounts are `credit`, `debit`, and `from_to_transfer_`, whose user-definable syntax is introduced in the following declarations:

```

msgs credit debit : Oid Nat -> Msg .
msg from_to_transfer_ : Oid Oid Nat -> Msg .

```

Notice the use of the `Oid` sort for specifying the addressee of a message, as in `credit` and `debit`, and for specifying the objects involved in a message, e.g., the source and the target accounts of an account transfer in the `from_to_transfer_` message. Note that, as explained in Chapter 11 for Core Maude, Full Maude also assumes that the message's destination object is the first argument mentioned in the message declaration. This convention is needed by the object-message fair rewriting strategy (see Section 11.2). The behavior associated with the messages is then specified by rewrite rules in a declarative way (see Section 21.1.4).

Given object identifiers `Smith` and `A-002`, the following term may represent a configuration with a person, his account, and a `credit` message sent to it.

```

< Smith : Person | name : "John", age : 34, account : A-002 >
< A-002 : Account | bal : 1000 >
credit(A-002, 100)

```

21.1.3 Inheritance

Class inheritance is directly supported by Maude's order-sorted type structure. Since class names are a particular case of sort names, a subclass declaration $C < C'$ in an object-oriented module is just a particular case of a subsort declaration $C < C'$. The effect of a subclass declaration is that the attributes, messages, and rules of all the superclasses, together with the newly defined attributes, messages, and rules of the subclass, characterize the structure and behavior of the objects in the subclass.

For example, we can define an object-oriented module `SAVING-ACCOUNT` of saving accounts introducing a subclass `SavingAccount` of `Account` with a new attribute `rate` recording the interest rate of the account.

```

class SavingAccount | rate : Float .
subclass SavingAccount < Account .

```

In this example there is only one class immediately above `SavingAccount`, namely, `Account`. In general, however, a class C may be defined as a subclass of several classes D_1, \dots, D_k , i.e., *multiple inheritance* is supported. If an attribute and its sort have already been declared in a superclass, they should not be declared again in the subclass. Indeed, all such attributes are *inherited*. In the case of multiple inheritance, the only requirement that is made is that if an attribute occurs in two different superclasses, then the sort associated with it in each of those superclasses must be the same.¹ In summary, a class inherits

¹ If a class inherits from two different superclasses that share an attribute but with different associated sorts, then both attributes are inherited in the subclass, thus muddling them up.

all the attributes, messages, and rules from all its superclasses. An object in the subclass behaves exactly as any object in any of the superclasses, but it may exhibit additional behavior due to the introduction of new attributes, messages, and rules in the subclass.

Objects in the class `SavingAccount` will have an attribute `bal` and can receive messages debiting, crediting and transferring funds exactly as any other object in the class `Account`. For example, the following object is a valid instance of class `SavingAccount`.

```
< A-002 : SavingAccount | bal : 5000, rate : 3.0 >
```

As for subort relationships, we can declare multiple subclass relationships in the same declaration. Thus, given classes `A, . . . , H`, we can have a declaration of subclasses such as

```
subclasses A B C < D E < F G H .
```

21.1.4 Object-oriented rules

The behavior associated with messages is specified by rewrite rules in a declarative way. For example, the semantics of the `credit`, `debit`, and `from_to_transfer` messages declared in Section 21.1.2 may be given as follows:

```
vars A B : Oid .
var M : Nat .
vars N N' : Int .

r1 [credit] :
  credit(A, M)
  < A : Account | bal : N >
  => < A : Account | bal : N + M > .

crl [debit] :
  debit(A, M)
  < A : Account | bal : N >
  => < A : Account | bal : N - M >
  if N >= M .

crl [transfer] :
  (from A to B transfer M)
  < A : Account | bal : N >
  < B : Account | bal : N' >
  => < A : Account | bal : N - M >
  < B : Account | bal : N' + M >
  if N >= M .
```

Note that the multiset structure of the configuration provides the top-level distributed structure of the system and allows concurrent application of the

rules. The reader is referred to [245] for a detailed explanation of the logical semantics of the object-oriented model of computation supported by Maude.

In object-oriented modules it is possible not to mention in a given rule those attributes of an object that are not relevant for that rule. The attributes mentioned only on the lefthand side of a rule are preserved unchanged, the original values of attributes mentioned only on the righthand side do not matter, and all attributes not explicitly mentioned are left unchanged (see Section 21.9 for more details). For instance, a message for changing the age of a person defined by the class `Person` (introduced in Section 21.1.2) may be defined as follows:

```
msg to_:'new'age_ : Oid Nat -> Msg .
var A : Nat .
var O : Oid .

rl [change-age] :
  < O : Person | >
  to O : new age A
  => < O : Person | age : A > .
```

The attributes `name` and `account`, which are not mentioned in this rule, are not changed when applying the rule. The value of the `age` attribute is replaced by the given new age, independently of its previous value.

The following module `ACCOUNT` contains all the declarations above defining the class `Account`. Note that `Qid` is declared as a subsort of `Oid`, making any quoted identifier a valid object identifier.

```
(omod ACCOUNT is
  protecting QID .
  protecting INT .

  subsort Qid < Oid .
  class Account | bal : Int .
  msgs credit debit : Oid Int -> Msg .
  msg from_to_transfer_ : Oid Oid Int -> Msg .

  vars A B : Oid .
  var M : Nat .
  vars N N' : Int .

  rl [credit] :
    credit(A, M)
    < A : Account | bal : N >
    => < A : Account | bal : N + M > .

  crl [debit] :
    debit(A, M)
    < A : Account | bal : N >
```

```

=> < A : Account | bal : N - M >
if N >= M .

crl [transfer] :
  (from A to B transfer M)
  < A : Account | bal : N >
  < B : Account | bal : N' >
  => < A : Account | bal : N - M >
     < B : Account | bal : N' + M >
  if N >= M .
endom)

```

We can now rewrite a simple configuration consisting of an account and a message as follows:

```

Maude> (rew < 'A-06238 : Account | bal : 2000 >
        debit('A-06238, 1000) .)

result Object :
  < 'A-06238 : Account | bal : 1000 >

```

The following module contains the declarations for the class `SavingAccount`.

```

(omod SAVING-ACCOUNT is
  including ACCOUNT .
  protecting FLOAT .
  class SavingAccount | rate : Float .
  subclass SavingAccount < Account .
endom)

```

We leave unspecified the rules for computing and crediting the interest of an account according to its rate, whose proper expression should introduce a real-time² attribute in account objects.

We can now rewrite a configuration, obtaining the following result.

```

Maude> (rew < 'A-73728 : SavingAccount | bal : 5000, rate : 3.0 >
        < 'A-06238 : Account | bal : 2000 >
        < 'A-28381 : SavingAccount | bal : 9000, rate : 3.0 >
        debit('A-06238, 1000)
        credit('A-73728, 1300)
        credit('A-28381, 200) .)

result Configuration :
  < 'A-06238 : Account | bal : 1000 >
  < 'A-73728 : SavingAccount | bal : 6300, rate : 3.0 >
  < 'A-28381 : SavingAccount | bal : 9200, rate : 3.0 >

```

² See [288, 291] for a general methodology to specify real-time systems, including object-oriented ones, in rewriting logic.

We can also search over configurations. In this case the search pattern takes into account object-oriented information, finding also states where a subclass matches the pattern. For example, we can look for final states having accounts with balance less than 8000 with the following command:

```
Maude> (search in SAVING-ACCOUNT :
  < 'A-73728 : SavingAccount | bal : 5000, rate : 3.0 >
  < 'A-06238 : Account | bal : 2000 >
  < 'A-28381 : SavingAccount | bal : 9000, rate : 3.0 >
  debit('A-06238, 1000)
  credit('A-73728, 1300)
  credit('A-28381, 200)
=>! C:Configuration
  < O:Oid : Account | bal : N:Nat >
  such that N:Nat < 8000 .)
search in SAVING-ACCOUNT :
  < 'A-73728 : SavingAccount | bal : 5000, rate : 3.0 >
  < 'A-06238 : Account | bal : 2000 >
  < 'A-28381 : SavingAccount | bal : 9000, rate : 3.0 >
  debit('A-06238, 1000)
  credit('A-73728, 1300)
  credit('A-28381, 200)
=>! C:Configuration
  < O:Oid : V#0:Account | bal : N:Nat, V#1:AttributeSet > .
```

Solution 1

```
C:Configuration -->
  < 'A-28381 : SavingAccount | bal : 9200,rate : 3.0 >
  < 'A-73728 : SavingAccount | bal : 6300,rate : 3.0 > ;
N:Nat --> 1000 ;
O:Oid --> 'A-06238 ;
V#0:Account --> Account ;
V#1:AttributeSet --> (none).AttributeSet
```

Solution 2

```
C:Configuration -->
  < 'A-06238 : Account | bal : 1000 >
  < 'A-28381 : SavingAccount | bal : 9200, rate : 3.0 > ;
N:Nat --> 6300 ;
O:Oid --> 'A-73728 ;
V#0:Account --> SavingAccount ;
V#1:AttributeSet --> rate : 3.0
```

No more solutions.

Notice that the search pattern has been transformed so that objects in subclasses match. In this example, we obtain as solutions both an object of class `Account` and an object in the subclass `SavingAccount`.

7	1	2
6		8
5	4	3

7	8	1
6		2
5	4	3

Fig. 21.1. Possible initial and final states for the 8-puzzle

21.2 More examples of object-oriented modules

We introduce several additional examples to illustrate the use of object-oriented modules in Full Maude.

21.2.1 A puzzle

We show in this section an object-oriented approach to a typical artificial intelligence planning problem, the 8-puzzle problem as presented in [170, page 283]. There is a set of eight numbered tiles and a blank tile, arranged in a 3×3 -grid. By moving the blank tile up, down, left, or right, the goal is to reach the state pictured on the righthand side of Figure 21.1, starting in an arbitrary state, like for example the one pictured on the lefthand side.

This puzzle can be solved using the ideas described in Chapter 7, and more concretely in the specification of the Khun Phan puzzle (Section 7.7), which also involves moving tiles; here we present instead an object-oriented approach.

Each tile is represented as an object $\langle (R,C) : \text{Tile} \mid \text{value} : N \rangle$, where R and C denote the row (from top to bottom) and column (from left to right), respectively, and N denotes either a natural number between 1 and 8, or the marker `blank`. We want useful coordinates to range between 1 and 3, and number values to range between 1 and 8, thus there are some ad-hoc overloaded constants. For the values there is no operation, while we need to check adjacent tiles using an auxiliary successor function `s_` on the `Coordinate` sort.

```
(omod 8-PUZZLE is
  sorts NumValue Value Coordinate .
  subsort NumValue < Value .
  ops 1 2 3 4 : -> Coordinate [ctor] .
  ops 1 2 3 4 5 6 7 8 : -> NumValue [ctor] .
  op blank : -> Value [ctor] .
  op s_ : Coordinate -> Coordinate .
  eq s 1 = 2 .
  eq s 2 = 3 .
  eq s 3 = 4 .
  eq s 4 = 4 .
  op '(_','_') : Coordinate Coordinate -> Oid .
  class Tile | value : Value .
```

```

msgs left right up down : -> Msg .
vars N M R : Coordinate .
var P : NumValue .

```

Moves are represented as messages sent to the configuration of tiles. Note that these messages are declared as operators of sort `Msg`. We have one rule for each message, so that each rule involves the blank tile and the neighbor tile, according to the direction of the move.

```

crl [r] :
  right
  < (N, R) : Tile | value : blank >
  < (N, M) : Tile | value : P >
  => < (N, R) : Tile | value : P >
      < (N, M) : Tile | value : blank >
  if R = s M .

crl [l] :
  left
  < (N, R) : Tile | value : blank >
  < (N, M) : Tile | value : P >
  => < (N, R) : Tile | value : P >
      < (N, M) : Tile | value : blank >
  if s R = M .

crl [u] :
  up
  < (R, M) : Tile | value : blank >
  < (N, M) : Tile | value : P >
  => < (R, M) : Tile | value : P >
      < (N, M) : Tile | value : blank >
  if s R = N .

crl [d] :
  down
  < (R, M) : Tile | value : blank >
  < (N, M) : Tile | value : P >
  => < (R, M) : Tile | value : P >
      < (N, M) : Tile | value : blank >
  if R = s N .

```

The initial and final states in Figure 21.1 are represented by the following tile configurations:

```

ops initial final : -> Configuration .
eq initial
  = < (1, 1) : Tile | value : 7 >
      < (1, 2) : Tile | value : 1 >
      < (1, 3) : Tile | value : 2 >
      < (2, 1) : Tile | value : 6 >

```

```

    < (2, 2) : Tile | value : blank >
    < (2, 3) : Tile | value : 8 >
    < (3, 1) : Tile | value : 5 >
    < (3, 2) : Tile | value : 4 >
    < (3, 3) : Tile | value : 3 > .

eq final
= < (1, 1) : Tile | value : 7 >
  < (1, 2) : Tile | value : 8 >
  < (1, 3) : Tile | value : 1 >
  < (2, 1) : Tile | value : 6 >
  < (2, 2) : Tile | value : blank >
  < (2, 3) : Tile | value : 2 >
  < (3, 1) : Tile | value : 5 >
  < (3, 2) : Tile | value : 4 >
  < (3, 3) : Tile | value : 3 > .
endom)

```

A plan that solves the planning problem in Figure 21.1 is (in a self-explanatory intuitive notation) `up`; `left`; `down`; `right`. However, one must be aware that this is a *sequential* plan, while messages in the tile configuration do not have any order upon them because of the structural axioms of associativity and commutativity. Therefore, this is another example where strategies are useful for controlling the application of rewrite rules in the order wanted by the user (see Sections 14.6 and 21.7).

Instead of applying a particular strategy, one can simply check that this plan works by using the `search` command:

```

Maude > (search up left down right initial =>* C:Configuration
        such that C:Configuration == final .)
rewrites: 974 in 50ms cpu (68ms real) (19480 rewrites/second)

```

```

Solution 1
C:Configuration --> < (1, 1) : Tile | value : 7 >
                   < (1, 2) : Tile | value : 8 >
                   < (1, 3) : Tile | value : 1 >
                   < (2, 1) : Tile | value : 6 >
                   < (2, 2) : Tile | value : blank >
                   < (2, 3) : Tile | value : 2 >
                   < (3, 1) : Tile | value : 5 >
                   < (3, 2) : Tile | value : 4 >
                   < (3, 3) : Tile | value : 3 >

```

No more solutions.

21.2.2 A simple spreadsheet

The following object-oriented module specifies the concurrent behavior of objects in a simple class `Cell` of cells in a spreadsheet, whose unique attribute

is the value stored in the cell. Cells are organized in a grid and are therefore identified by means of pairs (N,M) giving the row and column numbers. For each row N there is a cell (N,total) that keeps track of the corresponding total, and similarly for each column M there is a cell (total,M) . There is also a cell $(\text{total},\text{total})$ providing the sum of all the values in all the cells in the spreadsheet. The spreadsheet may receive messages $\text{add}(N,M,V)$ and $\text{sub}(N,M,V)$ for adding or subtracting (if possible) the amount V to the value stored in cell (N,M) .

```
(omod SPREADSHEET is
  protecting NAT .
  sort Name .
  subsort Nat < Name .
  op total : -> Name [ctor] .
  op '(_','_') : Name Name -> Oid .
  class Cell | val : Nat .
  msgs add sub : Nat Nat Nat -> Msg .
  vars M N V W X Y Z : Nat .

  r1 [add] :
    add(N, M, V)
    < (N, M) : Cell | val : W >
    < (total, total) : Cell | val : X >
    < (N, total) : Cell | val : Y >
    < (total, M) : Cell | val : Z >
    => < (N, M) : Cell | val : W + V >
        < (total, total) : Cell | val : X + V >
        < (N, total) : Cell | val : Y + V >
        < (total, M) : Cell | val : Z + V > .

  cr1 [sub] :
    sub(N, M, V)
    < (N, M) : Cell | val : W >
    < (total, total) : Cell | val : X >
    < (N, total) : Cell | val : Y >
    < (total, M) : Cell | val : Z >
    => < (N, M) : Cell | val : sd(W, V) >
        < (total, total) : Cell | val : sd(X, V) >
        < (N, total) : Cell | val : sd(Y, V) >
        < (total, M) : Cell | val : sd(Z, V) >
    if V <= W .
endom)
```

We show here how to transform synchronous object-oriented rules, like the ones in the module above, into asynchronous rules where there is only one object in the rule's lefthand side. The essential idea is to introduce new messages in the righthand side of the rules, creating new states in which the original computation is unfinished and is going to continue by further interaction of the new messages with the objects.

```

(omod SPREADSHEET-ASYNCH is
  protecting NAT .
  sort Name .
  subsort Nat < Name .
  op total : -> Name [ctor] .
  op '(_','_') : Name Name -> Oid .
  class Cell | val : Nat .
  msgs add sub : Nat Nat Nat -> Msg .
  msgs add-row add-col : Nat Nat -> Msg .
  msgs sub-row sub-col : Nat Nat -> Msg .
  msgs add-total sub-total : Nat -> Msg .
  vars M N V W : Nat .

  rl [add] :
    add(N, M, V)
    < (N, M) : Cell | val : W >
    => < (N, M) : Cell | val : W + V >
        add-row(N, V)
        add-col(M, V)
        add-total(V) .
  rl [add-row] :
    add-row(N, V)
    < (N, total) : Cell | val : W >
    => < (N, total) : Cell | val : W + V > .
  rl [add-col] :
    add-col(M, V)
    < (total, M) : Cell | val : W >
    => < (total, M) : Cell | val : W + V > .
  rl [add-total] :
    add-total(V)
    < (total, total) : Cell | val : W >
    => < (total, total) : Cell | val : W + V > .

  crl [sub] :
    sub(N, M, V)
    < (N, M) : Cell | val : W >
    => < (N, M) : Cell | val : sd(W, V) >
        sub-row(N, V)
        sub-col(M, V)
        sub-total(V)
    if V <= W .
  rl [sub-row] :
    sub-row(N, V)
    < (N, total) : Cell | val : W >
    => < (N, total) : Cell | val : sd(W, V) > .
  rl [sub-col] :
    sub-col(M, V)
    < (total, M) : Cell | val : W >
    => < (total, M) : Cell | val : sd(W, V) > .

```

```

rl [sub-total] :
  sub-total(V)
  < (total, total) : Cell | val : W >
  => < (total, total) : Cell | val : sd(W, V) > .
endom)

```

Because of the presence of new messages, there are configurations in the module `SPREADSHEET-ASYNCH` that do not correspond to any configuration in the original module `SPREADSHEET`. However, in any computation using the new rules that starts in a configuration from `SPREADSHEET` the new messages will eventually disappear by application of the new rules involving those messages on the lefthand side, reaching in this way a configuration in `SPREADSHEET`. Moreover, this configuration is exactly the same achieved by the original synchronous rules.

21.2.3 Blocks world

This is an object-oriented approach to a simpler version of the blocks world described in Section 6.5.2. Here, we have removed the robot arm to move blocks. A block is represented as an object with two attributes, `under`, saying whether it is under another block or it is clear, and `on`, saying whether the block is on top of another block or is on the table. Actions are represented as messages.

```

(omod 00-BLOCKS-WORLD is
  protecting QID .
  sorts BlockId Up Down .
  subsorts Qid < BlockId < Oid .
  subsorts BlockId < Up Down .
  op clear : -> Up [ctor] .
  op table : -> Down [ctor] .
  class Block | under : Up, on : Down .
  msg move : Oid Oid Oid -> Msg .
  msgs unstack stack : Oid Oid -> Msg .
  vars X Y Z : BlockId .

  rl [move] :
    move(X, Z, Y)
    < X : Block | under : clear, on : Z >
    < Z : Block | under : X >
    < Y : Block | under : clear >
    => < X : Block | on : Y >
        < Z : Block | under : clear >
        < Y : Block | under : X > .

  rl [unstack] :
    unstack(X,Z)
    < X : Block | under : clear, on : Z >
    < Z : Block | under : X >

```

```

=> < X : Block | on : table >
    < Z : Block | under : clear > .
rl [stack] :
  stack(X, Z)
  < X : Block | under : clear, on : table >
  < Z : Block | under : clear >
=> < X : Block | on : Z >
    < Z : Block | under : X > .

```

The states I and F in Figure 6.5 (without the robot arm) are respectively described now by the following two configurations:

```

ops initial final : -> Configuration .
eq initial
  = < 'a : Block | under : 'c, on : table >
    < 'c : Block | under : clear, on : 'a >
    < 'b : Block | under : clear, on : table > .

eq final
  = < 'a : Block | under : clear, on : 'b >
    < 'b : Block | under : 'a, on : 'c >
    < 'c : Block | under : 'b, on : table > .
endom)

```

To check that the “sequential plan” (again in an intuitive notation)

```
unstack(c, a); stack(b, c); stack(a, b)
```

moves objects from the `initial` to the `final` configuration, we can use a `search` command as follows:

```

Maude> (search unstack('c, 'a) stack('b, 'c) stack('a, 'b) initial
=>* C:Configuration
      such that C:Configuration == final .)
rewrites: 1318 in 20ms cpu (107ms real) (65900 rews/sec)

```

Solution 1

```

C:Configuration --> < 'a : Block | on : 'b, under : clear >
                   < 'b : Block | on : 'c, under : 'a >
                   < 'c : Block | on : table, under : 'b >

```

No more solutions.

Suppose that the blocks world is further refined, so that now blocks can have *colors*, say red, blue, and yellow. Of course, we want the rules for manipulating blocks to remain “exactly as before.” This is trivially achieved by class inheritance as illustrated by the following module, where the rules given for the class `Block` of blocks also apply without changes to blocks in the subclass `ColoredBlock` of colored blocks.

```

(omod OO-BLOCKS-WORLD-COLOR is
  including OO-BLOCKS-WORLD .
  sort Color .
  ops red blue yellow : -> Color [ctor] .
  class ColoredBlock | color : Color .
  subclass ColoredBlock < Block .
endom)

Maude> (rewrite
  unstack('c, 'a)
  < 'a : Block | color : red, under : 'c, on : table >
  < 'c : Block | color : blue, under : clear, on : 'a >
  < 'b : Block | color : yellow, under : clear, on : table > .)
result Configuration :
  < 'a : Block | color : red,    on : table, under : clear >
  < 'b : Block | color : yellow, on : table, under : clear >
  < 'c : Block | color : blue,   on : table, under : clear >

```

21.3 A bigger example: a rent-a-car store

In order to further illustrate Full Maude's object-oriented features, we specify a simple rent-a-car store example. Several rules in this specification have variables in their righthand sides or conditions not present in their lefthand sides; therefore, these rules are not directly executable by the rewrite engine and are declared as nonexecutable. In order to run the object-oriented system, we will have to use strategies; a possible such strategy will be presented in Section 21.7.

The regulations of the system, especially those that govern the rental processes, are the following:

1. Cars are rented for a specific number of days, after which they should be returned.
2. A car can be rented only if it is available.
3. No credit is allowed; customers must pay cash.
4. Customers must make a deposit at pick-up time of the estimated rental charges.
5. Rental charges depend on the car class. There are three categories: economy, mid-size, and full-size cars.
6. When a rented car is returned, the deposit is used to pay the rental charges.
7. If a car is returned before the due date, the customer is charged only for the number of days the car has been used. The rest of the deposit is reimbursed to the customer.
8. Customers who return a rented car after its due date are charged for all the days the car has been used, with an additional 20% charge for each day after the due date.

9. Failure to return the car on time or to pay a debt may result in the suspension of renting privileges.

Let us begin with the static aspects of this system, i.e., with its structure. We can identify three main classes, namely the store, customer, and car classes. There are three kinds of cars: economy, mid-size, and full-size cars.

Customers may rent cars. This relationship may be represented by a `Rental` class which, in addition to references to the objects involved in the relationship, has some extra attributes. The system also requires some control over time, which we get with a class representing calendars that provides the current date and simulates the passage of time.

The `Customer` class has three attributes, namely, `suspended`, `cash`, and `debt` to keep track of, respectively, whether he is suspended or not, the amount of cash that the customer currently has, and his debt with the store. Such a class is defined by the following Maude declaration:

```
class Customer | cash : Nat, debt : Nat, suspended : Bool .
```

The attribute `available` of the `Car` class indicates whether the car is currently available or not, and `rate` records the daily rental rate. We model the different types of cars for rent by three different subclasses, namely, `EconomyCar`, `MidSizeCar` and `FullSizeCar`.

```
class Car | available : Bool, rate : Nat .
class EconomyCar .
class MidSizeCar .
class FullSizeCar .
subclasses EconomyCar MidSizeCar FullSizeCar < Car .
```

Each object of class `Rental` will establish a relationship between a customer and a car, whose identifiers are kept in attributes `customer` and `car`, respectively. In addition to these, the class `Rental` is also declared with attributes `deposit`, `pickUpDate`, and `dueDate` to store, respectively, the amount of money left as a deposit by the customer, the date in which the car is picked up by the customer, and the date in which the car should be returned to the store.

```
class Rental | deposit : Nat, dueDate : Nat, pickUpDate : Nat,
              customer : Oid, car : Oid .
```

Given the simple use that we are going to make of dates, we can represent them, for example, as natural numbers. Then, we may have a calendar object that keeps the current date and gets increased by a rewrite rule as follows:

```
class Calendar | date : Nat .
rl [new-day] :
  < 0 : Calendar | date : F >
=> < 0 : Calendar | date : F + 1 > .
```

We do not worry here about the frequency with which the date gets increased, the possible synchronization problems in a distributed setting, or with any other issues related to the specification of time. See the papers [288, 291] on the specification of real-time systems in rewriting logic and Maude for a discussion on these issues.

Four actions can be identified in our example:

- a customer rents a car,
- a customer returns a rented car,
- a customer is suspended for being late in paying her debt or for being late in returning a rented car, and
- a customer pays (part of) her debt.

The rental of a car by a customer is specified by the `car-rental` rule below, which involves the customer renting the car, the car itself (which must be available, i.e., not currently rented), and a calendar object supplying the current date. The rental can take place if the customer is not suspended, that is, if her identifier is not in the set of identifiers of suspended customers of the store, and if the customer has enough cash to make the corresponding deposit. Notice that a customer could rent a car for less time she really is going to use it on purpose, because either she does not have enough money for the deposit, or prefers making a smaller deposit. According to the description of the system, the payment takes place when returning the car, although there is an extra charge for the days the car was not reserved.

```

cr1 [car-rental] :
  < U : Customer | cash : M, suspended : false >
  < I : Car | available : true, rate : Rt >
  < C : Calendar | date : Today >
=> < U : Customer | cash : M - Amnt >
    < I : Car | available : false >
    < C : Calendar | >
    < A : Rental | pickUpDate : Today, dueDate : Today + NumDays,
      car : I, deposit : Amnt, customer : U, rate : Rt >
  if Amnt := Rt * NumDays /\ M >= Amnt
  [nonexec] .

```

Note that, as already mentioned, those attributes of an object that are not relevant for a rule do not need to be mentioned. Attributes not appearing in the righthand side of a rule will maintain their previous values unmodified. Furthermore, since the variables `A` and `NumDays` appear in the righthand side or condition of the rule but not in its lefthand side, this rule has to be declared as `nonexec`. Note as well the use of the attributes `customer` and `car` in objects of class `Rental`, which makes explicit that a rental relationship is between the customer and the car specified by these attributes.

A customer returning a car late cannot be forced to pay the total amount of money due at return time. Perhaps she does not have such an amount of money in hand. The return of a rented car is specified by the rules below. The

first rule handles the case in which the car is returned on time, that is, the current date is smaller or equal to the due date, and therefore the deposit is greater or equal to the amount due.

```

crl [on-date-car-return] :
  < U : Customer | cash : M >
  < I : Car | rate : Rt >
  < A : Rental | customer : U, car : I, pickUpDate : PDt,
                dueDate : DDt, deposit : Dpst >
  < C : Calendar | date : Today >
=> < U : Customer | cash : (M + Dpst) - Amnt >
    < I : Car | available : true >
    < C : Calendar | >
  if (Today <= DDt) /\ Amnt := Rt * (Today - PDt)
  [nonexec] .

```

In this case, part of the deposit needs to be reimbursed. We can see that the `Rental` object disappears in the righthand side of the rule, that is, it is removed from the set of rentals and the availability of the car is restored.

The second rule deals with the case in which the car is returned late.

```

crl [late-car-return] :
  < U : Customer | debt : M >
  < I : Car | rate : Rt >
  < A : Rental | customer : U, car : I, pickUpDate : PDt,
                dueDate : DDt, deposit : Dpst >
  < C : Calendar | date : Today >
=> < U : Customer | debt : (M + Amnt) - Dpst >
    < I : Car | available : true >
    < C : Calendar | >
  if DDt < Today    *** it is returned late
    /\ Amnt := Rt * (DDt - PDt)
              + ((Rt * (Today - DDt)) * (100 + 20)) quo 100
  [nonexec] .

```

In this case, the customer's debt is increased by the portion of the amount due not covered by the deposit.

Debts may be paid at any time, the only condition being that the amount paid is between zero and the amount of money owed by the customer at that time.

```

crl [pay-debt] :
  < U : Customer | debt : M, cash : N >
=> < U : Customer | debt : M - Amnt, cash : N - Amnt >
  if 0 < Amnt /\ Amnt <= N /\ Amnt <= M
  [nonexec] .

```

Customers who are late in returning a rented car or in paying their debts “may” be suspended. The first rule deals with the case in which a customer

has a pending debt, and the second one handles the case in which a customer is late in returning a rented car.

```

crl [suspend-late-payers] :
  < U : Customer | debt : M, suspended : false >
  => < U : Customer | suspended : true >
  if M > 0 .

crl [suspend-late-returns] :
  < U : Customer | suspended : false >
  < I : Car | >
  < A : Rental | customer : U, car : I, dueDate : Ddt >
  < C : Calendar | date : Today >
  => < U : Customer | suspended : true >
  < I : Car | >
  < A : Rental | >
  < C : Calendar | >
  if Ddt < Today .

```

Since the system is not terminating, and there are several rules with variables in their righthand sides or conditions not present in their lefthand sides and not satisfying the admissibility conditions discussed in Section 6.3, strategies are necessary for controlling its execution. We can define many different strategies and use them in many different ways (see Section 14.6); a concrete possibility will be described later in Section 21.7.

21.4 Object-oriented parameterized programming

The notions of theory, view, and parameterized module (see Section 8.3) have been extended to the object-oriented case. In this section, we explain how to write object-oriented theories, views with object-oriented theories as sources and object-oriented modules or object-oriented theories as targets, and object-oriented parameterized modules with possibly object-oriented theories as parameters. In Section 21.5 we explain how the module operations available in Full Maude have been extended, so that they are also available on object-oriented modules. In particular, we will see how it is possible to rename an object-oriented module and to instantiate an object-oriented module parameterized by an object-oriented theory with a view having another object-oriented module as its target.

21.4.1 Theories

In addition to functional and system theories, Full Maude also supports *object-oriented theories*. Their structure is the same as that of object-oriented modules. Object-oriented theories can have classes, subclass relationships, and messages. These object-oriented notions may be useful for the definition of

theories; for example, the following theory `CELL` specifies the theory of classes with at least one attribute of any sort.

```
(oth CELL is
  sort Elt .
  class Cell | contents : Elt .
  endoth)
```

21.4.2 Views

For a view having an object-oriented theory as its source, the mapping of a class `C` in the source theory to a class `C'` in the target is expressed with syntax

```
class C to C' .
```

Attribute maps have the form

```
attr C . A to A' .
```

where `A` is the name of an attribute of class `C` in the source theory and `A'` is an attribute of the image class of `C` under the view.

The mapping of messages is expressed with syntax

```
msg M to M' .
```

where `M` is a message identifier or a message identifier together with its arity and value sort. As for operators, a message map in which explicit arity and coarity are given affects the entire family of subsort-overloaded message declarations associated with the declaration of the given message.

21.4.3 Parameterized object-oriented modules

Like any other type of module, object-oriented modules can be parameterized, and, like sort names, class names may also be parameterized. The naming of parameterized classes follows the same conventions discussed in Section 8.3.3 for parameterized sorts.

As an example of an object-oriented parameterized module, we define a stack of elements. We define a class `Stack{X}` as a linked sequence of node objects. Objects of class `Stack{X}` have a single attribute `first`, containing the identifier of the first node in the stack. If the stack is empty, the value of the `first` attribute is `null`. Each object of class `Node{X}` has an attribute `next` holding the identifier of the next node—which should be `null` if there is no next node—and an attribute `contents` to store a value of sort `X$Elt`. Notice that node identifiers are of the form `o(S, N)`, where `S` is the identifier of the stack object to which the node belongs, and `N` is a natural number. The messages `push`, `pop` and `top` have as their first argument the identifier of the object to which they are addressed, and will cause, respectively, the insertion at the top of the stack of a new element, the removal of the top element, and the sending of a response message `elt` containing the element at the top of the stack to the object making the request.

```

(omod OO-STACK{X :: TRIV} is
  protecting INT .
  protecting QID .
  subsort Qid < Oid .
  class Node{X} | next : Oid, contents : X$Elt .
  class Stack{X} | first : Oid .
  msg _push_ : Oid X$Elt -> Msg .
  msg _pop_ : Oid -> Msg .
  msg _top_ : Oid Oid -> Msg .
  msg _elt_ : Oid X$Elt -> Msg .

  op null : -> Oid .
  op o : Oid Int -> Oid .

  vars O O' O'' : Oid .
  var E : X$Elt .
  var N : Int .

  rl [top] : *** top on a non-empty stack
  < O : Stack{X} | first : O' >
  < O' : Node{X} | contents : E >
  (O top O'')
  => < O : Stack{X} | >
      < O' : Node{X} | >
      (O'' elt E) .

  rl [push1] : *** push on a non-empty stack
  < O : Stack{X} | first : o(O, N) >
  (O push E)
  => < O : Stack{X} | first : o(O, N + 1) >
      < o(O, N + 1) : Node{X} |
          contents : E, next : o(O, N) > .

  rl [push2] : *** push on an empty stack
  < O : Stack{X} | first : null >
  (O push E)
  => < O : Stack{X} | first : o(O, 0) >
      < o(O, 0) : Node{X} | contents : E, next : null > .

  rl [pop] : *** pop on a non-empty stack
  < O : Stack{X} | first : O' >
  < O' : Node{X} | next : O'' >
  (O pop)
  => < O : Stack{X} | first : O'' > .
endom)

```

Notice that `top` and `pop` messages are not received if the stack is empty.

We may want to define stacks storing not just data elements of a particular sort, but actually objects in a particular class. We can define an object-

oriented module with the intended behavior as the parameterized module `OO-STACK2` below, which is parameterized by the object-oriented theory `CELL` introduced in Section 21.4.1. Notice that the main difference with respect to the previous `STACK` version is in the attribute `node`, that keeps the identifier of the object where the contents can be found instead of the attribute `contents` that provided direct access to those contents.

```
(omod OO-STACK2{X :: CELL} is
  protecting INT .
  protecting QID .
  subsort Qid < Oid .
  class Node{X} | next : Oid, node : Oid .
  class Stack{X} | first : Oid .
  msg _push_ : Oid Oid -> Msg .
  msg _pop_ : Oid -> Msg .
  msg _top_ : Oid Oid -> Msg .
  msg _elt_ : Oid X$Elt -> Msg .

  op null : -> Oid .
  op o : Oid Int -> Oid .

  vars O 0' 0'' 0''' : Oid .
  var E : X$Elt .
  var N : Int .

  rl [top] : *** top on a non-empty stack
    < 0 : Stack{X} | first : 0' >
    < 0' : Node{X} | node : 0'' >
    < 0'' : X$Cell | contents : E >
    (0 top 0''')
    => < 0 : Stack{X} | >
      < 0' : Node{X} | >
      < 0'' : X$Cell | >
      (0''' elt E) .

  rl [push1] : *** push on a non-empty stack
    < 0 : Stack{X} | first : o(0, N) >
    (0 push 0')
    => < 0 : Stack{X} | first : o(0, N + 1) >
      < o(0, N + 1) : Node{X} |
        next : o(0, N), node : 0' > .

  rl [push2] : *** push on an empty stack
    < 0 : Stack{X} | first : null >
    (0 push 0')
    => < 0 : Stack{X} | first : o(0, 0) >
      < o(0, 0) : Node{X} | next : null, node : 0' > .

  rl [pop] : *** pop on a non-empty stack
```

```

    < 0 : Stack{X} | first : 0' >
    < 0' : Node{X} | next : 0'' >
    (0 pop)
    => < 0 : Stack{X} | first : 0'' > .
  endom)

```

21.5 Module operations on object-oriented modules

The module operations of summation, renaming, and instantiation have been extended so that they are also available on object-oriented modules.

21.5.1 Module summation and renaming

The summation and renaming of object-oriented modules is similar to their non-object-oriented counterparts. Renaming maps, however, are in this case available for mapping classes, attributes, and messages. Therefore, in addition to the renamings available in Core Maude, Full Maude also supports renaming maps of the form:

```

class <identifier> to <identifier>
attr <class-identifier> . <attr-identifier> to <class-identifier>
msg <identifier> to <identifier>
msg <identifier> : <type-list> -> <type> to <identifier>

```

We illustrate the renaming of object-oriented modules with the following example:³

```

Maude> (show module 00-STACK2 * (class Stack{X} to Stack{X},
                               class Node{X} to Node{X},
                               attr Stack{X} . first to head,
                               msg _elt_ to element,
                               sort Int to Integer) .)

omod 00-STACK2 * (sort Int to Integer,
                 msg _elt_ to element,
                 class Node'{X}' to Node'{X}',
                 class Stack'{X}' to Stack'{X}',
                 attr Stack'{X}' . first to head) {X :: CELL} is
protecting QID .
protecting INT * (sort Int to Integer) .
including CONFIGURATION+ .
including CONFIGURATION .
protecting BOOL .
subsort Qid < Oid .

```

³ The `including CONFIGURATION+ .` declaration in the shown module will be explained in Section 21.9.

```

class Node'{X}' | next : Oid, node : Oid .
class Stack'{X}' | head : Oid .
op null : -> Oid .
op o : Oid Integer -> Oid .
msg _pop_ : Oid -> Msg .
msg _push_ : Oid Oid -> Msg .
msg _top_ : Oid Oid -> Msg .
msg element : Oid X$Elt -> Msg .
r1 < 0:Oid : Stack{X}| head : 0':Oid >
  < 0':Oid : Node{X}| next : 0'':Oid >
  0:Oid pop
  => < 0:Oid : Stack{X}| head : 0'':Oid >
  [label pop] .
r1 < 0:Oid : Stack{X}| head : 0':Oid >
  < 0':Oid : Node{X}| node : 0'':Oid >
  < 0'':Oid : X$Cell | contents : E:X$Elt >
  0:Oid top 0'':Oid
  => < 0:Oid : Stack{X}| none >
  < 0':Oid : Node{X}| none >
  < 0'':Oid : X$Cell | none >
  element(0'':Oid,E:X$Elt)
  [label top] .
r1 < 0:Oid : Stack{X}| head : null >
  0:Oid push 0':Oid
  => < 0:Oid : Stack{X}| head : o(0:Oid,0)>
  < o(0:Oid,0): Node{X}| next : null,node : 0':Oid >
  [label push2] .
r1 < 0:Oid : Stack{X}| head : o(0:Oid,N:Integer)>
  0:Oid push 0':Oid
  => < 0:Oid : Stack{X}| head : o(0:Oid,N:Integer + 1)>
  < o(0:Oid,N:Integer + 1): Node{X}|
  next : o(0:Oid,N:Integer),node : 0':Oid >
  [label push1] .
endom

```

21.5.2 Module instantiation

We show in this section how, by instantiating the object-oriented module `OO-STACK2` given in Section 21.4.3, we can obtain a specification of a stack of banking accounts. We first specify a view `Account` from the object-oriented theory `CELL` (in Section 21.4.1) to the object-oriented module `ACCOUNT` (in Section 21.1.4).

```

(view Account from CELL to ACCOUNT is
  sort Elt to Int .
  class Cell to Account .
  attr Cell . contents to bal .
endv)

```

Now we can do the following rewriting on the module resulting from the instantiation.

```
Maude> (rew in 00-STACK2{Account}
      * (class Account to Account,
        class Stack{Account} to Stack{Account},
        class Node{Account} to Node{Account},
        attr Stack{Account} . first to head,
        attr Account . bal to balance,
        msg _elt_ to element,
        sort Int to Integer) :
  < 'stack : Stack{Account} | head : null >
  < 'A-73728 : Account | balance : 5000 >
  < 'A-06238 : Account | balance : 2000 >
  < 'A-28381 : Account | balance : 15000 >
  ('stack push 'A-73728)
  ('stack push 'A-06238)
  ('stack push 'A-28381)
  ('stack top 'A-06238)
  ('stack pop) .)

result Configuration :
  element('A-28381,15000)
  < 'A-06238 : Account | balance : 2000 >
  < 'A-28381 : Account | balance : 15000 >
  < 'A-73728 : Account | balance : 5000 >
  < 'stack : Stack{Account}| head : o('stack, 1)>
  < o('stack, 0) : Node{Account} | next : null, node : 'A-06238 >
  < o('stack, 1) : Node{Account} |
    next : o('stack, 0), node : 'A-73728 >
```

21.6 Example: extended rent-a-car store

This section describes a variant of the rent-a-car store example in Section 21.3 in which several interesting data structures are used to store relevant information.

Let us refine the specification of a rent-a-car store presented in Section 21.3 by adding the following regulations:

10. When a rented car is returned, the deposit is used to pay the rental charges, which are calculated in accordance with the conditions at pick-up time.
11. There are three different kinds of customers: staff, occasional, and preferred.
12. Staff members and preferred customers benefit from special discounts in all rentals.

13. A customer qualifies as “preferred” when the accumulated amount of money spent in the store by the customer is above a certain threshold.

The main differences introduced by these regulations are that we need to keep the conditions at pick-up time, so that the calculations at drop-off time are correct. We also need to distinguish the three different types of customers, with the possibility of an occasional customer being promoted to preferred if he spends a given amount of money.

As an alternative approach to the one followed previously in Section 21.3, we introduce a class `Store` of rental car stores, whose attributes represent the information concerning the general parameters of such stores: the rates applicable to each type of car, the discounts for each type of customer renting each type of car, the identifiers of the customers who are suspended, the amount of money above which occasional customers are qualified as preferred, the record with the amount of money spent in the store by each of the customers, and the daily penalty for late return (20%). In addition, attributes `customers`, `cars`, `rentals`, and `calendar` store the identifiers of the objects participating in the relationships with the `Store` composite object; those are *directed* binary relationships and therefore we need only store the identifiers of the subordinate objects as attributes of the object that references them.

```
class Store |
  discounts : PFun{Tuple{Cid, Cid}, Nat},
  payments  : PFun{Oid, Nat},
  penalty   : Nat,
  threshold : Nat,
  suspended : Set{Oid},
  rates     : PFun{Cid, Nat},
  customers : Set{Oid},
  cars      : Set{Oid},
  rentals   : Set{Oid},
  calendar  : Oid .
```

The information on rates, discounts, and money spent is modeled by attributes of sort `PFun` of partial functions⁴ (see Section 19.3.2), associating the appropriate values to each of the different agents involved. The rates for the different cars are stored in the attribute `rates`, of sort `PFun{Cid, Nat}`, that associates the per-day rate to be charged to a customer for renting a given type of car. Thus, assuming that `Rts` is a variable of sort `PFun{Cid, Nat}`, with value the partial function assigning the appropriate rates to each type of car, we have that `Rts[FullSizeCar]` is the per-day rate for renting a full size car. If we want to increase this rate by, say 20%, we can use the expression

```
Rts[FullSizeCar -> Rts[FullSizeCar] * (100 + penalty) / 100]
```

⁴ An alternative possibility is to use the maps specified in the predefined `MAP` module in Section 9.13.

with `penalty` a constant equal to 20. The discounts applied to each customer on each type of car and the amount of the purchases of each customer are stored, respectively, in attributes `payments` and `discounts`. The set of the identifiers of the customers who are suspended is stored in an attribute `suspended` of sort `Set{Oid}`. The predefined sorts `Oid` and `Cid` are used for object identifiers and class identifiers, respectively.

This specification will allow us, for instance, to easily “compose” systems with different particular details (e.g., discounts may change from one store to another), allowing them to easily co-exist.

The rest of the classes can be specified as follows:

```
class Customer | cash : Nat, debt : Nat .
class Staff .
class OccasionalCust .
class PreferredCust .
subclasses OccasionalCust PreferredCust Staff < Customer .

class Car | available : Bool .
class EconomyCar .
class MidSizeCar .
class FullSizeCar .
subclasses EconomyCar MidSizeCar FullSizeCar < Car .

class Rental |
  deposit      : Nat, discount   : Nat,
  dueDate      : Nat, pickUpDate : Nat,
  rate         : Nat, customer   : Oid,
  car          : Oid .
```

The different actions may then be defined as follows:

```
cr1 [car-rental] :
< U : Customer | cash : M >
< I : Car | available : true >          *** the car is available
< V : Store | suspended : US,
  rates : Rts, discounts : Dscnts, calendar : C,
  cars : (I, IS), customers : (U, SS), rentals : RS >
< C : Calendar | date : Today >
=> < U : Customer | cash : sd(M, Amnt) >
  < I : Car | available : false >
  < V : Store | rentals : (A, RS) >
  < C : Calendar | >
  < A : Rental | pickUpDate : Today, dueDate : Today + NumDays,
    car : I, deposit : Amnt, customer : U,
    rate : Rt, discount : Dscnt >
if not U in US          *** the customer is not suspended
  /\ Rt := Rts[class(< I : Car | >)]
  /\ Dscnt := Dscnts[(class(< U : Customer | >),
                    class(< I : Car | >))]
```

```

/\ Amnt := sd(Rt, Dscnt) * NumDays
/\ M >= Amnt          *** enough cash to make a deposit
[nonexec] .

```

Notice the use of `customer` and `car` attributes in objects of class `Rental`, which makes explicit that a rental relationship is between the customer and the car specified by these attributes. Likewise for attributes `customers`, `cars`, and `calendar` of object `V` of class `Store`, which indicate that the customer, car and calendar appearing on the rule should be known to the store. After the `car-rental` action, the rental is added to the set of rentals kept by the store.

Rules may be applied to objects of the classes specified in the rules or of any of their subclasses. Remember that the function `class` takes an object as argument and returns its actual class (see Section 21.1.2); for example, the `class` function applied to an object of the form `< 'c123 : MidSizeCar | ... >` returns `MidSizeCar`, and not `Car`. Finally, notice the use of *matching equations* of the form `t := t'` in the condition (see Section 4.3).

The return of a rented car is specified by the rules below. The first rule handles the case in which the car is returned on time, that is, the current date is smaller than or equal to the due date, and therefore the deposit is greater than or equal to the amount due. Notice that the rate and discount to be used in the calculation of the amount due are those at pick-up time, which are stored as attributes of the `Rental` object.

```

cr1 [on-date-car-return] :
  < U : Customer | cash : M >
  < I : Car | >
  < A : Rental | customer : U, car : I, rate : Rt, discount : Dscnt,
    pickUpDate : Ppdt, dueDate : Ddt, deposit : Dpst >
  < V : Store | payments : Pmnts, cars : (I, IS),
    customers : (U, SS), calendar : C, rentals : (A, RS) >
  < C : Calendar | date : Today >
=> < U : Customer | cash : M + sd(Dpst, Amnt) >
  < I : Car | available : true >
  < V : Store | rentals : RS,
    payments : (if Pmnts[U] == undefined *** no record for customer
      then Pmnts[U -> Amnt]
      else Pmnts[U -> ((Pmnts[U]) + Amnt)]
      fi) >
  < C : Calendar | >
  if (Today <= Ddt) /\ Amnt := sd(Rt, Dscnt) * sd(Today, Ppdt) .

```

In this case, the deposit is greater than the amount due and therefore part of the deposit needs to be reimbursed. Note also that the `Store` object keeps a record of the amount of money spent by each customer in the store, and thus it must be updated accordingly. We can see how the `Rental` object disappears in the righthand side of the rules: it is removed from the set of rentals known to the store and the availability of the car is restored.

The second rule deals with the case in which the car is returned late. The amount to be paid is calculated at drop-off time, but the rate and discount to be used, those at pick-up time, may have changed when returning the car.

```

crl [late-car-return] :
  < U : Customer | debt : M >
  < I : Car | >
  < A : Rental | customer : U, car : I, rate : Rt, discount : Dscnt,
    pickUpDate : Ppdt, dueDate : Ddt, deposit : Dpst >
  < V : Store | payments : Pmnts, penalty : Pnlt, rentals : (A, RS),
    cars : (I, IS), customers : (U, SS), calendar : C >
  < C : Calendar | date : Today >
=> < U : Customer | debt : M + sd(Amnt, Dpst) >
  < I : Car | available : true >
  < V : Store | rentals : RS,
    payments : (if Pmnts[U] == undefined
                then Pmnts[U -> Dpst]
                else Pmnts[U -> ((Pmnts[U]) + Dpst)]
              fi) >
  < C : Calendar | >
if Ddt < Today                               *** it is returned late
  /\ Amnt := (sd(Rt, Dscnt) * sd(Ddt, Ppdt))
  + (((sd(Rt, Dscnt) * sd(Today, Ddt))
    * (100 + Pnlt)) quo 100) .

```

In this case the customer's debt is increased by the portion of the amount due not covered by the deposit.

Debts may be paid at any time, the only condition being that the amount paid is between zero and the amount of money of the customer at that time.

```

crl [pay-debt] :
  < V : Store |
    payments : Pmnts, customers : (U, SS), calendar : C >
  < U : Customer | debt : M, cash : N >
  < C : Calendar | date : Today >
=> < V : Store | payments : Pmnts[U -> ((Pmnts[U]) + Amnt)] >
  < U : Customer | debt : sd(M, Amnt), cash : sd(N, Amnt) >
  < C : Calendar | >
if 0 < Amnt /\ Amnt <= N /\ Amnt <= M
[nonexec] .

```

We are assuming that, if there is a debt, then there has been a previous payment, and therefore there is already a record for that customer.

The text says that customers who are late in returning a rented car or in paying their debts “may” be suspended. However, nothing is said about the reasons for taking such a decision or when they should be suspended, that is, a customer could be suspended right after the car is returned without having paid all the charges, after some grace days, or never. In practice there will be

fixed criteria, as, for example, suspending customers that are two days late, or two months.

The first rule deals with the case in which a customer has a pending debt, and the second one handles the case in which a customer is late in returning a rented car.

```

crl [suspend-late-payers] :
  < V : Store | suspended : US, customers : (U, SS) >
  < U : Customer | debt : M >
  => < V : Store | suspended : (U, US) >
      < U : Customer | >
      if (not U in US) and M > 0 .

crl [suspend-late-returns] :
  < V : Store | suspended : US, cars : (I, IS),
              customers : (U, SS), calendar : C >
  < U : Customer | >
  < I : Car | >
  < A : Rental | customer : U, car : I, dueDate : F >
  < C : Calendar | date : Today >
  => < V : Store | suspended : (U, US) >
      < U : Customer | >
      < I : Car | >
      < A : Rental | >
      < C : Calendar | >
      if (not U in US) and F < Today .

```

The upgrade in the status of a customer can then be modeled with the following rule:

```

crl [upgrade-to-preferred] :
  < U : OccasionalCust | cash : M, debt : N >
  < V : Store | threshold : Thrshld, payments : Pmnts,
              customers : (U, SS), calendar : C >
  < C : Calendar | date : Today >
  => < U : PreferredCust | cash : M, debt : N >
      < V : Store | >
      < C : Calendar | >
      if (Pmnts[U]) >= Thrshld .

```

In this rule, a customer object `U` of subclass `OccasionalCust` becomes of subclass `PreferredCust` when the accumulated amount of purchases exceeds the store's threshold. The partial function stored in the attribute `payments` gives us the amount of money spent by each customer. In Maude, objects changing their classes must show all their attributes in the righthand sides of the rules.

As in the simpler rent-a-car system, the presence of nonterminating rules and of rules with new variables in the righthand side requires some kind of strategy for the execution of the system; we give an example of such a strategy in the next section.

21.7 A strategy for sequential rule execution

Strategies are necessary for controlling the execution of rules that are not terminating, or that do not satisfy the admissibility conditions discussed in Section 6.3. A simple but interesting strategy may be one that allows us to execute a given sequence of rules, that is, to accomplish sequentially a series of actions from a particular initial state. We introduce in this section such a generic strategy and illustrate its use by applying it for executing the systems specified in Sections 21.3 and 21.6. Dealing with strategies may become cumbersome, since we need to handle terms and modules at different levels of reflection, and expressions may become quite hard to read and handle. We show in this section how the `upModule` and `upTerm` functions and the `down` command introduced in Section 19.4 can help in alleviating this difficulty.

A strategy is represented as a sequence of rule applications. We instantiate the predefined module `LIST` with pairs formed by a rule label representing the rule to be applied, and a substitution to partially instantiate the variables in such a rule before its application. The pairs are obtained using the generic tuple construction described in Section 19.3.1. Thus, to get the module expression `LIST{Tuple{Qid, Substitution}}`, given the predefined view `Qid` and the *parameterized view* `Tuple`, that we have already used in the partial functions example of Section 19.3.2, we only need to define a view `Substitution` from `TRIV` to `META-LEVEL`.

```
(view Substitution from TRIV to META-LEVEL is
  sort Elt to Substitution .
endv)
```

This construction is put to work in the module `REW-SEQ` below. The operator `rewSeq` in this module takes the metarepresentation of a module, the metarepresentation of a term, and a list of pairs (each formed by a rule label and a substitution); the term obtained in this way is rewritten by applying the given rules sequentially, using in their applications their corresponding partial substitutions.

```
(mod REW-SEQ is
  including META-LEVEL .
  protecting LIST{Tuple{Qid, Substitution}} .

  var M : Module .
  var T : Term .
  var L : Qid .
  var S : Substitution .
  var LLS : List{Tuple{Qid, Substitution}} .

  op rewSeq :
    Module Term List{Tuple{Qid, Substitution}} -> [Term] .
```

```

r1 [seq] : rewSeq(M, T, (L, S) LLS)
=> rewSeq(M,
  getTerm(metaXapply(M, T, L, S, 0, unbounded, 0)), LLS) .

r1 [seq] : rewSeq(M, T, nil) => T .
endm)

```

The rules to be applied here are part of the module given as first argument. The strategy starts with the term given as initial state, which is replaced in each recursive call by the term representing the state obtained after the application of the next rule in the sequence (see Section 14.5.4). When all the rules have been applied, thus reaching the empty list as third argument, the current state is returned as the resulting final state.

We illustrate the use of the `rewSeq` strategy by applying a sequence of rules on a configuration of the rent-a-car system specified in Section 21.3. Let `RENT-A-CAR-STORE` be the name of the module containing the specification of such a system, and let `StoreConf` be a configuration of objects defined in the following module.

```

(fmod RENT-A-CAR-STORE-TEST is
  pr RENT-A-CAR-STORE .

  op StoreConf : -> Configuration [memo] .
  eq StoreConf
  = < 'C1 : Customer | cash : 5000, debt : 0, suspended : false >
    < 'C2 : Customer | cash : 5000, debt : 0, suspended : false >
    < 'A1 : EconomyCar | available : true, rate : 100 >
    < 'A3 : MidSizeCar | available : true, rate : 150 >
    < 'A5 : FullSizeCar | available : true, rate : 200 >
    < 'C : Calendar | date : 0 > .
endfm)

```

The `StoreConf` configuration consists of two clients `C1` and `C2`, three cars `A1`, `A3` and `A5`, and a calendar object `C`. Now, let `StoreStrat` be a sequence of pairs (rule label - substitution) that defines the strategy declared in the following module as a sequence of actions:

```

(fmod REW-SEQ-TEST is
  pr REW-SEQ .

  op StoreStrat : -> List{Tuple{Qid, Substitution}} [memo] .
  eq StoreStrat
  = ('car-rental,
    'U:Qid <- ''C1.Qid ;          *** size car A3 for 2 days
    'I:Qid <- ''A3.Qid ;
    'NumDays:Int <- 's_~2['0.Zero] ;
    'A:Qid <- ''a0.Qid)
    ('new-day, none)             *** two days pass
    ('new-day, none)

```

```

('on-date-car-return, none)      *** car A3 is returned
('new-day, none)
('car-rental,                    *** client C1 rents the full
  'U:0id <- ''C1.Qid ;           *** size car A5 for 1 day
  'I:0id <- ''A5.Qid ;
  'NumDays:Int <- 's_~1['0.Zero] ;
  'A:0id <- ''a1.Qid)
('new-day, none)                *** two days pass
('new-day, none)
('late-car-return, none)        *** car A5 is returned
('new-day, none)
('suspend-late-payers, none)    *** client C1 is suspended
('new-day, none)
('new-day, none)
('pay-debt,                      *** client C1 pays 100$
  'Amnt:Int <- 's_~100['0.Zero]) .
endfm)

```

Comments on the righthand side of the code above explain the sequence of rules defining the strategy. Basically, the execution trace specified consists of client C1 renting two cars, one of which is returned on time and the other one is returned late. After the second car is returned, the client is suspended for being late in his payments. The client then pays part of his debt. Note how the passage of time is modeled by the application of the rule `new-day`.

Now, in order to execute the system specifications using this strategy, we just need to use `rewSeq` to apply the given rules sequentially, using their corresponding partial substitutions in their applications. Note how the first two arguments are metarepresented with the `upModule` and `upTerm` functions, since they need to be the metarepresentations of the actual module and term, respectively.

```

Maude> (down RENT-A-CAR-STORE :
      rew rewSeq(upModule(RENT-A-CAR-STORE-TEST),
                upTerm(RENT-A-CAR-STORE-TEST, StoreConf),
                StoreStrat) .)

result Configuration :
  < 'C : Calendar | date : 8 >
  < 'C1 : Customer | suspended : true, debt : 140, cash : 4400 >
  < 'C2 : Customer | suspended : false, debt : 0, cash : 5000 >
  < 'A1 : EconomyCar | rate : 100, available : true >
  < 'A3 : MidSizeCar | rate : 150, available : true >
  < 'A5 : FullSizeCar | rate : 200, available : true >

```

We can see in this configuration that eight days have passed, after which the client C1 is suspended. The client C1 has paid a total of \$600 ($= 2 \times 150 + 200 + 100$), and has still a debt of \$140 ($= 200 + 20 \% 200 - 100$).

The same strategy can be used to execute the extended specification in Section 21.6, contained in a module named `EXTENDED-RENT-A-CAR-STORE`. First, we define a module with an initial configuration `ExtStoreConf`.

```
(fmod EXTENDED-RENT-A-CAR-STORE-TEST is
  pr EXTENDED-RENT-A-CAR-STORE .

  op ExtStoreConf : -> Configuration [memo] .
  eq ExtStoreConf
    = < 'S : Store |
      discounts :
        ((Staff, EconomyCar), 20),
        ((Staff, MidSizeCar), 30),
        ((Staff, FullSizeCar), 40),
        ((OccasionalCust, EconomyCar), 0),
        ((OccasionalCust, MidSizeCar), 0),
        ((OccasionalCust, FullSizeCar), 0),
        ((PreferredCust, EconomyCar), 10),
        ((PreferredCust, MidSizeCar), 15),
        ((PreferredCust, FullSizeCar), 20)),
      payments : empty, penalty : 0,
      threshold : 1000, suspended : empty,
      rates : ((EconomyCar, 100),
              (MidSizeCar, 150),
              (FullSizeCar, 200)),
      customers : ('C1, 'C2),
      cars : ('A1, 'A3, 'A5),
      rentals : empty, calendar : 'C >
  < 'C1 : Staff | cash : 5000, debt : 0 >
  < 'C2 : OccasionalCust | cash : 5000, debt : 0 >
  < 'A1 : EconomyCar | available : true >
  < 'A3 : MidSizeCar | available : true >
  < 'A5 : FullSizeCar | available : true >
  < 'C : Calendar | date : 0 > .
endfm)
```

Now we execute a command completely analogous to the previous one, obtaining a resulting state that shows how, after eight days, client C1 has paid \$500, and has a debt of \$60.

```
Maude> (down EXTENDED-RENT-A-CAR-STORE :
  rew rewSeq(upModule(EXTENDED-RENT-A-CAR-STORE),
    upTerm(EXTENDED-RENT-A-CAR-STORE, ExtStoreConf),
    StoreStrat) .)
result Configuration :
  < 'A1 : EconomyCar | available : true >
  < 'A3 : MidSizeCar | available : true >
  < 'A5 : FullSizeCar | available : true >
  < 'C : Calendar | date : 8 >
```

```

< 'C1 : Staff | cash : 4500, debt : 60 >
< 'C2 : OccasionalCust | cash : 5000, debt : 0 >
< 'S : Store | calendar : 'C,
  cars : ('A1, 'A3, 'A5),
  customers : ('C1, 'C2),
  discounts : (((OccasionalCust, EconomyCar), 0),
               ((OccasionalCust, FullSizeCar), 0),
               ((OccasionalCust, MidSizeCar), 0),
               ((PreferredCust, EconomyCar), 10),
               ((PreferredCust, FullSizeCar), 20),
               ((PreferredCust, MidSizeCar), 15),
               ((Staff, EconomyCar), 20),
               ((Staff, FullSizeCar), 40),
               ((Staff, MidSizeCar), 30)),
  payments : ('C1, 500),
  penalty : 0,
  rates : ((EconomyCar, 100),
           (FullSizeCar, 200),
           (MidSizeCar, 150)),
  rentals : empty,
  suspended : 'C1,
  threshold : 1000 >

```

21.8 Model checking a round-robin scheduling algorithm

In this section we present a specification of a round-robin scheduling algorithm, and the mutual exclusion and guaranteed reentrance properties are proven about it. Both the algorithm and the property guaranteeing that all processes reenter their critical sections are parameterized by the number of processes. We use Maude's model checker to prove the mutual exclusion and guaranteed reentrance properties. As we said in Section 19.2, to use the `MODEL-CHECKER` module, or any other Core Maude module, we just need to make sure that it has been loaded; we suggest loading the `model-checker.maude` file *before* starting Full Maude.

We first give a specification of natural numbers modulo. Since we want to be able to have any number of processes, we define the `NAT/` module parameterized by the functional theory `NZNAT#`, which requires a constant of sort `Nat`. Thus, having a view, say `5` from `TRIV` to `NZNAT#` mapping `#` to the natural number `5`, the module expression `NAT/{5}` specifies the natural numbers modulo `5`.

```

(fth NZNAT# is
  protecting NAT .
  op # : -> NzNat .
endfth)

```

```
(fmod NAT/{N} :: NZNAT#} is
  sort Nat/{N} .
  op '[' : Nat -> Nat/{N} [ctor] .
  op '+_ ' : Nat/{N} Nat/{N} -> Nat/{N} .
  op '*_ ' : Nat/{N} Nat/{N} -> Nat/{N} .
  vars N M : Nat .
  ceq [N] = [N rem #] if N >= # .
  eq [N] + [M] = [N + M] .
  eq [N] * [M] = [N * M] .
endfm)
```

The round-robin scheduling algorithm is specified in the module `RROBIN` below. Processes are represented as objects of class `Proc`, which may be in `wait` or `critical` mode, meaning that a process may be either in its critical section or waiting to enter into it. The process getting the token, which is represented as the message `go`, can enter its critical section. Once a process gets out of its critical section it forwards the token to the next process. The `init` operator sets up the initial configuration for a given number of processes. Note that `Nat/{N}` is made a subsort of `Oid`, making in this way natural numbers modulo `N` valid object identifiers.

```
(omod RROBIN{N} :: NZNAT#} is
  protecting NAT/{N} .

  sort Mode .
  ops wait critical : -> Mode [ctor] .

  subsort Nat/{N} < Oid .

  class Proc | mode : Mode .
  msg go : Nat/{N} -> Msg .

  var N : Nat .

  rl [enter] :
    go([N])
    < [N] : Proc | mode : wait >
    => < [N] : Proc | mode : critical > .

  rl [exit] :
    < [N] : Proc | mode : critical >
    => < [N] : Proc | mode : wait >
    go([s(N)]) .

  op init : -> Configuration .
  op make-init : Nat/{N} -> Configuration .

  ceq init = go([0]) make-init([N]) if s(N) := # .
  ceq make-init([s(N)])
```

```

    = < [s(N)] : Proc | mode : wait > make-init([N])
    if N < # .
    eq make-init([0]) = < [0] : Proc | mode : wait > .
  endom)

```

For proving mutual exclusion and guaranteed reentrance, we declare the propositions `inCrit` and `twoInCrit` in the module `CHECK-RROBIN` below (see Chapter 13 for a discussion on the use of Maude's model checker). The property `inCrit` takes a `Nat/{N}` as argument, thus making this property parameterized by the number of processes, and is true when such a process is in its critical section. The property `twoInCrit` is true if any two processes are in their critical sections simultaneously. Mutual exclusion will be proved directly below, while for proving guaranteed reentrance we use the auxiliary formula `guaranteedReentrance`, which allows us to specify the property of all processes reentering their critical sections in exactly $2N$ steps, for N the number of processes. For a formula F , `nextIter F` returns $0 \dots 0 F$ (where 0 denotes the modal next operator), which specifies that the property is true in the next iteration, that is, $2N$ steps later. Note that the expression `2 * #` will become two times N once the module is instantiated.

```

(omod CHECK-RROBIN{N :: NZNAT#} is
  pr RROBIN{N} .
  inc MODEL-CHECKER .
  inc SATISFACTION .
  ex LTL-SIMPLIFIER .
  inc LTL .

  subsort Configuration < State .

  op inCrit : Nat/{N} -> Prop .
  op twoInCrit : -> Prop .

  var N : Nat .
  vars X Y : Nat/{N} .
  var C : Configuration .
  var F : Formula .

  eq < X : Proc | mode : critical > C |= inCrit(X) = true .
  eq < Y : Proc | mode : critical > < Y : Proc | mode : critical > C
    |= twoInCrit = true .

  op guaranteedReentrance : -> Formula .
  op allProcessesReenter : Nat -> Formula .
  op nextIter_ : Formula -> Formula .
  op nextIterAux : Nat Formula -> Formula .

  eq guaranteedReentrance = allProcessesReenter(#) .

```

```

eq allProcessesReenter(s N)
  = (inCrit([s N]) -> nextIter inCrit([s N])) /\
    allProcessesReenter(N) .
eq allProcessesReenter(0) = inCrit([0]) -> nextIter inCrit([0]) .

eq nextIter F = nextIterAux(2 * #, F) .

eq nextIterAux(s N, F) = 0 nextIterAux(N, F) .
eq nextIterAux(0, F) = F .
endom)

```

Note that the LTL formula describing the `guaranteedReentrance` property is not a single LTL formula, but an *infinite parametric family of formulas*

$$\text{guaranteedReentrance} = \{\text{allProcessesReenter}(n) \mid n \in \mathbb{N}\}.$$

The use of equations in the above CHECK-RRROBIN parameterized module allows us to define this infinite family of formulas by means of a few recursive equations. When this module is instantiated for a concrete value of n , we then obtain the concrete LTL formula `allProcessesReenter(n)` for that n .

We now prove mutual exclusion and guaranteed reentrance for the case of five processes using the model checker.

```

(view 5 from NZNAT# to NAT is
  op # to term 5 .
endv)

Maude> (reduce in CHECK-RRROBIN{5} :
  modelCheck(init, [] ~ twoInCrit) .)
result Bool :
  true

Maude> (reduce in CHECK-RRROBIN{5} :
  modelCheck(init, [] guaranteedReentrance) .)
result Bool :
  true

```

Of course the answer depends on the property checked and is not always `true`. The following example shows how the model checker gives a counterexample as result when trying to prove that, for a configuration of five processes, process [1] is in its critical section three steps after it was in it.

```

Maude> (red in CHECK-RRROBIN{5} :
  modelCheck(init, [] (inCrit([1]) -> 0 0 0 inCrit([1]))) .)
result ModelCheckResult :
  counterexample(
    {go([0]) <[0]: Proc | mode : wait >
      <[1]: Proc | mode : wait > <[2]: Proc | mode : wait >
      <[3]: Proc | mode : wait > <[4]: Proc | mode : wait >, 'enter}
    {<[0]: Proc | mode : critical > <[1]: Proc | mode : wait >
      <[2]: Proc | mode : wait > <[3]: Proc | mode : wait >

```

```

    <[4]: Proc | mode : wait >, 'exit}
{go([1]) <[0]: Proc | mode : wait >
  <[1]: Proc | mode : wait > <[2]: Proc | mode : wait >
  <[3]: Proc | mode : wait > <[4]: Proc | mode : wait >, 'enter}
{<[0]: Proc | mode : wait > <[1]: Proc | mode : critical >
  <[2]: Proc | mode : wait > <[3]: Proc | mode : wait >
  <[4]: Proc | mode : wait >, 'exit}
{go([2]) <[0]: Proc | mode : wait >
  <[1]: Proc | mode : wait > <[2]: Proc | mode : wait >
  <[3]: Proc | mode : wait > <[4]: Proc | mode : wait >, 'enter}
{<[0]: Proc | mode : wait > <[1]: Proc | mode : wait >
  <[2]: Proc | mode : critical > <[3]: Proc | mode : wait >
  <[4]: Proc | mode : wait >, 'exit},
{go([3]) <[0]: Proc | mode : wait >
  <[1]: Proc | mode : wait > <[2]: Proc | mode : wait >
  <[3]: Proc | mode : wait > <[4]: Proc | mode : wait >, 'enter}
{<[0]: Proc | mode : wait > <[1]: Proc | mode : wait >
  <[2]: Proc | mode : wait > <[3]: Proc | mode : critical >
  <[4]: Proc | mode : wait >, 'exit}
{go([4]) <[0]: Proc | mode : wait >
  <[1]: Proc | mode : wait > <[2]: Proc | mode : wait >
  <[3]: Proc | mode : wait > <[4]: Proc | mode : wait >, 'enter}
{<[0]: Proc | mode : wait > <[1]: Proc | mode : wait >
  <[2]: Proc | mode : wait > <[3]: Proc | mode : wait >
  <[4]: Proc | mode : critical >, 'exit}
{go([0]) <[0]: Proc | mode : wait >
  <[1]: Proc | mode : wait > <[2]: Proc | mode : wait >
  <[3]: Proc | mode : wait > <[4]: Proc | mode : wait >, 'enter}
{<[0]: Proc | mode : critical > <[1]: Proc | mode : wait >
  <[2]: Proc | mode : wait > <[3]: Proc | mode : wait >
  <[4]: Proc | mode : wait >, 'exit}
{go([1]) <[0]: Proc | mode : wait >
  <[1]: Proc | mode : wait > <[2]: Proc | mode : wait >
  <[3]: Proc | mode : wait > <[4]: Proc | mode : wait >, 'enter}
{<[0]: Proc | mode : wait > <[1]: Proc | mode : critical >
  <[2]: Proc | mode : wait > <[3]: Proc | mode : wait >
  <[4]: Proc | mode : wait >, 'exit}
{go([2]) <[0]: Proc | mode : wait >
  <[1]: Proc | mode : wait > <[2]: Proc | mode : wait >
  <[3]: Proc | mode : wait > <[4]: Proc | mode : wait >, 'enter}
{<[0]: Proc | mode : wait > <[1]: Proc | mode : wait >
  <[2]: Proc | mode : critical > <[3]: Proc | mode : wait >
  <[4]: Proc | mode : wait >, 'exit})

```

21.9 From object-oriented modules to system modules

The best way to understand classes and class inheritance in Full Maude is by making explicit the full structure of an object-oriented module, which is left

somewhat implicit by the syntactic conventions adopted for them. Indeed, although object-oriented modules provide convenient syntax for programming object-oriented systems, their semantics can be reduced to that of system modules. We can regard the special syntax reserved for object-oriented modules as syntactic sugar. In fact, each object-oriented module can be translated into a corresponding system module whose semantics *is* by definition that of the original object-oriented module.

In the translation process, the most basic structure shared by all object-oriented modules is made explicit by the `CONFIGURATION` functional module. The translation of a given object-oriented module extends this structure with the classes, messages and rules introduced by the module. For example, the following system module is the translation of the `ACCOUNT` module introduced earlier. Note that a subsort `Account` of `Cid` is introduced. The purpose of this subsort is to range over the class identifiers of the subclasses of `Account`. For the moment, no such subclasses have been introduced; therefore, at present the only constant of sort `Account` is the class identifier `Account`.

```

mod ACCOUNT is
  protecting INT .
  protecting QID .
  including CONFIGURATION+ .
  including CONFIGURATION .
  sorts Account .
  subsort Qid < Oid .
  subsort Account < Cid .
  op Account : -> Account .
  op credit : Oid Int -> Msg [msg] .
  op debit : Oid Int -> Msg [msg] .
  op from_to_transfer_ : Oid Oid Int -> Msg [msg] .
  op bal :_ : Int -> Attribute .
  var A : Oid .
  var B : Oid .
  var M : Int .
  var N : Int .
  var N' : Int .
  var V@Account : Account .
  var ATTS@0 : AttributeSet .
  var V@Account1 : Account .
  var ATTS@2 : AttributeSet .
  rl [credit] :
    credit(A, M)
    < A : V@Account | bal : N, ATTS@0 >
    => < A : V@Account | bal : (N + M), ATTS@0 > .
  crl [debit] :
    debit(A, M)
    < A : V@Account | bal : N, ATTS@0 >
    => < A : V@Account | bal : (N - M), ATTS@0 >
    if N >= M = true .

```

```

cr1 [transfer] :
  (from A to B transfer M)
  < A : V@Account | bal : N, ATTS@0 >
  < B : V@Account1 | bal : N', ATTS@2 >
  => < A : V@Account | bal : (N - M), ATTS@0 >
     < B : V@Account1 | bal : (N' + M), ATTS@2 >
  if N >= M = true .
endm

```

We can describe the desired transformation from an object-oriented module to a system module as follows:⁵

- The module CONFIGURATION+ is imported, which in turn imports the module CONFIGURATION from Section 11.1. It adds a function `class` which returns the actual class of the given object, and also syntax for objects with no attributes `<_:_| >`.

```

mod CONFIGURATION+ is
  protecting CONFIGURATION .
  op <_:_| > : Oid Cid -> Object .
  op class : Object -> Cid .
  eq < 0:Oid : C:Cid | > = < 0:Oid : C:Cid | none > .
  eq class(< 0:Oid : C:Cid | A >) = C:Cid .
endm

```

- For each class declaration of the form `class C | a1:S1, ..., an:Sn`, the following is introduced: a subsort `C` of sort `Cid`, a constant `C` of sort `C`, and declarations of operations

```

ai :- : Si -> Attribute

```

for each attribute a_i .

- For each subclass relation `C < D` a subsort declaration

```

subsort C < D .

```

is introduced, and the set of attributes for objects of class `C` is completed with those of `D`.

- The system modules resulting from the transformation have the special features supported in Core Maude for object-based programming explained in Chapter 11. Specifically, the `msg` attribute is added to message declarations starting with the `msg` keyword.
- The rewrite rules are modified to make them applicable to all objects of the given classes and of their subclasses, that is, not only to objects whose class identifiers are those explicitly given. The rules are then “inherited” by all objects in their subclasses by replacing the class identifiers in the objects in the rules by variables of the corresponding class sort. Variables

⁵ We have simplified the transformation of object-oriented modules into system modules that originally appeared in [245].

of sort **AttributeSet** are also introduced, to range over the additional attributes that may appear in objects of a subclass. That is, each object expression $\langle O : C \mid \dots \rangle$ appearing in a rule is translated into $\langle O : X \mid \dots, Atts \rangle$, where the new variable X is declared of sort C , and the new variable $Atts$ has sort **AttributeSet**.

- The rewrite rules are modified to give the user the possibility of not mentioning in a given rule those attributes of an object that are not relevant for that rule. To explain the transformation, let $\overrightarrow{a} : \overrightarrow{v}$ denote the attribute-value pairs $a_1 : v_1, \dots, a_n : v_n$, where \overrightarrow{a} are the attribute identifiers of a given class C (after completing it with all the attributes in its superclasses) having \overrightarrow{S} as the corresponding sorts of values prescribed for those attributes. Then, in object-oriented modules we allow rules where the attributes for an object O , mentioned in the lefthand and righthand sides of a rule, need not exhaust all the object's attributes, but can instead be in any of two arbitrary subsets of the object's attributes. We can picture this as follows

$$\dots \langle O : C \mid \overrightarrow{al} : \overrightarrow{vl}, \overrightarrow{ab} : \overrightarrow{vb} \rangle \dots \longrightarrow \dots \langle O : C \mid \overrightarrow{ab} : \overrightarrow{vb'}, \overrightarrow{ar} : \overrightarrow{vr'} \rangle \dots$$

where \overrightarrow{al} are the attributes appearing only on the *left*, \overrightarrow{ab} are the attributes appearing on *both* sides, and \overrightarrow{ar} are the attributes appearing only on the *right*. In the transformation into a system module, this rule is translated into

$$\begin{aligned} & \dots \langle O : X \mid \overrightarrow{al} : \overrightarrow{vl}, \overrightarrow{ab} : \overrightarrow{vb}, \overrightarrow{ar} : \overrightarrow{x}, \overrightarrow{ac} : \overrightarrow{x'}, Atts \rangle \dots \\ & \longrightarrow \dots \langle O : X \mid \overrightarrow{al} : \overrightarrow{vl}, \overrightarrow{ab} : \overrightarrow{vb'}, \overrightarrow{ar} : \overrightarrow{vr'}, \overrightarrow{ac} : \overrightarrow{x'}, Atts \rangle \dots \end{aligned}$$

where X is a variable of sort C , \overrightarrow{ac} are the attributes defined in the class C that do not appear in \overrightarrow{al} , \overrightarrow{ab} , or \overrightarrow{ar} , the \overrightarrow{x} and $\overrightarrow{x'}$ are new variables of the appropriate sorts, and $Atts$ matches the remaining attribute-value pairs.

The rewrite rules given in the original **ACCOUNT** module are interpreted here—according to the conventions already explained—in a form that can be inherited by subclasses of **Account** that could be defined later. Thus, **SavingAccount** inherits the rewrite rules for crediting and debiting accounts, and for transferring funds between accounts that had been defined for **Account**.

Let us illustrate the treatment of class inheritance with the system module resulting from the transformation of the module **SAVING-ACCOUNT** introduced previously.

```
mod SAVING-ACCOUNT is
  including CONFIGURATION+ .
  including CONFIGURATION .
  including ACCOUNT .
  sorts SavingAccount .
```

```

subsort SavingAccount < Cid .
subsort SavingAccount < Account .
op SavingAccount : -> SavingAccount .
op rate :_ : Int -> Attribute .
endm

```

Note that by translating a rule like `credit` above

```

r1 [credit] :
  credit(A, M)
  < A : Account | bal : N >
  => < A : Account | bal : (N + M) > .

```

into its corresponding transformed form

```

r1 [credit] :
  credit(A, M)
  < A : V0@:Account | bal : N, V1@:AttributeSet >
  => < A : V0@:Account | bal : (N + M), V1@:AttributeSet > .

```

it is guaranteed that the rule will be applicable to objects of class `Account` as well as of any of its subclasses.

Note also that a rule like `change-age` (discussed in Section 21.1.4)

```

r1 [change-age] :
  < 0 : Person | >
  to 0 : new age A
  => < 0 : Person | age : A > .

```

is translated into a rule like

```

r1 [change-age] :
  < 0 : V0@:Person | name : V1:String, age : V2:Nat,
                    account : V3:Oid, V4@:AttributeSet >
  to 0 : new age A
  => < 0 : V0@:Person | age : A, name : V1:String,
                    account : V3:Oid, V4@:AttributeSet > .

```

With this translation we allow the rule to be applied to objects in subclasses of `Person`. Furthermore, we guarantee that it is only applied to well-formed objects, that is, to objects with all the required attributes.

See [111] for a detailed explanation of the transformation of object-oriented modules into system modules and how their semantics *is* by definition that of the original object-oriented module.

Applications and Tools

A Sampler of Application Areas

This chapter gives an overview of some application areas of rewriting logic and Maude, with pointers to papers and web sites where more information can be found. Some of the material is adapted, with some modifications and extensions, from [232, 3, 250]. Since Maude is a general-purpose declarative programming language, there is in principle no limit to the applications that could be developed using it. Therefore, the areas discussed, although quite diverse, are only a sample of types of applications for which Maude seems particularly well suited. But there are many others. For example, the availability of built-in internet sockets since Maude 2.2 (see Section 11.4.1) opens up interesting possibilities for a new style of declarative internet programming which have already begun to be exploited (see Chapter 17).

22.1 Models of computation

A wide variety of models of computation, including concurrent ones, can be naturally and directly expressed as rewrite theories in rewriting logic and can be executed as system modules in Maude. In this way, models hitherto quite different from each other can be naturally unified and interrelated within a common framework.

The following is a concise list of models of computation that have been represented in rewriting logic, with relevant literature citations; more details can be found in [247, 249] and in the following sections:

1. equational programming, which is the special case of rewrite theories whose set of rules is empty and whose equations are Church-Rosser, possibly modulo some axioms A ;
2. lambda calculi and combinatory reduction systems [243, 216, 217, 340, 334] (see also Section 8.3.6);
3. labeled transition systems [243];
4. grammars and string-rewriting systems [243];

5. Petri nets, including place/transition nets, contextual nets, algebraic nets, colored nets, and timed Petri nets [243, 247, 335, 342, 288, 332] (see also Section 6.5.1);
6. Gamma and the Chemical Abstract Machine [243];
7. CCS and LOTOS [252, 231, 374, 45, 87, 368, 367, 220, 34] (see also Section 23.2.3 for a description of related Maude tools);
8. the π calculus [375, 334, 357];
9. concurrent objects and actors [243, 245, 348, 350, 353] (see also Chapters 11, 17, and 21);
10. the UNITY language [243];
11. concurrent graph rewriting [247];
12. dataflow [247];
13. neural networks [247];
14. real-time systems, including timed automata, timed transition systems, hybrid automata, and timed Petri nets [288, 283] (see Sections 22.5 and 23.1.6);
15. probabilistic systems [214, 215, 3] (see Section 22.6); and
16. the tile logic [166, 167, 160] model of synchronized concurrent computation [255, 39, 35, 168].

The above specifications of models of computation as rewrite theories are typically executable in Maude, establishing that rewriting logic is a very flexible *operational* framework in which to specify the semantics of such models. What is not immediately apparent from the above list is that it is also a flexible *mathematical* semantic framework at the level of concurrency models. That is, quite often a well-known mathematical model of concurrency happens to be isomorphic to the initial model $\mathcal{T}_{\mathcal{R}}$ of the rewrite theory \mathcal{R} axiomatizing that particular model, or at least closely related to such an initial model. Some examples will illustrate this point:

1. In [217] it is shown that for rewrite theories of the form $\mathcal{R} = (\Sigma, \emptyset, L, R)$, with the rules R left-linear, $\mathcal{T}_{\mathcal{R}}$ is isomorphic to a model based on residuals and permutation equivalence proposed by Boudol;
2. The same paper also shows that for $\mathcal{R} = (\Sigma, E, L, R)$ a rewrite theory axiomatizing an orthogonal combinatory reduction system, including the λ -calculus, a quotient of $\mathcal{T}_{\mathcal{R}}$ by a few additional equations is isomorphic to a well-known model of parallel reductions based on residuals and permutation equivalence;
3. The paper [342] shows in detail that for $\mathcal{R} = (\Sigma, E, L, R)$ a rewrite theory axiomatizing a place/transition net, $\mathcal{T}_{\mathcal{R}}$ is naturally isomorphic (in the categorical sense) to the Best-Devillers net process model—a result essentially known from the coincidence of $\mathcal{T}_{\mathcal{R}}$ with the Meseguer-Montanari algebraic model of nets [243] and the Degano-Meseguer-Montanari algebraic characterization of net processes—and then generalizes this natural isomorphism to one between $\mathcal{T}_{\mathcal{R}}$ and a Best-Devillers-like model for \mathcal{R} the axiomatization of an algebraic net;

4. The papers [45, 87] show that for $\mathcal{R} = (\Sigma, E, L, R)$ a rewrite theory axiomatizing CCS, a truly concurrent semantics causal model based on proved transition systems is isomorphic to a quotient of $\mathcal{T}_{\mathcal{R}}$ by a few additional axioms;
5. The paper [264] shows that for $\mathcal{R} = (\Sigma, E, L, R)$ a rewrite theory axiomatizing a concurrent object-oriented system satisfying reasonable requirements, a subcategory of $\mathcal{T}_{\mathcal{R}}$ is isomorphic to a partial order of events model which, for asynchronous object systems corresponding to actors, coincides with the finitary part of the Hewitt-Baker partial order of events model.

An important additional development in this area is the ρ -calculus of Cirstea and Kirchner [59, 58, 61, 62]. This is a very general rewrite theory that can play for rewriting logic specifications a role similar to that played by the λ -calculus in functional computing; its generality is shown by the fact that ρ -terms generalize the rewriting logic proof terms defined in [243]. Furthermore, the ρ -calculus can simulate the λ -calculus itself. In fact, by replacing and generalizing the λ -calculus idea of function application by that of *rule application*, the ρ -calculus unifies both the λ -calculus and first-order rewriting. In analogy with λ -calculi, there are typed versions, including a simply typed ρ -calculus and a “ ρ cube” [60, 63].

22.2 Semantics of programming languages and software analysis

As illustrated by means of a simple programming language in Section 13.6, rewriting logic is a promising semantic framework for formally specifying programming languages as rewrite theories. Since those specifications usually can be executed in Maude, they in fact become *interpreters* for the languages in question. In addition, such formal specifications allow both formal reasoning and a variety of formal analyses for the languages so specified.

The use of rewrite rules to define the semantics of programming languages is of course not new. In a higher-order version it goes back to the use of semantic equations in denotational semantics; in a first-order version, the power of equational specifications to give semantic definitions of conventional languages has been understood and used for a long time. However, both the lambda calculus and executable equational specifications implicitly assume that such language definitions can be given in terms of *functions*, and rely on the Church-Rosser property to ensure the result of an execution is meaningful. For sequential languages, by making the state of the computation explicit, a functional description of this kind can always be achieved. The situation becomes more difficult for languages that support highly concurrent and non-deterministic applications, and where the possibly nonterminating *interactions* between processes or components—as opposed to the computation of an output value from given inputs—are often the whole point of a program. Such

languages and applications do not have a natural equational description in terms of functions, but do have a very natural rewriting logic semantics, not only operationally (by means of rewriting steps) but also denotationally (\mathcal{T}_R and related models).

A number of case studies giving rewriting logic definitions of programming languages have been carried out. Firstly, some of the models of computation discussed in Section 22.1 are so closely connected with languages that their rewriting logic specifications are also language specifications. Good examples are rewriting logic definitions of the lambda calculus and (mini-) ML, CCS, the π -calculus, and sketches of UNITY and Gamma, which are given in some of the references cited in Section 22.1. Secondly, the usefulness of rewriting strategies to specify program evaluations has been clearly demonstrated in ELAN specifications, for example of Prolog and of the functional-logic programming language Babel [378], and also in the Maude executable specifications for CCS developed by Bruni and Clavel [65, 35], and by Verdejo and Martí-Oliet [368, 366, 369, 371, 365]. Thirdly, the fact that rewriting logic naturally supports concurrent objects has proved very useful in formally specifying a number of novel concurrent and mobile languages. For example, Ishikawa et al. have given a Maude specification of a representative subset of GAEA, a reflective concurrent logic programming language developed at ETL, Japan [198, 197]. Mason and Talcott have used rewriting logic to give semantic definitions of actor languages, and to “compile away” certain language features by defining semantics-preserving translations between actor languages that are formalized as translations between their corresponding rewrite theories [237]. Van Baalen, Caldwell, and Mishra have used Maude to give a formal semantics to the DaAgent mobile agent system and to analyze key fault-tolerant protocols in that language [13]; their analysis has revealed mistakes and inconsistencies in the protocols’ informal specifications. Yet another example is the formal executable specification in Maude of UPenn’s PLAN active network programming language [261, 379]. Maude itself has been used to define the semantics of its Mobile Maude extension [113]. Finally, Maude has been used not only to specify programming languages, but also to specify and verify microprocessors in work by Harman [180, 181]; and to analyze hardware/software co-designs in hardware description languages in work by Katelman and Meseguer [204].

Many languages have already been given semantics in this way using Maude. The language definitions can then be used as interpreters, and—in conjunction with Maude’s search command and its LTL model checker (Chapters 12 and 13, respectively)—to formally analyze programs in those languages. For example, large fragments of Java and the JVM have been specified in Maude this way, with the Maude rewriting logic semantics being used as the basis of Java and JVM program analysis tools that for some examples outperform well-known Java analysis tools [156, 153] (see Section 23.2.5). A similar Maude specification of the semantics of Scheme at the University of Illinois at Urbana-Champaign (UIUC) yields an interpreter which on average,

for the benchmarks tested, has a speed which is 75% the speed of the standard Scheme interpreter. The specification of a C-like language and the corresponding formal analyses are discussed in detail in [259]. A semantics of a Caml-like language with threads was discussed in detail in [258], and a modular rewriting logic semantics of CML has been given by Chalub and Braga in [54]. d’Amorim and Roşu have given a definition of the Scheme language in [86]. Grigore Roşu and his collaborators have developed several domain-specific program analysis tools based on their rewriting semantics in Maude [56, 320]. Other language case studies, all specified in Maude, include: bc [34], CCS [368, 369, 34], CIAO [345], Creol [199], ELOTOS [365], MSR [51, 52, 339], PLAN [344, 345], Orc [5], the π -calculus [357], the ambient calculus [318], and Eden [194]. In fact, the semantics of large fragments of conventional languages are by now routinely developed by UIUC graduate students as course projects in a few weeks, including, besides the languages already mentioned: Beta, Haskell, Lisp, LLVM, Pict, Python, Ruby, and Smalltalk.

Since structural operational semantics definitions can be used for languages not amenable to a functional description, it is natural to compare them with rewriting logic definitions. Their relationship has been discussed in detail in [231], and, more recently, in [371, 251, 258, 259]. In fact, both “big-step” and “small-step” structural operational semantics definitions can be naturally regarded as special formats of corresponding rewrite theory definitions [231]. Tile models provide yet another systematic way of understanding structural operational semantics definitions as tile rewrite theories [166, 167, 168], which can then be mapped into rewriting logic for execution purposes [255, 39, 35]. There is also a close connection between rewriting logic and Mosses’s modular structural operational semantics (MSOS) which has been recognized from the beginning [268, 269], and that has led to subsequent research on:

- a Maude Action Tool to execute MSOS definitions and Action Semantics definitions [33, 32];
- a faithful translation from MSOS to rewriting logic [251, 34]; and
- based on such a translation, a tool to execute MSOS definitions [53, 55], namely the Maude MSOS tool described in Section 23.2.2.

Besides the obvious interest of having a formal executable specification of the semantics of a programming language, if one does this in rewriting logic using Maude, one gets much more than just an interpreter. The point is that now Maude can be used as a *metatool* to obtain powerful *software analysis tools* for the programming language in question, essentially *for free*. The generic formal analysis capabilities of Maude are specialized to get software analysis tools for a given programming language precisely by providing the Maude specification of that language’s semantics. For example, Maude’s generic breadth-first search capability (see Sections 6.4 and 25.4) becomes specialized to a semi-decision procedure for finding violations of invariants in any concurrent program in the language [258, 259]. Similarly, Maude’s generic LTL model-checking capability (see Chapter 13) becomes specialized to a model

checker for the programming language in question [258, 259]. When model checking a concurrent program in a given language, in addition to the enormous state space reduction afforded by rewriting logic’s distinction between equations and rules [258, 259], it is even possible to obtain a further drastic state space reduction by using a *generic* partial order reduction (POR) technique based on transforming the Maude semantic definition of the language into a semantically equivalent POR-enabled one [155].

It is also possible to base on the Maude semantics of a programming language various *deductive tools and analyses*. For example, Ahrendt, Roth, and Sasse [4, 322] have used the Maude specification of Java developed in the JavaFAN project (see Section 23.2.5) to automatically validate a large set of transformation rules in a dynamic logic for Java Card programs. Both the ASIP+ITP tool [81] mentioned in Section 23.1.1, and the Java+ITP tool [324] discussed in Section 23.2.6 support, to varying degrees, Hoare logic reasoning and ITP dischargement of verification conditions for, respectively, a fragment of a Pascal-like language, and a fragment of sequential Java. In a similar fashion, as further discussed in Section 23.2.3, a Maude semantics of CCS has been extended to a tool to check whether a formula in the Hennessy-Milner modal logic [193] is satisfied by a finite CCS process [368, 370].

22.3 Maude as a metalanguage

Rewriting logic is like a coin with two inseparable sides: one computational and another logical. The generality and expressiveness of rewriting logic as a semantic framework for concurrent computation has also a logical counterpart. Indeed, rewriting logic is also a promising *logical framework* in which many different logics and formal systems can be naturally represented and interrelated [231, 229]. Furthermore, it has also good properties as *metalogical framework*, in which one can reason about the metalogical properties of the represented logics [18, 19]. With Maude, such representations can then be used to generate a wide range of formal tools.

22.3.1 Representing, mapping, and reasoning about logics

The basic idea is that we can represent a logic \mathcal{L} with a finitary syntax and inference system within rewriting logic by means of a representation map

$$\Phi : \mathcal{L} \longrightarrow RWLogic.$$

The map Φ should be *conservative*, that is, it should preserve and reflect theoremhood. The reason why rewriting logic is a good framework is that the formulas of a logic \mathcal{L} can typically be axiomatized by an equational theory, and the rules of inference can then be typically understood as rewrite rules, that may be conditional if the inference rules have “side conditions.” Therefore, the mappings Φ are usually very simple and direct. In addition, using reflection

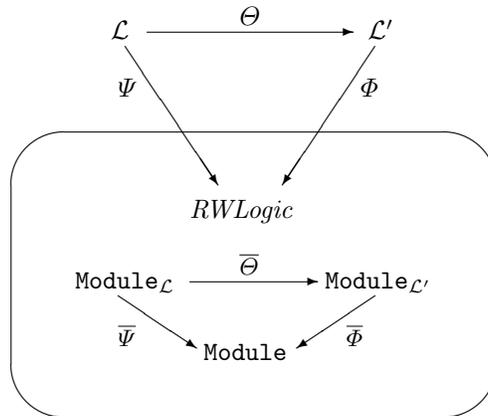
we can define and execute a map Φ of this kind *inside rewriting logic itself* by means of an equationally defined map

$$\bar{\Phi} : \text{Module}_{\mathcal{L}} \longrightarrow \text{Module}.$$

The map $\bar{\Phi}$ can be defined by extending the universal theory \mathcal{U} , which has a sort **Module** representing rewrite theories (see Section 14.3), with the equational definition of a new sort $\text{Module}_{\mathcal{L}}$ whose terms represent (finitely presentable) theories in the logic \mathcal{L} .

In fact, we can go a step further, and represent inside rewriting logic a mapping $\Theta : \mathcal{L} \longrightarrow \mathcal{L}'$ between any two finitary logics \mathcal{L} and \mathcal{L}' as an equationally defined function $\bar{\Theta} : \text{Module}_{\mathcal{L}} \longrightarrow \text{Module}_{\mathcal{L}'}$. If the map Θ is computable, then, by a metatheorem of Bergstra and Tucker [21], it is possible to define the function $\bar{\Theta}$ by means of a finite set of Church-Rosser and terminating equations. That is, such functions can be effectively defined and executed within rewriting logic.

In summary, using reflection, computable mappings between logics, including maps representing other logics in rewriting logic, can be internalized and executed within rewriting logic, as indicated in the picture below.



There is yet another reason why rewriting logic is very useful for logical framework applications. Thanks to reflection and the existence of initial models, rewriting logic can not only be used as a logical framework in which the deduction of a logic \mathcal{L} can be faithfully simulated, but also as a *metalogical framework* in which we can reason about the metalogical properties of a logic \mathcal{L} . Basin, Clavel, and Meseguer have studied the use of reflection, induction, and Maude’s inductive theorem prover (see Section 23.1.1) enriched with reflective reasoning principles to prove such metalogical properties [17, 18, 19].

A good number of examples of representations of logics in rewriting logic have been given by different authors, often in the form of executable specifications, including:

1. The logics represented by Martí-Oliet and Meseguer in [231, 229], including equational logic, Horn logic with equality, linear logic, logics with quantifiers, and any sequent calculus presentation of a logic for a very general notion of “sequent”;
2. The map $LinLogic \rightarrow RWLogic$ in [231, 229] representing propositional linear logic was subsequently specified in a reflective way in Maude by Clavel and Martí-Oliet [65, 66];
3. The map $HOL \rightarrow Nuprl$ between the logics of the HOL and Nuprl theorem provers has been specified in Maude by Stehr, Naumov, and Meseguer [276, 343];
4. Dowek, Hardin, and Kirchner have presented (what obviously are) rewrite theories for doing deduction *modulo* an equational theory of equivalence between formulas specified by the equations E of the rewriting logic axiomatization, both for first-order and higher-order logics [108, 106, 107];
5. The connections with rewriting logic of that work have been made explicit by Viry, who has given a coherent sequent calculus rewrite theory in this style in [376, 377] (see also [100]);
6. Stehr and Meseguer have defined a natural representation map $PTS \rightarrow RWLogic$ of pure type systems (a parametric family of higher-order logics generalizing the λ -cube) in rewriting logic [340]; and
7. Bruni, Meseguer, and Montanari have defined a mapping $Tile Logic \rightarrow RWLogic$ from tile logic into rewriting logic that can be used to execute tile logic specifications [35, 37, 38, 39, 40].

22.3.2 Specifying and building formal tools

Theorem provers and other formal tools have underlying *inference systems* that can be naturally specified and prototyped in rewriting logic. Furthermore, the strategy aspects of such tools and inference systems can then be specified by rewriting strategies. In Maude, formal tools have typically a *reflective* design that, by metarepresenting theories as data, easily allows inference steps that may transform the object theory. Strategies are then specified as rewrite theories controlling the application of such metalevel inference rules at the meta-metalevel. A simple example of this reflective methodology, namely, an order-sorted unification procedure modulo commutativity, has been explained in detail in Sections 16.1 and 19.7. Several formal tools developed in this reflective way are part of the Maude formal environment, namely, an inductive theorem prover (see Section 23.1.1); Church-Rosser (see Section 23.1.3), coherence (see Section 23.1.4), termination (see Section 23.1.2), and sufficient completeness (see Section 23.1.5) checkers; a Knuth-Bendix completion tool; Real-Time Maude (see Section 23.1.6); and the Maude predicate abstraction tool (see Section 23.1.7) [71, 72, 73, 74, 109, 110, 120, 119, 117, 187]. Also closely related to Maude itself is the Full Maude tool (see Part II), which extends Maude with special syntax for object-oriented specifications, and with

a rich *module algebra* of parameterized modules and module composition operations [121, 111, 125].

This method of building formal tools is not restricted to Maude-related tools. One can generate tools from their rewriting logic specifications for *any* finitary logic, such as:

1. A proof assistant built by Stehr for the Open Calculus of Constructions, which extends Coquand and Huet's calculus of constructions with equational reasoning and a flexible universe hierarchy [336] (see Section 23.2.1);
2. The Maude Action Tool [33] already mentioned in Section 22.2;
3. The Maude MSOS tool, which supports execution and analysis of MSOS semantic definitions [53, 55] (see Section 23.2.2);
4. CCS and LOTOS execution and verification environments developed by Verdejo and Martí-Oliet [368, 366, 369, 371, 365] (see Section 23.2.3);
5. The MSR Tool, which supports specification, simulation, and analysis of cryptographic protocols expressed in the multiset rewriting formalisms [51, 52, 313] (see Section 23.2.4);
6. The JavaFAN tool, which supports efficient execution and formal analysis of Java and JVM programs [156, 153] (see Section 23.2.5);
7. The Java+ITP tool, which supports Hoare logic reasoning on a subset of sequential Java and discharging of first-order verification conditions in the Maude ITP [324] (see Section 23.2.6);
8. The ITP/OCL tool to validate UML static diagrams with respect to OCL invariants [75] (see Section 23.2.7);
9. The Pathway Logic Assistant, a tool to visualize and formally analyze biological processes, such as metabolism or signaling, specified in Maude as rewrite theories [138, 355, 354] (see Section 23.2.8);
10. A tool by Havelund and Roşu for testing linear temporal logic formulas on finite execution traces [182, 183, 184, 185, 319];
11. A tool by Fischer and Roşu to automatically check an abstract interpretation against user-given properties [161].

Furthermore, as already explained in Section 22.2, tools like JavaFAN are not an isolated instance, but a legion; since any rewriting logic semantics of a programming language in Maude can be automatically endowed not only with an interpreter, but also with a semi-decision procedure for invariant failures, an LTL model checker, and even with partial order reduction capabilities [258, 259, 155].

22.4 Modeling and analysis of networks and distributed systems

22.4.1 Distributed architectures and components

It is very important to detect errors and inconsistencies as early as possible in the software design cycle. For this reason, formal approaches that can

increase the analytic power of architectural notations such as architectural description languages (ADLs) and object-oriented design formalisms like UML are quite valuable. A related concern is the formal specification and analysis of *distributed component architectures*.

Rewriting logic has been used by several authors in these areas to allow formal analysis of software designs and, in some cases, to support code generation from the associated executable specifications. Relevant work in this direction includes:

1. work of Nodelman and Talcott representing both the Wright architecture description language and its underlying CSP semantics in Maude;
2. work of Durán, Meseguer, and Talcott on semantic interoperation of heterogeneous software architectures based on their rewriting logic semantics [263] (see also Appendix E of [67]);
3. work of Wirsing and Knapp on the systematic transformation of UML diagrams and similar object-oriented notations into formal executable rewriting logic specifications in Maude, which can then be used to execute and formally analyze the designs, and even to generate code in a conventional language such as Java [382, 209, 210, 383];
4. work by Fernández and Toval formalizing in Maude the UML metamodel and its evolution [361, 159], with applications to formal analysis and prototyping [158, 362];
5. work by Nakajima and Futatsugi on the transformation of scenario-based object-oriented design diagrams for execution and formal analysis [275];
6. work by Talcott on a rewriting logic semantics for actor systems axiomatized by actor theories [348, 349, 350, 352, 353]; such systems can be extended by an algebra of *components*, that are encapsulated by interfaces, and that can include actors, messages, and other (sub-)components; in addition Talcott has developed methods to reason formally about such open component systems;
7. work by Denker, Meseguer, and Talcott on a general middleware architecture for composable distributed communication services such as fault-tolerance, security, and so on, that can be composed and can be dynamically added to selected subsets of a distributed communications system [92];
8. work by Meseguer and Talcott on models of distributed object reflection proposing the “Russian Dolls” model [262];
9. work by Najm and Stefani giving a rewriting logic semantics to the operational subset of the Reference Model for Open Distributed Processing (RM-ODP) [271, 272, 273] (see also [130, 129]);
10. work by Nakajima that uses rewriting logic specifications in CafeOBJ to formally specify the architecture of WEB-NMS, a Java/ORB implementation of a network management system [274];

11. work by Albarrán, Durán, and Vallecillo on interoperating Maude executable specifications with distributed component platforms such as CORBA and SOAP [6, 7, 8];
12. work by Denker and Talcott modeling architectures for goal operated autonomous systems [98, 99] (also see <http://www.csl.sri.com/~denker/remoteAgents/> and <http://pagoda.csl.sri.com>);
13. work by Clavel and Egea on modeling UML class diagrams and validating OCL invariants against them by means of the ITP/OCL tool [132, 75]; and
14. work by Boronat, Carsí, and Ramos on using Maude within the MOMENT framework [26], whose purpose is to provide support for model management within the Eclipse Modeling Framework, and in particular for model transformations through the standard QVT language [280].

22.4.2 Specification and analysis of communication protocols

Because of its flexibility to model distributed objects with different modes of communication and interaction, rewriting logic is very well suited to specify and analyze communication protocols, including cryptographic protocols, and, more generally, network software such as active network programming languages, active network algorithms, and network management systems.

Applications of this kind include:

1. work by researchers at Stanford, SRI, and at the Computer Communications Research Group at University of California Santa Cruz using Maude to analyze the early design of a new reliable broadcast protocol for active networks [88, 89];
2. work of Wang, Gunter, and Meseguer using Maude to formally specify and analyze a PLAN active network algorithm [379];
3. work by Ölvecký, Keaton, Meseguer, Talcott, and Zabele using Real-Time Maude (see Section 23.1.6) to specify and analyze the AER/NCA suite of active network protocol components for reliable multicast [285, 292];
4. work by Lien using Real-Time Maude to specify and analyze the NORM multicast protocol developed by the Internet Engineering Task Force [221];
5. work by Ölvecký and Thordvalsen using Real-Time Maude to specify and analyze the OGDC wireless sensor network algorithm [359, 293];
6. work of Verdejo, Pita, and Martí-Oliet on the Maude specification and verification of the FireWire leader election protocol [372];
7. work of Mason and Talcott on modeling, simulation and analysis of network architectures and communication protocols [239];
8. work of Pita and Martí-Oliet using the reflective features of Maude to specify some management processes of broadband telecommunication networks [299, 300, 301]; and
9. work of Stehr and Talcott modeling the Spread group communication protocols http://formal.cs.uiuc.edu/stehr/spread_eng.html; and of Gutierrez-

Nolasco, Venkatasubramanian, Stehr, and Talcott on the formal specification and analysis of Secure Spread [178].

22.4.3 Modeling and analysis of security protocols

Security is a concern of great practical importance for many systems, making it worthwhile to subject system designs and implementations to rigorous formal analysis. Security, however, is *many-faceted*: on the one hand, we are concerned with properties such as *secrecy*: malicious attackers should not be able to get secret information; on the other, we are also concerned with properties such as *availability*, which may be destroyed by a denial-of-service (DoS) attack: a highly reliable communication protocol ensuring secrecy may be rendered useless because it spends all its time checking spurious signatures generated by a DoS attacker. Rewriting logic and Maude have been successfully applied to analyze security properties, including both secrecy and availability, for a wide range of systems. More generally, using distributed object-oriented reflection techniques [92, 262], it is possible to analyze *tradeoffs* between different security properties, and between them and other system properties; and it is possible to develop system composition and adaptation techniques allowing systems to behave adequately in changing environments.

Work in this general area includes:

1. work of Denker, Meseguer, and Talcott on the specification and analysis of cryptographic protocols using Maude [90, 91] (see also [317]);
2. work of Basin and Denker on an experimental comparison of the advantages and disadvantages of using Maude versus using Haskell to analyze security protocols [20];
3. work of Millen and Denker using Maude to give a formal semantics to the cryptographic protocol specification language CAPSL, and to endow CAPSL with an execution and formal analysis environment [93, 94, 95, 96];
4. work of Gutierrez-Nolasco, Venkatasubramanian, Stehr, and Talcott on the Secure Spread protocol [178];
5. work of Goodloe, Gunter, and Stehr on the formal specification and analysis of the L3A security protocol [176];
6. work of Cervesato, Stehr, and Reich on the rewriting logic semantics of the MSR security specification formalism, leading to the first executable environment for MSR [51, 52, 339]; and
7. work by Agha, Gunter, Greenwald, Khanna, Meseguer, Sen, and Thati on the specification and analysis of a DoS-resistant TCP/IP protocol using probabilistic rewrite theories [2].

A related technique with important security applications is *narrowing*, a symbolic procedure like rewriting, except that rules, instead of being applied by matching a subterm, are applied by unifying the lefthand side with a non-variable subterm. Traditionally, narrowing has been used as a method to solve equations in a confluent and terminating equational theory. In rewriting logic,

narrowing has been generalized by Meseguer and Thati to a semi-decision procedure for *symbolic reachability analysis* [265, 266]. That is, instead of solving equational goals $\exists \mathbf{x}. t = t'$, we solve reachability goals $\exists \mathbf{x}. t \longrightarrow t'$. The relevant point for security applications is that, since narrowing with a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ is performed *modulo* the equations E , this allows more sophisticated analyses than those performed under the usual Dolev-Yao “perfect cryptography assumption.” It is well known that protocols that had been proved secure under this assumption can be broken if an attacker uses knowledge of the algebraic properties satisfied by the underlying cryptographic functions. In rewriting logic we can specify a cryptographic protocol as a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, and can model those algebraic properties as equations in E . Under suitable assumptions that are typically satisfied by cryptographic protocols, narrowing then gives us a complete semi-decision procedure to find attacks *modulo* the equations E ; therefore, any attack making use of algebraic properties can be found this way [265, 266]. Very recent work in this direction by Escobar, Meadows, and Meseguer [143, 144, 145] is using rewriting logic and narrowing to give a precise rewriting semantics to the inference system of one of the most effective analysis tools for cryptographic protocols, namely the NRL Analyzer [242]. Further recent work on narrowing with rewrite theories focuses on generalizing the procedure to so-called “back-and-forth narrowing,” so as to ensure completeness under very general assumptions about the rewrite theory \mathcal{R} [356], and efficient lazy strategies to restrict as much as possible the narrowing search space [150].

22.5 Real-time systems

In many reactive and distributed systems, real-time properties are essential to their design and correctness. Therefore, the question of how systems with real-time features can be best specified, analyzed, and proved correct in the semantic framework of rewriting logic is an important one. This question has been investigated by several authors from two perspectives. On the one hand, an extension of rewriting logic called *timed rewriting logic* has been investigated, and has been applied to some examples and specification languages [212, 286, 333]. On the other hand, a simple way to express real-time and hybrid system specifications *directly* in rewriting logic is presented in [288, 291]. Such specifications are called *real-time rewrite theories* and have “ordinary” rules representing instantaneous transitions that take no time and only change some part of the state, as well as “tick” rewrite rules that model time elapse in the system. Tick rules have the form

$$l : \{t\} \xrightarrow{r} \{t'\} \text{ if } C$$

with r a term denoting the *duration* of the transition (where the time domain can be chosen to be either discrete or continuous), $\{t\}$ representing the *whole* state of a system, and C an equational condition. By making the clock an

explicit part of the state, these theories can be *desugared* into semantically equivalent ordinary rewrite theories [288, 291]. That is, in the desugared version we can model the state of a real-time or hybrid system as a pair $(\{t\}, r)$, with $\{t\}$ the current state and with r the current global clock time. Then the above rule becomes desugared as

$$l : (\{t\}, x) \longrightarrow (\{t'\}, x + r) \text{ if } C$$

where x is a new variable which does not appear in t, t', r or C .

By characterizing equationally the enabledness of each rule and using conditional rules and *frozen* operators (see Section 4.4.9), it is always possible to define tick rules so that instantaneous rules are always given higher priority; that is, so that a tick rule can never fire when an instantaneous rule is enabled [291]. When time is continuous, tick rules are typically *nondeterministic*, in the sense that the time r advanced by the rule is not uniquely determined, but is instead a parametric expression. In such cases, tick rules need a *time sampling strategy* to choose suitable values for time advance. For dense time such a time sampling strategy would in general make search and model-checking analyses feasible (when model checking time-bounded properties) but incomplete; however, under very widely applicable and relatively easy to check conditions such analyses turn out to be complete [290].

Besides being able to show that a wide range of known real-time models (including, for example, timed automata, hybrid automata, timed Petri nets, and timed object-oriented systems), and of discrete or dense time values, can be naturally expressed in a direct way in rewriting logic (see [288]), an important advantage of this approach is that one can use an existing implementation of rewriting logic like Maude to execute and analyze real-time specifications. Indeed, Real-Time Maude [283, 287, 289, 291] is a specification language and a formal tool built in Maude by reflection (see Section 23.1.6) which provides special syntax to specify real-time systems. It systematically exploits the underlying Maude efficient rewriting, search, and LTL model-checking capabilities to both execute and formally analyze real-time specifications. A number of substantial applications have been specified and analyzed in Real-Time Maude, including the AER-NCA active network protocol suite [285, 292]; the NORM multicast protocol [221]; the OGDC wireless sensor network algorithm [359, 293]; and the CASH adaptive scheduling algorithm [284].

22.6 Probabilistic systems

Many systems are probabilistic in nature. This can be due either to the uncertainty of the environment in which they must operate, such as message losses and other failures in an unreliable environment, or to the probabilistic nature of some of their algorithms, or to both. In general, particularly for distributed systems, both probabilistic and nondeterministic aspects may coexist, in the sense that different transitions may take place nondeterministically, but the

outcomes of some of those transitions may be probabilistic in nature. To specify systems of this kind, rewrite theories have been generalized to *probabilistic rewrite theories* in [214, 215, 3]. Rules in such theories are *probabilistic rewrite rules* of the form

$$l : t(\mathbf{x}) \rightarrow t'(\mathbf{x}, \mathbf{y}) \text{ if } \text{cond}(\mathbf{x}) \text{ with probability } \mathbf{y} := \pi_l(\mathbf{x})$$

where the first thing to observe is that the term t' has new variables \mathbf{y} disjoint from the variables \mathbf{x} appearing in t . Therefore, such a rule is *nondeterministic*; that is, the fact that we have a matching substitution θ for the variables \mathbf{x} such that $\theta(\text{cond})$ holds does not uniquely determine the next state fragment: there can be many different choices for the next state, depending on how we instantiate the extra variables \mathbf{y} in t' . In fact, we can denote the different such next states by expressions of the form $t'(\theta(\mathbf{x}), \rho(\mathbf{y}))$, where θ is fixed as the given matching substitution, but ρ ranges over all the possible substitutions for the new variables \mathbf{y} . The probabilistic nature of the rule is expressed by the notation: *with probability* $\mathbf{y} := \pi_l(\mathbf{x})$, where $\pi_l(\mathbf{x})$ is a probability distribution *which may depend on the matching substitution* θ . We then choose the values for \mathbf{y} , that is, the substitution ρ , probabilistically according to the distribution $\pi_l(\theta(\mathbf{x}))$.

The fact that the probability distribution may depend on the substitution θ can be illustrated by means of a simple example. Consider a battery-operated clock. We may represent the state of the clock as a term `clock(T, C)`, with T a natural number denoting the time, and C a positive real number denoting the amount of battery charge. Each time the clock ticks, the time is increased by one unit, and the battery charge slightly decreases; however, the lower the battery charge, the greater the chance that the clock will stop, going into a state of the form `broken(T, C')`. We can model this system by means of the probabilistic rewrite rule

```

r1 [tick] : clock(T, C)
=> if B
    then clock(s(T), C - (C / 1000.0))
    else broken(T, C)
fi
with probability B := BERNOULLI(C / 1000.0) .

```

that is, the probability of the clock breaking down instead of ticking normally *depends on the battery charge*, which is here represented by the battery-dependent bias of the coin in a Bernoulli trial. Note that here the new variable on the rule's righthand side is the Boolean variable B , corresponding to the result of tossing the biased coin.

As shown in [214], probabilistic rewrite theories can express a wide range of models of probabilistic systems, including continuous-time Markov chains [346], probabilistic nondeterministic systems [308, 327], and generalized semi-Markov processes [171]. They can also naturally express probabilistic object-based distributed systems [215, 3], including real-time ones. Yet another class

of probabilistic models that can be simulated by probabilistic rewrite theories is the class of object-based stochastic hybrid systems discussed in [260].

The PMAude language [215, 3] is an experimental specification language whose modules are probabilistic rewrite theories. Note that, due to their non-determinism, probabilistic rewrite rules *are not directly executable*. However, probabilistic systems specified in PMAude *can be simulated in Maude*. This is accomplished by transforming a PMAude specification into a corresponding Maude specification in which actual values for the new variables appearing in the righthand side of a probabilistic rewrite rule are obtained by *sampling* the corresponding probability distribution functions. This theory transformation uses three key Maude modules as basic infrastructure, namely, COUNTER, RANDOM, and SAMPLER. The predefined module COUNTER (see Section 9.3) provides a built-in strategy for the application of the nondeterministic rewrite rule

```
rl counter => N:Nat .
```

that rewrites the constant `counter` to a natural number. The built-in strategy applies this rule so that the natural number obtained after applying the rule is exactly the successor of the value obtained in the preceding rule application. The RANDOM module is a predefined Maude module (see Section 9.3) providing a (pseudo-)random number generator function called `random`. The SAMPLER module supports sampling for different probability distributions. It has a rule

```
rl [rnd] : rand => float(random(counter + 1) / 4294967295) .
```

which rewrites the constant `rand` to a floating point number between 0 and 1 pseudo-randomly chosen according to the uniform distribution. SAMPLER has rewrite rules supporting sampling according to different probability distributions; this is based on first sampling a floating point number between 0 and 1 pseudo-randomly chosen according to the uniform distribution by means of the above `rnd` rule. For example, to sample the Bernoulli distribution we use the following operator and rewrite rule in SAMPLER:

```
op BERNOULLI : Float -> Bool .
rl BERNOULLI(R) => rand < R .
```

that is, to sample a result of tossing a coin with bias `R`, we first sample the uniform distribution. If the sampled value is strictly smaller than `R`, then the answer is `true`; otherwise the answer is `false`. Any discrete probability distribution on a finite set can be sampled in a similar way. The ordinary Maude specification that *simulates* the PMAude specification for a clock with the above `tick` probabilistic rewrite rule imports COUNTER, RANDOM, and SAMPLER, and has then a corresponding Maude rewrite rule

```
rl [tick] : clock(T, C)
=> if BERNOULLI(C / 1000.0)
then clock(s(T), C - (C / 1000.0))
```

```

else broken(T, C)
fi .

```

In general, provided that sampling for the probability distributions used in a PMAude module are supported in the underlying SAMPLER module, we can associate with it a corresponding Maude module. We can then use this associated Maude module to perform Monte Carlo simulations of the probabilistic systems thus specified. The point is that, given a probabilistic rewrite theory and a term t describing a given state, there can be several different rewrites, perhaps with different rules, at different positions, and with different matching substitutions, that can be applied to t . Therefore, the choice of rule, position, and substitution is *nondeterministic*. To eliminate all nondeterminism, at most one rule at exactly one position and with a unique substitution should be applicable to any term t . As explained in [3], for many systems, including probabilistic real-time object-oriented systems, this can be naturally achieved, essentially by scheduling events at real-valued times that are all different, because we sample a continuous probability distribution on the real numbers. Provided all nondeterminism has been eliminated from the original PMAude module, we can then use the results of such Monte Carlo simulations to perform a *statistical model-checking analysis* of the given system to verify certain properties. For example, for a PMAude specification of a TCP/IP protocol variant that is resistant to denial-of-service (DoS) attacks, we may wish to establish that, even if an attacker controls 90% of the network bandwidth, it is still possible for the protocol to establish a connection in less than 30 seconds with 99% probability. Properties of this kind, including properties that measure quantitative aspects of a system, can be expressed in the QATEX probabilistic temporal logic [3], and can be model checked using the VeStA tool [328]. See [2] for a substantial case study specifying a DoS-resistant TCP/IP protocol as a PMAude module, performing Monte Carlo simulations by means of its associated Maude module, and formally analyzing in VeStA its properties, expressed as QATEX specifications, according to the methodology just described. More recently, several object-based stochastic hybrid system case studies have been specified in PMAude, and have been simulated in Maude. Then, relevant formal properties for each case study, expressed as QATEX specifications, have been statistically model checked in VeStA using Monte Carlo simulations performed in Maude [260].

22.7 Modeling and analysis of biological systems

Biology lacks at present adequate mathematical models that can provide something analogous to the analytic and predictive power that mathematical models provide for, say, physics. Of course, the mathematical models of chemistry describing, say, molecular structures are still applicable to biochemistry. The problem is that they *do not scale up* to something like a cell, because they

are too low-level. One can of course model biological phenomena at different *levels of abstraction*. Higher, more abstract levels seem both the most crucial and the least supported. The most abstract the level, the better the chances to scale up.

All this is analogous to the use of different levels of abstraction to model digital systems. There are great scaling up advantages in treating digital systems and computer designs at a *discrete* level of abstraction, above the continuous level provided by differential equations, or, even lower, the quantum electrodynamics level. The discrete models, when they can be had, can also be more *robust and predictable*: there is greater difficulty in predicting the behavior of a system that can only be modeled at lower levels. Indeed, the level at which biologists like to reason about cell behavior is typically the discrete level; however, at present descriptions at this level consist of semi-formal notations for the elementary reactions, together with informal and potentially ambiguous notations for things like pathways, cycles, feedback, etc. Furthermore, such notations are static and therefore offer little predictive power. What are needed are *new computable mathematical models of cell biology* that are at a high enough level of abstraction so that they fit biologist's intuitions, make those intuitions mathematically precise, and provide biologists with the *predictive power* of mathematical models, so that the consequences of their hypotheses and theories can be analyzed, and can then suggest laboratory experiments to prove them or disprove them.

Rewriting logic seems ideally suited for this task. The basic idea is that we can *model a cell as a concurrent system* whose concurrent transitions are precisely its biochemical reactions. In fact, the chemical notation for a reaction like $A B \longrightarrow C D$ is *exactly* a rewriting notation. In this way we can develop *symbolic* models of biological systems which we can then analyze just as we would analyze any other rewrite theory, for example using search and model checking.

Implicit in the view of modeling a cell as a rewrite theory (Σ, E, R) is the idea of modeling the cell states as elements of an algebraic data type specified by (Σ, E) . This can of course be done at different levels of abstraction. We can, for example, introduce basic sorts such as **AminoAcid**, **Protein**, and **DNA** and declare the most basic building blocks as constants of the appropriate sort. For example,

```
ops T U Y S K P : -> AminoAcid [ctor] .
ops 14-3-3 cdc37 GTP Hsp90 Raf1 Ras : -> Protein [ctor] .
```

But sometimes a protein is *modified*, for example by one of its component amino acids being phosphorylated at a particular *site* in its structure. Consider, for example, the c-Raf protein, denoted above by **Raf1**. Two of its **S** amino acid components can be phosphorylated at sites, say, 259 and 261. We then obtain a modified protein that we denote by the symbolic expression

```
[Raf1 - phos(S 259) phos(S 621)]
```

A fragment, relevant for this example, of the signature Σ needed to symbolically express and analyze such modified proteins is given by the following sorts, subsorts, and operators:

```
sorts Site Modification ModSet .
subsort Modification < ModSet .

op phos : Site -> Modification [ctor] .
op none : -> ModSet [ctor] .
op __ : ModSet ModSet -> ModSet [ctor assoc comm id: none] .
op __ : AminoAcid Int -> Site [ctor] .
op [_-] : Protein ModSet -> Protein [ctor right id: none] .
```

Proteins can stick together to form *complexes* and complexes can combine with proteins or other complexes to form larger complexes. This can be modeled by the following subsort and operator declarations:

```
sorts Thing Complex .
subsorts Protein Complex < Thing .
op _:_ : Thing Thing -> Complex [ctor comm] .
```

In the cell, proteins and other molecules exist in “soups,” such as the cytosol, or the soups of proteins inside the cell and nucleus membranes, or the soup inside the nucleus. All these soups, as well as the “structured soups” making up the different structures of the cell, can be modeled by the following fragment of sort, subsort, and operator declarations:

```
sort Soup .
subsort Thing < Soup .
op empty : -> Soup [ctor] .
op __ : Soup Soup -> Soup [ctor assoc comm id: empty] .

sorts Enclosure MemType .
subsort Enclosure < Soup .

ops CM NM : -> MemType [ctor] .
op {_|_{_}} : MemType Soup Soup -> Enclosure [ctor] .
```

that is, soups are made up out of complexes and individual proteins by means of the above binary “soup union” operator (with juxtaposition syntax) that combines two soups into a bigger soup. This union operator models the fluid nature of soups by obeying *associative and commutative* laws. A *cell* is an *enclosure*, composed of two subsoups, namely the soup in the membrane and that inside the membrane. The nucleus is also an enclosure with a similar abstract structure. This is expressed by the operator $\{_|_{_}\}$, where different enclosure types are distinguished by the `MemType` argument, `CM` for Cell Membrane and `NM` for Nuclear Membrane. The following expression gives a partial description of a cell:

```

{CM | cm [Ras - GTP]
  {cy (([Raf1 - phos(S 259) phos(S 621)]
        : (cdc37 : Hsp90)) : 14-3-3)
    {NM | nm {n}} }}

```

where `cm` denotes the rest of the soup in the cell membrane, `cy` denotes the rest of the soup in the cytoplasm, and `nm` and `n` likewise denote the remaining soups in the nucleus membrane and inside the nucleus.

Once we have cell states defined as elements of an algebraic data type specified by (Σ, E) , the only missing information has to do with cell *dynamics*, that is, with its biochemical reactions. They can be modeled by suitable rewrite rules R , giving us a full model (Σ, E, R) . Consider, for example, the following reaction described in a survey by Kolch [211]:

“Raf-1 resides in the cytosol, tied into an inactive state by the binding of a 14-3-3 dimer to phosphosterines-259 and -621. When activation ensues, Ras-GTP binding ... brings Raf-1 to the membrane.”

We can model this reaction by the following rewrite rule:

```

r1 [10] :
{CM | cm [Ras - GTP]
  {cy (([Raf1 - phos(S 259) phos(S 621)]
        : (cdc37 : Hsp90)) : 14-3-3)}}
=> {CM | cm ([Ras - GTP] : (([Raf1 - phos(S 259) phos(S 621)]
                             : (cdc37 : Hsp90)) : 14-3-3))
    {cy}} .

```

where `cm` and `cy` are variables of sort `Soup`, representing, respectively, the rest of the soup in the cell membrane, and the rest of the soup inside the cell (including the nucleus). Note that in the new state of the cell represented by the righthand side of the rule, the complex has indeed migrated to the membrane.

Given a type of cell specified as a rewrite theory (Σ, E, R) , rewriting logic then allows us to reason about the *complex changes* that are possible in the system, given the basic changes specified by R . That is, we can then use (Σ, E, R) together with Maude and its supporting formal tools to simulate, study, and analyze *cell dynamics*. In particular, we can study in this way *biological pathways*, that is, complex processes involving partially ordered sets of biological reactions and leading to important cell changes. In particular we can:

- observe progress in time of the cell state by *symbolic simulation*, obtaining a corresponding trace;
- answer questions of *reachability* from a given cell state to another state satisfying some property; this can be done both *forwards* and *backwards*;
- answer more complex questions by *model checking* LTL properties; and

- do *meta-analysis* of proposed models of the cell to weed out spurious conjectures and to identify *consequences* of a given model that could be settled by *experimentation*.

As part of the Pathway Logic project (<http://www.csl.sri.com/users/clt/PLweb/>), substantial sets of rules have been curated from the literature using representations similar to that discussed above, and tools have been developed to help the biologist visualize and query these models (see Section 23.2.8 for a discussion of the Pathway Logic Assistant) using familiar concepts and graphical representations [138, 355]. These tools make use of reflection to analyze and transform the models and Maude's search and model-checking tools are used to answer queries. Maude models satisfying certain conditions can be transformed into 1-safe Petri nets and analyzed using special purpose tools such as LoLA [325] that take advantage of the simple structure and can answer reachability queries for very large models (Petri nets with more than 3000 places and transitions) within seconds. The paper [355] contains a good discussion of related work in this area, using other formalisms, such as Petri nets or process calculi, that can also be understood as particular rewrite theories; and shows how cell behavior can be modeled with rewrite rules and can be analyzed *at different levels of abstraction*, and even across such levels. An important future direction for this work is to use more advanced specification and analysis techniques—for example, techniques based on real-time and probabilistic rewrite theories as discussed in Sections 22.5 and 22.6—so as to develop a *range of complementary models* for cell biology. In this way, aspects such as the probabilistic nature of cell reactions, their dependence on the concentration of certain substances, and their real-time behavior could also be modeled, and more sophisticated analyses could be developed.

Some Tools

with Christiano Braga,
Azadeh Farzan,
Joe Hendrix,
Peter Ölveczky,
Miguel Palomino,
Ralf Sasse,
Mark-Oliver Stehr,
and Alberto Verdejo

This chapter describes some existing Maude-based tools that are available for download. The first section describes tools concerned with analysis of either Maude specifications, or of extensions of such specifications. The second section describes tools that are not Maude-specific; they are instead concerned with specification and analysis in various application domains and support domain-specific notations. For each tool, its description addresses the following questions:

- What does the tool do?
- When would you want to use it?
- How can the tool and documentation be obtained?

The material in this chapter is a reformulation and adaptation of the material kindly provided to us by the authors of each tool. We cordially thank them all for their help.

23.1 Maude tools

23.1.1 The ITP: an Inductive Theorem Prover

The ITP tool [79] is a theorem-proving *assistant*. It can be used to interactively verify properties of membership equational specifications with respect to its *ITP-models*. An ITP-model is a model of the specification such that:

- it is an inductive model, in the sense that the sets interpreting the sorts in the model are inductively generated by the membership axioms defining the sorts in the specification;

- it is a standard model of the theory of arithmetic for the integer numbers.

By default, the ITP tool assumes that the sets interpreting the sorts are also *freely* generated [140], that is, two different ground constructor-terms always denote different elements. However, users can “customize” the tool for verifying properties about models that do not satisfy the freeness requirement; of course, some of the commands, whose soundness is based on this requirement, would not be then available.

An important feature of the proposed semantic framework is that it supports proofs by structural induction and complete induction. Another interesting feature is that incompletely specified operations can be reasoned about so as to support incrementality. That is, unlike in most reasoning systems, including RRL [202] and ACL2 [205], operations do not have to be completely specified before inductive properties about them can be verified mechanically.

Notice that the class of the ITP-models includes the *initial* model of the specification,¹ but possibly also many other models. This tolerance provides the extra freedom that is needed to inductively reason about incompletely specified operations.

The ITP tool is a Maude program. It comprises over 8000 lines of Maude code that make extensive use of the reflective capabilities of the system. In fact, rewriting-based proof simplification steps are directly executed by the powerful underlying Maude rewriting engine.

The ITP tool is currently available as a web-based application that includes a *module editor*, a *formula editor*, and a *command editor*. These editors allow users to create and modify their specifications, to formalize properties about them, and to guide the proofs by filling in and submitting web forms. The web application also offers a goal viewer, a script viewer, and a log viewer. They generate web pages that allow the user to check, print, and save the current state of a proof, the commands that have guided it, and the logs generated in the process by the Maude system.

The Web ITP tool can be executed in two different ways: either as a remote or as a local application. It comprises 2000 lines of Maude code, 3000 lines of JSP, and 7500 lines of Java. The only requirements to run the remote application are a computer with an internet connection and with JDK 1.4.1 installed (it should be available on most computers) and a browser. The remote application can be accessed at the URL <http://maude.sip.ucm.es:8080/webitp/>.

Running the ITP tool as a local application is more demanding: in addition to JDK 1.4.1 and a browser, it is also necessary to have the Tomcat server installed, as well as Maude and the files containing the specification of the ITP. The concrete details of the files needed and the steps to follow to complete the installation can be found at <http://maude.sip.ucm.es/itp/>.

¹ In the initial model [248], sorts are interpreted as the smallest sets satisfying the axioms in the theory, and equality is interpreted as the smallest congruence satisfying those axioms.

The ITP is still an experimental tool, but the results obtained so far are quite encouraging. The ITP tool is the only theorem prover at present that supports reasoning about membership equational logic specifications. The powerful integration of term rewriting with a decision procedure for linear arithmetic with uninterpreted function symbols [80], while also available in other rewriting-based theorem provers like RRL [201], has been easily and efficiently implemented in the ITP by exploiting the reflective design of the tool and the reflective capabilities of the Maude system. This fact has encouraged us to plan to add other decision procedures to our tool in the near future. Another interesting extension of the tool is the implementation of the cover set induction method [384], a feature already available in RRL [202].

23.1.2 The Maude Termination Tool

The Maude Termination Tool (MTT) is a tool that checks the termination of Maude equational specifications. MTT takes Maude functional modules as inputs and tries to prove them terminating by using a number of existing termination tools as backends. Maude, as other equational and rule-based programming languages, has expressive features such as:

- advanced typing constructs including sorts, subsorts, kinds, and memberships;
- matching modulo;
- evaluation strategies; and
- very general conditional rules.

Proving termination of expressive equational programs having such features is nontrivial, since some of these features are not supported by standard termination methods and tools. Yet, the use of such features may be essential to ensure termination. MTT uses two theory transformations described in [119, 117] to bridge the gap between expressive equational programs and conventional termination tools for term rewriting systems, which are used as backends. It can send the transformed termination problems to any tool supporting the TPDB syntax currently shared by many such tools (see <http://www.lri.fr/~marche/termination-competition>). In particular it can use CiME, MU-TERM, and AProVE as backends.

The tool implementation distinguishes two parts:

- a reflective Maude specification implements the theory transformations described in [119, 117], and
- a Java application connects Maude to various term rewriting termination tools—including CiME, MU-TERM, and AProVE—and provides a graphical user interface.

The Java application is in charge of sending the Maude specification provided by the user to Maude to perform transformations. Depending on the selections, one transformation or another will be accomplished. The resulting unsorted

unconditional rewriting system obtained from such transformations is proved terminating by using the above-mentioned tools as backends.

MTT is based on the theoretical work described in [119, 117], and has been developed by Francisco Durán, with the collaboration of David Martín. The application, its source code, and all the related information is available at <http://www.lcc.uma.es/~duran/MTT>. The application itself consists of a single jar file; the reflective Maude specification of the transformations from Maude specifications to unconditional untyped specifications is also provided. The tool depends on the external tools CiME, MU-TERM, and AProVE, which must be available in the system (in addition to Maude, of course). To alleviate the requirements on external tools, the application includes support for connecting to the external tools remotely. That is, MTT can interact either with a local copy of the termination tool, which it internally executes, or with a remote copy of it. This feature is particularly attractive for those platforms for which there is no version available of some of the tools (including Maude itself). For example, one may be running MTT on a Windows box which connects to the external tools running on different machines. Of course, the Java Runtime Environment is required for the execution of the jar file.

Examples making use of the MTT can be found in Sections 12.4 and 13.4.

23.1.3 The Church-Rosser Checker

The *Church-Rosser Checker* (CRC) is a tool to help checking whether an order-sorted equational specification satisfies the Church-Rosser property. The tool can be used to prove such a property for equational specifications in Maude, that is, for Maude functional modules.

The goal of *executable* equational specification languages is to make *computable* the abstract data types specified in them by initial algebra semantics. In practice this is accomplished by using specifications that are *ground* Church-Rosser and terminating, so that the equations can be used from left to right as simplification rules; the result of evaluating an expression is then the canonical form that stands as a unique representative for the equivalence class of terms equal to the original term according to the equations. For order-sorted specifications, being Church-Rosser and terminating means not only confluence—so that a unique normal form will be reached—but also a *sort decreasingness* property, namely that the normal form will have the least possible sort among those of all other equivalent terms. Therefore, the tool's output consists of a set of critical pairs and a set of membership assertions that must be shown, respectively, ground-joinable, and ground-rewritable to a term with the required sort.

For computational purposes it becomes very important to know whether a given specification is indeed ground Church-Rosser and terminating. Establishing the ground Church-Rosser property for a terminating specification is a thorny issue. The problem is that a specification with an initial algebra

semantics can often be ground Church-Rosser even though some of its critical pairs may not be joinable. That is, the specification can often be ground Church-Rosser without being Church-Rosser for arbitrary terms with variables. In such a situation, blindly applying a completion procedure that is trying to establish the Church-Rosser property for arbitrary terms may be both quite hopeless—the procedure may diverge or get stuck because of unorientable rules, and even with success may return a specification that is quite different from the original one—and even unnecessary if the specification was already ground Church-Rosser.

The Church-Rosser Checker can be used to check specifications with an initial algebra semantics that have already been proved terminating and now need to be checked to be ground Church-Rosser. Since, for the reasons mentioned above, user interaction will typically be quite essential, completion is not attempted. Instead, if the specification cannot be shown to be ground Church-Rosser by the tool, proof obligations are generated and are given back to the user as a useful guide in the attempt to establish the ground Church-Rosser property. Since this property is in fact inductive, in some cases an inductive theorem prover can be enlisted to prove some of these proof obligations. In other cases, the user may in fact have to modify the original specification by carefully considering the information conveyed by the proof obligations.

The tool is written entirely in Maude, and is in fact a rewriting logic *executable specification* of the formal inference system that it implements. A complete execution environment for the tool has been built in Maude, and it has been integrated within Full Maude. The tool treats equational specifications as *data* that is manipulated. The CRC computes critical pairs and membership assertions by inspecting the equations in the original specification. This makes a *reflective* design—in which theories become data at the metalevel—ideally suited for the task. Indeed, the fact that rewriting logic is a reflective logic, and that Maude efficiently supports reflective rewriting logic computations is systematically exploited in the tool. The same reflective design has been followed for other tools, like the Knuth-Bendix completion tool [110], and the coherence checker tool (see Section 23.1.4).

The very high level of abstraction at which the tool has been developed has made it relatively easy for us to build it, makes understanding its implementation, as well as its maintenance and extension, much easier than if a conventional implementation, say in C, C++, or Java, had instead been chosen. Thanks to the high performance of the Maude engine, these important benefits in ease of development, understandability, extensibility, and in flexibility for introducing formally-defined proof tactics, are achieved without sacrificing performance. Even though it has not been optimized for performance, and in spite of using reflection and sophisticated rewriting modulo associativity and associativity-commutativity, the tool has competitive performance.

The tool and its documentation are available from <http://maude.lcc.uma.es/CRChC/>. The documentation includes some methodological guidelines

for the use of the tool, and its use is illustrated with some examples; it is also explained there that the issue of finding general inductive proof techniques for proving the ground Church-Rosser property is at the moment an interesting open problem. Additional information on the tool can be obtained from [126, 128, 73, 71, 72] and [120].

Examples making use of the Church-Rosser Checker can be found in Sections 12.4 and 13.4.

A previous version of this tool, together with an inductive theorem prover to prove theorems about such specifications developed by Manuel Clavel, was developed as part of the Cafe project [165] by Francisco Durán. The tool was updated to Maude 2.2 with the help of Miguel Palomino. The current tool only accepts order-sorted conditional specifications where each of the operators has either no equational attributes, or only the commutativity attribute. Furthermore, it is assumed that such specifications do not contain any built-in sort or function, and that they have already been proved terminating (using, for example, the MTT tool described in Section 23.1.2 above). The tool attempts to establish the Church-Rosser property *modulo* the commutativity of some of the operators by checking a sufficient condition. We plan to develop a version of the Church-Rosser Checker that will handle different combinations of `assoc`, `comm`, and `id` attributes.

23.1.4 The Maude Coherence Checker

As explained in Section 6.3 and illustrated with examples in Sections 12.4 and 13.4, *coherence* [377] is a key executability requirement for rewrite theories, and therefore for Maude system modules. It allows reducing the, in general undecidable, problem of computing rewrites of the form $[t]_{E \cup A} \rightarrow [t']_{E \cup A}$, with A a set of equational attributes (associativity, commutativity, identity) for which matching algorithms exist (see Section 4.4.1), to the much simpler and decidable problem of computing rewrites of the form $[t]_A \rightarrow [t']_A$.

The *Maude Coherence Checker* [127, 128] (ChC), which is written in Maude using a reflective design as an extension of Full Maude, provides a *decision procedure* for order-sorted² system modules whose equations and rules are unconditional. The only equational attributes supported by the current implementation are commutativity axioms, declared with the `comm` keyword. A future version will cover other attributes such as associativity-commutativity, and identity. The tool generates a set of *critical pairs*, whose coherence guarantees that of the entire system module [377]. It then checks whether each of these pairs is coherent. The system module given as input to the tool is always assumed to be ground Church-Rosser and terminating. The Church-

² These are system modules whose only membership axioms are *implicit* ones, taking the form of subsort and operator declarations, and such that only equations (resp., only equations and rewrites) can appear in conditional equations (resp., conditional rules).

Rosser Checker (see Section 23.1.3) and the Maude Termination Tool (see Section 23.1.2) can be used to try to prove such properties.

For Maude system modules, which always have an initial model semantics (see Section 6.3), the weaker requirement of *ground coherence*, that is, coherence for ground terms, is enough. When the ChC tool cannot prove coherence—either because this fails, or because the input specification falls outside the class of decidable theories—it outputs a set of *proof obligations* associated with the critical pairs that it could not prove coherent. The user can then interact with the ChC tool to try to prove the ground coherence of the input system module by a constructor-based process of reasoning by cases. In the end, either:

1. all proof obligations are discharged and the module is shown to be ground coherent; or
2. proving ground coherence can be reduced to proving that the inductive validity of a set of equations follows from the equational part of the input system module, for which the ITP can be used (see Section 23.1.1); or
3. it is not possible to reduce some of the proof obligations to inductively proving some equations.

Case (3) may be a clear indication that the specification is not ground coherent, so that a new specification should be developed (see Section 7.8 for an example of this kind). Sections 12.4 and 13.4 illustrate the use of the ChC tool in proving the coherence or ground coherence of some Maude specifications.

The tool and its documentation are available at <http://maude.lcc.uma.es/CRChC/>, including some methodological guidelines for the use of the tool and some examples.

23.1.5 The Sufficient Completeness Checker

Assuming an equational specification in which a subsignature of constructors has been specified and such that the equations are terminating, the *sufficient completeness problem* consists in verifying that the canonical forms of all well-typed ground terms are constructor terms. Intuitively, this means that all defined operations have been fully defined, without leaving out any special cases. In Maude, constructors are specified with the `ctor` attribute, and sufficient completeness must be understood with respect to the given operator declarations and memberships [191]. This is because a ground term having a kind but not a sort may very well contain defined function symbols. It is only ground terms having a sort that should be provably equal to constructor terms.

The tree automata-based *sufficient completeness checker* (SCC) [190] is a tool for checking sufficient completeness of order-sorted specifications with rewriting modulo axioms, that are left-linear, weakly-normalizing, ground confluent, and ground sort-decreasing modulo the axioms. The tool is still

under active development, and may be downloaded from its website at <http://maude.cs.uiuc.edu/tools/scc>.

A detailed example of how to use the SCC can be found in Section 4.4.3, and additional examples in Sections 12.4 and 13.4.

The SCC checker is a decision procedure for left-linear, unconditional order-sorted specifications with any combination of associativity, commutativity, and identity axioms, except associativity or associativity and identity alone. In this last case, the tool has a specialized algorithm that will always eventually find a counterexample if the specification is not sufficiently complete, and will often show sufficient completeness in practice. However, the sufficient completeness problem itself is undecidable in this context, and so no checker will always show sufficient completeness for specifications with associative symbols.

The checker casts this sufficient completeness problem with rewriting modulo axioms as a decision problem for *equational tree automata* [281]. The paper [191] shows in detail how to convert the sufficient completeness property into a propositional tree automata emptiness problem. The key idea is that given an order-sorted specification $\mathcal{E} = (\Sigma, A \cup R)$ with sorts S , we can construct the following automata for each sort $s \in S$:

- an automaton \mathcal{A}_s^c accepting constructor terms of sort s ;
- an automaton \mathcal{A}_s^d accepting terms whose root is a defined symbol of sort s and whose subterms are constructor terms; and
- an automaton \mathcal{A}^r accepting any term reducible by equations in R .

If \mathcal{E} is left-linear, weakly-normalizing, ground confluent, and ground sort-decreasing modulo A , then \mathcal{E} is sufficiently complete iff $\mathcal{L}(\mathcal{A}_s^d) \subseteq \mathcal{L}(\mathcal{A}^r) \cup \mathcal{L}(\mathcal{A}_s^c)$ for each sort $s \in S$.

Since equational tree automata with associative symbols are not closed under Boolean operations [281], we found it useful to introduce a new tree automata framework in [191, 192], called *propositional tree automata* (PTA), that is closed with respect both to Boolean operations and an equational theory. Using our propositional tree automata framework, we in turn translate the previous equational tree automata problem into the problem of checking the emptiness of $\bigcup_{s \in S} \mathcal{L}(\mathcal{A}_s^d) - (\mathcal{L}(\mathcal{A}^r) \cup \mathcal{L}(\mathcal{A}_s^c))$.

The SCC has two major components: an analyzer written in Maude that generates the tree automaton emptiness problem from a Maude specification; and a C++ library called CETA that performs the emptiness check.

Analyzer

The analyzer accepts commands from the user, generates PTA from Maude specifications, forwards the PTA decision problems to CETA, and presents the user with the results. If the specification is not sufficiently complete, the tool shows the user a counterexample illustrating the error. The analyzer consists of approximately 900 lines of Maude code, and exploits Maude's support for reflection. The specifications it checks are also written in Maude.

If the user asks the tool to check the sufficient completeness of an order-sorted specification that is not linear and unconditional, the tool transforms the specification by renaming variables and dropping conditions into a checkable order-sorted linear specification. Even if the tool is able to verify the sufficient completeness of the transformed specification, it warns the user that it cannot show the sufficient completeness of the original specification. However, any counterexamples found in the transformed specification are also counterexamples in the original specification. We have found this feature quite useful to identify errors in Maude specifications falling outside the decidable class, including the sufficient completeness checker itself.

CETA

The propositional tree automaton generated by the analyzer is forwarded to the tree automata library CETA which we have developed. CETA is a complex C++ library with approximately ten thousand lines of code. Emptiness checking is performed by a subset construction algorithm extended with support for associativity and commutativity axioms as described in [192]. The performance of CETA is quite good: most examples can be verified in seconds. The slowest specification that has been checked is the sufficient completeness analyzer itself; the library requires just under a minute on a Pentium 4 desktop to check the 900 lines of Maude code in the analyzer.

23.1.6 The Real-Time Maude tool

Real-Time Maude [283, 287, 289, 291] is a specification language and a formal tool that extends Maude to specify and analyze real-time systems, defined as real-time rewrite theories (see Section 22.5). It provides special syntax to specify object-oriented real-time systems, for which useful specification techniques have been developed [291].

The Real-Time Maude tool systematically exploits the underlying Maude efficient rewriting, search, and LTL model-checking capabilities to both execute and formally analyze real-time specifications. Reflection is crucially exploited in the Real-Time Maude 2.1 implementation. On the one hand Real-Time Maude specifications are internally desugared into ordinary Maude specifications by transforming their metarepresentations. On the other, reflection is also used for execution and analysis purposes. The point is that the desired modes of execution and formal properties to be analyzed have real-time aspects with no clear counterpart at the Maude level. To faithfully support these real-time aspects a *reflective transformational approach* is adopted: the original real-time theory and query (for either execution or analysis) are *simultaneously* transformed into a semantically equivalent pair of a Maude rewrite theory and a Maude query [291]. In practice, this makes those executions and analyses quite efficient and allows scaling up to highly nontrivial specifications and case studies.

Typical application domains for Real-Time Maude include high-level modeling and analysis of all kinds of advanced time-dependent systems, such as, e.g., timed communication protocols, scheduling algorithms, time-sensitive security protocols, and so on. In particular, Real-Time Maude, which does not rely on fixed communication primitives, may be particularly useful to model systems with advanced and novel models of communication, such as, e.g., wireless sensor networks.

Existing Real-Time Maude applications include the formal specification and analysis of the following:

- the AER/NCA suite of protocols [203], intending to achieve reliable, scalable, and TCP-friendly multicast in active networks [292];
- the NORM multicast protocol developed by the Internet Engineering Task Force [221];
- the OGDC wireless sensor network algorithm [359, 293];
- the CASH adaptive scheduling algorithm [284]; and
- the wide-mouthed frog and Kerberos security protocols [177].

The analysis of the AER/NCA suite of protocols uncovered subtle design errors and independently found all bugs discovered by traditional testing [283, 285, 292].

The Real-Time Maude tool is a mature and quite efficient tool available (with source code, a tool manual, examples, case studies, and papers) from <http://www.ifi.uio.no/RealTimeMaude>.

23.1.7 Predicate abstraction in Maude

Model checking is a verification technique that can be used to prove properties of finite-state systems in an automatic way, essentially by exhaustively enumerating all states. Unfortunately, many interesting systems are infinite and therefore outside the scope of standard model-checking algorithms. In those cases, or in cases where the state space is finite but too large to be model checked in practice, a possible way of dealing with the difficulty consists in computing a finite-state system that *simulates* the concrete system at hand, in the sense that if a certain property holds in the simpler one then it must be true also of the original system.

Predicate abstraction is a technique to automatically compute finite-state systems that simulate more complex and possibly infinite ones. More precisely, assume a system with states a, b, c, \dots , belonging to a set S of states, and with transitions given by a relation $\rightarrow \subseteq S \times S$. Then, a *predicate abstraction* is defined by a set of predicates ϕ_1, \dots, ϕ_n over the set S of states in the following manner:

- The set of states of the abstract system is the set of n -tuples of Boolean values, where n is the number of predicates.
- A concrete state a is mapped to the tuple $\alpha(a) = \langle \phi_1(a), \dots, \phi_n(a) \rangle$.

The transition relation in the abstract system is then determined in a standard way:

- There is a transition from the abstract state $\langle b_1, \dots, b_n \rangle$ to $\langle b'_1, \dots, b'_n \rangle$ iff there are concrete states a and b such that $\alpha(a) = \langle b_1, \dots, b_n \rangle$, $\alpha(b) = \langle b'_1, \dots, b'_n \rangle$, and $a \rightarrow b$.

Computing the abstract state associated with a concrete state a , though dependent on the complexity of the predicates, is usually straightforward. It is in the computation of the abstract transition relation that the difficulty really lies, since a single transition step potentially depends on an infinite number of concrete states.

The Maude's predicate abstraction prototype automates the construction of this abstract system. Each rule defining the transition relation of the concrete system is directly transformed into another rule that is then used to define the transition relation in the abstract system. This relation does not usually coincide with the exact abstract transition relation as defined above, but it is a useful over-approximation. More precisely, given a module named M specifying the concrete system as well as the predicates $\text{phi1}, \dots, \text{phiN}$, the tool implements an operation `abstractionGen` such that the term `abstractionGen('M, 'phi1 ... 'phiN)` reduces to the *metarepresentation* of the corresponding abstract system. This metarepresentation can then be analyzed with the `metaSearch` command for checking safety properties, or can be manually transformed into a module at the object level to study it with Maude's model checker.

The tool runs on top of Maude 2.1.1 and makes use of a slightly modified version of the ITP theorem prover described in Section 23.1.1. It can be obtained, together with its manual and some examples and the modified ITP, from <http://maude.sip.ucm.es/~miguelpt/bibliography.html>.

23.2 Other tools

23.2.1 The Open Calculus of Constructions

Logical type theories in the line of Martin-Löf's type theory [298] or the calculus of constructions [85] can be motivated by the observation that, when we give proofs a formal status in the deductive system of a logic, we naturally obtain a type theory where the proofs correspond to elements and theorems correspond to their types. Intuitively, a type conveys certain abstract information about the proof, namely, its corresponding theorem, and thus it hides the structure of the proof, which is usually desirable if the theorem is applied as a lemma as part of a larger development. The logical type theories mentioned above are typed λ -calculi with dependent types of the form $\{X : S\} T$, that is, types that can depend on universally quantified variables. This makes these type theories at least as expressive as higher-order logic.

Several applications studied in [336] suggest that rewriting logic, including its membership equational sublogic, and logical type theories have features that are nearly complementary, so that much could be gained from a unification of these two lines of research. As an example, rewriting logic can serve as a framework logic [231, 229, 247], and higher-order logic can be used to formulate induction principles and to reason about the formalism that has been embedded into the framework. Furthermore, rewriting logic can benefit from an expressive type system with dependent types and universes, which in particular provides type operators and explicit polymorphism. Conversely, type theory could benefit from equational logic and rewriting logic. For instance, the lack of a notion of executable specification, providing a notion of abstract execution, is a clear weakness of logical type theories, which in turn leads to severe difficulties with principles of modularity and information hiding. Furthermore, the notion of computation in type theories is rather rigid, compared with the powerful notion of computation based on conditional rewriting modulo the axioms of an equational theory as in rewriting logic.

In [336] a unified formalism called the *open calculus of constructions* (OCC) was presented. The calculus is parameterized by a flexible universe hierarchy, which admits impredicative universes in the style of the calculus of constructions and predicative universes in the style of Martin-Löf's type theory. Similar to the calculus of inductive constructions, it is inspired by the extended calculus of constructions [225], but has more general computational capabilities. Specifically, OCC incorporates membership equational logic and the corresponding version of rewriting logic as computational sublanguages, and as a consequence supports conditional assertions, equations, and rules together with an operational semantics based on conditional rewriting and goal-oriented proof search modulo equational theories. A unique feature of OCC is that the operational semantics is context-dependent, in the sense that checking a judgement like $\Gamma \vdash M : T$ can involve reduction and goal-oriented search that depends on operational propositions in Γ . All subproofs which are subsumed by the operational semantics are performed automatically and hence are of a purely computational nature, so that no structured proof object is needed.

Executable specifications

A common use of dependent types, where the quantification ranges over a type universe is the declaration of polymorphic functions. This is illustrated by the following sample specification of polymorphic finite multisets. In addition to the type constructor, and three multiset constructors, we have three equational axioms designated by the syntax `||` as *structural equations*, meaning that everything that follows operates modulo those equations.

```
( fms : Type -> Type )

( fms_nil : {T | Type} (fms T) )
```

```

(fms_single : {T | Type} T -> (fms T) )
(fms_union : {T | Type} (fms T) -> (fms T) -> (fms T) )

(fms_union_comm : || {T : Type}{m1,m2 : (fms T)}
  (fms_union m1 m2) = (fms_union m2 m1) )
(fms_union_assoc : || {T : Type}{m1,m2,m3 : (fms T)}
  (fms_union m1 (fms_union m2 m3)) = (fms_union (fms_union m1 m2) m3))
(fms_union_right_id : || {T : Type}{m : (fms T)}
  (fms_union m fms_nil) = m )

```

Like in LEGO [304], the *implicitly dependent type* $\{x \mid A\} B$ is syntactic sugar for the (explicitly) dependent type $\{x : A\} B$, and denotes the type of functions with a result of type B and an implicit argument of type A .

Assuming a polymorphic predicate `fms_empty` to verify emptiness, we can write the following specification of a polymorphic higher-order function `fms_select`, which selects all elements from a multiset satisfying a given predicate. The specification has four equational axioms, designated by the syntax `!!` as *computational equations*, meaning that their operational semantics corresponds to equational simplification.

```

(fms_select : {T | Type} (T -> Prop) -> (fms T) -> (fms T) )

(fms_select_nil : !! {T : Type}{P : (T -> Prop)}
  (fms_select P (fms_nil | T)) = (fms_nil | T) )
(fms_select_single_1 : !! {T : Type}{P : (T -> Prop)}{x : T}
  (P x) -> (fms_select P (fms_single x)) = (fms_single x) )
(fms_select_single_2 : !! {T : Type}{P : (T -> Prop)}{x : T}
  (Not (P x)) -> (fms_select P (fms_single x)) = fms_nil )
(fms_select_union : !! {T : Type}{m,m' : (fms T)}{P : (T -> Prop)}
  (Not (fms_empty m)) -> (Not (fms_empty m')) ->
  (fms_select P (fms_union m m')) =
    (fms_union (fms_select P m) (fms_select P m')) )

```

Assuming a specification of, say, natural numbers, we can now perform sample executions like the following. In the first line, we define a predicate P using a typed λ -abstraction, then we invoke a reduction. What is interesting in this simple example is that a combination of both reduction (just β -reduction in this case) and goal-oriented search (in this case to verify the equality predicate) is needed to arrive at the given result.

```

(P := [x : nat](x = 0) )

(red (fms_select P (fms_union (fms_single 0)
  (fms_union (fms_single 1)
    (fms_single 0)))))) )

(fms_union (fms_single 0) (fms_single 0) )

```

Interactive theorem proving

Proof principles such as induction can be naturally expressed in higher-order logic, as for instance in the case of natural number induction:

```
( nat_ind : {P : nat -> Prop}
  (P 0) ->
  ({i : nat} (P i) -> (P (suc i))) ->
  {n : nat} (P n) )
```

To give just a flavor of interactive theorem proving we use it to prove monotonicity of the predicate `nat_le` by stepwise refinement in two steps. First we apply the induction principles, and then we prove the remaining subgoal (the base case is automatic).

```
( nat_suc_is_monotone = ? : {n : nat} (nat_le n (suc n)) )

-----
?7585 : { n : nat } ( nat_le n ( suc n ) )
1 new goal

( Inst (nat_ind ([n : ?] ?) ? ?) )

-----
?7616 : { i : nat }
  ( ( [ n : nat ] ( nat_le n ( suc n ) ) ) i ) ->
  ( ( [ n : nat ] ( nat_le n ( suc n ) ) ) ( suc i ) )
1 new goal

( Inst ([i : nat] ?) )

{ i : nat }
-----
?7693 : ( ( [ n : nat ] ( nat_le n ( suc n ) ) ) i ) ->
  ( ( [ n : nat ] ( nat_le n ( suc n ) ) ) ( suc i ) )
1 new goal

( Inst ([H : ?? ?] ?) )
all goals solved
```

The last step of this inductive proof is noteworthy, since by forcing the premise `H` into the form `?? ?` we express that it should be added to the context as an assertional hypothesis, to support an automatic proof (using goal-oriented search) of the righthand side of the implication.

Prototype implementations

The implementation of a type theory like OCC presents a challenge, mainly because of the context-dependent operational semantics and the tricky interdependency between the computational mechanism and the typing rules. Two

prototype implementations for OCC language fragments of different generality are available. Maude and Prolog, respectively, are used as implementation frameworks. Both prototypes are based on a reflective architecture to address the challenges mentioned above.

For the term representation and associated operations, both prototypes use CINNI, the explicit substitution calculus with names and indices that has been developed in [336, 334]. Thanks to its use of names, the translation between internal and external syntax becomes trivial. The typing rules are directly represented as rewrite rules (in Maude) or as Horn clauses (in Prolog). This is done in a way that generalizes the idea of a first-order representation of pure type systems developed in [341] and hence avoids potential closure problems associated with α -conversion [305]. Metavariables have been added to the syntax to delay decisions and to support type inference based on various heuristics. The interactive proof mode then becomes a special case of type checking, where the metavariables represent unknown proof terms (corresponding to subgoals) that are too complicated to be synthesized by the system. As we have seen, proof by stepwise refinement amounts to nothing other than instantiating unsolved metavariables.

But where does reflection come into play? It is used to represent the operational semantics of OCC in terms of the operational semantics of the underlying framework (Maude or Prolog). The basic idea is the following: Given a judgement $\Gamma \vdash J$ that needs to be solved, the system applies a function *extr* that extracts the operational contents from Γ and represents it either as a first-order rewrite theory over membership-equational logic (in the case of Maude) or simply as a Horn clause theory (in case of Prolog). After this translation the judgement J is solved in the new theory *extr*(Γ) using computational reflection. The function *extr* abstracts from all non-computational aspects, e.g., from purely logical theorems that are not equipped with a computational interpretation.

With all this one needs to keep in mind that a typical type or proof-checking problem can involve a huge number of judgements with many different associated contexts, and hence may trigger computations in many different contexts (possibly with different structural axioms). In Maude, the module resulting from the extraction is internally compiled into automata, so that rewriting can be performed efficiently, but clearly the overhead of this compilation process is not negligible. Although some optimizations, like caching for example, have been applied, it is clear that much more can be done here in order to obtain a more efficient implementation.

In summary, OCC offers a general higher-order framework for specification, programming, and interactive theorem proving, which, due to the flexibility of its underlying equational/rewriting logic, is widely applicable in various domains. Among the topics covered by the examples in [337, 338, 336] we find executable equational/behavioral specifications, programming with dependent types, symbolic execution of system models, formalization of algebraic and categorical concepts, inductive/coinductive theorem proving, and theorem prov-

ing modulo equational theories. Also the implementation of the cryptoprotocol specification language MSR (discussed in Section 23.2.4) presents an interesting application of the Maude prototype of the open calculus of constructions as part of a larger system.

The OCC prototypes together with libraries and many examples are available at <http://formal.cs.uiuc.edu/stehr/occ.html>. A good starting point to explore the features of OCC are the examples from the tutorial part of [336], which are all available at the given website.

23.2.2 The Maude MSOS tool

The Maude MSOS Tool (MMT) [53, 55] is an execution environment for *modular structural operational semantics* (MSOS) specifications that brings the power of analysis available in the Maude system to MSOS specifications.

MSOS [270], developed by Mosses, is a modular variant of Plotkin’s SOS [303]. Like SOS, MSOS is a framework suitable for the specification of a wide range of computational formalisms, including programming language semantics and concurrent systems. Unlike SOS, MSOS has the advantage of supporting completely modular specifications, an important advantage when considering the engineering decisions that one must face when some features to be implemented are not known in advance. As an example, consider an SOS specification of a functional language fragment with environments; if imperative features must be added later to the language, therefore requesting a store component to be considered, all SOS rules written for the functional language fragment would have to be retracted and replaced by new rules that also involve the store component. MSOS makes this unnecessary by structuring the labels in SOS rules as records of semantic components, such as the environment and the store, and by abstracting from the rules the fact that new components may be added in future extensions.

MMT is formally designed, which means that it is an implementation of a semantics-preserving mapping from MSOS to rewriting logic [251]. It supports a specification language, designed by Mosses and Chalub, called MSDF, the *modular SOS specification formalism*, which is a language that combines an extended-BNF notation for the definition of abstract grammar and a textual representation for MSOS transitions that captures mathematical notation commonly used in papers and textbooks, making the language itself based on well-known “user-friendly” notations available in the literature.

MMT is implemented as an extension of Full Maude, therefore the bulk of the tool consists of a read-eval-print loop where a user inputs MSDF modules and MMT commands. The tool then compiles MSDF modules into Full Maude system modules, which enables their use with the suite of Full Maude’s commands. MMT’s commands include a pretty-printing and a compilation flag that enables and disables MMT’s default rewrite strategy, a necessary feature when combining MMT with other Maude-based tools such as Martí-Oliet, Meseguer, and Verdejo’s interpreter for a Maude strategy language [234].

Several case studies, both in the programming languages and concurrent systems realms, have been developed using MMT, with very positive results, including the following:

- Constructive MSOS (CMSOS), a library of reusable common abstract language features,
- the semantics of two programming languages based on CMSOS: subsets of Concurrent ML and Java,
- the semantics of Mini-Freja, a lazy functional language, and
- many distributed algorithms, such as dining philosophers, bakery algorithm, leader election on an asynchronous ring, and mutual exclusion using semaphores.

The paper [55] gives an overview of MMT. The tool itself is available from <http://maude-msos-tool.sourceforge.net/> and runs on top of Maude 2.2 and Full Maude 2.2. The tool's website provides all the necessary information on how to download and execute the tool, together with a simple example and many fully documented case studies.

23.2.3 The CCS and LOTOS tools

The CCS and LOTOS tools are part of a broader effort in which several tools have been developed with the purpose of exploring the features of rewriting logic, and in particular of Maude, as a logical and semantic framework for representing and executing inference systems. Two different methodologies for representing these inference systems have been investigated, and structural operational semantics definitions illustrating these two methodologies on two process algebras have been developed.

In the first methodology, the basic idea of the representation used is that judgments in the inference system are represented as terms in Maude, and inference rules are mapped to rewrite rules. In order to illustrate the general ideas, we have represented both the semantics of Milner's CCS [267] and a modal logic for describing properties of CCS processes. Although a rewriting logic representation of the CCS semantics was given previously [231], it cannot be directly executed in Maude. Moreover, it cannot be used to answer questions such as which are the successors of a process after performing an action, which is used to define the semantics of the Hennessy-Milner modal logic [193]. Basically, the problems are the existence of new variables in the righthand side of the rewrite rules, and the nondeterministic application of the semantic rules, inherent to CCS. These problems can be solved in a general, not CCS-dependent, way by exploiting the reflective properties of rewriting logic. Furthermore, the semantics can be extended to traces of actions and to the CCS weak transition relation. This executable specification plus the reflective control of the rewriting process can be used to analyze CCS processes. The papers [368, 370] explain all the details of these representations and how the combined tool can be used by showing examples. The

Maude code (for Maude version 1.05) and examples can be downloaded from <http://maude.cs.uiuc.edu/maude1/casestudies/ccs/>.

A second possible way of representing in Maude inference systems in general, and structural operational semantics definitions in particular, is one where transitions are represented as rewrites and the inference rules are mapped to conditional rules with rewrites in the conditions. This is possible because Maude allows conditional rules with rewrites in the conditions, where those rewrites are solved at execution time by means of a built-in search mechanism. In this way, it becomes possible, for example, to represent in Maude in a fully *executable* way the CCS operational semantics considering transitions as rewrites. In fact, both the usual transition semantics and the weak transition semantics, where internal actions are not observed, can be implemented this way. On top of them it is also possible to implement in Maude the Hennessy-Milner modal logic for describing processes. The paper [369] describes in detail the results obtained in this second implementation and compares both implementations. The Maude 2 code for this second approach can be also downloaded from <http://maude.cs.uiuc.edu/maude1/casestudies/ccs>.

The same techniques have also been used to implement a symbolic semantics for LOTOS [44]. The Full LOTOS case study extends those techniques to a bigger language and, moreover, does this in such a way that the ACT ONE algebraic specifications [133] used in LOTOS to define data types are really integrated into the operational semantics (this is the reason for talking about Full LOTOS instead of just LOTOS), something that really breaks new ground in this approach. In addition, by means of the metalanguage features supported by Maude, the Full LOTOS semantics tool is also integrated with Full Maude: this way, in the same semantic framework one can build an entire environment with parsing, pretty printing, and input/output processing of LOTOS specifications and commands for executing them, hiding from the user the underlying use of Maude. The paper [364] explains how this complete tool has been developed, shows examples of its usage, and describes the available commands. The Maude 2 code can be downloaded from <http://maude.sip.ucm.es/~alberto/esf>.

23.2.4 The MSR cryptoprotocol specification language

MSR originated as a simple logic-based language aimed at investigating the decidability of protocol analysis under a variety of assumptions [50]. It evolved into a precise, powerful, flexible, and still relatively simple framework for the specification of complex cryptographic protocols, possibly structured as a collection of coordinated subprotocols [43, 48]. It uses strongly-typed multiset rewriting rules over first-order atomic formulas to express protocol actions and relies on a form of existential quantification to symbolically model the generation of nonces and other fresh data. Dependent types are a useful abstraction mechanism not available in other languages. For instance, the dependency of public/private keys on their owner can be naturally expressed

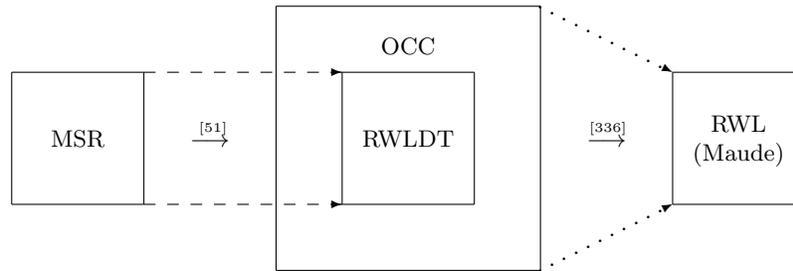


Fig. 23.1. Embedding of MSR into rewriting logic

at the type level. Finally, MSR supports an array of useful static checks that include type checking [47] and data access verification [49].

Architecture of the MSR tool

An efficient embedding from MSR into rewriting logic with dependent types (RWLDT) was presented in [51, 52]. RWLDT is a restricted instance of the open calculus of constructions (OCC). This mapping forms the basis for the implementation of an MSR execution and analysis environment [313]. The MSR tool also makes use of the mapping from RWLDT into rewriting logic (RWL), implemented as part of the OCC prototype in Maude [336] (see also Section 23.2.1).

This two-level approach, which is summarized in Figure 23.1, has some advantages over a direct mapping into RWL. The first is modularity and separation of concerns: the mapping from MSR into RWLDT is only concerned with the dynamics (given by the rules) but preserves the static part (given by declarations, types, and terms). The second advantage is that RWLDT seems to be the right level for user interaction, because terms and types closely correspond to those of MSR. Finally, the preservation of types and the fact that RWLDT is a sublogic of OCC provides a suitable level of abstraction for formal reasoning.

The MSR tool supports symbolic execution and symbolic static space exploration of the translated MSR specification. The user interaction takes place at the level of RWLDT terms, which directly correspond to MSR terms, and hence the user does not need to be concerned with the resulting translation into RWL. The MSR tool implements not only symbolic execution at the MSR language level, but also symbolic search, both with several options similar to those available in Maude. To facilitate the interactive analysis of security protocols, the tool also supports interactive state space navigation, where symbolic executions and searches can be composed in an interactive session to explore the protocol state space. A similar MSR interface to Maude’s model checker is left for future work. At the moment, we can instead export the

RWL translation of the RWLDT specification and perform model checking at the Maude level.

Example

To just give a flavour of the syntax and the interaction with the MSR tool we use the well-known Needham-Schroeder public-key two-party protocol, which can be specified in MSR syntax as shown below. The type `princ` contains the principals participating in the protocol, and the dependent type `pubK A` represents the public keys of principal A, which can be used to encrypt messages using `enc`. Here, `msg` is the built-in type of messages, which comes with a concatenation operator `&`. The overall specification consists of two roles, modeling the initiator and the responder, respectively, each of them containing two labeled multiset-rewrite rules. To control the instantiation of roles and to observe termination of the protocol we have added tokens `START-i` and `TERMINATED-i`.

```

pubK: princ -> type. %enum
enc: pubK A -> msg -> msg.

START-1: princ -> state.
START-2: princ -> state.
TERMINATED-1, TERMINATED-2: princ -> princ -> nonce -> state.

initiator : forall A: princ.
{ exists L: princ -> nonce -> state.
  Init1: forall B: princ. forall PKB: pubK B.
    START-1 A
    => exists nA: nonce. A & B & enc PKB (nA & A), L B nA .
  Init2: forall B: princ. forall PKA: pubK A. forall PKB: pubK B.
    forall nA: nonce. forall nB: nonce.
    B & A & enc PKA (nA & nB), L B nA
    => A & B & enc PKB nB, TERMINATED-1 A B nB.
}

responder : forall B: princ.
{ exists L: princ -> nonce -> nonce -> state.
  Resp1: forall A: princ. forall PKA: pubK A. forall PKB: pubK B.
    forall nA: nonce.
    START-2 B, A & B & enc PKB (nA & A)
    => exists nB: nonce. B & A & enc PKA (nA & nB), L A nA nB.
  Resp2: forall A: princ. forall PKB: pubK B. forall nA: nonce.
    forall nB: nonce.
    A & B & enc PKB nB, L A nA nB
    => TERMINATED-2 B A nA.
}

```

Given this specification of the protocol we can prepare a sample execution by introducing two concrete principals with their public keys, and define the initial configuration using the keyword `config`.

```
A, B: princ. pkA: pubK A. pkB: pubK B.
config START-1 A, START-2 B.
```

Now we have different options to explore the behavior of the protocol. For instance, we could use `search` to ask for all possible reachable states:

```
(search)
1: N(princ-msg A & princ-msg B & enc pkB(nonce-msg F1 & princ-msg A)),
   START-2 B, T_initiator_Init2 A F0, F0 B F1
2: N(princ-msg B & princ-msg A & enc pkA(nonce-msg F1 & nonce-msg F3)),
   T_initiator_Init2 A F0, T_responder_Resp2 B F2, F0 B F1,F2 A F3
3: N(princ-msg A & princ-msg B & enc pkB(nonce-msg F3)),
   TERMINATED-1 A, T_responder_Resp2 B F2, F2 A F3
4: TERMINATED-1 A,TERMINATED-2 B
```

Alternatively, we could simply perform a complete state space exploration asking for all reachable states that cannot be further reduced:

```
(search !)
1 : TERMINATED-1 A,TERMINATED-2 B
```

Of course, the analysis becomes more interesting when an explicit attacker model is added to the specification, which is unfortunately beyond the scope of this brief overview.

The MSR tool, examples, and related documentation are available at <http://formal.cs.uiuc.edu/stehr/msr.html>.

23.2.5 JavaFAN

JavaFAN (Java Formal ANalyzer) [153, 156] is a tool to simulate and formally analyze multithreaded Java programs at source code and/or bytecode levels. A novel feature of JavaFAN's design is that it is directly based on *formal definitions* of the Java and the JVM semantics in the form of *rewrite theories* in Maude, following the general methodology outlined in Section 22.2. The following types of analysis are directly supported by Maude:

- *symbolic simulation*, with Java and JVM specifications used as *interpreters* executing programs with actual or symbolic inputs;
- *breadth-first search* within a concurrent program's state space to find violations of safety properties; and
- *model checking* of linear temporal logic (LTL) properties for programs whose state space is finite.

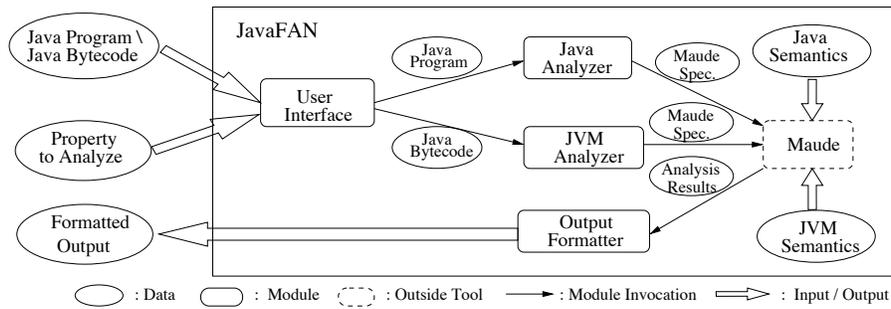


Fig. 23.2. Architecture of JavaFAN

To keep the framework user-friendly, JavaFAN wraps the Maude specifications and accepts Java or JVM code from the user as input.

Figure 23.2 presents the architecture of JavaFAN. The *user interface* module hides the Maude back-end behind a user-friendly environment. It also plays the role of a dispatcher, sending the Java source code and/or the bytecode to Java and/or JVM analyzers, respectively. The analyzers wrap the input programs into properly defined Maude modules and invoke Maude, which analyzes the code based on formal specifications of the Java language and the JVM. The output formatter collects the output of Maude, transforms it into a user-readable format, and sends it to the user.

Maude has been used to specify the operational semantics of all of Java and the JVM (except for the libraries). Java and the JVM are modeled differently. For Java, a continuation-based style is adopted, while for the JVM an object-oriented style, that makes the specification simpler and easier to understand, is used.

Although the development effort of JavaFAN has been quite modest, it compares well with other Java analysis tools in its application to the analysis of Java and JVM programs [153]. JavaFAN's efficiency relies both on the efficiency of Maude and on the choice of representation. For example, the use of equations instead of rules to express the semantics of deterministic features, the use of continuation-based interpretation, and distinguishing between the *static* and *dynamic* parts of a program, so that only the dynamic component is kept in the state representation, all result in significant resource savings for large programs.

JavaFAN can be downloaded from <http://javafan.cs.uiuc.edu/>, where the reader can find more information.

23.2.6 Java+ITP

Java+ITP [324] is an experimental tool for the verification of properties of a sequential imperative subset of the Java language. It is based on an algebraic

continuation passing style (CPS) semantics of this fragment as an equational theory in Maude. It supports compositional reasoning in a Hoare logic for this Java fragment that has been proposed and proven correct with respect to the algebraic semantics in [323]. After being decomposed, Hoare triples are translated into semantically equivalent first-order verification conditions (VCs) which are then sent to Maude's Inductive Theorem Prover (ITP) (see Section 23.1.1) to be discharged. The long-term goal of this project is to use extensible and modular rewriting logic semantics of programming languages, for which CPS axiomatizations are indeed very useful, to develop similarly extensible and modular Hoare logics on which *generic* program verification tools can be based.

The Maude specification of Java+ITP's underlying Java language fragment adapts the Maude-based rewriting logic semantics of Java developed for JavaFAN (see Section 23.2.5) by adding to it extra features making it suitable for theorem-proving purposes. Although Java+ITP focuses for the moment on a modest sequential Java fragment, there is ample evidence, both in Java and in other languages (see [259]), supporting the claim that CPS-based rewriting logic definitions are modular and extensible; therefore, the present work should naturally extend to more ambitious language fragments in Java and in other languages.

A Hoare logic for this Java fragment has been proposed, and its correctness has been mathematically justified based on the CPS semantics in [323]. Even for this modest fragment this turns out to be nontrivial, because some of the standard Hoare rules, including the rules for conditionals and for while loops, are in fact *invalid* and have to be properly generalized in order to be applicable to Java programs.

The Java+ITP tool is a mechanization of this Hoare logic supporting:

1. compositional reasoning with the Hoare rules to decompose Hoare triples into simpler ones;
2. generation of first-order *verification conditions* (VCs); and
3. discharging of such VCs by Maude's inductive theorem prover (ITP) (see Section 23.1.1), using the underlying CPS semantics.

Java+ITP has been developed as an extension of Maude's ITP and is entirely written in Maude.

Although Java+ITP is primarily a research vehicle to advance the longer-term goal of developing generic logics of programs and generic program verifiers based on modular rewriting logic semantic definitions, it has proved to be also quite useful as a *teaching tool* at the University of Illinois at Urbana-Champaign to teach graduate students and seniors the essential ideas of algebraic semantics and Hoare logic. It has been used quite extensively by students on several graduate courses on program verification and formal methods.

To summarize, Java+ITP is a research tool to investigate modularity and extensibility of programming languages and of Hoare logics. It has proved to be useful for this purpose by uncovering subtleties in the Hoare logic needed

for Java not present in toy languages, and not even present in the Hoare logics of Java tools like Jive. Keeping the compositional Hoare logic reasoning at the source code level is also one of the goals that, in contrast to other approaches, has been advanced. But of course this is just a snapshot of work in progress. The current Java fragment is still quite modest, so new features should be added to it such as exceptions and objects; this should be easy thanks to the extensible CPS semantics. After this, threads and concurrency should also be added, and Hoare rules for these new features should also be investigated. The long-term goal is of course modularity, so that Hoare rules will be applicable not just to Java, but to any other languages using some of the same features in a modular way, but this still remains an exciting goal for the future.

An overview of this tool can be found in [324]. For a more detailed description, including proofs, a technical report is also available [323]. The actual tool can be downloaded from <http://maude.cs.uiuc.edu/tools/javaitp> where more documentation about technicalities is also available.

23.2.7 The ITP/OCL tool

The Unified Modeling Language (UML) [279, 321, 23] is a general-purpose visual modeling language that is used to specify, visualise, construct, and document the artifacts of a software system. The UML notation is largely based on diagrams; however, for certain aspects of a design, diagrams do not provide the level of conciseness and expressiveness that a textual language can offer. The Object Constraint Language (OCL) [278, 380] is a textual constraint language with a notational style similar to common object-oriented languages. OCL's purpose is to increase the expressive power of the modelers specifying and documenting UML diagrams.

Validation and testing in software development has been recognized of key importance for a long time. There are many different approaches to validation: simulation, rapid prototyping, etc. We *validate* a model by checking whether its instances (also called “snapshots”) fulfill the desired invariants. This can lead to several consequences with respect to the design. First, if there are reasonable snapshots that do not fulfill the invariants, this may indicate that the invariants are too strong or the model is not adequate in general. On the other hand, invariants may be too weak, allowing undesirable system states.

The ITP/OCL tool [75] is a rewriting-based tool that supports automatic validation of UML static class diagrams with respect to OCL invariants. It is intended as a *lightweight* formal method: it should help software modelers to find flaws in UML class diagrams in the early phases of the software development process. It is intended also as a *practical* formal method: it should be directly usable by UML+OCL modelers. The ITP/OCL tool is directly based on the equational specification of UML+OCL class diagrams developed in [132]: basically, class and object UML diagrams are specified as membership equational theories, and OCL invariants are represented as Boolean terms over extensions of those theories. Then, validating object diagrams with respect to

invariants is reduced to checking whether the corresponding Boolean terms rewrite to true or false. Although the equational semantics developed in [132] only covers UML+OCL static class diagrams, it provides a solid ground upon which to develop equational extensions to express the semantics of other UML diagrams.

The ITP/OCL tool is written entirely in Maude, making extensive use of its reflective capabilities to implement the user interface. In this way, the tool underlying equational semantics remains hidden to the user, who only needs to be familiar with the standard notions of UML diagrams and OCL constraints. The ITP/OCL commands can be grouped in four classes:

- *Commands that create a diagram.* They are defined by equations that add to the tool database the module that specifies an empty class or object diagram. For example, to create a class diagram we use the command `(create-class-diagram CD .)`, where *CD* is the class diagram name.
- *Commands that insert an element (class, attribute, association, and so on) in a diagram.* They are defined by equations that add to the module specifying the diagram in the tool database the declarations (sorts, operators, memberships, equations) that specify that the diagram has this element. For example, to insert a class we use the command `(insert-class CD : C .)`, where *CD* is the class diagram name and *C* is the class name.
- *Commands that state a constraint over a class diagram.* They are defined by equations that associate with the module specifying the class diagram in the tool database the Boolean term that represents this constraint. For instance, to state a *contextualized invariant* we use the command `(insert-invariant CD : C : INV .)`, where *CD* is the class diagram name, *C* is the contextual class name, and *INV* is the invariant.
- *Commands that validate an object diagram (and evaluate queries).* They are defined by equations that check whether the Boolean terms representing the invariants reduce to true or false in the module that specifies the union of the class diagram, with its invariants, and the object diagram. For example, to check whether an object diagram validates the invariants stated over the class diagram of which it is an instance, we use the command `(check-invariants CD : OD .)`, where *CD* is the class diagram name and *OD* is the object diagram name.

The implementation of the ITP/OCL tool comprises around 4000 lines of Maude code. The latest version of the ITP/OCL tool, with the available documentation (including a comparison with related tools) and a collection of examples, can be found at <http://maude.sip.ucm.es/mova>. We are also currently developing the MOVA tool, a Java visual front-end for the ITP/OCL tool: events on the MOVA worksheets and toolbars are transformed into ITP/OCL commands and are interpreted and executed in a Maude process running the ITP/OCL tool.

23.2.8 The Pathway Logic Assistant

Pathway logic [138, 355, 354] is an approach to modeling biological systems as executable rewriting logic specifications, using formal methods tools to analyze these models. As discussed in Section 22.7, a pathway logic model consists of a collection of Maude modules specifying the structure and components of a cell; giving rules describing how signals are propagated in order to control cellular processes such as transcription, metabolism, proliferation, or self-destruction; and defining one or more initial states (called dishes) to study.

For such a model to be useful to biologists, it is crucial to have an interactive visual representation that can be used to navigate and query the model. A few examples of what a biologist might want to do are:

- List the dishes that are available to study.
- Display the network of signaling reactions for a given dish.
- Locate a particular network element, a reaction, or reactant.
- Ask for more information about a particular network element.
- Formulate and submit a query about pathways in the network.

The Pathway Logic Assistant (PLA) was designed to meet the requirements listed above and many others. From an architectural point of view, PLA is a collection of components that communicate using IOP [240], an asynchronous message passing infrastructure designed to support interoperation of formally based tools (see Section 18.5.2). The key components of PLA are PLA-M, an extension of IMAude (see Section 18.5.1) that serves both as coordinator and reasoning engine, and PLA-V, a Java-based viewer that provides interactive visual representations of both models and query results.

Because of the restricted nature of the pathway logic rules, Petri nets are used as the basis for visualization and efficient analysis. In particular, given a specific initial state, the Maude rules are specialized to rule instances reachable from the initial state, and the resulting specialized rules are mapped to Petri net transitions. Figure 23.3 shows the display of a typical model. The full network is displayed in the upper right as a thumbnail with a portion displayed in the main frame at a readable magnification. The graph is generated by PLA-M and a description of the graph is sent to PLA-V as an expression in the JLambda language (see <http://turing.une.edu.au/~iop>) (a Scheme-like interpreted language with access to Java classes and primitives for interactive display.) Graph nodes are made interactive by defining event listeners (JLambda closures) for them. A PLA graph is made interactive by associating actions with the graph as a whole, using tools and menus associated with the graph to provide user access. The result of invoking an event listener or action is typically to send a message to PLA-M requesting some analysis, or informing PLA-M of some graph annotation that has been updated by the user.

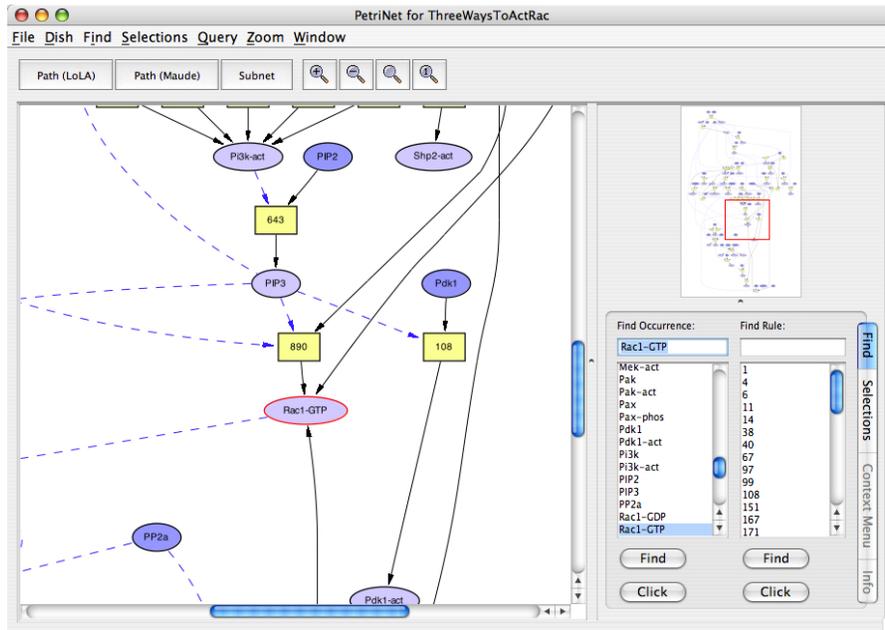


Fig. 23.3. A screen shot of the PLA Viewer

The Pathway Logic Assistant runs on Linux and MacOS X and can be downloaded from the pathway logic web site <http://pl.csl.sri.com/> along with sample models and tutorial material.

Part IV

Reference

Debugging and Troubleshooting

24.1 Debugging approaches

There are several approaches to debugging and optimizing Maude programs: tracing, term coloring, using the debugger, and using the profiler.

24.1.1 Tracing

The tracing facilities allow us to follow the execution of our specifications, that is, the sequence of rewrites or equational simplification reductions that take place. Tracing is turned on with the command

```
set trace on .
```

A log of the trace can be captured using *script* or xterm logging. This can then be studied using a text editor. Since the trace is usually voluminous, there are a number of trace options to control just what is traced. We refer to Section 25.6 for a complete list of tracing commands and options.

One of the most useful options is selective tracing:

```
set trace select on .
trace select foo bar ([_,_]) .
```

This will cause only rewrites where the statement (equation, membership or rule) is labeled with a selected name or the redex is headed by operators with a selected name to be traced. In the above example, suppose `foo` and `bar` are rule labels, `[_,_]` is an operator name, and `foo` is also an operator name. Then, rewrites using the rules labeled by `foo` or `bar` will be reported, as will also rewrites with redex whose top-level operator is either `foo` or `[_,_]`. Note that these labels or operators need not be in existence at the time the `trace select` command is executed; thus it is possible to select statements and operators that will only be created at runtime via the metalevel.

A useful option for metaprogramming is

```
trace exclude FOO BAR .
```

This will exclude the named modules from being traced and thus allows one to selectively avoid tracing the chosen object and/or metalevel modules. This is particularly useful when using Full Maude to localize the tracing to the “object modules” being executed and *not* to the FULL-MAUDE module itself (see Chapter 19). After loading Full Maude, its specification is excluded from the tracing, which allows us to trace Full Maude specifications as Core Maude specifications.

As we have mentioned, there are different commands that may help us in the control of the trace of the execution at hand. If the number of rewrites is small, we may use the whole trace to check the behavior of our specification. However, the number of rewrites is usually big, and considering the whole trace is completely impossible. The different options may help us, for example, to focus on a particular rule or set of rules, exclude certain modules from the trace, or not tracing the rewrites happening in the conditions.

Let us illustrate some of these commands to trace the bank accounts example presented in Section 11.1. To see the trace we just need to set the trace on. After it, the trace of any rewrite command will be given, according to the active options. By default, the application of every equation, membership axiom, and rule will be printed, showing the corresponding substitution, the current whole term, and the subterm on which the axiom is being applied before and after its application. To get a flavor of the information we get, let us rewrite the `bankConf` term with a bound of 1.

```
Maude> set trace on .
Maude> rew [1] bankConf .
rewrite [1] in BANK-ACCOUNT-TEST : bankConf .
***** equation
eq bankConf = (((((< A-003 : Account | bal : 1250 > from A-003 to
  A-002 transfer 300) debit(A-002, 400)) < A-002 : Account | bal :
  250 >) debit(A-001, 150)) debit(A-001, 200)) < A-001 : Account |
  bal : 300 > .
empty substitution
bankConf
--->
((((< A-003 : Account | bal : 1250 > from A-003 to A-002 transfer
  300) debit(A-002, 400)) < A-002 : Account | bal : 250 >) debit(
  A-001, 150)) debit(A-001, 200)) < A-001 : Account | bal : 300 >
***** trial #1
crl debit(A:Oid, M:Nat) < A:Oid : Account | bal : N:Nat > => < A:Oid
  : Account | bal : (N:Nat - M:Nat) > if N:Nat >= M:Nat = true
[label debit] .
A:Oid --> A-001
M:Nat --> 150
N:Nat --> 300
***** solving condition fragment
N:Nat >= M:Nat = true
```

```

***** equation
(built-in equation for symbol _>=_)
300 >= 150
--->
true
***** success for condition fragment
N:Nat >= M:Nat = true
A:Oid --> A-001
M:Nat --> 150
N:Nat --> 300
***** success #1
***** rule
crl debit(A:Oid, M:Nat) < A:Oid : Account | bal : N:Nat > => < A:Oid
  : Account | bal : (N:Nat - M:Nat) > if N:Nat >= M:Nat = true
  [label debit] .
A:Oid --> A-001
M:Nat --> 150
N:Nat --> 300
debit(A-001, 150) debit(A-001, 200) debit(A-002, 400) < A-001 :
  Account | bal : 300 > < A-002 : Account | bal : 250 > < A-003 :
  Account | bal : 1250 > from A-003 to A-002 transfer 300
--->
(debit(A-001, 200) debit(A-002, 400) < A-002 : Account | bal : 250 >
  < A-003 : Account | bal : 1250 > from A-003 to A-002 transfer
  300) < A-001 : Account | bal : (300 - 150) >
***** equation
(built-in equation for symbol _-_)
300 - 150
--->
150
rewrites: 4 in 1ms cpu (1ms real) (4000 rewrites/second)
result Configuration: debit(A-001, 200) debit(A-002, 400) < A-001 :
  Account | bal : 150 > < A-002 : Account | bal : 250 > < A-003 :
  Account | bal : 1250 > from A-003 to A-002 transfer 300

```

Notice that, even though the bound for the rewrite command is one, there are four rewrites. Recall that the bound only concerns rule application. In this trace we see how three equations—two of them built-in—are also applied. In addition to the statement used in each rewriting step, the trace shows the matching substitution and the whole term, before and after the application of the statement. Notice also the information concerning the evaluation of conditions. We can see that, although there is a match with the `debit` rule, this rule is not applied until the success of its condition has been checked.

Suppose we are mainly concerned with the application of rules. In this case we may think that there is too much “noise” due to the application of equations. We may request hiding the information about the application of equations with the command `set trace eq off`. Then the trace for rewriting the same `bankConf` term with the same bound of 1 is as follows:

```

Maude> set trace eq off .
Maude> rew [1] bankConf .
rewrite [1] in BANK-ACCOUNT-TEST : bankConf .
***** trial #1
crl debit(A:Oid, M:Nat) < A:Oid : Account | bal : N:Nat > => < A:Oid
  : Account | bal : (N:Nat - M:Nat) > if N:Nat >= M:Nat = true
  [label debit] .
A:Oid --> A-001
M:Nat --> 150
N:Nat --> 300
***** solving condition fragment
N:Nat >= M:Nat = true
***** success for condition fragment
N:Nat >= M:Nat = true
A:Oid --> A-001
M:Nat --> 150
N:Nat --> 300
***** success #1
***** rule
crl debit(A:Oid, M:Nat) < A:Oid : Account | bal : N:Nat > => < A:Oid
  : Account | bal : (N:Nat - M:Nat) > if N:Nat >= M:Nat = true
  [label debit] .
A:Oid --> A-001
M:Nat --> 150
N:Nat --> 300
debit(A-001, 150) debit(A-001, 200) debit(A-002, 400) < A-001 :
  Account | bal : 300 > < A-002 : Account | bal : 250 > < A-003 :
  Account | bal : 1250 > from A-003 to A-002 transfer 300
--->
(debit(A-001, 200) debit(A-002, 400) < A-002 : Account | bal : 250 >
  < A-003 : Account | bal : 1250 > from A-003 to A-002 transfer
  300) < A-001 : Account | bal : (300 - 150) >
rewrites: 4 in 1ms cpu (0ms real) (4000 rewrites/second)
result Configuration: debit(A-001, 200) debit(A-002, 400) < A-001 :
  Account | bal : 150 > < A-002 : Account | bal : 250 > < A-003 :
  Account | bal : 1250 > from A-003 to A-002 transfer 300

```

The selection of the concrete operator or statement label to trace may also be a good alternative when looking for something specific. Suppose that we are suspicious of a particular rule, say `transfer`. We may get the applications of such a rule for the unbounded rewrite of the `bankConf` term by using the `trace select` command as follows.

```

Maude> set trace select on .
Maude> trace select transfer .
Maude> rew bankConf .
rewrite in BANK-ACCOUNT-TEST : bankConf .
***** trial #1
crl < A:Oid : Account | bal : N:Nat > < B:Oid : Account | bal :

```

```

N':Nat > fromA:Oid to B:Oid transfer M:Nat => < A:Oid : Account |
bal : (N:Nat - M:Nat) > < B:Oid : Account | bal : (M:Nat +
N':Nat) > if N:Nat >= M:Nat = true [label transfer] .
A:Oid --> A-003
N:Nat --> 1250
B:Oid --> A-002
N':Nat --> 250
M:Nat --> 300
***** solving condition fragment
N:Nat >= M:Nat = true
***** success for condition fragment
N:Nat >= M:Nat = true
A:Oid --> A-003
N:Nat --> 1250
B:Oid --> A-002
N':Nat --> 250
M:Nat --> 300
***** success #1
***** rule
crl < A:Oid : Account | bal : N:Nat > < B:Oid : Account | bal :
N':Nat > from A:Oid to B:Oid transfer M:Nat => < A:Oid : Account
| bal : (N:Nat - M:Nat) > < B:Oid : Account | bal : (M:Nat +
N':Nat) > if N:Nat >= M:Nat = true [label transfer] .
A:Oid --> A-003
N:Nat --> 1250
B:Oid --> A-002
N':Nat --> 250
M:Nat --> 300
debit(A-001, 200) debit(A-002, 400) < A-001 : Account | bal : 150 >
< A-002 : Account | bal : 250 > < A-003 : Account | bal : 1250 >
from A-003 to A-002 transfer 300
--->
(debit(A-001, 200) debit(A-002, 400) < A-001 : Account | bal : 150 >)
< A-003 : Account | bal : (1250 - 300) > < A-002 : Account |
bal : (300 + 250) >
rewrites: 13 in 1ms cpu (1ms real) (13000 rewrites/second)
result Configuration: debit(A-001, 200) < A-001 : Account |
bal : 150 > < A-002 : Account | bal : 150 > < A-003 : Account |
bal : 950 >

```

We may also hide some of the information being shown. For example, we may get the same trace without the substitutions being shown with the `set trace substitution off` command.

```

Maude> set trace substitution off .
Maude> rew bankConf .
rewrite in BANK-ACCOUNT-TEST : bankConf .
***** trial #1
crl < A:Oid : Account | bal : N:Nat > < B:Oid : Account | bal :

```

```

      N':Nat > from A:Oid to B:Oid transfer M:Nat => < A:Oid : Account
      | bal : (N:Nat - M:Nat) > < B:Oid : Account | bal : (M:Nat +
      N':Nat) > if N:Nat >= M:Nat = true [label transfer] .
***** solving condition fragment
N:Nat >= M:Nat = true
***** success for condition fragment
N:Nat >= M:Nat = true
***** success #1
***** rule
crl < A:Oid : Account | bal : N:Nat > < B:Oid : Account | bal :
      N':Nat > from A:Oid to B:Oid transfer M:Nat => < A:Oid : Account
      | bal : (N:Nat - M:Nat) > < B:Oid : Account | bal : (M:Nat +
      N':Nat) > if N:Nat >= M:Nat = true [label transfer] .
debit(A-001, 200) debit(A-002, 400) < A-001 : Account | bal : 150 >
< A-002 : Account | bal : 250 > < A-003 : Account | bal : 1250 >
from A-003 to A-002 transfer 300
--->
(debit(A-001, 200) debit(A-002, 400) < A-001 : Account | bal : 150 >)
< A-003 : Account | bal : (1250 - 300) > < A-002 : Account |
bal : (300 + 250) >
rewrites: 13 in 0ms cpu (0ms real) (~ rewrites/second)
result Configuration: debit(A-001, 200) < A-001 : Account |
bal : 150 > < A-002 : Account | bal : 150 > < A-003 : Account |
bal : 950 >

```

Let us consider now a different example, namely, the `PATH` module presented in Sections 3.5 and 4.3. This module was already used for illustrating the possibility of using the tracing facilities in Full Maude in Section 19.1, but we did not show there some of the more interesting issues on the traces for such a module. We use it here to illustrate the trace given for membership axioms and for conditional axioms with multiple fragments. We recall first the conditional membership axiom defining multi-edge paths and the conditional equation defining the associativity of path concatenation.

```

var E : Edge .
vars P Q R S : Path .
cmb E ; P : Path if target(E) = source(P) .
ceq (P ; Q) ; R = P ; (Q ; R)
if target(P) = source(Q) /\ target(Q) = source(R) .

```

Now we request the trace for the reduction of the term `length((b ; c) ; d)`. The information shown is particularly illustrative for understanding the way in which the membership axioms are used and the way conditions are evaluated. Note that the equation expressing the associativity of path concatenation has two fragments, one of which is evaluated after the other. In case the condition of a matching equation fails another equation is attempted; furthermore, equations with matching conditions have unbounded variables initially.

Since the full trace is more than six pages long, we use the `set trace condition off` command, so that the evaluation of the conditions is omitted.

```

Maude> set trace on .
Maude> set trace condition off .
Maude> red length((b ; c) ; d) .
reduce in PATH : length((b ; c) ; d) .
***** trial #1
cmb E ; P : Path if target(E) = source(P) .
E --> b
P --> c
***** solving condition fragment
target(E) = source(P)
***** success for condition fragment
target(E) = source(P)
E --> b
P --> c
***** success #1
***** membership axiom
cmb E ; P : Path if target(E) = source(P) .
E --> b
P --> c
[Path]: b ; c becomes Path
***** trial #2
ceq (P ; Q) ; R = P ; (Q ; R)
  if target(P) = source(Q) /\ target(Q) = source(R) .
P --> b
Q --> c
R --> d
***** solving condition fragment
target(P) = source(Q)
***** success for condition fragment
target(P) = source(Q)
P --> b
Q --> c
R --> d
***** solving condition fragment
target(Q) = source(R)
***** success for condition fragment
target(Q) = source(R)
P --> b
Q --> c
R --> d
***** success #2
***** equation
ceq (P ; Q) ; R = P ; (Q ; R)
  if target(P) = source(Q) /\ target(Q) = source(R) .
P --> b
Q --> c
R --> d
(b ; c) ; d
---->

```

```

b ; (c ; d)
***** trial #3
cmb E ; P : Path if target(E) = source(P) .
E --> c
P --> d
***** solving condition fragment
target(E) = source(P)
***** success for condition fragment
target(E) = source(P)
E --> c
P --> d
***** success #3
***** membership axiom
cmb E ; P : Path if target(E) = source(P) .
E --> c
P --> d
[Path]: c ; d becomes Path
***** trial #4
cmb E ; P : Path if target(E) = source(P) .
E --> b
P --> c ; d
***** solving condition fragment
target(E) = source(P)
***** success for condition fragment
target(E) = source(P)
E --> b
P --> c ; d
***** success #4
***** membership axiom
cmb E ; P : Path if target(E) = source(P) .
E --> b
P --> c ; d
[Path]: b ; (c ; d) becomes Path
***** trial #5
ceq length(E ; P) = 1 + length(P) if E ; P : Path .
E --> b
P --> c ; d
***** solving condition fragment
E ; P : Path
***** success for condition fragment
E ; P : Path
E --> b
P --> c ; d
***** success #5
***** equation
ceq length(E ; P) = 1 + length(P) if E ; P : Path .
E --> b
P --> c ; d
length(b ; (c ; d))

```

```

--->
1 + length(c ; d)
***** trial #6
ceq length(E ; P) = 1 + length(P) if E ; P : Path .
E --> c
P --> d
***** solving condition fragment
E ; P : Path
***** success for condition fragment
E ; P : Path
E --> c
P --> d
***** success #6
***** equation
ceq length(E ; P) = 1 + length(P) if E ; P : Path .
E --> c
P --> d
length(c ; d)
--->
1 + length(d)
***** equation
eq length(E) = 1 .
E --> d
length(d)
--->
1
***** equation
(built-in equation for symbol _+_)
1 + 1
--->
2
***** equation
(built-in equation for symbol _+_)
1 + 2
--->
3
rewrites: 20 in 2ms cpu (1ms real) (10000 rewrites/second)
result NzNat: 3

```

But the trace is too long to observe what we were interested in. Suppose we just wanted to check a possible mistake in the specification of the `length` function. We may select it for filtering the equations defining it.

```

Maude> set trace select on .
Maude> trace select length .
Maude> red length((b ; c) ; d) .
reduce in PATH : length((b ; c) ; d) .
***** trial #1
ceq length(E ; P) = 1 + length(P) if E ; P : Path .

```

```

E --> b
P --> c ; d
***** solving condition fragment
E ; P : Path
***** success for condition fragment
E ; P : Path
E --> b
P --> c ; d
***** success #1
***** equation
ceq length(E ; P) = 1 + length(P) if E ; P : Path .
E --> b
P --> c ; d
length(b ; (c ; d))
--->
1 + length(c ; d)
***** trial #2
ceq length(E ; P) = 1 + length(P) if E ; P : Path .
E --> c
P --> d
***** solving condition fragment
E ; P : Path
***** success for condition fragment
E ; P : Path
E --> c
P --> d
***** success #2
***** equation
ceq length(E ; P) = 1 + length(P) if E ; P : Path .
E --> c
P --> d
length(c ; d)
--->
1 + length(d)
***** equation
eq length(E) = 1 .
E --> d
length(d)
--->
1
rewrites: 20 in 0ms cpu (1ms real) (~ rewrites/second)
result NzNat: 3

```

24.1.2 Term coloring

A common failure mode of Maude programs is when a term does not fully reduce. This is a lack of sufficient completeness. For linear unconditional order-sorted specifications, sufficient completeness can be checked with the SCC tool

[190](see Section 23.1.5). However, for general Maude specifications proving sufficient completeness may require inductive theorem proving. If a term does not fully reduce, that is, if nonconstructor symbols remain in the term’s canonical form, it can be difficult to determine just where the problem began, since when a subterm fails to reduce, the enclosing term often fails to reduce, and so on, leading to a large unreduced term. If the specification makes consistent use of the `ctor` attribute, problem subterms can be pinpointed by switching on term coloring with the command

```
set print color on .
```

Symbols within terms that are being executed (i.e., in a trace or in the final result of a `reduce` command) are colored as follows:

reduced, ctor	not colored
reduced, non-ctor, strangeness below	blue
reduced, non-ctor, no strangeness below	red
unreduced, no reduced above	green
unreduced, reduced directly above	magenta
unreduced, reduced not directly above	cyan

If an operator is colored, this means that the term contains nonconstructors, that is, that there is “strangeness” in the term. The different colors indicate the source of the strangeness. The idea is that red and magenta indicate the initial locus of a bug, while blue and cyan indicate secondary damage. Green denotes reduction pending and cannot appear in the final result. An example is the following module, in which there is a missing case in each of the definitions of the `_<_` and `min` operators ($0 < 0$ and `min(N N)`, respectively).

```
fmod NAT-MSET-MIN is
  protecting BOOL .
  sorts Nat NatMSet .
  subsort Nat < NatMSet .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  op _ _ : NatMSet NatMSet -> NatMSet [assoc comm ctor] .
  op _<_ : Nat Nat -> Bool .
  op min : NatMSet -> Nat .

  vars N M : Nat .
  var S : NatMSet .
  eq 0 < s(N) = true .
  eq s(N) < 0 = false .
  eq s(N) < s(M) = N < M .
  eq min(N N S) = min(N S) .
  ceq min(N M S) = min(N S) if N < M .
  ceq min(N M) = N if N < M .
  eq min(N) = N .
endfm
```

With color printing turned on, reducing `min(s(s(0)) s(s(0)))` returns the term with the `min` operator colored red, indicating a nonconstructor that can't be reduced. Reducing `min(s(s(0)) min(s(0) s(0)))` returns the term with the inner occurrence of the `min` operator colored red as above, and the outer occurrence colored blue, indicating that the problem probably lies in a subterm.

To avoid confusion, any colors that may have been specified using the `format` attribute (see Section 4.4.5) are ignored in this mode.

24.1.3 The debugger

There are three ways to get into the Maude debugger:

- a control-C interrupt during rewriting,
- prefixing a command with the keyword `debug`, and
- hitting a break point.

Break points are set with the command

```
break select foo bar ([_,_]) .
```

where the names refer to operators or statement (equation, membership or rule) labels in a way that is completely analogous to the `trace select` command described in Section 24.1.1. Break points are enabled with the command

```
set break on .
```

On entering the debugger, the prompt changes to `Debug(n)>` where *n* is the debug level, that is, the number of times the debugger has been re-entered (it is fully re-entrant). All top-level commands can be executed from the debugger, along with four commands that are special to the debugger:

`where` . Prints out the stack of pending rewrites, explaining how each one arose.

`step` . Executes the next rewrite with tracing turned on.

`resume` . Exits the debugger and continues with the current rewriting task.

`abort` . Exits the debugger and abandons the current rewriting task.

We illustrate these commands using the bank accounts example presented in Section 11.1 (assuming it is in the file `bank-account-test.maude`).

We first use the `debug` command to activate the debugger from the beginning of a rewrite. Note the use of the `where`, `step`, and `resume` commands.

```
Maude> load bank-account-test.maude
Maude> debug rew bankConf .
rewrite in BANK-ACCOUNT-TEST : bankConf .
Debug(1)> where .
Current term is:
```

```

bankConf
which arose while executing a top level command.
Debug(1)> step .
***** equation
eq bankConf = (((((A-003 : Account | bal : 1250 > from A-003 to
    A-002 transfer 300) debit(A-002, 400)) < A-002 : Account | bal :
    250 >) debit(A-001, 150)) debit(A-001, 200)) < A-001 : Account |
    bal : 300 > .
empty substitution
bankConf
--->
((((((A-003 : Account | bal : 1250 > from A-003 to A-002 transfer
    300) debit(A-002, 400)) < A-002 : Account | bal : 250 >) debit(
    A-001, 150)) debit(A-001, 200)) < A-001 : Account | bal : 300 >
Debug(1)> where .
Current term is:
debit(A-001, 150) debit(A-001, 200) debit(A-002, 400) < A-001 :
    Account | bal : 300 > < A-002 : Account | bal : 250 > < A-003 :
    Account | bal : 1250 > from A-003 to A-002 transfer 300
which arose while executing a top level command.
Debug(1)> resume .
rewrites: 13 in 2ms cpu (85160ms real) (6500 rewrites/second)
result Configuration: debit(A-001, 200) < A-001 : Account | bal :
    150 > < A-002 : Account | bal : 150 > < A-003 : Account | bal :
    950 >

```

As said above, we can also enter into the debugger by reaching a break point or typing control-c. In the following example we set a break point on the debit rule, take a step, and then abort the rewrite process.

```

Maude> set break on .
Maude> break select debit .
Maude> rew bankConf .
rewrite in BANK-ACCOUNT-TEST : bankConf .
break on labeled rule:
crl debit(A:Oid, M:Nat) < A:Oid : Account | bal : N:Nat > => < A:Oid
    : Account | bal : (N:Nat - M:Nat) > if N:Nat >= M:Nat = true [
    label debit] .
Debug(1)> where .
Current term is:
debit(A-001, 150) debit(A-001, 200) debit(A-002, 400) < A-001 :
    Account | bal : 300 > < A-002 : Account | bal : 250 > < A-003 :
    Account | bal : 1250 > from A-003 to A-002 transfer 300
which arose while executing a top level command.
Debug(1)> step .
***** trial #1
crl debit(A:Oid, M:Nat) < A:Oid : Account | bal : N:Nat > => <
    A:Oid : Account | bal : (N:Nat - M:Nat) > if N:Nat >= M:Nat =
    true [label debit] .

```

```

A:Oid --> A-001
M:Nat --> 150
N:Nat --> 300
***** solving condition fragment
N:Nat >= M:Nat = true
Debug(1)> where .
Current term is:
300 >= 150
which arose while checking a condition during the evaluation of:
debit(A-001, 150) debit(A-001, 200) debit(A-002, 400) < A-001 :
  Account | bal : 300 > < A-002 : Account | bal : 250 > < A-003 :
  Account | bal : 1250 > from A-003 to A-002 transfer 300
which arose while executing a top level command.
Debug(1)> abort .
Maude>

```

Our last example illustrates the re-entering nature of the debugger. As said above, any command can be used during the debugging process, allowing, for example, starting an execution while debugging another one. We execute a `debug rew` command, entering the debugger, where we set a break point on the `transfer` rule. Notice the `Debug(2)>` prompt. Notice also how after getting out of the inner debugger the break point is still active.

```

Maude> debug rew bankConf .
rewrite in BANK-ACCOUNT-TEST : bankConf .
Debug(1)> step .
***** equation
eq bankConf = ((((( < A-003 : Account | bal : 1250 > from A-003 to
  A-002 transfer 300) debit(A-002, 400)) < A-002 : Account | bal
  : 250 >) debit(A-001, 150)) debit(A-001, 200)) < A-001 : Account
  | bal : 300 > .
empty substitution
bankConf
--->
(((( < A-003 : Account | bal : 1250 > from A-003 to A-002 transfer
  300) debit(A-002, 400)) < A-002 : Account | bal : 250 >) debit(
  A-001, 150)) debit(A-001, 200)) < A-001 : Account | bal : 300 >
Debug(1)> set break on .
Debug(1)> break select transfer .
Debug(1)> rew bankConf .
rewrite in BANK-ACCOUNT-TEST : bankConf .
break on labeled rule:
crl < A:Oid : Account | bal : N:Nat > < B:Oid : Account | bal :
  N':Nat > from A:Oid to B:Oid transfer M:Nat => < A:Oid : Account
  | bal : (N:Nat - M:Nat) > < B:Oid : Account | bal : (M:Nat +
  N':Nat) > if N:Nat >= M:Nat = true [label transfer] .
Debug(2)> where .
Current term is:
debit(A-001, 200) debit(A-002, 400) < A-001 : Account | bal : 150 >

```

```

    < A-002 : Account | bal : 250 > < A-003 : Account | bal : 1250 >
    from A-003 to A-002 transfer 300
which arose while executing a top level command.
Debug(2)> resume .
break on labeled rule:
crl < A:Oid : Account | bal : N:Nat > < B:Oid : Account | bal :
    N':Nat > from A:Oid to B:Oid transfer M:Nat => < A:Oid : Account |
    bal : (N:Nat - M:Nat) > < B:Oid : Account | bal : (M:Nat + N':Nat)
    > if N:Nat >= M:Nat = true [label transfer] .
Debug(2)> abort .
Debug(1)> resume .
break on labeled rule:
crl < A:Oid : Account | bal : N:Nat > < B:Oid : Account | bal :
    N':Nat > from A:Oid to B:Oid transfer M:Nat => < A:Oid : Account
    | bal : (N:Nat - M:Nat) > < B:Oid : Account | bal : (M:Nat +
    N':Nat) > if N:Nat >= M:Nat = true [label transfer] .
Debug(1)> set break off .
Debug(1)> resume .
rewrites: 13 in 4ms cpu (63920ms real) (2600 rewrites/second)
result Configuration: debit(A-001, 200) < A-001 : Account | bal :
    150 > < A-002 : Account | bal : 150 > < A-003 : Account | bal :
    950 >

```

24.1.4 The profiler

Tuning up of specifications is something that may be useful in many practical situations. We illustrate the use of the profiling facilities available in Maude to understand better the execution of our specifications, helping us in this way to make them more efficient. We will discuss the use of the profiler on two examples, namely, the specification of the Fibonacci function already discussed in Section 4.4.8 and the specification of sorted lists presented in Section 10.5.

First of all, it must be clear that there is no magic recipe on how to optimize our specifications. On the contrary, although there are some guidelines that we may try to follow when possible, it is not always the case that they work, or that they are applicable. For example, conditional rules are generally expensive from a computational point of view, as are membership axioms, but in some cases we may be interested in using proving tools for which using them could be a better alternative. Similarly, in Section 4.4.8 we saw that using the `memo` attribute was a big win in the case of the Fibonacci function, but it is not always applicable; for some specifications, the consumption of memory can become so big that we may be getting a slower specification. There is always a tradeoff between the speedup obtained using memoization and the amount of memory and the cost of handling it. We illustrate all these and other concerns in this section.

Profiling is switched on by the command `set profile on`. When profiling is switched on, a count of the number of executions of each statement

(equation, membership, and rule) is kept. For unconditional statements, the profile information is just the number of rewrites using that statement. For conditional ones there is also the number of matches, since not every match leads to a rewrite, due to condition failure. Moreover, when searching there can be multiple rewrites for each match, since the condition may be solved in multiple ways. There is a table that for each condition fragment gives:

1. the number of times the fragment was initially tested,
2. the number of times the fragment was tested due to backtracking,
3. the number of times the fragment succeeded, and
4. the number of times the fragment failed.

Normally, $(1) + (2) = (3) + (4)$.

Special rewrites such as built-in rewrites and memoized rewrites are also tracked, but these are associated with symbols rather than with statements. For conciseness, symbols with no special rewrites, and statements that are not matched are omitted. There are some limitations: metalevel rewrites are not displayed, due to the ephemeral nature of metamodules. In addition, condition fragments associated with a match or search command are not tracked (though any rewrites initiated by such a fragment are). If you turn profiling on or off in the debugger you may get inconsistent results.

The profile information is associated with each module and is usually cleared at the start of any command that can do rewrites, except `continue`. This behavior can be changed with the `set clear profile on / off` command.

Let us first consider the Fibonacci function described in Section 4.4.8.

```
fmod FIBONACCI is
  protecting NAT .
  op fibo : Nat -> Nat .

  var N : Nat .
  eq fibo(0) = 0 .
  eq fibo(1) = 1 .
  eq fibo(s s N) = fibo(N) + fibo(s N) .
endfm
```

Notice in the following reductions that the times given when the profile is active are slightly higher.

```
Maude> red fibo(30) .
reduce in FIBONACCI : fibo(30) .
rewrites: 4038805 in 3920ms cpu (3960ms real) (1030201 rews/sec)
result NzNat: 832040

Maude> set profile on .

Maude> red fibo(30) .
reduce in FIBONACCI : fibo(30) .
```

```
rewrites: 4038805 in 4170ms cpu (4194ms real) (968453 rews/sec)
result NzNat: 832040
```

After doing the reduction with the profiler activated, we can request the collected information by means of the command `show profile`. In this example, since the module has no memberships or rules and there are no conditional axioms, the profiler gives the number of times each of the equations has been applied and also the number of times built-in functions are called.

```
Maude> show profile .

op _+_ : [Nat] [Nat] -> [Nat] .
built-in eq rewrites: 1346268 (33.3333%)

eq fibo(0) = 0 .
rewrites: 514229 (12.7322%)

eq fibo(1) = 1 .
rewrites: 832040 (20.6011%)

eq fibo(s^2(N)) = fibo(N) + fibo(s N) .
rewrites: 1346268 (33.3333%)
```

In this very simple example we observe that only the three equations in the `FIBONACCI` module plus the predefined addition operation on natural numbers have been used. We can also observe how the equations are applied a number of times relatively similar, with percentages 12, 20, and 33, respectively. More interesting is the number of times each of them is applied, which goes to 1346268 for the third equation. Taking into account that we reduced `fibo(30)`, it means that the calculations have been repeated many times. As we saw in Section 4.4.8, this is a good place to use the `memo` attribute: calculations on small arguments are repeated many times and a small amount of memory is needed for storing the result of such calculations.

After adding the `memo` attribute to the `fibo` operator, we get the following results from the profiler:

```
Maude> set profile on .

Maude> red fibo(30) .
reduce in FIBONACCI : fibo(30) .
rewrites: 88 in 1ms cpu (1ms real) (88000 rews/sec)
result NzNat: 832040

Maude> show profile .
op _+_ : [Nat] [Nat] -> [Nat] .
built-in eq rewrites: 29 (32.9545%)

op fibo : [Nat] -> [Nat] .
memo rewrites: 28 (31.8182%)
```

```

eq fibo(0) = 0 .
rewrites: 1 (1.13636%)

eq fibo(1) = 1 .
rewrites: 1 (1.13636%)

eq fibo(s_~2(N)) = fibo(N) + fibo(s N) .
rewrites: 29 (32.9545%)

```

As we already saw in Section 4.4.8, the number of rewrites and the time consumed in the computation have decreased dramatically. We may observe that now each of the values in the Fibonacci sequence is calculated only once.

Let us consider now another example, namely, the parameterized module `SORTED-LIST` presented in Section 10.5, which defines a sort `SortedList{X}` of sorted lists as a subsort of the sort `List{TOSET}{X}` of lists. In this case we deal with a parameterized module which imports several other modules, and which has membership axioms and equations, some of which are conditional.

First of all, notice that by default the profiler provides information on a particular computation. In this example, it is not the same sorting a list in reverse order as an already sorted list, and is not the same using insertion sort, mergesort, or quicksort for sorting. To have a better insight about our specification, and thus gaining chances of improving it, we should consider several reductions, dealing with different cases, the different sorting algorithms in our case.

To be able to run examples on big lists, with numbers initially sorted in different ways, let us consider the following module `NAT-LIST-GENERATOR`, which imports the module `SORTED-LIST{NatAsToset}` defining sorted lists of natural numbers, and specifies functions `nats-upto`, that builds lists from zero to the specified value, and `random-nats`, which generates a list of the specified number of random numbers.

```

fmod NAT-LIST-GENERATOR is
  protecting SORTED-LIST{NatAsToset} .
  protecting RANDOM .

vars N M : Nat .

op nats-upto : Nat -> NeSortedList{NatAsToset} .
eq nats-upto(s N) = nats-upto(N) ++ s N : [] .
eq nats-upto(0) = 0 : [] .

op random-nats : Nat -> List{TOSET}{NatAsToset} .
op random-nats : Nat Nat -> List{TOSET}{NatAsToset} .
eq random-nats(N) = random-nats(0, N) .
ceq random-nats(N, M)
  = random(N) : random-nats(s N, M)
  if N <= M .

```

```

    eq random-nats(N, M) = [] [owise] .
endfm

```

We execute each one of the `insertion-sort`, `mergesort`, and `quicksort` algorithms on three lists, namely, a sorted list, a list in reverse order, and a random one, each of them with 1000 elements.

```

Maude> red insertion-sort(nats-upto(1000)) .
rewrites: 2012009 in 1032ms cpu (1079ms real) (1948029 rews/sec)
result NeSortedList{NatAsToset}: 0 : 1 : 2 : 3 : 4 : 5 : 6 : 7 : ...

```

```

Maude> red insertion-sort(reverse(nats-upto(1000))) .
rewrites: 5519515 in 3634ms cpu (3694ms real) (1518667 rews/sec)
result NeSortedList{NatAsToset}: 0 : 1 : 2 : 3 : 4 : 5 : 6 : 7 : ...

```

```

Maude> red insertion-sort(random-nats(1000)) .
rewrites: 1535372 in 1286ms cpu (1397ms real) (1193166 rews/sec)
result NeSortedList{NatAsToset}: 23772 : 1738648 : 2016694 : ...

```

```

Maude> red mergesort(nats-upto(1000)) .
rewrites: 2082358 in 1079ms cpu (1134ms real) (1928402 rews/sec)
result NeSortedList{NatAsToset}: 0 : 1 : 2 : 3 : 4 : 5 : 6 : 7 : ...

```

```

Maude> red mergesort(reverse(nats-upto(1000))) .
rewrites: 2578581 in 1210ms cpu (1221ms real) (2129622 rews/sec)
result NeSortedList{NatAsToset}: 0 : 1 : 2 : 3 : 4 : 5 : 6 : 7 : ...

```

```

Maude> red mergesort(random-nats(1000)) .
rewrites: 88005 in 71ms cpu (77ms real) (1222478 rews/sec)
result NeSortedList{NatAsToset}: 23772 : 1738648 : 2016694 : ...

```

```

Maude> red quicksort(nats-upto(1000)) .
rewrites: 6519514 in 5065ms cpu (5344ms real) (1287111 rews/sec)
result NeSortedList{NatAsToset}: 0 : 1 : 2 : 3 : 4 : 5 : 6 : 7 : ...

```

```

Maude> red quicksort(reverse(nats-upto(1000))) .
rewrites: 6528518 in 4096ms cpu (4130ms real) (1593729 rews/sec)
result NeSortedList{NatAsToset}: 0 : 1 : 2 : 3 : 4 : 5 : 6 : 7 : ...

```

```

Maude> red quicksort(random-nats(1000)) .
rewrites: 97858 in 75ms cpu (84ms real) (1287791 rews/sec)
result NeSortedList{NatAsToset}: 23772 : 1738648 : 2016694 : ...

```

Instead of considering the profiling information separately, we use the `set clear profile off` command, so that the profiling information gets accumulated.

```

Maude> set clear profile off .

```

```

Maude> set profile on .

```

```

Maude> red insertion-sort(nats-upto(1000)) .
rewrites: 2012009 in 1169ms cpu (1351ms real) (1719927 rews/sec)
result NeSortedList{NatAsToset}: 0 : 1 : 2 : 3 : 4 : 5 : 6 : 7 : ...

Maude> red insertion-sort(reverse(nats-upto(1000))) .
...

```

As mentioned above, with the profiler active, the times taken by the reductions are slightly higher. The number of rewrites and cpu time for each of the cases is presented in the tables displayed in Figure 24.1 (page 787), where we also include these values for all the different executions discussed in the rest of the section.¹ The information shown by the profiler with the `show profile` command is three pages long; we just comment the most interesting pieces.

1. Predefined operators `_+_`, `_quo_`, `_<=_`, `_>_`, and `random` are used an important number of times, in particular `_<=_` and `_>_`, which are applied, respectively, 7599823 (28.2%) and 1770593 times (6.6%). Notice that `_<=_` is only used in the conditions of one of the membership axioms and in some of the conditions of the equations for `insert-list`, `merge`, `leq-elems`, `gr-elems`, and `random-nats`; the operator `_>_` is used in the conditions of the equations for `insert-list`, `mergesort`, `merge`, `leq-elems`, and `gr-elems`.

```

op _+_ : [Nat] [Nat] -> [Nat] .
built-in eq rewrites: 29961 (0.111124%)

op _quo_ : [Nat] [Nat] -> [Nat] .
built-in eq rewrites: 3000 (0.0111269%)

op _<=_ : [Nat] [Nat] -> [Bool] .
built-in eq rewrites: 7599823 (28.1874%)

op _>_ : [Nat] [Nat] -> [Bool] .
built-in eq rewrites: 1770593 (6.56706%)

op random : [Nat] -> [Nat] .
built-in eq rewrites: 3003 (0.011138%)

```

2. From the numbers for the membership axioms we may conclude that they are applied a considerable number of times, in particular the conditional one—4790561 rewrites (17.8%)—. Note that for conditional membership axioms, as for all conditional axioms, the system gives information on the number of matches, that is, the number of times that the conditions

¹ All these figures have been obtained running Maude during a hot summer night on a Linux platform with an Intel Pentium M760 2GHz processor and 1GB of memory.

are reduced. It also provides the number of times each one of the fragments of the condition is reduced. In the case of the membership axioms in this specification, there is only one fragment. The part of the output corresponding to the membership axioms is the following:

```
mb [] : SortedList{NatAsToset} .
rewrites: 22040 (0.0817455%)

mb N : [] : NeSortedList{NatAsToset} .
rewrites: 17505 (0.0649254%)

cmb N : NEOL:NeSortedList{NatAsToset} : NeSortedList{NatAsToset}
  if N <= head(NEOL:NeSortedList{NatAsToset}) = true .
lhs matches: 4795972    rewrites: 4790561 (17.768%)
Fragment   Initial tries   Resolve tries   Successes   Failures
1           4795972                0               4790561    5411
```

We see that the condition has been checked 4795972 times, out of which only 5411 failed.

- From the equations specifying the insertion-sort algorithm, the ones used more times are the two conditional ones for the `insert-list` function. From the information in the profile we see that these conditional equations have been attempted almost the same number of times, 756888 and 754895. We also see that both have been applied almost the same number of times, because the one that was attempted first almost always failed the evaluation of its condition, and then the second equation was applied.

```
eq insertion-sort([]) = [] .
rewrites: 3 (1.11269e-05%)

eq insertion-sort(N : L:List{TOSET}{NatAsToset}) = insert-list(
  insertion-sort(L:List{TOSET}{NatAsToset}), N) .
rewrites: 3003 (0.011138%)

eq insert-list([], M) = M : [] .
rewrites: 1010 (0.00374605%)

ceq insert-list(N : OL:SortedList{NatAsToset}, M) = M : N :
  OL:SortedList{NatAsToset} if M <= N = true .
lhs matches: 756888    rewrites: 1993 (0.00739196%)
Fragment   Initial tries   Resolve tries   Successes   Failures
1           756888                0               1993        754895

ceq insert-list(N : OL:SortedList{NatAsToset}, M) = N :
  insert-list(OL:SortedList{NatAsToset}, M) if M > N = true .
lhs matches: 754895    rewrites: 754895 (2.79988%)
Fragment   Initial tries   Resolve tries   Successes   Failures
1           754895                0               754895      0
```

4. The information on the equations for `mergesort` presents a similar pattern, the main difference being that there is only one conditional equation, since `merge` is declared as commutative. In this case the number of rewrites of all the equations is relatively small, much smaller than the total of rewrites in the computation. This makes us think that the weight of the computation of the memberships and generation of the lists to sort is much higher than the sorting itself.

```

eq mergesort(N : []) = N : [] .
rewrites: 3003 (0.011138%)

ceq mergesort(L:List{TOSET}{NatAsToset}) = merge(mergesort(take
  length(L:List{TOSET}{NatAsToset}) quo 2 from L:List{TOSET}{
  NatAsToset}),mergesort(throw length(L:List{TOSET}{
  NatAsToset}) quo 2 from L:List{TOSET}{NatAsToset})) if
  length(L:List{TOSET}{NatAsToset}) > 1 = true .
lhs matches: 3000      rewrites: 3000 (0.0111269%)
Fragment   Initial tries   Resolve tries   Successes   Failures
1           3000             0               3000        0

eq merge([], OL:SortedList{NatAsToset}) = OL:SortedList{
  NatAsToset} .
rewrites: 3000 (0.0111269%)

ceq merge(N : OL:SortedList{NatAsToset}, M : OL':SortedList{
  NatAsToset}) = N :merge(OL:SortedList{NatAsToset}, M :
  OL':SortedList{NatAsToset}) if N <= M = true .
lhs matches: 18705     rewrites: 18705 (0.0693761%)
Fragment   Initial tries   Resolve tries   Successes   Failures
1           18705             0               18705        0

```

5. The information for the `quicksort` algorithm follows a similar pattern as well. However, in this case it is interesting to notice that the equations for the `leq-elems` and `gr-elems` operations are attempted the same number of times, and for each of these operations the condition (say, $N \leq M$) of one of the equations fails in around half of the cases, being then used the other equation (with condition, say, $N > M$).

```

eq quicksort([]) = [] .
rewrites: 3006 (0.0111491%)

eq quicksort(N : L:List{TOSET}{NatAsToset}) = quicksort(
  leq-elems(L:List{TOSET}{NatAsToset}, N)) ++ N : quicksort(
  gr-elems(L:List{TOSET}{NatAsToset}, N)) .
rewrites: 3003 (0.011138%)

eq leq-elems([], M) = [] .
rewrites: 3003 (0.011138%)

```

```

ceq leq-elems(N : L>List{TOSET}{NatAsToset}, M) = N :
  leq-elems(L>List{TOSET}{NatAsToset}, M) if N <= M = true .
lhs matches: 1012626    rewrites: 506277 (1.87776%)
Fragment   Initial tries   Resolve tries   Successes   Failures
1          1012626         0              506277     506349

ceq leq-elems(N : L>List{TOSET}{NatAsToset}, M) = leq-elems(
  L>List{TOSET}{NatAsToset}, M) if N > M = true .
lhs matches: 506349    rewrites: 506349 (1.87803%)
Fragment   Initial tries   Resolve tries   Successes   Failures
1          506349         0              506349     0

eq gr-elems([], M) = [] .
rewrites: 3003 (0.011138%)

ceq gr-elems(N : L>List{TOSET}{NatAsToset}, M) = gr-elems(
  L>List{TOSET}{NatAsToset}, M) if N <= M = true .
lhs matches: 1012626    rewrites: 506277 (1.87776%)
Fragment   Initial tries   Resolve tries   Successes   Failures
1          1012626         0              506277     506349

ceq gr-elems(N : L>List{TOSET}{NatAsToset}, M) = N : gr-elems(
  L>List{TOSET}{NatAsToset}, M) if N > M = true .
lhs matches: 506349    rewrites: 506349 (1.87803%)
Fragment   Initial tries   Resolve tries   Successes   Failures
1          506349         0              506349     0

```

6. From the rest of the equations applied we may highlight those for the head and `_++_` operations.

```

eq head(E:Nat : L>List{TOSET}{NatAsToset}) = E:Nat .
rewrites: 4795972 (17.7881%)

eq [] ++ L>List{TOSET}{NatAsToset} = L>List{TOSET}{
  NatAsToset} .
rewrites: 12006 (0.0445298%)

eq (E:Nat : L>List{TOSET}{NatAsToset}) ++ L':List{TOSET}{
  NatAsToset} =E:Nat : (L>List{TOSET}{NatAsToset} ++ L':List{
  Toset}{NatAsToset}) .
rewrites: 5010777 (18.5848%)

```

The equation for the `head` function is used in the evaluation of the condition of the membership axiom. The concatenation operator is used in the `quicksort`, `nats-upto`, and `reverse` functions.

Taking all the information provided by the profiler into account, we may think of doing different types of modifications to the original specification.

- None of the operators seems to be appropriate for memoization, since they are used on many different arguments, and if repeated, the size of

the argument lists is so big that it is probably not worthy storing the results.

Let us, in any case, add the `memo` attribute, e.g., to the `head` operator; the result for one of the reductions above is the following:

```
Maude> red insertion-sort(random-nats(1000)) .
rewrites: 1535372 in 18500ms cpu (19221ms real) (82992 rews/sec)
result NeSortedList{NatAsToset}: 23772 : 1738648 : 2016694 : ...
```

That is, it goes from 1286 milliseconds cpu time to 18500. See Figure 24.1 for the rest of the values.

- Conditional equations are in general computationally expensive. Let us write the two conditional equations for `insertion-sort` as one single unconditional equation:

```
eq insert-list(N : OL, M)
= if M <= N
  then M : N : OL
  else N : insert-list(OL, M)
fi .
```

The result for the same reduction is the following:

```
Maude> reduce insertion-sort(random-nats(1000)) .
rewrites: 1536365 in 638ms cpu (704ms real) (2404692 rews/sec)
result NeSortedList{NatAsToset}: 23772 : 1738648 : 2016694 : ...
```

Although the number of rewrites increases slightly—from 1535372 to 1536365—the amount of cpu time has dropped to a half—from 1286 ms. to 638 ms.

- We may use the `owise` attribute for making the conditional equation of the `mergesort` function unconditional, that is, we may write it as:

```
eq mergesort(L)
= merge(mergesort(take (length(L) quo 2) from L),
        mergesort(throw (length(L) quo 2) from L))
[owise] .
```

Although this clearly improves the equation, the win is not very significant. To test it, we run the `mergesort` algorithm on a randomly generated list.

```
Maude> red mergesort(random-nats(1000)) .
rewrites: 87005 in 68ms cpu (110ms real) (1261124 rews/sec)
result NeSortedList{NatAsToset}: 237728 : 17386481 : ...
```

The number of rewrites goes from 88005 to 87005 and the cpu time goes from 71 ms. to 68 ms.

- A more important gain may be obtained by improving the splitting of the lists for the quicksort algorithm. Let us join the `leq-elems` and `gr-elems` functions in one single `leq-gr-elems` returning a pair of lists, one with the smaller or equal elements and the other with the greater ones.

		original spec.	memo head	unconditional insert-list	
		profiler on	profiler off		
nats-upto	rews	2012009			2013009
	ms	1169	1032	96188	1029
reverse-nats-upto	rews	5519515			5519515
	ms	4189	3634	193977	2376
random-nats	rews	1535372			1536365
	ms	1434	1286	18500	638

insertion-sort

		original spec.	memo head	merge owise	
		profiler on	profiler off		
nats-upto	rews	2082358			2081358
	ms	1223	1079	95760	1079
reverse-nats-upto	rews	2578581			2577581
	ms	1409	1210	99579	1204
random-nats	rews	88005			87005
	ms	108	71	559	68

mergesort

		original spec.	memo head	improved splitting	
		profiler on	profiler off		
nats-upto	rews	6519514			5267013
	ms	5671	5065	221387	2523
reverse-nats-upto	rews	6528518			5777017
	ms	4451	4096	196991	2650
random-nats	rews	97858			69612
	ms	83	75	693	31

quicksort

Fig. 24.1. Number of rewrites and cpu time for different versions of the sorting algorithms

```

op quicksort : List{TOSET}{X} -> SortedList{X} .
op leq-gr-elems : List{TOSET}{X} X$Elt -> LGRResult .
op leq-gr-elems : List{TOSET}{X} List{TOSET}{X} List{TOSET}{X}
  X$Elt -> LGRResult .

sort LGRResult .
op {_,_} : List{TOSET}{X} List{TOSET}{X} -> LGRResult .

eq quicksort([]) = [] .
ceq quicksort(N : L)
  = quicksort(L') ++ (N : quicksort(L''))
  if {L', L''} := leq-gr-elems(L, N).

eq leq-gr-elems(L, M) = leq-gr-elems(L, [], [], M) .
eq leq-gr-elems([], L, L', M) = {L, L'} .
eq leq-gr-elems(N : L, L', L'', M)
  = if N <= M
    then leq-gr-elems(L, N : L', L'', M)
    else leq-gr-elems(L, L', N : L'', M)
  fi .

```

The execution of the `quicksort` function on a list of randomly generated numbers takes now 69612 rewrites (against 97858) and does the reduction in 31 milliseconds (around 75 before).

```

Maude> red quicksort(random-nats(1000)) .
reduce in SORTED-LIST-TEST : quicksort(random-nats(1000)) .
rewrites: 69612 in 31ms cpu (72ms real) (2175714 rews/sec)
result NeSortedList{NatAsToset}: 237728 : 17386481 : ...

```

There is still room for improvement in this specification. For instance, some operations on lists can be made more efficient by means of *tail-recursive* definitions with accumulator arguments, in the style of the definitions shown in Section 9.12.1.

24.1.5 Performance note

Turning on tracing, break points, or profiling causes Maude to run much more slowly, because these options force execution through a slow path that performs extensive bookkeeping before and after each rewrite, membership application, and condition fragment check. Therefore, execution will be significantly slower if one or more of these options is on, even if no tracing information is output, no break points are encountered, and the profile information is never examined.

To ensure Maude is running at full speed one must use:

```

set trace off .
set break off .
set profile off .

```

24.2 Traps and known problems

We list some commonly encountered problems with Maude.

24.2.1 Associativity and idempotency

Remember that the attributes `assoc` and `idem` (see Section 4.4.1) cannot be used together in any combination of attributes, because the appropriate matching and normalization algorithms have not been developed yet.

This requirement is quietly enforced by ignoring the attribute `idem` where necessary.

Let us consider the following example, in which we wrongly declare an operator with the attributes `assoc` and `idem` appearing together.

```
fmod WRONG-NAT-SET is
  pr NAT .
  sort WNatSet .
  subsort Nat < WNatSet .
  op none : -> WNatSet [ctor] .
  op _ : WNatSet WNatSet -> WNatSet
    [ctor assoc comm idem id: none] .
endfm
```

When we reduce a term like, e.g., `4 4 5 2`, the duplication does not disappear, because Maude has ignored the idempotency attribute; the remaining attributes are applied as usual.

```
Maude> red 4 4 5 2 .
result WNatSet: 2 4 4 5
```

We can solve this by adding explicitly an idempotency equation, as we have seen, for example, in Section 9.12.2.

Combining `idem` with attributes other than `assoc` is all right. For example, the following module combines idempotency with commutativity.

```
fmod COMM-IDEM-EX is
  pr NAT .
  sort CI .
  subsort Nat < CI .
  op f : CI CI -> CI [ctor comm idem] .

  vars N M : Nat .
  var C : CI .

  op g : CI -> Nat .
  eq g(f(N, M)) = 0 .
  eq g(C) = 1 [owise] .
endfm
```

```
Maude> red f(2, 2) .
result NzNat: 2
```

```
Maude> red f(2, 3) == f(3, 2) .
result Bool: true
```

Notice that in this module, because of matching modulo commutativity and idempotency, the first equation for `g` can be applied to a term such as, e.g., `g(2)`. For example, we have the following reductions:

```
Maude> red g(2) .
result Zero: 0
```

```
Maude> red g(f(2, f(2, 2))) .
result Zero: 0
```

```
Maude> red g(f(2, f(3, 4))) .
result NzNat: 1
```

24.2.2 Segmentation fault (core dumped)

This looks like a bug in Maude, but in fact it is a stack overflow (a real segmentation fault is caught and reported as an “internal error”). On a Unix box you can find out the current limit on your stack size with the (shell) command

```
limit stacksize
```

This is often set to 8192K by default, which is quite inappropriate for a highly recursive system like Maude. You can set the stack size to a larger value with, for example,

```
limit stacksize 100M
```

or remove the limit altogether with

```
unlimit stacksize
```

Note that stack overflows are reported as `Illegal instruction` on both PowerPC- and Intel-based Macs.

24.2.3 Bare variable lefthand sides

The use of a bare variable lefthand side for an equation, rule, or membership axiom may lead to unexpected nontermination. The recommended place to use them is in statements declared with the `nonexec` attribute, which are only going to be applied via a strategy language. Using them in membership axioms is seductive, but very tricky. For example:

```

subsort Prime < Nat .
var N : Nat .
cmb N : Prime if favoritePrimeTest(N) .

```

will end up with the membership axiom and `favoritePrimeTest` being applied to every reduced term of sort `Nat`, including those that arise during evaluation of `favoritePrimeTest(N)` with likely nontermination.

24.2.4 Operator overloading and associativity

The situation where two ad-hoc overloaded operators have the same kinds in their arities but different ones in their coarities causes a warning to be emitted, as already mentioned in Section 3.6. For example, loading the file `overloading-assoc-warning.maude` containing the module

```

fmod OVER-ASSOC-EX1 is
  sorts Foo Bar .
  op f : Foo -> Foo .
  op f : Foo -> Bar .
endfm

```

causes the following warning:

```

Warning: "overloading-assoc-warning.maude", line 8 (fmod
OVER-ASSOC-EX1): declaration for f has the same domain kinds as
the declaration on "overloading-assoc-warning.maude", line 7
(fmod OVER-ASSOC-EX1) but a different range kind.

```

A similar warning is obtained in the case where the arities differ but might look the same because of associativity, like in the following example (loaded as before):

```

fmod OVER-ASSOC-EX2 is
  sort Foo .
  op f : Foo Foo -> Foo [assoc] .
  op f : Foo Foo Foo -> Foo .
endfm

```

```

Warning: "overloading-assoc-warning.maude", line 22
(fmod OVER-ASSOC-EX2): declaration for f clashes with
declaration on "overloading-assoc-warning.maude", line 21 (fmod
OVER-ASSOC-EX2) because of associativity.

```

24.2.5 Preregularity and equational attributes

We recall from Section 3.8 that Maude assumes that modules are *preregular* and generates warnings when a module contains operator declarations that do not satisfy this property. This means that for each possible combination of argument sorts the resulting term has a unique least type, which is usually

a sort but might also be the kind, depending on the operator declarations. However, as also explained in Section 3.8, in the presence of equational attributes, such as `assoc`, `comm`, `id:`, and `idem` (see Section 4.4.1), preregularity must be understood *modulo* the axioms A declared by such attributes. That is, we want not just each term t , but also each equivalence class $[t]_A$ to have a least sort. Therefore, there is an additional requirement for an operator that is declared associative, namely, that the least type of a term should not depend on the way nested operators are associated. Let us explain this situation in some detail.

The `assoc` attribute, stating that a binary operator is associative, appears usually associated with declarations of operators whose arguments are both of the same sort, like, for example,

```
op _+_ : Nat Nat -> Nat [assoc] .
```

However, in the presence of subsorts and overloaded operators it also makes sense to have binary operators whose arguments are not the same, but are related via subsorting; for example, to make it explicit that the addition of a natural number to a nonzero natural number produces a nonzero natural number, we can have an additional declaration

```
op _+_ : NzNat Nat -> NzNat [assoc] .
```

or also (see Section 4.4.6)

```
op _+_ : NzNat Nat -> NzNat [ditto] .
```

Thus, in general, the `assoc` attribute is allowed for binary operators such that the two argument sorts and the result sort all belong to the same connected component. Therefore, it is possible to consider a module like the following:

```
fmod NON-ASSOCIATIVE-EX is
  sorts s1 s2 .
  subsort s1 < s2 .
  op f : s1 s2 -> s2 [assoc] .
  op a : -> s1 .
  eq f(a,a) = a .
endfm
```

If we try to reduce the term $f(a,a)$, we get the following warning:

```
Maude> red f(a, a) .
Warning: sort declarations for associative operator f are
  non-associative on 2 out of 27 sort triples. First such triple is
  (s1, s1, s2).
reduce in NON-ASSOCIATIVE-EX : f(a, a) .
rewrites: 1 in 0ms cpu (0ms real) (~ rews/sec)
result s1: a
```

Maude has checked the preregularity property on the associative operator f . It is enough to check this property on each triple of types, and when it fails to hold Maude returns the first such triple. In this example we have three possible types for each one of the two arguments and also for the result, namely, the sorts $s1$ and $s2$, and the corresponding kind $[s2]$, and therefore we have $3^3 = 27$ possible triples. Among those, the triple $(s1, s1, s2)$ does *not* satisfy the preregularity checking, because $f(X:s1, X:s1)$ has sort $s2$, $f(X:s1, X:s2)$ has sort $s2$, and $f(X:s2, X:s2)$ has kind $[s2]$, but no sort; thus the flattened term $f(X:s1, X:s1, X:s2)$ could have either sort $s2$, by grouping the arguments as $f(X:s1, f(X:s1, X:s2))$, or kind $[s2]$, by grouping the arguments as $f(f(X:s1, X:s1), X:s2)$. To sum up, the sort structure for the operator f is said to be *non-associative* on the triple $(s1, s1, s2)$.

Two ways of avoiding this undesirable situation are the following: either having a unique declaration at the top sort with both arguments of the same sort,

```
fmod ASSOCIATIVE-EX1 is
  sorts s1 s2 .
  subsort s1 < s2 .
  op f : s2 s2 -> s2 [assoc] .
  op a : -> s1 .
  eq f(a,a) = a .
endfm

Maude> red f(a, a) .
reduce in ASSOCIATIVE-EX1 : f(a, a) .
rewrites: 1 in 0ms cpu (0ms real) (~ rews/sec)
result s1: a
```

or adding enough declarations to cover all possible combinations of arguments; in this case only one more declaration is enough, as follows:

```
fmod ASSOCIATIVE-EX2 is
  sorts s1 s2 .
  subsort s1 < s2 .
  op f : s2 s2 -> s2 [assoc] .
  op f : s1 s2 -> s2 [assoc] .
  op a : -> s1 .
  eq f(a,a) = a .
endfm

Maude> red f(a, a) .
reduce in ASSOCIATIVE-EX2 : f(a, a) .
rewrites: 1 in 0ms cpu (0ms real) (~ rews/sec)
result s1: a
```

When an associative operator is also declared to be commutative using the `comm` attribute, Maude computes the commutative completion (switching the

order of the argument sorts) of the given operator declarations before checking preregularity.

In the case of the `id:` and `idem` attributes, preregularity modulo those axioms requires that all *collapses*, that is, all passages from a term $f(t, e)$ to an equivalent term t by application of an identity equation $f(x, e) = x$; or from a term $f(t, t)$ to an equivalent term t by application of an idempotency equation $f(x, x) = x$ (where in both cases the top function symbol f disappears), should be such that the least sort of the resulting term t is smaller than or equal to the least sort of $f(t, e)$ (resp. $f(t, t)$). A term that can collapse from one sort to a greater or incomparable sort breaks the sort calculations and violates preregularity modulo such axioms. Therefore, syntactic conditions ensuring that a collapse is also into a lesser or equal sort are checked by Maude for both the `id:` and `idem` attributes.

24.2.6 Collapse theories

Using `id:` or `idem` attributes means that you are (conceptually) working with infinite equivalence classes, and that many lefthand side patterns will match in unexpected ways. Unlike OBJ3, Maude has true collapse matching algorithms, rather than identity completion, and it does not try to omit problematic matches. Consider for example the module

```
fmod COLLAPSE-ID-EX is
  sort Foo .
  ops a e : -> Foo .
  op f : Foo Foo -> Foo [left id: e] .
  var X : Foo .
  eq f(X, a) = ...
endfm
```

Then we have

```
a = f(e, a) = f(e, f(e, a)) = ...
```

In particular, the pattern $f(X, a)$ matches a with $X \leftarrow e$, leading to possible nontermination. You should be wary of having an operator with an identity element as the top symbol for a lefthand side. One useful trick when you need a pattern like $f(X, a)$ is to use a pattern $f(Y, a)$ where Y has a sort lower than that of the identity element. For example,

```
fmod COLLAPSE-NAT-EX is
  sorts Nat NzNat .
  subsort NzNat < Nat .
  op 0 : -> Nat .
  op s : Nat -> NzNat .
  op + : Nat Nat -> Nat [assoc comm id: 0] .
  op + : Nat NzNat -> Nat [assoc comm id: 0] .
  var X : Nat .
```

```

var Y : NzNat .
eq +(s(X), Y) = s(+ (X, Y)) .
endfm

```

Here $+(s(X), Y)$ cannot match $s(0)$ because, although $s(0) = +(s(0), 0)$ by the identity attribute, Y cannot match 0 .

Rewriting with the `idem` attribute is even riskier. For example,

```

fmod COLLAPSE-IDEM-EX is
  sort Foo .
  ops a b : -> Foo .
  op f : Foo Foo -> Foo [idem] .
  var X : Foo .
  eq a = b .
endfm

```

We then have

```
a = f(a, a) = f(f(a, a), f(a, a)) = ...
```

Thus, if `a` can be rewritten by an equation, then any number of rewrites can be done by using the `idem` axiom to create new copies of `a`. In fact, the current implementation would choose the obvious rewrite and just produce `b`, but this should not be relied upon; `COLLAPSE-IDEM-EX` is a nonterminating system. The only safe way to use `idem` is as follows. Whenever a connected component is the domain and range of an operator having the `idem` attribute, then its sorts are *poisoned*. Terms of poisoned sorts must never rewrite other than by rules under the control of a strategy, that is, using metalevel descent functions. Such terms must be built out of free constructors—operators that may have equational attributes such as `comm`, but may not have equations with these operators at the top. Of course, it is ok to have defined functions that work on such constructor terms; it is just that the terms themselves may not rewrite.

24.2.7 One-sided identities and associativity

When the associativity axiom is combined with a one-sided identity axiom some unexpected matching properties result. Consider the module:

```

fmod ASSOC-ID-EX is
  sort Foo .
  ops a b 1f : -> Foo .
  op f : Foo Foo -> Foo [assoc left id: 1f] .
  var X Y : Foo .
endfm

```

Then (see Section 25.3 for matching commands),

```
match f(X, Y) <=? f(a, b) .
```

yields three solutions:

```
Solution 1
X:Foo --> 1f
Y:Foo --> f(a, b)

Solution 2
X:Foo --> a
Y:Foo --> b

Solution 3
X:Foo --> f(a, 1f)
Y:Foo --> b
```

whereas the naive user may not have expected the last solution.

Matching with extension can be even more surprising. The command

```
xmatch f(X, Y) <=? f(a, b) .
```

yields five solutions:

```
Solution 1
Matched portion = f(a, 1f)
X:Foo --> a
Y:Foo --> 1f

Solution 2
Matched portion = f(a, 1f)
X:Foo --> f(a, 1f)
Y:Foo --> 1f

Solution 3
Matched portion = (whole)
X:Foo --> 1f
Y:Foo --> f(a, b)

Solution 4
Matched portion = (whole)
X:Foo --> a
Y:Foo --> b

Solution 5
Matched portion = (whole)
X:Foo --> f(a, 1f)
Y:Foo --> b
```

Here the first two solutions match a portion $f(a, 1f)$ of the subject that was not apparent from the original problem. However, if one considers the equivalence class of $f(a, b)$ they are valid solutions that are necessary for correct simulation of (conditional) rewriting on equivalence classes.

24.2.8 Memberships for associative operators

Membership axioms can interact with `assoc` or `iter` operator attributes in undesirable ways.

The reason is that, for completeness, the operator declarations would have to be tried on every subterm of every member of the equivalence class, and this is not done (for efficiency reasons) in the current implementation, giving rise to some warnings.

For associative operators declared at the sort level, membership axioms will be applied only at the top, they will not be applied to subterms in the process of applying an operator declaration to compute the sort. For example in the following module

```
fmod ASSOC-MB-EX1 is
  sort Foo .
  op f : Foo Foo -> Foo [assoc comm] .
  op e : -> [Foo] .
  ops a b c d : -> Foo .

  mb f(a, e) : Foo .
endfm
```

the membership axiom will not be used to lower the sort of `f(a, f(b, e))` to `foo` as it does not match at the top.

Recall from Sections 3.9.3 and 4.8 that terms built with associative operators can be written in flattened form. This is the notation used for `f`-terms in the following examples.

```
Maude> red f(a, b, e) .
Warning: membership axioms are not guaranteed to work correctly for
  associative symbol f as it has declarations that are not at the
  kind level.
reduce in ASSOC-MB-EX1 : f(e, a, b) .
rewrites: 0 in 0ms cpu (0ms real) (~ rews/sec)
result [Foo]: f(e, a, b)

Maude> red f(a, b, e, a) .
reduce in ASSOC-MB-EX1 : f(e, a, a, b) .
rewrites: 0 in 0ms cpu (0ms real) (~ rews/sec)
result [Foo]: f(e, a, a, b)

Maude> red f(e, b, e, a) .
reduce in ASSOC-MB-EX1 : f(e, e, a, b) .
rewrites: 0 in 0ms cpu (0ms real) (~ rews/sec)
result [Foo]: f(e, e, a, b)

Maude> red f(a, b, e, e, a) .
reduce in ASSOC-MB-EX1 : f(e, e, a, a, b) .
rewrites: 0 in 0ms cpu (0ms real) (~ rews/sec)
result [Foo]: f(e, e, a, a, b)
```

```
Maude> red f(a, a, b, e, e, a) .
reduce in ASSOC-MB-EX1 : f(e, e, a, a, a, b) .
rewrites: 0 in 0ms cpu (0ms real) (~ rews/sec)
result [Foo]: f(e, e, a, a, a, b)
```

Here the intuition is that each `e` forces the result to the kind level, unless there is an `a` to bring it back down. Unfortunately, for `f(a, b, e)` we would need to use the membership axiom on a proper subterm, and then use the declaration at the top to arrive at the sort `Foo`, and this is not allowed.

Note that the warning produced by Maude is a per module warning and is only printed once, when the first reduction or rewriting command is given in the module.

The module `ASSOC-MB-EX1` above can be rewritten so that sort computations work as expected as follows:

```
fmod ASSOC-MB-EX2 is
  sort Foo .
  op f : [Foo] [Foo] -> [Foo] [assoc comm] .
  op e : -> [Foo] .
  ops a b c d : -> Foo .

  mb f(X:Foo, Y:Foo) : Foo .
  mb f(a, e) : Foo .
endfm

Maude> red f(a, b, e) .
reduce in ASSOC-MB-EX2 : f(e, a, b) .
rewrites: 2 in 0ms cpu (1ms real) (~ rews/sec)
result Foo: f(e, a, b)

Maude> red f(a, b, e, a) .
reduce in ASSOC-MB-EX2 : f(e, a, a, b) .
rewrites: 3 in 0ms cpu (0ms real) (~ rews/sec)
result Foo: f(e, a, a, b)

Maude> red f(e, b, e, a) .
reduce in ASSOC-MB-EX2 : f(e, e, a, b) .
rewrites: 6 in 0ms cpu (0ms real) (~ rews/sec)
result [Foo]: f(e, e, a, b)

Maude> red f(a, b, e, e, a) .
reduce in ASSOC-MB-EX2 : f(e, e, a, a, b) .
rewrites: 11 in 0ms cpu (0ms real) (~ rews/sec)
result Foo: f(e, e, a, a, b)

Maude> red f(a, a, b, e, e, a) .
reduce in ASSOC-MB-EX2 : f(e, e, a, a, a, b) .
rewrites: 12 in 0ms cpu (0ms real) (~ rews/sec)
result Foo: f(e, e, a, a, a, b)
```

Here the operator declaration is at the *kind level*, and the effect of the declaration of `f` in `ASSOC-MB-EX1` is obtained by an extra membership axiom.²

Let us see another example of this situation, starting with a module specifying non-empty lists of natural numbers.

```
fmod SIMPLE-NAT-LIST is
  protecting NAT .
  sort NatList .
  subsort Nat < NatList .
  op __ : NatList NatList -> NatList [assoc] .
endfm
```

It seems natural to specify *sorted* lists of natural numbers by importing `SIMPLE-NAT-LIST` and then defining a subsort of `NatList`.

```
fmod NAIVE-SORTED-NAT-LIST is
  protecting SIMPLE-NAT-LIST .
  sort SortedNatList .
  subsort Nat < SortedNatList < NatList .

  vars I J : Nat .
  var SNL : SortedNatList .
  cmb I J : SortedNatList if I <= J .
  cmb I J SNL : SortedNatList if I <= J /\ J SNL : SortedNatList .
endfm
```

```
Maude> red 0 1 2 3 4 5 6 7 8 9 .
Warning: membership axioms are not guaranteed to work correctly for
  associative symbol __ as it has declarations that are not at the
  kind level.
reduce in NAIVE-SORTED-NAT-LIST : 0 1 2 3 4 5 6 7 8 9 .
rewrites: 1354 in 0ms cpu (0ms real) (~ rews/sec)
result SortedNatList: 0 1 2 3 4 5 6 7 8 9
```

To avoid this, we can rewrite the module above so that we only use *kind-level operator declarations* (notice the form of the arrow) and convert all sort-level operator declarations into memberships.

```
fmod NAT-LIST-KIND is
  protecting NAT .
  sort NatList .
  subsort Nat < NatList .

  op __ : NatList NatList ~> NatList [assoc] .
  mb I:NatList J:NatList : NatList .
endfm
```

² Maude 1 did not allow multiple membership axioms on associative operators. In Maude 2 this works, although it will be extremely inefficient for large terms, since matching the extra membership essentially amounts to expanding out the equivalence class.

```

fmod SORTED-NAT-LIST-KIND is
  protecting NAT-LIST-KIND .
  sort SortedNatList .
  subsort Nat < SortedNatList < NatList .

  vars I J : Nat .
  var SNL : SortedNatList .
  cmb I J : SortedNatList if I <= J .
  cmb I J SNL : SortedNatList if I <= J /\ J SNL : SortedNatList .
endfm

Maude> red 0 1 2 3 4 5 6 7 8 9 .
reduce in SORTED-NAT-LIST-KIND : 0 1 2 3 4 5 6 7 8 9 .
rewrites: 1354 in 0ms cpu (0ms real) (~ rews/sec)
result SortedNatList: 0 1 2 3 4 5 6 7 8 9

```

24.2.9 Memberships for iterated operators

In analogy to interaction of associative operators and membership declarations, terms constructed with a stack of iterated operators may not be assigned the expected sort when it is necessary to apply a membership axiom to a subterm in order to infer the sort. Again, if an `iter` operator is declared at the sort level, Maude will not apply membership axioms to subterms in order to calculate the sort of a subterm before attempting to apply the operator declaration to calculate the sort of the whole term. As an example, consider the following module:

```

fmod ITER-MB-EX1 is
  sort Foo .
  op f : Foo -> Foo [iter] .
  op e : -> [Foo] .

  mb f(e) : Foo .
endfm

Maude> red f(e) .
Warning: membership axioms are not guaranteed to work correctly for
  iterated symbol f as it has declarations that are not at the
  kind level.
reduce in ITER-MB-EX1 : f(e) .
rewrites: 1 in 0ms cpu (0ms real) (~ rews/sec)
result Foo: f(e)

Maude> red f(f(e)) .
reduce in ITER-MB-EX1 : f^2(e) .
rewrites: 0 in 0ms cpu (0ms real) (~ rews/sec)
result [Foo]: f^2(e)

```

```
Maude> red f(f(f(e))) .
reduce in ITER-MB-EX1 : f^3(e) .
rewrites: 0 in 0ms cpu (0ms real) (~ rews/sec)
result [Foo]: f^3(e)
```

Here the intuition is that e is at the kind level, but $f(e)$ is not. Unfortunately, for $f(f(e))$ we would need to use the membership axiom on a proper subterm and then use the declaration at the top to arrive at the sort `Foo`, and declarations applying above membership axioms for iterated operators are not allowed.

Again, recall that the warning that membership axioms may not work is only given once per module. Here it just happens that it is given in response to a reduction command that does give the right answer.

The example can be rewritten so that membership axioms can be used to compute the desired sort as follows:

```
fmod ITER-MB-EX2 is
  sort Foo .
  op f : [Foo] -> [Foo] [iter] .
  op e : -> [Foo] .

  mb f(X:Foo) : Foo .
  mb f(e) : Foo .
endfm

Maude> red f(e) .
reduce in ITER-MB-EX2 : f(e) .
rewrites: 1 in 0ms cpu (0ms real) (~ rews/sec)
result Foo: f(e)

Maude> red f(f(e)) .
reduce in ITER-MB-EX2 : f^2(e) .
rewrites: 2 in 0ms cpu (0ms real) (~ rews/sec)
result Foo: f^2(e)

Maude> red f(f(f(e))) .
reduce in ITER-MB-EX2 : f^3(e) .
rewrites: 3 in 0ms cpu (0ms real) (~ rews/sec)
result Foo: f^3(e)
```

Here the operator declaration is at the *kind level*, and as in the associativity example in the previous section, the effect of the old declaration is obtained by an extra membership axiom. Note that using membership axioms in this way loses the efficiency for big towers of operators, which is the whole point of the `iter` theory.

24.2.10 Ambiguity in print attribute items

Since Maude has few restrictions on variable names, it is possible to introduce ambiguity with the `print` attribute by using strings or attribute names

as variables. Consider, for example, the following module, where the string "here" and the keyword `metadata` are also declared as variables.

```
fmod PRINT-ATTR-AMBIGUOUS is
  sort Foo .
  op a : -> Foo .
  ops f g h : Foo -> Foo .
  vars metadata "here" : Foo .
  eq f("here") = g("here") [print "here"] .
  eq g(metadata) = h(metadata)
    [print "metadata = " metadata "g->h equation"] .
endfm
```

In the `print` attribute of the first equation, Maude cannot decide whether "here" is a string constant or a variable. Similarly, in the `print` attribute of the second equation, Maude will not be able to decide whether `metadata` is a variable or a keyword. Under these circumstances, Maude will output warnings about the multiple parses and then make an undefined choice between them.

```
Warning: <standard input>, line 6 (fmod PRINT-ATTR-AMBIGUOUS):
  multiple distinct parses for statement
  eq f ("here") = g ("here") [print "here"] .
Warning: <standard input>, line 7 (fmod PRINT-ATTR-AMBIGUOUS):
  multiple distinct parses for statement
  eq g (metadata) = h (metadata)
    [print "metadata = " metadata "g->h equation"] .
```

In this particular example, the equations work but, as a consequence of the ambiguity, the user does not get the expected information provided by the `print` attribute.

```
Maude> set print attribute on .
Maude> red f(g(a)) .
reduce in PRINT-ATTR-AMBIGUOUS : f(g(a)) .
metadata =
h(a)
metadata =
rewrites: 3 in 0ms cpu (0ms real) (25210 rewrites/second)
result Foo: h(h(a))
```

Complete List of Maude Commands

In this chapter we use curly bracket pairs, ‘{’ and ‘}’, to enclose optional syntax.

25.1 Command line flags

The following command line flags are supported.

--help

Displays information on the usage of the Maude command and its line flags.

--version

Displays the Maude version number.

-no-mixfix

Turns off mixfix printing; useful if Maude is being run by some other program that does not want to deal with the intricacies of mixfix parsing.

-ansi-color, -no-ansi-color

By default ANSI escape codes for color and other effects are disabled if the standard output is not a terminal or the `TERM` environment variable is set to `dumb`. These flags allow the default behavior to be overridden.

-tecla, -no-tecla

By default Tecla-based command line editing is disabled if the standard output is not a terminal or the `TERM` environment variable is set to `dumb` or `emacs`. These flags allow the default behavior to be overridden.

-no-prelude

Causes Maude not to read in the standard prelude.

-batch

Disables control-C handling.

-interactive

Pretends to be interactive, and enables control-C handling even though standard output is not a terminal.

-xml-log=*file-name*

Generates an XML log for selected commands in the given file.

-no-banner

Causes Maude not to show the welcome banner at start-up.

-random-seed=*number*

Specifies the natural number *number* in the range $[0, 2^{32} - 1]$ as the seed for the pseudo-random number generator **random** in module **RANDOM** (see Section 9.3). The default seed is 0.

-no-advise

Switches off advisories at start up.

-no-wrap

Disables the automatic line wrapping of output.

25.2 Rewriting commands

reduce {**in** *module* :} *term* .

Causes the specified term to be reduced using the equations and membership axioms in the given module. **reduce** may be abbreviated to **red**. If the **in** clause is omitted, the current module is assumed. For examples, see Section 4.9.

rewrite {[*bound*]} {**in** *module* :} *term* .

Causes the specified term to be rewritten using the rules, equations, and membership axioms in the given module. The default interpreter for rules applies them using a rule-fair top-down (lazy) strategy and stops when the number of rule applications reaches the given bound. No rule will be applied if an equation can be applied. If the **in** clause is omitted, the current module is assumed. If the upper bound clause is omitted, infinity is assumed. **rewrite** may be abbreviated to **rew**. For examples, see Section 6.4.1.

frewrite {[*bound* {,*number*}]} {**in** *module* :} *term* .

Like the previous command, causes the specified term to be rewritten using the rules, equations, and membership axioms in the given module. But now the default interpreter for rules applies them using a rule- and position-fair strategy and stops when the number of rule applications reaches the given bound. This strategy causes multiple passes over the term, with at most *number* rule rewrites taking place at each position. If the **in** clause is omitted, the current module is assumed. If the upper

bound clause is omitted, infinity is assumed. If the number of rewrites per position is omitted, 1 is assumed. **frewrite** may be abbreviated to **frew**. For examples, see Section 6.4.2.

Unlike **rewrite**, which uses a leftmost outermost strategy for applying rules and reduces the whole term with equations after each successful rule rewrite, **frewrite** attempts to be position fair by making a number of depth-first traversals of the term. On each traversal, each position that existed at the start of the traversal is entitled to at most *number* rule rewrites when its turn comes around. After a rule rewrite succeeds, only the subterm that was rewritten is reduced with equations in order to avoid destroying positions that have not yet had their turn for the current traversal. Traversals are made until *bound* rule rewrites have been made, or until no more rewrites are possible. When operators have the **assoc** or **iter** attributes, term depth and positions are relative to the flattened or compact form of the term, respectively. Thus, fair rewriting treats a whole stack of an **iter** operator as a single position for the purposes of position fairness.

There are a couple of caveats with **frewrite**:

- If position-fair rewriting stops in mid traversal, then the sort of the (incompletely reduced) result has not yet been calculated and is printed as **(sort not calculated)**.
- Position-fair rewriting is not substitution fair; this is particularly apparent if you have a multiset of messages and objects, as in Section 11.2.

erewrite **{[bound {,number}]}** **{in module :}** *term* .

Works like the **frewrite** command and in addition it allows messages to be exchanged with external objects that do not reside in the configuration. It is abbreviated to **erew**.

continue **{number}** .

Attempts to continue rewriting the result of the last rewriting command using the rules, equations, and membership axioms, stopping if the upper bound on the number of rule applications is reached. This command is only usable if the current module has not changed since the last rewriting command, and the last rewriting command was not **reduce**. If no upper bound clause is given, infinity is assumed. In the case where the last rewriting command was **frewrite**, the number given to the **continue** command increases the bound on the number of traversals, leaving the number of rewrites per position unchanged. In particular, considerable extra information about the current traversal is saved by the **frewrite** command so that, for example,

```
frewrite [n, k] t .
continue m .
```

produces the same final answer as

`frewrite [s, k] t .`

when $s = n + m$. For an `erewrite` command, the same state information is preserved as for `frewrite`, but in this case nothing can be guaranteed, due to the interaction with the external environment.

loop {in *module* :} *term* .

This command is used to initialize the read-eval-print loop in a module importing LOOP-MODE (see Section 18.1). The specified term is rewritten as far as possible using the rules, equations, and membership axioms in the given module. If the result has a loop constructor symbol at the top, then it becomes the current state of the loop; also, the list of quoted identifiers in the output position of the loop constructor is printed as a sequence of identifiers.

(*identifier**)

This command is used to input a list of identifiers to the loop in a module importing LOOP-MODE (see Section 18.1). If the current module has not changed since the last rewriting command, the result of previous rewrites has a loop constructor symbol at the top, and the last rewriting command was not **reduce** then:

1. the sequence of identifiers in the parentheses is converted into a list of quoted identifiers and is placed under the input position of the loop constructor;
2. a nil list of quoted identifiers is placed under the output position of the loop constructor;
3. the new term is rewritten as far as possible using the rules, equations, and membership axioms in the module to which the term belongs; and
4. if the new result has a loop constructor symbol at the top, the list of quoted identifiers in the output position of the loop constructor is printed as a sequence of identifiers.

set clear rules on . / set clear rules off .

Normally, each `rewrite` or `frewrite` command and each loop mode invocation resets the rule state for each symbol. For most symbols the rule state consists of the next rule to be executed in a round-robin scheme but for counter symbols the rule state consists of the next number to rewrite to. Setting *clear rules* to off means the rule state will *not* be reset between commands.

25.3 Matching commands

Matching commands are used to directly invoke the rewriting engine's term pattern matcher. They can be useful for figuring out exactly what subjects can be matched by a complex pattern.

match `{[number]}` `{in module :}` *pattern* `<=?` *subject-term* `{such that condition}` .

Performs straightforward matching in the given module. This kind of matching is used by the engine for applying membership axioms. The result is a list of at most *number* matching substitutions such that the subject term matches the pattern and the substitution satisfies the optional *condition* (whose syntactic form is the same as the one of conditions for conditional equations and memberships; see Section 4.3). If the upper bound clause is omitted, infinity is assumed. For examples, see Section 4.9.

xmatch `{[number]}` `{in module :}` *pattern* `<=?` *subject-term* `{such that condition}` .

Works similarly to the previous command, except that it performs matching with extension for those theories that need it (those including the `assoc` or `iter` attributes). If the subject term (after theory normalization) has a symbol *f* from an extension theory on top, only a piece of the top theory layer with *f* on top need be matched. This kind of matching is used by the engine for applying equations and rules in order to accurately simulate equivalence class rewriting. The result is a list of all matches satisfying the given condition. If only part of the subject was matched, that part is given. For examples, see Sections 4.8 and 4.9.

25.4 Searching commands

search `{[bound {,depth}]}` `{in module :}` *subject* *searchtype* *pattern* `{such that condition}` .

Performs a breadth-first search for rewrite proofs starting at *subject* to a final state that matches *pattern* and satisfies an optional *condition* (whose syntactic form is the same as the one of conditions for conditional equations and memberships; see Section 4.3). Possible values for *searchtype* are

- `=>1` one step proof
- `=>+` one or more steps proof
- `=>*` zero or more steps proof
- `=>!` only canonical final states, that cannot be further rewritten, are allowed as solutions

The optional *bound* argument provides an upper bound in the number of solutions to be found; if it is omitted, infinity is assumed.

The optional *depth* argument indicates the maximum depth of the search. If it is omitted, infinity is assumed. It is also possible to give a depth bound without giving a bound on the number of solutions returned by requesting a search of the form `search [,m] . . .`

The search type `=>1` is an abbreviation of the search type `=>+` with the depth bound set to 1.

As usual, if the **in** clause is omitted, the current module is assumed.
For examples, see Section 6.4.3.

show search graph .

Displays the search graph generated by the last search.

show path *number* .

Displays the path to a given state, identified by the number, in a search graph.

show path labels *number* .

Works like the command above, but only shows labels of applied rules instead of the full path.

25.5 Unification and variants commands

unify {[*bound*]} {**in** *module* :} *term*₁ =? *term*'₁ { /\ ... /\ *term*_{*k*} =? *term*'_{*k*} } .

Computes a complete set of order-sorted unifiers modulo the (supported) equational axioms in the given module for the provided unification problem. If the cardinality of the set of unifiers is greater than the specified *bound*, the unifiers beyond that bound are omitted. The *module* can be any module or theory declared in the current Maude session; as usual, if the **in** clause is omitted, the current module is used.

For examples, see Section 15.4.

variant unify {[*bound*]} {**in** *module* :} *term*₁ =? *term*'₁ { /\ ... /\ *term*_{*k*} =? *term*'_{*k*} } .

Computes a complete set of order-sorted unifiers modulo the equations declared with the **variant** attribute (which must satisfy the finite variant property) plus the (supported) equational axioms in the given module for the provided unification problem. If the cardinality of the set of unifiers is greater than the specified *bound*, the unifiers beyond that bound are omitted. The *module* can be any module or theory declared in the current Maude session; as usual, if the **in** clause is omitted, the current module is used.

For more details, see Section 15.10.1.

get variants {[*bound*]} {**in** *module* :} *term* .

Computes a complete set of variants of the given term in the (supported) equational theory of the given module, where the equations of interest must be declared with the **variant** attribute. If the cardinality of the set of variants is greater than the specified *bound*, the variants beyond that bound are omitted. The *module* can be any module or theory declared in the current Maude session; as usual, if the **in** clause is omitted, the current module is used.

For examples and more details, see Section 15.9.2.

25.6 Tracing commands

Tracing produces detailed information about each rewrite performed and each conditional rewrite attempted. Since this typically results in an unmanageably huge volume of output, there are commands to control what is actually displayed.

set trace on . / set trace off .

These commands turn tracing on and off. If tracing is turned on, all trace information will be generated internally, even if none of it is displayed, thus considerably slowing the speed of interpretation.

set trace condition on . / set trace condition off .

Determines whether the evaluations of conditions are traced.

set trace whole on . / set trace whole off .

Determines whether the whole term is printed before and after a rewrite.

set trace substitution on . / set trace substitution off .

Determines whether the substitution is printed.

set trace mb on . / set trace mb off .

Determines whether membership axiom applications are printed.

set trace eq on . / set trace eq off .

Determines whether equation applications are printed.

set trace rl on . / set trace rl off .

Determines whether rule applications are printed.

set trace select on . / set trace select off .

Determines whether only trace information for selected operator symbols is printed (rather than all symbols).

trace select *symbols* . / trace deselect *symbols* .

Selects/deselects operator symbols and labels from the current module for tracing with the select option. Examples:

```
trace select foo bar baz .
trace deselect baz .
```

trace exclude *modules* . / trace include *modules* .

Controls which modules are traced. Examples:

```
trace exclude META-LEVEL .
trace include MY-MOD1 MY-MOD2 .
```

set trace rewrite on . / set trace rewrite off .

Determines whether the redex and its replacement are printed.

set trace body on . / set trace body off .

Determines whether the “start of rewrite” line (i.e., the one beginning with *’s) and the body of the equation/rule/membership being used are printed; if turned off, just the label and the substitution are printed. By setting both body and rewrite to **off** (see previous command), these options reduce a trace to a list of labels much like that produced by the `show path labels number` command.

set trace builtin on . / set trace builtin off .

Determines whether trace information for built-in operator symbols is printed.

25.7 Print attribute commands

In print attribute mode, when a statement is executed, the items in its `print` attribute are printed, with variables taking their value in the current substitution.

set print attribute on . / set print attribute off .

These commands turn print attribute mode on and off. It is **off** by default.

set print attribute newline on . / set print attribute newline off .

These commands determine whether a newline is printed following the items of a print attribute. By default a newline is printed (even if there are no items).

Note that print attribute mode is like trace mode, break mode, and profile mode in that in this mode all execution takes the slow path. This is true even if no print attributes are encountered.

25.8 Print option commands

set print mixfix on . / set print mixfix off .

Controls whether operators with mixfix syntax are printed in either mixfix or prefix form. User-defined syntax is supported for pretty-printing, even though it is not currently supported for parsing. It is sometimes advantageous to have uniform prefix notation for output; for example, if the output is going to be postprocessed by some other tool. Default is **on**.

set print graph on . / set print graph off .

If **on**, terms that are internally represented by graphs (currently, result terms together with terms being reduced and terms in substitutions during tracing) are printed as graph representations rather than as terms,

together with the number of operator symbols in the full term. This can be useful in some pathological cases where the size of the term is exponential on the size of the graph. Default is **off**.

set print flattened on . / set print flattened off .

Controls whether arguments under operators with the associative attribute are printed in flattened form or not. Default is **on**.

set print with parentheses on . / set print with parentheses off .

If on, mixfix terms are printed with additional parentheses to make grouping explicit. Default is **off**.

set print with aliases on . / set print with aliases off .

Controls if variables aliases are used. Default is **on**.

set print number on . / set print number off .

Controls if special output convention for natural numbers is used. Default is **on**.

set print rational on . / set print rational off .

Controls if special output convention for rational numbers is used. Default is **on**.

set print color on . / set print color off .

Controls if reduction status coloring is used. Default is **off**.

set print format on . / set print format off .

Controls if format attributes are obeyed. Default is **on**.

set print conceal on . / set print conceal off .

Controls if argument hiding is used. Default is **off**.

print conceal *symbols* . / print reveal *symbols* .

Controls which operators have their arguments hidden.

25.9 Show option commands

set show stats on . / set show stats off .

Determines whether the number of rewrites is printed with the results of the **reduce**, **rewrite**, and **continue** commands in Section 25.2. Default is **on**.

set show loop stats on . / set show loop stats off .

As above but for loop mode.

set show timing on . / set show timing off .

Determines whether the cpu and real time used during rewriting is printed with the results of the **reduce**, **rewrite**, and **continue** commands in Section 25.2. Default is **on**.

set show loop timing on . / set show loop timing off .

As above but for loop mode.

set show command on . / set show command off .

Determines whether the full form of certain commands is printed before they are executed. Default is **on**.

set show breakdown on . / set show breakdown off .

Determines whether a breakdown of rewrites is displayed. Default is **off**.

set show gc on . / set show gc off .

Determines which message is printed when a garbage collect is performed. Default is **off**.

set show advisories on . / set show advisories off .

Determines whether advisories are displayed. Default is **on**.

25.10 Show commands

show modules .

Lists the names of all the modules that are currently in the module database maintained by the system.

show module {*module*} .

Prints out a representation of the given module (or of the current module if none is given).

show all {*module*} .

Prints out a *flattened* representation of the given module (or of the current module if none is given).

show sorts {*module*} .

Prints out a representation of the sort and subsort information for the given module (or for the current module if none is given).

show ops {*module*} .

Lists the operators in the given module (or in the current module if none is given).

show vars {*module*} .

Lists the variables in the given module (or in the current module if none is given).

show mbs {*module*} .

Lists the membership axioms in the given module (or in the current module if none is given).

show eqs {*module*} .

Lists the equations in the given module (or in the current module if none is given).

show rls {*module*} .

Lists the rules in the given module (or in the current module if none is given).

show components {*module*} .

Lists the connected components (kinds) of the poset of sorts for the given module (or for the current module if none is given).

show summary {*module*} .

Shows a summary of statistics for the context free grammar and term rewriting system generated for the given module (or for the current module if none is given).

show views .

Lists the names of all the views that are currently in the view database maintained by the system.

show view {*view*} .

Prints out the given view (or of the last view entered into the system if none is given).

25.11 Profiler commands

set profile on . / **set profile off** .

Turns profiling on and off. Default is **off**.

set clear profile on . / **set clear profile off** .

Controls whether profile is clear before each command. Default is **on**.

show profile {*module*} .

Shows current profile for the given module (or in the current module if none is given). It shows both percentages and absolute rewrite counts.

25.12 Debugger commands

set break on . / **set break off** .

Controls whether break points are obeyed.

break select *symbols* . / **break deselect** *symbols* .

Selects/deselects operator symbols and labels from the current module for break points with the select option. Examples:

```
break select foo bar baz .
break deselect baz .
```

debug reduce {*in module :*} *term* .

Works just like the **reduce** command in Section 25.2, except that it drops into the debugger before executing the first rewrite.

debug rewrite {[*number*]} {*in module :*} *term* .

Works just like the **rewrite** command in Section 25.2, except that it drops into the debugger before executing the first rewrite.

debug frewrite {[*bound* {,*number*}]} {*in module :*} *term* .

Works just like the **frewrite** command in Section 25.2, except that it drops into the debugger before executing the first rewrite.

debug erewrite {[*bound* {,*number*}]} {*in module :*} *term* .

Works just like the **erewrite** command in Section 25.2, except that it drops into the debugger before executing the first rewrite.

resume .

Only usable from the debugger. Exits the debugger and resumes the current rewriting activity.

abort .

Only usable from the debugger. Exits the debugger and abandons the current rewriting activity.

step .

Only usable from the debugger. Performs a single step of the current rewriting activity with tracing switched on.

where .

Only usable from the debugger. Prints the stack of pending rewrite tasks together with explanations of how they arose.

25.13 Miscellaneous commands

parse {*in module :*} *term* .

Causes the specified term to be parsed using the signature of the given module. If the **in** clause is omitted, the current module is assumed.

select module .

Selects a named module to be the current module. All commands that require a module refer to the current module, unless a module is explicitly given. The current module is usually the last module entered or used; for example, after the command **show module** *AMODULE*, the *AMODULE* module becomes the current module.

set protect module on . / set protect module off .

Adds or removes the named module from the set of modules that are automatically imported in **protecting** mode in every module.

set extend *module* on . / set extend *module* off .

Adds or removes the named module from the set of modules that are automatically imported in **extending** mode in every module.

set include *module* on . / set include *module* off .

Adds or removes the named module from the set of modules that are automatically imported in **including** mode in every module.

set verbose on . / set verbose off .

Controls display of extra information, depending on command. Default is **off**.

set clear memo on . / set clear memo off .

Controls whether the memoization tables are cleared before each command.

25.14 System level commands

These commands control system level activities. Unlike all the above commands they are not followed by a period.

pwd

Prints the path of the working directory.

ls {*flags*} {*directories*}

Runs the UNIX **ls** command to list the files in the specified directories or working directory if none specified. The allowable flags depend on your local implementation of **ls**. Example:

```
ls -lF /usr/bin/usr/local
```

cd *directory-name*

Changes the working directory to *directory-name*.

pushd *directory-name*

Saves the current working directory on a stack and then changes the working directory to *directory-name*.

popd

Changes the working directory to that which is on the top of the directory stack and pops the directory stack.

in *file-name*

Causes a specified file to be included at this point. For files specified by a bare file name, it checks (with **.maude**, **.fm**, **.obj** extensions) if the filename is in one of these locations: (a) the current directory; (b) the directories in the **MAUDE_LIB** environment variable, and (c) the directory containing the executable. Otherwise, the full file name must be given,

together with a full path name if the file is not in the current working directory. The `in` command may be nested, i.e., the included file may contain `in` commands. Example:

```
in ../Examples/foo.maude
```

Notice that compilation of operator declarations and statements is done lazily, so that the module is not necessarily fully compiled when included. This implies that some warnings and advisories will only show up when a reduction actually takes place in the module. This also holds for a module that is entered by writing it in the prompt instead of a file.

load *file-name*

Performs the same job as `in` but does not produce detailed output as modules are entered. Example:

```
load ../Examples/foo.maude
```

eof

Causes the interpreter to respond as if it had reached the end of file.

quit

Causes the interpreter to exit.

Core Maude Grammar

This chapter describes the syntax of Maude using the following extended BNF notation: the symbols ‘(’ and ‘)’ are used as metaparentheses; the symbol ‘|’ is used to separate alternatives; square bracket pairs, ‘[’ and ‘]’, enclose optional syntax; ‘*’ indicates zero or more repetitions of preceding unit; ‘+’ indicates one or more repetitions of preceding unit; and the string “x” denotes x literally. As an application of this notation, A(, A)* indicates a non-empty list of A’s separated by commas. Finally, %% indicates comments in the syntactic description, as opposed to comments in the Maude code.

26.1 The grammar

```

<MaudeTop> ::=
  ( <SystemCommand> | <Command> | <DebuggerCommand> |
    <Module> | <Theory> | <View> )+

<SystemCommand> ::= in <FileName> | load <FileName> |
  quit | eof | popd | pwd |
  cd <Directory> | push <Directory> |
  ls [ <LsFlag> ] [ <Directory> ]

<Command> ::= select <ModId> . |
  parse [ in <ModId> : ] <Term> . |
  [ debug ] reduce [ in <ModId> : ] <Term> . |
  [ debug ] rewrite [ [ <Nat> ] ] [ in <ModId> : ] <Term> . |
  [ debug ] frewrite [ [ <Nat> [ , <Nat> ] ] ] [ in <ModId> :
    ] <Term> . |
  [ debug ] erewrite [ [ <Nat> [ , <Nat> ] ] ] [ in <ModId> : ]
    <Term> . |
  ( match | xmatch ) [ [ <Nat> ] ] [ in <ModId> : ]
    <Term> <=? <Term> [ such that <Condition> ] . |

```

```

unify [ [ <Nat> ] ] [ in <ModId> : ]
  <UnificationEquation> ( /\ <UnificationEquation> )* . |
[ debug ] variant unify [ [ <Nat> ] ] [ in <ModId> : ]
  <UnificationEquation> ( /\ <UnificationEquation> )* . |
[ debug ] get variants [ [ <Nat> ] ] [ in <ModId> : ] <Term> . |
search [ [ <Nat> ] ] [ in <ModId> : ]
  <Term> <SearchType> <Term> [ such that <Condition> ] . |
[ debug ] continue <Nat> . |
loop [ in <ModId> : ] <Term> . |
( <TokenString> ) |
trace ( select | deselect | include | exclude )
  ( <OpId> | ( <OpForm> ) )+ . |
print ( conceal | reveal ) ( <OpId> | ( <OpForm> ) )+ . |
break ( select | deselect ) ( <OpId> | ( <OpForm> ) )+ . |
show <ShowItem> [ <ModId> ] . |
show view [ <ViewId> ] . |
show modules . |
show views . |
show search graph . |
show path [ labels ] <Nat> .
do clear memo . |
set <SetOption> ( on | off ) .

<UnificationEquation> ::= <Term> =? <Term>

<ShowItem> ::= module | all | sorts | ops | vars | mbs |
  eqs | rls | summary | kinds | profile

<SetOption> ::= show <ShowOption> |
  print <PrintOption> |
  trace [ <TraceOption> ] |
  break | verbose | profile |
  clear ( memo | rules | profile ) |
  protect <ModId> |
  extend <ModId> |
  include <ModId>

<ShowOption> ::= advise | stats | loop stats | timing |
  loop timing | breakdown | command | gc

<PrintOption> ::= mixfix | flat | with parentheses |
  with aliases | conceal | number | rat | color |
  format | graph | attribute | attribute newline

<TraceOption> ::= condition | whole | substitution | select |

```

```

mbs | eqs | rls | rewrite | body

⟨DebuggerCommand⟩ ::= resume . | abort . | step . | where .

⟨Module⟩ ::= fmod ⟨ModId⟩ [ ⟨ParameterList⟩ ] is ⟨ModElt⟩* endfm |
           mod ⟨ModId⟩ [ ⟨ParameterList⟩ ] is ⟨ModElt'⟩* endfm

⟨Theory⟩ ::= fth ⟨ModId⟩ is ⟨ModElt⟩* endfth |
           th ⟨ModId⟩ is ⟨ModElt'⟩* endth

⟨View⟩ ::= view ⟨ViewId⟩ from ⟨ModExp⟩ to ⟨ModExp⟩ is
        ⟨ViewElt⟩*
        endv

⟨ParameterList⟩ ::= { ⟨ParameterDecl⟩ ( , ⟨ParameterDecl⟩ )* }

⟨ParameterDecl⟩ ::= ⟨ParameterId⟩ :: ⟨ModExp⟩

⟨ModElt⟩ ::= including ⟨ModExp⟩ . |
           extending ⟨ModExp⟩ . |
           protecting ⟨ModExp⟩ . |
           sorts ⟨Sort⟩+ . |
           subsorts ⟨Sort⟩+ ( < ⟨Sort⟩+ )+ . |
           op ⟨OpForm⟩ : ⟨Type⟩* ⟨Arrow⟩ ⟨Type⟩ [ ⟨Attr⟩ ] . |
           ops ( ⟨OpId⟩ | ( ⟨OpForm⟩ ) )+ : ⟨Type⟩* ⟨Arrow⟩ ⟨Type⟩
           [ ⟨Attr⟩ ] . |
           vars ⟨VarId⟩+ : ⟨Type⟩ . |
           ⟨Statement⟩ [ ⟨StatementAttr⟩ ] .

⟨ViewElt⟩ ::= var ⟨varId⟩+ : ⟨Type⟩ . |
           sort ⟨Sort⟩ to ⟨Sort⟩ . |
           label ⟨LabelId⟩ to ⟨LabelId⟩ . |
           op ⟨OpForm⟩ to ⟨OpForm⟩ . |
           op ⟨OpForm⟩ : ⟨Type⟩* ⟨Arrow⟩ ⟨Type⟩ to ⟨OpForm⟩ . |
           op ⟨Term⟩ to ⟨Term⟩ .

⟨ModExp⟩ ::= ⟨ModId⟩ |
           ( ⟨ModExp⟩ ) |
           ⟨ModExp⟩ + ⟨ModExp⟩ |
           ⟨ModExp⟩ * ⟨Renaming⟩
           ⟨ModExp⟩ { ⟨ViewId⟩ ( , ⟨ViewId⟩ )* }

⟨Renaming⟩ ::= ( ⟨RenamingItem⟩ ( , ⟨RenamingItem⟩ )* )

⟨RenamingItem⟩ ::= sort ⟨Sort⟩ to ⟨Sort⟩ |

```

```

label <LabelId> to <LabelId> |
op <OpForm> <ToPartRenamingItem> |
op <OpForm> : <Type>* <Arrow> <Type> <ToPartRenamingItem>

<ToPartRenamingItem> ::= to <OpForm> [ <Attr> ]

<Arrow> ::= -> | ~>

<Type> ::= <Sort> | <Kind>

<Kind> ::= [ <Sort> ( , <Sort> )* ]

<Sort> ::= <SortId> | <Sort> { <Sort> ( , <Sort> )* }

<ModElt'> ::= <ModElt> |
  <Statement'> [ <StatementAttr> ] .

<Statement> ::= mb [ <Label> ] <Term> : <Sort> |
  cmb [ <Label> ] <Term> : <Sort> if <Condition> |
  eq [ <Label> ] <Term> = <Term> |
  ceq [ <Label> ] <Term> = <Term> if <Condition>

<Statement'> ::= rl [ <Label> ] <Term> => <Term> |
  crl [ <Label> ] <Term> => <Term> if <Condition'>

<Label> ::= [ <LabelId> ] :

<Condition> ::= <ConditionFragment> ( /\ <ConditionFragment> )*

<Condition'> ::= <ConditionFragment'>
  ( /\ <ConditionFragment'> )*

<ConditionFragment> ::= <Term> = <Term> | <Term> := <Term>
  | <Term> : <Sort>

<ConditionFragment'> ::= <ConditionFragment> | <Term> => <Term>

<Attr> ::=
  [ ( assoc | comm |
    [ left | right ] id: <Term> |
    idem | iter | memo | ditto |
    config | obj | msg |
    metadata <StringId>
    strat ( <Nat>+ ) |
    poly ( <Nat>+ ) |

```

```

frozen [ ( <Nat>+ ) ] |
prec <Nat> |
gather ( ( e | E | & )+ ) |
format ( <Token>+ ) |
special ( <Hook>+ ) )+ ]

<StatementAttr> ::=
  [ ( nonexec | otherwise | variant |
    metadata <StringId> |
    label <LabelId> |
    print <PrintItem>* )+ ]

<PrintItem> ::= <StringId> | <VarId> | <VarAndSortId>

<Hook> ::= id-hook <Token> [ ( <TokenString> ) ] |
  ( op-hook | term-hook ) ( <TokenString> )

<FileName>      %%% OS dependent
<Directory>    %%% OS dependent
<LsFlag>       %%% OS dependent

<StringId>     %%% characters enclosed in double quotes "...
<ModId>       %%% simple identifier, by convention all capitals
<ViewId>      %%% simple identifier, by convention capitalized
<ParameterId> %%% simple identifier, by convention single capital
<SortId>      %%% simple identifier, by convention capitalized
<VarId>       %%% simple identifier, by convention capitalized
<VarAndSortId> %%% an identifier consisting of a variable name
                followed by a colon followed by a sort name
<OpId>        %%% identifier possibly with underscores
<OpForm> ::= <OpId> | ( <OpForm> ) | <OpForm>+
<Nat>         %%% a natural number
<Term> ::= <Token> | ( <Term> ) | <Term>+
<Token>       %%% Any symbol other than ( or )
<TokenString> ::= <Token> | ( <TokenString> ) | <TokenString>*
<LabelId>    %%% simple identifier

```

In parsing module expressions, instantiation has higher precedence than renaming, which in turn has higher precedence than summation.

26.2 Synonyms

sort = sorts

```
subsort = subsorts  
var = vars
```

Command only synonyms:

```
advise = advisory = advisories  
alias = aliases  
attr = attribute  
cmd = command  
cond = condition  
cont = continue  
eqs = eq  
erew = erewrite  
flat = flattened  
frew = frewrite  
kinds = components  
label = labels  
mbs = mb  
norm = normalize  
paren = parens = parentheses  
q = quit  
rat = rational  
red = reduce  
rew = rewrite  
rls = rl = rule = rules  
s.t. = such that  
subst = substitution
```

Module only synonyms:

```
assoc = associative  
ceq = cq  
comm = commutative  
config = configuration  
ctor = constructor  
ex = extending  
id: = identity:  
idem = idempotent  
inc = including  
iter = iterated  
msg = message  
obj = object  
owise = otherwise  
poly = polymorphic  
prec = precedence  
pr = protecting  
strat = strategy
```

26.3 Lexical Issues

Tokens are sequences of printable ASCII characters delimited by white space, except that ‘(’, ‘)’, ‘[’, ‘]’, ‘{’, ‘}’, and ‘,’ are always considered as single character tokens, unless backquoted.

Single line comments are started by one of `***` or `---`, and ended by the end of line. Multiline comments are started by `***(` and ended by `)`. Parentheses (whether backquoted or not) must balance within multiline comments.

String identifiers use C backslash conventions [206, Section A2.5.2].

References

1. G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.
2. G. Agha, C. Gunter, M. Greenwald, S. Khanna, J. Meseguer, K. Sen, and P. Thati. Formal modeling and analysis of DoS using probabilistic rewrite theories. In A. Sabelfeld, editor, *Proceedings Workshop on Foundations of Computer Security, FCS'05, (Affiliated with LICS'05), Chicago, IL, June 30-July 1, 2005*, pages 91–102, 2005. <http://www.cs.chalmers.se/~andrei/FCS05/fcs05.pdf>.
3. G. Agha, J. Meseguer, and K. Sen. PMAude: Rewrite-based specification language for probabilistic object systems. In A. Cerone and H. Wiklicky, editors, *Proceedings Third Workshop on Quantitative Aspects of Programming Languages, QAPL'05, Edinburgh, UK, April 2005*, volume 153(2) of *Electronic Notes in Theoretical Computer Science*, pages 213–239. Elsevier, 2006. <http://www.sciencedirect.com/science/journal/15710661>.
4. W. Ahrendt, A. Roth, and R. Sasse. Automatic validation of transformation rules for Java verification against a rewriting semantics. In G. Sutcliffe and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 12th International Conference, LPAR 2005, Montego Bay, Jamaica, December 2-6, 2005, Proceedings*, volume 3835 of *Lecture Notes in Artificial Intelligence*, pages 412–426. Springer, 2005.
5. M. Al-Turki. A rewriting logic approach to the semantics of Orc. Master's thesis, Computer Science Department, University of Illinois at Urbana-Champaign, December 2005.
6. A. Albarrán, F. Durán, and A. Vallecillo. From Maude specifications to SOAP distributed implementations: A smooth transition. In *VI Jornadas Ingeniería del Software y Bases de Datos (JISBD'01)*, pages 419–433, 2001.
7. A. Albarrán, F. Durán, and A. Vallecillo. Maude meets CORBA. In *Proceedings 2nd Argentine Symposium on Software Engineering, Buenos Aires, Argentina, September 10–11, 2001*.
8. A. Albarrán, F. Durán, and A. Vallecillo. On the smooth implementation of component-based system specifications. In *Proceedings 6th ECOOP International Workshop on Component-Oriented Programming, WCOP'01, Budapest, Hungary, June 2001, 2001*.

9. M. Alpuente, S. Escobar, and J. Iborra. Modular termination of basic narrowing. In A. Voronkov, editor, *Rewriting Techniques and Applications, 19th International Conference, RTA 2008, Hagenberg, Austria, July 15-17, 2008, Proceedings*, volume 5117 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2008.
10. M. Alpuente, M. Falaschi, and G. Vidal. Partial evaluation of functional logic programs. *ACM Transactions on Programming Languages and Systems*, 20(4):768–844, 1998.
11. M. Alpuente, J. Iborra, and S. Escobar. Termination of narrowing revisited. *Theoretical Computer Science*, 410(46):4608–4625, 2009.
12. T. Arts and H. Zantema. Termination of logic programs using semantic unification. In M. Proietti, editor, *Logic Programming Synthesis and Transformation, 5th International Workshop, LOPSTR'95, Utrecht, The Netherlands, September 20-22, 1995, Proceedings*, volume 1048 of *Lecture Notes in Computer Science*, pages 219–233. Springer, 1996.
13. J. van Baalen, J. L. Caldwell, and S. Mishra. Specifying and checking fault-tolerant agent-based protocols using Maude. In J. L. Rash, C. A. Rouff, W. Truszkowski, D. Gordon, and M. G. Hinchey, editors, *Formal Approaches to Agent-Based Systems: First International Workshop, FAABS 2000, Greenbelt, MD, USA, April 2000, Revised Papers*, volume 1871 of *Lecture Notes in Computer Science*, pages 180–193. Springer, 2001.
14. K. Bae, S. Escobar, and J. Meseguer. Abstract logical model checking of infinite-state systems using narrowing. In F. van Raamsdonk, editor, *24th International Conference on Rewriting Techniques and Applications, RTA 2013, June 24-26, 2013, Eindhoven, The Netherlands*, volume 21 of *LIPICs*, pages 81–96. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.
15. H. Baker and C. Hewitt. Laws for communicating parallel processes. In *Proceedings of the 1977 IFIP Congress*, pages 987–992. IFIP Press, 1977.
16. W. W. R. Ball and H. S. M. Coxeter. *Mathematical Recreations and Essays*. Dover, 1987.
17. D. Basin, M. Clavel, and J. Meseguer. Reflective metalogical frameworks. In *Proceedings of LFM'99: Workshop on Logical Frameworks and Meta-languages*, Paris, France, September 28, 1999. <http://www.site.uottawa.ca/~afelty/LFM99/index.html>.
18. D. Basin, M. Clavel, and J. Meseguer. Rewriting logic as a metalogical framework. In S. Kapoor and S. Prasad, editors, *Twentieth Conference on the Foundations of Software Technology and Theoretical Computer Science, New Delhi, India, December 13-15, 2000, Proceedings*, volume 1974 of *Lecture Notes in Computer Science*, pages 55–80. Springer, 2000.
19. D. Basin, M. Clavel, and J. Meseguer. Reflective metalogical frameworks. *ACM Transactions on Computational Logic*, 5(3):528–576, 2004.
20. D. Basin and G. Denker. Maude versus Haskell: An experimental comparison in security protocol analysis. In Futatsugi [164], pages 235–256. <http://www.sciencedirect.com/science/journal/15710661>.
21. J. Bergstra and J. Tucker. Characterization of computable data types by means of a finite equational specification method. In J. W. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming, 7th Colloquium, Noordwijkerhout, The Netherlands, July 14-18, 1980, Proceedings*, volume 85 of *Lecture Notes in Computer Science*, pages 76–90. Springer, 1980.

22. K. J. Berkling and E. Fehr. A consistent extension of the lambda-calculus as a base for functional programming languages. *Information and Control*, 55:89–101, 1982.
23. M. Blaha and J. Rumbaugh. *Object-oriented modeling and design with UML, Second Edition*. Prentice Hall, 2005.
24. H.-J. Boehm, R. R. Atkinson, and M. F. Plass. Ropes: An alternative to strings. *Software Practice and Experience*, 25(12):1315–1330, 1995.
25. A. Bogomolny. Interactive mathematics miscellany and puzzles. <http://www.cut-the-knot.org/games.shtml>.
26. A. Boronat, J. A. Carsí, and I. Ramos. Algebraic specification of a model transformation engine. In L. Baresi and R. Heckel, editors, *Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2006, Held as Part of ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings*, volume 3922 of *Lecture Notes in Computer Science*, pages 262–277. Springer, 2006.
27. P. Borovanský, C. Kirchner, H. Kirchner, and P.-E. Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 285(2):155–185, 2002.
28. C. Bouchard, K. A. Gero, C. Lynch, and P. Narendran. On forward closure and the finite variant property. In P. Fontaine, C. Ringeissen, and R. A. Schmidt, editors, *Frontiers of Combining Systems - 9th International Symposium, FroCoS 2013, Nancy, France, September 18-20, 2013. Proceedings*, volume 8152 of *Lecture Notes in Computer Science*, pages 327–342. Springer, 2013.
29. A. Boudet. Unification in a combination of equational theories: an efficient algorithm. In M. E. Stickel, editor, *10th International Conference on Automated Deduction, Kaiserslautern, FRG, July 24-27, 1990, Proceedings*, volume 449 of *Lecture Notes in Computer Science*, pages 292–307. Springer, 1990.
30. A. Boudet, E. Contejean, and H. Devie. A new AC-unification algorithm with an algorithm for solving systems of diophantine equations. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science (LICS 1990)*, pages 289–299. IEEE Computer Society Press, June 1990.
31. A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.
32. C. Braga. *Rewriting Logic as a Semantic Framework for Modular Structural Operational Semantics*. PhD thesis, Departamento de Informática, Pontificia Universidade Católica de Rio de Janeiro, Brasil, 2001.
33. C. Braga, H. Haeusler, J. Meseguer, and P. Mosses. Maude Action Tool: Using reflection to map action semantics to rewriting logic. In T. Rus, editor, *Algebraic Methodology and Software Technology, 8th International Conference, AMAST 2000, Iowa City, Iowa, USA, May 20–27, 2000, Proceedings*, volume 1816 of *Lecture Notes in Computer Science*, pages 407–421. Springer, 2000.
34. C. Braga and J. Meseguer. Modular rewriting semantics in practice. In Martí-Oliet [228], pages 393–416. <http://www.sciencedirect.com/science/journal/15710661>.
35. R. Bruni. *Tile Logic for Synchronized Rewriting of Concurrent Systems*. PhD thesis, Dipartimento di Informatica, Università di Pisa, 1999. Technical Report TD-1/99. http://www.di.unipi.it/phd/tesi/tesi_1999/TD-1-99.ps.gz.
36. R. Bruni and J. Meseguer. Generalized rewrite theories. In J. C. M. Baeten, J. K. Lenstra, J. Parrow, and G. J. Woeginger, editors, *Automata, Languages*

- and Programming, 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4, 2003. *Proceedings*, volume 2719 of *Lecture Notes in Computer Science*, pages 252–266. Springer, 2003.
37. R. Bruni, J. Meseguer, and U. Montanari. Implementing tile systems: Some examples from process calculi. In P. Degano, U. Vaccaro, and G. Pirillo, editors, *Proceedings 6th Italian Conference on Theoretical Computer Science, ICTCS'98*, pages 168–179. World Scientific, 1998.
 38. R. Bruni, J. Meseguer, and U. Montanari. Internal strategies in a rewriting implementation of tile systems. In Kirchner and Kirchner [208], pages 95–116. <http://www.sciencedirect.com/science/journal/15710661>.
 39. R. Bruni, J. Meseguer, and U. Montanari. Process and term tile logic. Technical Report SRI-CSL-98-06, Computer Science Laboratory, SRI International, July 1998. Also TR-98-09, Dipartimento di Informatica, Università di Pisa. <http://www.di.unipi.it/~bruni/publications/techrep98.ps.gz>.
 40. R. Bruni, J. Meseguer, and U. Montanari. Executable tile specifications for process calculi. In J.-P. Finance, editor, *Fundamental Approaches to Software Engineering, Second International Conference, FASE'99, Held as Part of ETAPS'99, Amsterdam, The Netherlands, March 22–28, 1999, Proceedings*, volume 1577 of *Lecture Notes in Computer Science*, pages 60–76. Springer, 1999.
 41. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
 42. R. Burstall and J. A. Goguen. The semantics of Clear, a specification language. In D. Bjørner, editor, *Abstract Software Specifications, 1979 Copenhagen Winter School, January 22 - February 2, 1979, Proceedings*, volume 86 of *Lecture Notes in Computer Science*, pages 292–332. Springer, 1980.
 43. F. Butler, I. Cervesato, A. D. Jaggard, and A. Scedrov. A formal analysis of some properties of Kerberos 5 using MSR. In *Fifteenth Computer Security Foundations Workshop — CSFW-15*, pages 175–190, Cape Breton, NS, Canada, 24–26 June 2002. IEEE Computer Society Press.
 44. M. Calder and C. Shankland. A symbolic semantics and bisimulation for Full LOTOS. In M. Kim, B. Chin, S. Kang, and D. Lee, editors, *Proceedings of FORTE 2001, 21st International Conference on Formal Techniques for Networked and Distributed Systems*, pages 184–200. Kluwer Academic Publishers, 2001.
 45. G. Carabetta, P. Degano, and F. Gadducci. CCS semantics via proved transition systems and rewriting logic. In Kirchner and Kirchner [208], pages 253–272. <http://www.sciencedirect.com/science/journal/15710661>.
 46. F. M. Carrano and J. J. Prichard. *Data Abstraction and Problem Solving with C++*. Third Edition. Addison-Wesley, 2002.
 47. I. Cervesato. A specification language for crypto-protocols based on multi-set rewriting, dependent types and subsorting. In *Workshop on Specification, Analysis and Validation for Emerging Technologies*, pages 1–22, 2001.
 48. I. Cervesato. Typed MSR: Syntax and examples. In V. I. Gorodetski, V. A. Skormin, and L. J. Popyack, editors, *Information Assurance in Computer Networks: Methods, Models, and Architectures for Network Security: International Workshop MMM-ACNS 2001, St. Petersburg, Russia, May 21-23, 2001*, volume 2052 of *Lecture Notes in Computer Science*, pages 159–177. Springer, 2001.

49. I. Cervesato. Data access specification and the most powerful symbolic attacker in MSR. In M. Okada, B. Pierce, A. Scedrov, H. Tokuda, and A. Yonezawa, editors, *Software Security - Theories and Systems: Next-NSF-JSPS International Symposium, ISSS 2002 Tokyo, Japan, November 8-10, 2002 Revised Papers*, volume 2609 of *Lecture Notes in Computer Science*, pages 384–416. Springer, 2003.
50. I. Cervesato, N. Durgin, P. D. Lincoln, J. C. Mitchell, and A. Scedrov. A meta-notation for protocol analysis. In *12th Computer Security Foundations Workshop — CSFW-12*, pages 55–69, Mordano, Italy, 28–30 June 1999. IEEE Computer Society Press.
51. I. Cervesato and M.-O. Stehr. Representing the MSR cryptoprotocol specification language in an extension of rewriting logic with dependent types. In Martí-Oliet [228], pages 183–207. <http://www.sciencedirect.com/science/journal/15710661>.
52. I. Cervesato and M.-O. Stehr. Representing the MSR cryptoprotocol specification language in an extension of rewriting logic with dependent types. *Higher-Order and Symbolic Computation*, 20(1-2):3–35, 2007.
53. F. Chalub. An Implementation of Modular SOS in Maude. Master’s thesis, Universidade Federal Fluminense, May 2005. <http://www.ic.uff.br/~frosario/dissertation.pdf>.
54. F. Chalub and C. Braga. A modular rewriting semantics for CML. *Journal of Universal Computer Science*, 10(7):789–807, July 2004. http://www.jucs.org/jucs_10_7/a_modular_rewriting_semantics.
55. F. Chalub and C. Braga. Maude MSOS tool. In Denker and Talcott [97], pages 3–17. <http://www.sciencedirect.com/science/journal/15710661>.
56. F. Chen, G. Roşu, and R. P. Venkatesan. Rule-based analysis of dimensional safety. In R. Nieuwenhuis, editor, *Rewriting Techniques and Applications, 14th International Conference, RTA 2003, Valencia, Spain, June 9-11, 2003, Proceedings*, volume 2706 of *Lecture Notes in Computer Science*, pages 197–207. Springer, 2003.
57. A. Cholewa, J. Meseguer, and S. Escobar. Variants of variants and the finite variant property. Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign, <http://hdl.handle.net/2142/47117>, 2014.
58. H. Cirstea. *Calcul de Réécriture: Fondements et Applications*. PhD thesis, Université Henri Poincaré – Nancy I, 2000.
59. H. Cirstea and C. Kirchner. Combining higher-order and first-order computations using ρ -calculus: Towards a semantics of ELAN. In D. Gabbay and M. de Rijke, editors, *Frontiers of Combining Systems 2*, pages 95–121. Research Studies Press/Wiley, 1999.
60. H. Cirstea and C. Kirchner. The simply typed rewriting calculus. In Futatsugi [164], pages 23–41. <http://www.sciencedirect.com/science/journal/15710661>.
61. H. Cirstea and C. Kirchner. The rewriting calculus — Part I. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9(3):363–399, 2001.
62. H. Cirstea and C. Kirchner. The rewriting calculus — Part II. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9(3):401–434, 2001.
63. H. Cirstea, C. Kirchner, and L. Liquori. The rho cube. In F. Honsell and M. Miculan, editors, *Foundations of Software Science and Computation Structures, 4th International Conference, FOSSACS 2001, Held as Part of ETAPS*

- 2001, Genova, Italy, April 2001, *Proceedings*, volume 2030 of *Lecture Notes in Computer Science*, pages 168–183. Springer, 2001.
64. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
 65. M. Clavel. *Reflection in General Logics and in Rewriting Logic, with Applications to the Maude Language*. PhD thesis, Universidad de Navarra, Spain, February 1998.
 66. M. Clavel. *Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications*. CSLI Publications, 2000.
 67. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: specification and programming in rewriting logic. SRI International, January 1999, <http://maude.cs.uiuc.edu/maude1/manual/>.
 68. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. A tutorial on Maude. SRI International, March 2000, <http://maude.cs.uiuc.edu/maude1/tutorial/>.
 69. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Towards Maude 2.0. In Futatsugi [164], pages 294–315. <http://www.sciencedirect.com/science/journal/15710661>.
 70. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.
 71. M. Clavel, F. Durán, S. Eker, and J. Meseguer. Building equational proving tools by reflection in rewriting logic. In *Proceedings of the CafeOBJ Symposium '98, Numazu, Japan*. CafeOBJ Project, April 1998. <http://maude.cs.uiuc.edu/papers/>.
 72. M. Clavel, F. Durán, S. Eker, and J. Meseguer. Design and implementation of the Cafe prover and the Church-Rosser checker tools. Technical report, Computer Science Laboratory, SRI International, March 1998. <http://maude.cs.uiuc.edu/papers/>.
 73. M. Clavel, F. Durán, S. Eker, and J. Meseguer. Building equational proving tools by reflection in rewriting logic. In K. Futatsugi, A. T. Nakagawa, and T. Tamai, editors, *CAFE: An Industrial-Strength Algebraic Formal Method*. Elsevier, 2000. <http://maude.cs.uiuc.edu/papers/>.
 74. M. Clavel, F. Durán, S. Eker, J. Meseguer, and M.-O. Stehr. Maude as a formal meta-tool. In J. M. Wing, J. Woodcock, and J. Davies, editors, *FM'99 — Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 20–24, 1999 Proceedings, Volume II*, volume 1709 of *Lecture Notes in Computer Science*, pages 1684–1703. Springer, 1999.
 75. M. Clavel and M. Egea. ITP/OCL: A rewriting-based validation tool for UML+OCL static class diagrams. In M. Johnson and V. Vene, editors, *Algebraic Methodology and Software Technology, 11th International Conference, AMAST 2006, Kuressaare, Estonia, July 2006, Proceedings*, volume 4019 of *Lecture Notes in Computer Science*, pages 368–373. Springer, 2006.
 76. M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. In Meseguer [246], pages 126–148. <http://www.sciencedirect.com/science/journal/15710661>.
 77. M. Clavel and J. Meseguer. Reflection in conditional rewriting logic. *Theoretical Computer Science*, 285(2):245–288, 2002.

78. M. Clavel, J. Meseguer, and M. Palomino. Reflection in membership equational logic, many-sorted equational logic, Horn logic with equality, and rewriting logic. In Gadducci and Montanari [169], pages 91–107. <http://www.sciencedirect.com/science/journal/15710661>.
79. M. Clavel, M. Palomino, and A. Riesco. Introducing the ITP tool: a tutorial. *Journal of Universal Computer Science*, 12(11):1618–1650, 2006.
80. M. Clavel, M. Palomino, and J. Santa-Cruz. Integrating decision procedures in reflective rewriting-based theorem provers. In S. Antoy and Y. Toyama, editors, *Proceedings 4th International Workshop on Reduction Strategies in Rewriting and Programming (WRS'04)*, pages 15–24, Aachen, Germany, June 2004. <http://www-i2.informatik.rwth-aachen.de/WRS04/WRS-proceedings.pdf>.
81. M. Clavel and J. Santa-Cruz. ASIP + ITP: A verification tool based on algebraic semantics. In F. J. López-Fraguas, editor, *V Jornadas sobre Programación y Lenguajes, PROLE 2005*, pages 149–158. Thomson, 2005.
82. H. Comon-Lundh and S. Delaune. The finite variant property: how to get rid of some algebraic properties. In J. Giesl, editor, *Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings*, volume 3467 of *Lecture Notes in Computer Science*, pages 294–307. Springer, 2005.
83. E. Contejean and H. Devie. An efficient incremental algorithm for solving systems of linear diophantine equations. *Information and Computation*, 113(1):143–172, 1994.
84. E. Contejean, C. Marché, and X. Urbain. CiME, 2004. Available at <http://cime.lri.fr/>.
85. T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, 1988.
86. M. d’Amorim and G. Roşu. An equational specification for the Scheme language. *Journal of Universal Computer Science*, 11(7):1327–1348, 2005. Selected papers from the 9th Brazilian Symposium on Programming Languages (SBLP’05). Also Technical Report No. UIUCDCS-R-2005-2567, April 2005.
87. P. Degano, F. Gadducci, and C. Priami. A causal semantics for CCS via rewriting logic. *Theoretical Computer Science*, 275(1-2):271–304, 2002.
88. G. Denker, J. J. García-Luna-Aceves, J. Meseguer, P. C. Ölveczky, J. Raju, B. Smith, and C. L. Talcott. Specification and analysis of a reliable broadcasting protocol in Maude. In B. Hajek and R. S. Sreenivas, editors, *Proceedings 37th Allerton Conference on Communication, Control and Computation*, pages 738–747. University of Illinois, 1999.
89. G. Denker, J. J. García-Luna-Aceves, J. Meseguer, P. C. Ölveczky, J. Raju, B. Smith, and C. L. Talcott. Specifying a reliable broadcasting protocol in Maude. Technical report, Computer Science Laboratory, SRI International, 1999.
90. G. Denker, J. Meseguer, and C. L. Talcott. Protocol specification and analysis in Maude. In N. Heintze and J. Wing, editors, *Proceedings of Workshop on Formal Methods and Security Protocols, June 25, 1998, Indianapolis, Indiana*, 1998.
91. G. Denker, J. Meseguer, and C. L. Talcott. Formal specification and analysis of active networks and communication protocols: The Maude experience. In D. Maughan, G. Koob, and S. Saydjari, editors, *Proceedings DARPA Information Survivability Conference and Exposition, DISCEX 2000, Hilton Head*

- Island, South Carolina, January 25–27, 2000*, pages 251–265. IEEE Computer Society Press, 2000.
92. G. Denker, J. Meseguer, and C. L. Talcott. Rewriting semantics of meta-objects and composable distributed services. In Futatsugi [164], pages 407–427. <http://www.sciencedirect.com/science/journal/15710661>.
 93. G. Denker and J. Millen. CAPSL and CIL language design: A common authentication protocol specification language and its intermediate language. Technical Report SRI-CSL-99-02, Computer Science Laboratory, SRI International, 1999. http://www.cs1.sri.com/~denker/pub_99.html.
 94. G. Denker and J. Millen. CAPSL intermediate language. In N. Heintze and E. Clarke, editors, *Proceedings of Workshop on Formal Methods and Security Protocols, FMSP'99, July 1999, Trento, Italy*, 1999.
 95. G. Denker and J. Millen. CAPSL integrated protocol environment. In D. Maughan, G. Koob, and S. Saydjari, editors, *Proceedings DARPA Information Survivability Conference and Exposition, DISCEX 2000, Hilton Head Island, South Carolina, January 25-27, 2000*, pages 207–222. IEEE Computer Society Press, 2000.
 96. G. Denker and J. Millen. The CAPSL integrated protocol environment. Technical report, Computer Science Laboratory, SRI International, 2000. Technical Report SRI-CSL-2000-02.
 97. G. Denker and C. Talcott, editors. *Proceedings Sixth International Workshop on Rewriting Logic and its Applications, WRLA 2006, Vienna, Austria, April 1–2, 2006*, volume 176(4) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2007. <http://www.sciencedirect.com/science/journal/15710661>.
 98. G. Denker and C. L. Talcott. Formal checklists for remote agent dependability. In Martí-Oliet [228], pages 229–248. <http://www.sciencedirect.com/science/journal/15710661>.
 99. G. Denker and C. L. Talcott. A formal framework for goal net analysis. In *Workshop on Verification and Validation of Planning Systems*. AAAI, 2005.
 100. E. Deplagne. Sequent calculus viewed modulo. In C. Pilière, editor, *Proceedings of the Fifth ESSLLI Student Session*, pages 66–76, 2000.
 101. N. Dershowitz. Goal solving as operational semantics. In *International Logic Programming Symposium, Portland, OR*, pages 3–17. MIT Press, December 1995.
 102. N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 243–320. North-Holland, 1990.
 103. A. van Deursen. *Executable Language Definitions*. PhD thesis, University of Amsterdam, 1994.
 104. A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*. World Scientific, 1996.
 105. E. Domenjoud. Solving systems of linear diophantine equations: An algebraic approach. In A. Tarlecki, editor, *Mathematical Foundations of Computer Science 1991, 16th International Symposium, MFCS'91, Kazimierz Dolny, Poland, September 9-13, 1991, Proceedings*, volume 520 of *Lecture Notes in Computer Science*, pages 141–150. Springer, 1991.
 106. G. Dowek, T. Hardin, and C. Kirchner. Higher order unification via explicit substitutions. *Information and Computation*, 157(1/2):183–235, 2000.

107. G. Dowek, T. Hardin, and C. Kirchner. HOL- $\lambda\sigma$: An intentional first-order expression of higher-order logic. *Mathematical Structures in Computer Science*, 11(1):21–45, 2001.
108. G. Dowek, T. Hardin, and C. Kirchner. Theorem proving modulo. *Journal of Automated Reasoning*, 31(1):33–72, 2003.
109. F. Durán. Coherence checker and completion tools for Maude specifications. Manuscript, Computer Science Laboratory, SRI International, 2000. <http://www.lcc.uma.es/~duran/ChC/>.
110. F. Durán. Termination checker and Knuth-Bendix completion tools for Maude equational specifications. Manuscript, Computer Science Laboratory, SRI International, 2000. <http://maude.cs.uiuc.edu/papers/>.
111. F. Durán. *A Reflective Module Algebra with Applications to the Maude Language*. PhD thesis, University of Málaga, Spain, 1999. <http://maude.cs.uiuc.edu/papers/>.
112. F. Durán. The extensibility of Maude’s module algebra. In T. Rus, editor, *Algebraic Methodology and Software Technology, 8th International Conference, AMAST 2000, Iowa City, Iowa, USA, May 20-27, 2000, Proceedings*, volume 1816 of *Lecture Notes in Computer Science*, pages 422–437. Springer, 2000.
113. F. Durán, S. Eker, P. Lincoln, and J. Meseguer. Principles of Mobile Maude. In D. Kotz and F. Mattern, editors, *Agent Systems, Mobile Agents, and Applications, Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents, ASA/MA 2000, Zurich, Switzerland, September 13–15, 2000, Proceedings*, volume 1882 of *Lecture Notes in Computer Science*, pages 73–85. Springer, 2000.
114. F. Durán, S. Escobar, and S. Lucas. Towards (constructor) normal forms for Maude within Full Maude. In S. Lucas, M.-M. Gallardo, and E. Pimentel, editors, *IV Jornadas sobre Programación y Lenguajes, PROLE 2004*, pages 125–136, 2004.
115. F. Durán, S. Escobar, and S. Lucas. New evaluation commands for Maude within Full Maude. In Martí-Oliet [228], pages 263–284. <http://www.sciencedirect.com/science/journal/15710661>.
116. F. Durán, S. Escobar, and S. Lucas. On-demand evaluation for Maude. In S. Abdennadher and C. Ringeissen, editors, *Proceedings Fifth International Workshop on Rule-Based Programming, RULE 2004, Aachen, Germany*, volume 124(1) of *Electronic Notes in Theoretical Computer Science*, pages 25–39. Elsevier, 2005. <http://www.sciencedirect.com/science/journal/15710661>.
117. F. Durán, S. Lucas, C. Marché, J. Meseguer, and X. Urbain. Proving operational termination of membership equational programs. *Higher-Order and Symbolic Computation*, 21(1-2):59–88, 2008.
118. F. Durán, S. Lucas, and J. Meseguer. MTT: The Maude termination tool (system description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, volume 5195 of *Lecture Notes in Computer Science*, pages 313–319. Springer, 2008.
119. F. Durán, S. Lucas, J. Meseguer, C. Marché, and X. Urbain. Proving termination of membership equational programs. In P. Sestoft and N. Heintze, editors, *Proceedings ACM SIGPLAN 2004 Symposium on Partial Evaluation and Program Manipulation, PEPM’04*, pages 147–158. ACM Press, 2004.

120. F. Durán and J. Meseguer. A Church-Rosser checker tool for Maude equational specifications. Manuscript, Computer Science Laboratory, SRI International, 2000. <http://www.lcc.uma.es/~duran/CRC/>.
121. F. Durán and J. Meseguer. An extensible module algebra for Maude. In Kirchner and Kirchner [208], pages 174–195. <http://www.sciencedirect.com/science/journal/15710661>.
122. F. Durán and J. Meseguer. The Maude specification of Full Maude. Technical report, Computer Science Laboratory, SRI International, February 1999. <http://maude.cs.uiuc.edu/papers/>.
123. F. Durán and J. Meseguer. Parameterized theories and views in Full Maude 2.0. In Futatsugi [164], pages 316–338. <http://www.sciencedirect.com/science/journal/15710661>.
124. F. Durán and J. Meseguer. Structured theories and institutions. *Theoretical Computer Science*, 309(1-3):357–380, 2003.
125. F. Durán and J. Meseguer. Maude’s module algebra. *Science of Computer Programming*, 66(2):125–153, 2007.
126. F. Durán and J. Meseguer. A Church-Rosser checker tool for conditional order-sorted equational Maude specifications. In P. C. Ölveczky, editor, *8th International Workshop on Rewriting Logic and its Applications, WLRA 2010, Paphos, Cyprus, March 20-21, 2010, Proceedings*, Lecture Notes in Computer Science. Springer, 2010. To appear.
127. F. Durán and J. Meseguer. A Maude coherence checker tool for conditional order-sorted rewrite theories. In P. C. Ölveczky, editor, *8th International Workshop on Rewriting Logic and its Applications, WLRA 2010, Paphos, Cyprus, March 20-21, 2010, Proceedings*, Lecture Notes in Computer Science. Springer, 2010. To appear.
128. F. Durán and J. Meseguer. On the Church-Rosser and coherence properties of conditional order-sorted rewrite theories. *Journal of Logic and Algebraic Programming*, 81(7-8):816–850, 2012.
129. F. Durán, M. Roldán, and A. Vallecillo. Using Maude to write and execute ODP information viewpoint specifications. *Computer Standards and Interfaces*, 27(6):597–620, 2005.
130. F. Durán and A. Vallecillo. Formalizing ODP enterprise specifications in Maude. *Computer Standards and Interfaces*, 25(2):83–102, 2003.
131. F. Durán and A. Verdejo. A conference reviewing system in Mobile Maude. In Gadducci and Montanari [169], pages 79–95. <http://www.sciencedirect.com/science/journal/15710661>.
132. M. Egea. ITP/OCL: A theorem prover-based tool for UML+OCL class diagrams. Master’s thesis, Facultad de Informática, Universidad Complutense de Madrid, September 2005. <http://maude.sip.ucm.es/~marina/pubs/dea.pdf>.
133. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1, Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1985.
134. S. Eker. Fast matching in combination of regular equational theories. In Meseguer [246], pages 90–109. <http://www.sciencedirect.com/science/journal/15710661>.
135. S. Eker. Term rewriting with operator evaluation strategies. In Kirchner and Kirchner [208], pages 311–330. <http://www.sciencedirect.com/science/journal/15710661>.

136. S. Eker. Associative-commutative rewriting on large terms. In R. Nieuwenhuis, editor, *Rewriting Techniques and Applications, 14th International Conference, RTA 2003, Valencia, Spain, June 9-11, 2003, Proceedings*, volume 2706 of *Lecture Notes in Computer Science*, pages 14–29. Springer, 2003.
137. S. Eker. Unification in Maude, January 2007. Talk at the “Protocol eXchange Seminar”, Naval Postgraduate School. Available at <http://maude.cs.uiuc.edu/talks/eker-unification.pdf>.
138. S. Eker, M. Knapp, K. Laderoute, P. Lincoln, and C. Talcott. Pathway Logic: Executable models of biological networks. In Gadducci and Montanari [169], pages 125–142. <http://www.sciencedirect.com/science/journal/15710661>.
139. S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker. In Gadducci and Montanari [169], pages 143–168. <http://www.sciencedirect.com/science/journal/15710661>.
140. H. B. Enderton. *A Mathematical Introduction to Logic, Second Edition*. Academic Press, 2000.
141. S. Escobar. Functional logic programming in maude. In S. Iida, J. Meseguer, and K. Ogata, editors, *Specification, Algebra, and Software - Essays Dedicated to Kokichi Futatsugi*, volume 8373 of *Lecture Notes in Computer Science*, pages 315–336. Springer, 2014.
142. S. Escobar, J. Hendrix, C. Meadows, and J. Meseguer. Diffie-Hellman cryptographic reasoning in the Maude-NRL protocol analyzer. In M. Nesi and R. Treinen, editors, *Proceedings Second International Workshop on Security and Rewriting Techniques, SecReT 2007, Paris, France, June 29, 2007*, 2007.
143. S. Escobar, C. Meadows, and J. Meseguer. A rewriting-based inference system for the NRL protocol analyzer: Grammar generation. In *Proceedings FMSE’05, Formal Methods in Security Engineering*, pages 1–12, Alexandria, Virginia, November 2005. ACM Press.
144. S. Escobar, C. Meadows, and J. Meseguer. A rewriting-based inference system for the NRL Protocol Analyzer and its meta-logical properties. *Theoretical Computer Science*, 367(1-2):162–202, 2006.
145. S. Escobar, C. Meadows, and J. Meseguer. Equational cryptographic reasoning in the Maude-NRL Protocol Analyzer. In M. Fernández and C. Kirchner, editors, *Proceedings First International Workshop on Security and Rewriting Techniques, SecReT 2006, Venice, Italy, July 15, 2006*, volume 171(4) of *Electronic Notes in Theoretical Computer Science*, pages 23–36. Elsevier, 2007. <http://www.sciencedirect.com/science/journal/15710661>.
146. S. Escobar, C. Meadows, and J. Meseguer. Maude-NPA: Cryptographic protocol analysis modulo equational properties. In A. Aldini, G. Barthe, and R. Gorrieri, editors, *Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures*, volume 5705 of *Lecture Notes in Computer Science*, pages 1–50. Springer, 2009.
147. S. Escobar and J. Meseguer. Symbolic model checking of infinite-state systems using narrowing. In F. Baader, editor, *Term Rewriting and Applications, 18th International Conference, RTA 2007, Paris, France, June 26-28, 2007, Proceedings*, volume 4533 of *Lecture Notes in Computer Science*, pages 153–168. Springer, 2007.
148. S. Escobar, J. Meseguer, and R. Sasse. Effectively checking the finite variant property. In A. Voronkov, editor, *Rewriting Techniques and Applications, 19th*

- International Conference, RTA 2008, Hagenberg, Austria, July 15-17, 2008, Proceedings*, volume 5117 of *Lecture Notes in Computer Science*, pages 79–93. Springer, 2008.
149. S. Escobar, J. Meseguer, and R. Sasse. Variant narrowing and equational unification. In G. Roşu, editor, *Proceedings 7th International Workshop on Rewriting Logic and its Applications, WRLA 2008, Budapest, Hungary, March 29-30, 2008*, volume 238(3) of *Electronic Notes in Theoretical Computer Science*, pages 88–102. Elsevier, 2009. <http://www.sciencedirect.com/science/journal/15710661>.
 150. S. Escobar, J. Meseguer, and P. Thati. Natural narrowing for general term rewriting systems. In J. Giesl, editor, *Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings*, volume 3467 of *Lecture Notes in Computer Science*, pages 279–293. Springer, 2005.
 151. S. Escobar, R. Sasse, and J. Meseguer. Folding variant narrowing and optimal variant termination. *Journal of Logic and Algebraic Programming*, 81(7-8):898–928, 2012.
 152. F. Fages. Associative-commutative unification. *Journal of Symbolic Computation*, 3(3):257–275, 1987.
 153. A. Farzan, F. Cheng, J. Meseguer, and G. Roşu. Formal analysis of Java programs in JavaFAN. In R. Alur and D. A. Peled, editors, *Computer Aided Verification: 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004. Proceedings*, volume 3114 of *Lecture Notes in Computer Science*, pages 501–505, 2004.
 154. A. Farzan and J. Meseguer. State space reduction of rewrite theories using invisible transitions. In M. Johnson and V. Vene, editors, *Algebraic Methodology and Software Technology, 11th International Conference, AMAST 2006, Kuressaare, Estonia, July 2006, Proceedings*, volume 4019 of *Lecture Notes in Computer Science*, pages 142–157. Springer, 2006.
 155. A. Farzan and J. Meseguer. Partial order reduction for rewriting semantics of programming languages. In Denker and Talcott [97], pages 56–75. <http://www.sciencedirect.com/science/journal/15710661>.
 156. A. Farzan, J. Meseguer, and G. Roşu. Formal JVM code analysis in JavaFAN. In C. Rattray, S. Maharaj, and C. Shankland, editors, *Algebraic Methodology and Software Technology, 10th International Conference, AMAST 2004, Stirling, Scotland, UK, July 12-16, 2004, Proceedings*, volume 3116 of *Lecture Notes in Computer Science*, pages 132–147. Springer, 2004.
 157. M. Fay. First-order unification in an equational theory. In W. H. Joyner, editor, *Proceedings of the 4th Workshop on Automated Deduction, Austin, Texas, USA*, pages 161–167. Academic Press, 1979.
 158. J. L. Fernández and A. Toval. Can intuition become rigorous? Foundations for UML model verification tools. In F. M. Titsworth, editor, *International Symposium on Software Reliability Engineering*, pages 344–355. IEEE Press, 2000.
 159. J. L. Fernández and A. Toval. Seamless formalizing the UML semantics through metamodels. In K. Siau and T. Halpin, editors, *Unified Modeling Language: Systems Analysis, Design, and Development Issues*, pages 224–248. Idea Group Publishing, 2001.
 160. G. Ferrari and U. Montanari. Tile formats for located and mobile systems. *Information and Computation*, 156(1/2):173–235, 2000.

161. B. Fischer and G. Roşu. Interpreting abstract interpretations in membership equational logic. Technical report, Research Institute for Advanced Computer Science, 2001. Technical Report RIACS 01.16.
162. D. Fomin, S. Genkin, and I. Itenberg. *Mathematical Circles: The Russian Experience*, volume 7 of *Mathematical World*. American Mathematical Association, 1996.
163. T. Frühwirth and S. Abdennadher. *Essentials of Constraint Programming*. Springer, 2003.
164. K. Futatsugi, editor. *Proceedings Third International Workshop on Rewriting Logic and its Applications, WRLA 2000, Kanazawa, Japan, September 18–20, 2000*, volume 36 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2000. <http://www.sciencedirect.com/science/journal/15710661>.
165. K. Futatsugi and R. Diaconescu. *CafeOBJ Report*. World Scientific, AMAST Series, 1998.
166. F. Gadducci and U. Montanari. Tiles, rewriting rules, and CCS. In Meseguer [246], pages 1–19. <http://www.sciencedirect.com/science/journal/15710661>.
167. F. Gadducci and U. Montanari. The tile model. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 133–166. The MIT Press, 2000.
168. F. Gadducci and U. Montanari. Comparing logics for rewriting: Rewriting logic, action calculi and tile logic. *Theoretical Computer Science*, 285(2):319–358, 2002.
169. F. Gadducci and U. Montanari, editors. *Proceedings Fourth International Workshop on Rewriting Logic and its Applications, WRLA 2002, Pisa, Italy, September 19–21, 2002*, volume 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2004. <http://www.sciencedirect.com/science/journal/15710661>.
170. M. R. Genesereth and N. J. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann Publishers, 1987.
171. P. Glynn. On the role of generalized semi-Markov processes in simulation output analysis. In *Proceedings of the 1983 Winter Simulation Conference*, pages 38–42, 1983.
172. J. Goguen and J. Meseguer. Eqlog: Equality, types and generic modules for logic programming. In D. DeGroot and G. Lindstrom, editors, *Logic Programming, Functions, Relations and Equations*, pages 295–363. Prentice-Hall, 1986.
173. J. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992.
174. J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In J. A. Goguen and G. Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Action*, pages 3–167. Kluwer Academic Publishers, 2000.
175. J. A. Goguen, J.-P. Jouannaud, and J. Meseguer. Operational semantics for order-sorted algebra. In W. Brauer, editor, *Automata, Languages and Programming, 12th Colloquium, Nafplion, Greece, July 15-19, 1985, Proceedings*, volume 194 of *Lecture Notes in Computer Science*, pages 221–231. Springer, 1985.
176. A. Goodloe, C. A. Gunter, and M.-O. Stehr. Formal prototyping in early stages of protocol design. In *Proceedings Workshop on Issues in the Theory*

- of Security (WITS'05), Long Beach, California, January 2005, pages 67–80. ACM Press, 2005.
177. M. Grimeland. Modeling and analysis of time-dependent security protocols in Real-Time Maude. Master's thesis, Dept. of Informatics, University of Oslo, June 2006.
178. S. Gutierrez-Nolasco, N. Venkatasubramanian, M.-O. Stehr, and C. Talcott. Exploring adaptability of secure group communication using formal prototyping techniques. In *Proceedings Third Workshop on Adaptive and Reflective Middleware (RM2004)*, pages 232–237, Toronto, Ontario, Canada, October 19, 2004. ACM Press.
179. M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
180. N. A. Harman. Correctness and verification of hardware systems using Maude. Technical Report 3-2000, Department of Computer Science, University of Wales at Swansea, 2000.
181. N. A. Harman. Verifying a simple pipelined microprocessor using Maude. In M. Cerioli and G. Reggio, editors, *Recent Trends in Algebraic Development Techniques, 15th International Workshop, WADT 2001, Joint with the CoFI WG Meeting, Genova, Italy, April 1-3, 2001, Selected Papers*, volume 2267 of *Lecture Notes in Computer Science*, pages 128–151. Springer, 2001.
182. K. Havelund and G. Roşu. Java PathExplorer — A runtime verification tool. In *Proceedings 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space, ISAIRAS'01*, Montreal, Canada, June 18–22, 2001.
183. K. Havelund and G. Roşu. Monitoring Java programs with Java PathExplorer. In K. Havelund and G. Rosu, editors, *Proceedings First Workshop on Runtime Verification, RV'2001, Paris, France, July 23, 2001*, volume 55(2) of *Electronic Notes in Theoretical Computer Science*, pages 200–217. Elsevier, 2001. <http://www.sciencedirect.com/science/journal/15710661>.
184. K. Havelund and G. Roşu. Monitoring programs using rewriting. Technical report, Research Institute for Advanced Computer Science, 2001.
185. K. Havelund and G. Roşu. Testing linear temporal logic formulae on finite execution traces. Technical Report RIACS 01.08, Research Institute for Advanced Computer Science, May 2001.
186. P. J. Hayes. What the frame problem is and isn't. In Z. W. Pylyshyn, editor, *The Robot's Dilemma: The Frame Problem in Artificial Intelligence*, pages 123–137. Ablex Publishing Corp., 1987.
187. J. Hendrix, M. Clavel, and J. Meseguer. A sufficient completeness reasoning tool for partial specifications. In J. Giesl, editor, *Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings*, volume 3467 of *Lecture Notes in Computer Science*, pages 165–174. Springer, 2005.
188. J. Hendrix and J. Meseguer. On the completeness of context-sensitive order-sorted specifications. Technical Report UIUCDCS-R-2007-2812, Computer Science Dept., University of Illinois at Urbana-Champaign, February 2007.
189. J. Hendrix and J. Meseguer. Order-sorted equational unification revisited. In G. Kniesel and J. S. Pinto, editors, *Proceedings Ninth International Workshop on Rule-Based Programming, RULE 2008, Hagenberg Castle, Austria*, volume 290 of *Electronic Notes in Theoretical Computer Science*, pages 37–50. Elsevier, 2012. <http://www.sciencedirect.com/science/journal/15710661>.

190. J. Hendrix, J. Meseguer, and H. Ohsaki. A sufficient completeness checker for linear order-sorted specifications modulo axioms. In U. Furbach and N. Shankar, editors, *Automated Reasoning - Third International Joint Conference, IJCAR 2006, Seattle, Washington, August 17 - 20, 2006, Proceedings*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 151–155. Springer, 2006.
191. J. Hendrix, H. Ohsaki, and J. Meseguer. Sufficient completeness checking with propositional tree automata. Technical Report UIUCDCS-R-2005-2635, University of Illinois at Urbana-Champaign, 2005. <http://maude.cs.uiuc.edu/tools/scc/>.
192. J. Hendrix, H. Ohsaki, and M. Viswanathan. Propositional tree automata. In F. Pfenning, editor, *Term Rewriting and Applications, 17th International Conference, RTA 2006, Seattle, Washington, August 12 - 14, 2006, Proceedings*, volume 4098 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2006.
193. M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the Association for Computing Machinery*, 32(1):137–172, 1985.
194. M. Hidalgo-Herrero, A. Verdejo, and Y. Ortega-Mallén. Looking for Eden through Maude and its strategies. In F. J. López-Fraguas, editor, *V Jornadas sobre Programación y Lenguajes, PROLE 2005*, pages 13–23. Thomson, 2005.
195. E. Horowitz and S. Sahni. *Fundamentals of Data Structures in Pascal. Fourth Edition*. Computer Science Press, 1994.
196. J.-M. Hullot. Canonical forms and unification. In W. Bibel and R. A. Kowalski, editors, *Fifth Conference on Automated Deduction, CADE 1980, Les Arcs, France, July 8-11, 1980, Proceedings*, volume 87 of *Lecture Notes in Computer Science*, pages 318–334. Springer, 1980.
197. H. Ishikawa, K. Futatsugi, and T. Watanabe. An operational semantics of GAEA in CafeOBJ. In K. Futatsugi, J. A. Goguen, and J. Meseguer, editors, *OBJ/CafeOBJ/Maude Workshop at Formal Methods '99: Formal Specification, Proof, and Applications*, pages 213–227. Theta, 1999.
198. H. Ishikawa, J. Meseguer, T. Watanabe, K. Futatsugi, and H. Nakashima. On the semantics of GAEA — An object-oriented specification of a concurrent reflective language in rewriting logic. In *Proceedings IMSA'97*, pages 70–109. Information-Technology Promotion Agency, Japan, 1997.
199. E. B. Johnsen, O. Owe, and E. W. Axelsen. A runtime environment for concurrent objects with asynchronous method calls. In Martí-Oliet [228], pages 375–392. <http://www.sciencedirect.com/science/journal/15710661>.
200. J.-P. Jouannaud, C. Kirchner, and H. Kirchner. Incremental construction of unification algorithms in equational theories. In J. Díaz, editor, *Automata, Languages and Programming, 10th Colloquium, ICALP83, Barcelona, Spain, July 18-22, 1983, Proceedings*, volume 154 of *Lecture Notes in Computer Science*, pages 361–373. Springer, 1983.
201. D. Kapur and M. Subramaniam. New uses of linear arithmetic in automated theorem proving by induction. *Journal of Automated Reasoning*, 16(1-2):39–78, 1996.
202. D. Kapur and H. Zhang. An overview of rewrite rule laboratory (RRL). *J. Computer and Mathematics with Applications*, 29(2):91–114, 1995.

203. S. Kasera, S. Bhattacharyya, M. Keaton, D. Kiwior, J. Kurose, D. Towsley, and S. Zabele. Scalable fair reliable multicast using active services. Technical Report TR 99-44, University of Massachusetts, Amherst, CMPSCI, 1999.
204. M. Katelman and J. Meseguer. A rewriting semantics for ABEL with applications to hardware/software co-design and analysis. In Denker and Talcott [97], pages 95–109. <http://www.sciencedirect.com/science/journal/15710661>.
205. M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Press, 2000.
206. B. W. Kernighan and D. M. Ritchie. *The C Programming Language, Second Edition*. Prentice Hall, 1988.
207. C. Kirchner. Order-sorted equational unification. Technical Report 954, INRIA Lorraine & LORIA, Nancy, France, December 1988.
208. C. Kirchner and H. Kirchner, editors. *Proceedings Second International Workshop on Rewriting Logic and its Applications, WRLA'98, Pont-à-Mousson, France, September 1-4, 1998*, volume 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998. <http://www.sciencedirect.com/science/journal/15710661>.
209. A. Knapp. Generating rewrite theories from UML collaborations. In K. Futatsugi, A. T. Nakagawa, and T. Tamai, editors, *Cafe: An Industrial-Strength Algebraic Formal Method*, pages 97–120. Elsevier, 2000.
210. A. Knapp. *A Formal Approach to Object-Oriented Software Engineering*. Shaker Verlag, Aachen, Germany, 2001. PhD thesis, Institut für Informatik, Universität München, 2000.
211. W. Kolch. Meaningful relationships: The regulation of the Ras/ Raf/ MEK/ ERK pathway by protein interactions. *Biochemical Journal*, 351:289–305, 2000.
212. P. Kosiuczenko and M. Wirsing. Timed rewriting logic with application to object-oriented specification. Technical report, Institut für Informatik, Universität München, 1995.
213. D. Kotz and R. Gray. Mobile agents and the future of the internet. *ACM Operating Systems Review*, 33(3):7–13, 1999.
214. N. Kumar, K. Sen, J. Meseguer, and G. Agha. Probabilistic rewrite theories: Unifying models, logics and tools. Technical Report UIUCDCS-R-2003-2347, Computer Science Dept., University of Illinois at Urbana-Champaign, May 2003.
215. N. Kumar, K. Sen, J. Meseguer, and G. Agha. A rewriting based model of probabilistic distributed object systems. In E. Najm, U. Nestmann, and P. Stevens, editors, *Formal Methods for Open Object-Based Distributed Systems: 6th IFIP WG 6.1 International Conference, FMOODS 2003, Paris, France, November 19-21, 2003. Proceedings*, volume 2884 of *Lecture Notes in Computer Science*, pages 32–46. Springer, 2003.
216. C. Laneve and U. Montanari. Axiomatizing permutation equivalence in the λ -calculus. In H. Kirchner and G. Levi, editors, *Algebraic and Logic Programming, Third International Conference, Volterra, Italy, September 2-4, 1992, Proceedings*, volume 632 of *Lecture Notes in Computer Science*, pages 350–363. Springer, 1992.
217. C. Laneve and U. Montanari. Axiomatizing permutation equivalence. *Mathematical Structures in Computer Science*, 6:219–249, 1996.
218. D. Lange and M. Oshima. Seven good reasons for mobile agents. *Communications of the ACM*, 42:88–89, 1999.

219. P. Lescanne. From $\lambda\sigma$ to $\lambda\nu$, a journey through calculi of explicit substitutions. In H. Boehm, editor, *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, January 17–21, 1994*, pages 60–69. ACM, 1994.
220. M. Leucker and T. Noll. Rewriting logic as a framework for generic verification tools. In Futatsugi [164], pages 117–133. <http://www.sciencedirect.com/science/journal/15710661>.
221. E. Lien. Formal modeling and analysis of the NORM multicast protocol in Real-Time Maude. Master's thesis, Department of Linguistics, University of Oslo, April 2004. <http://wo.uio.no/as/WebObjects/theses.woa/wo/0.3.9>.
222. S. Lucas. Context-sensitive computations in functional and functional logic programs. *Journal of Functional and Logic Programming*, 1998(1):1–61, 1998.
223. S. Lucas. Termination of rewriting with strategy annotations. In R. Nieuwenhuis and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 8th International Conference, LPAR 2001, Havana, Cuba, December 3–7, 2001, Proceedings*, volume 2250 of *Lecture Notes in Artificial Intelligence*, pages 669–684. Springer, 2001.
224. S. Lucas. Context-sensitive rewriting strategies. *Information and Computation*, 178(1):294–343, 2002.
225. Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science*. International Series of Monographs on Computer Science. Oxford University Press, 1994.
226. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems — Specification*. Springer, 1992.
227. A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.
228. N. Martí-Oliet, editor. *Proceedings Fifth International Workshop on Rewriting Logic and its Applications, WRLA 2004, Barcelona, Spain, March 27–28, 2004*, volume 117 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2005. <http://www.sciencedirect.com/science/journal/15710661>.
229. N. Martí-Oliet and J. Meseguer. General logics and logical frameworks. In D. M. Gabbay, editor, *What is a Logical System?*, volume 4 of *Studies in Logic and Computation*, pages 355–392. Oxford University Press, 1994.
230. N. Martí-Oliet and J. Meseguer. Action and change in rewriting logic. In R. Pareschi and B. Fronhöfer, editors, *Dynamic Worlds: From the Frame Problem to Knowledge Management*, volume 12 of *Applied Logic Series*, pages 1–53. Kluwer Academic Publishers, 1999.
231. N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In D. M. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic, Second Edition, Volume 9*, pages 1–87. Kluwer Academic Publishers, 2002. First published as SRI Technical Report SRI-CSL-93-05, August 1993.
232. N. Martí-Oliet and J. Meseguer. Rewriting logic: roadmap and bibliography. *Theoretical Computer Science*, 285(2):121–154, 2002.
233. N. Martí-Oliet, J. Meseguer, and M. Palomino. Theoretical maps as algebraic simulations. In J. L. Fiadeiro, P. Mosses, and F. Orejas, editors, *Recent Trends in Algebraic Development Techniques, 17th International Workshop, WADT 2004, Barcelona, Spain, March 27–30, 2004, Revised Selected Papers*, volume 3423 of *Lecture Notes in Computer Science*, pages 126–143. Springer, 2005.

234. N. Martí-Oliet, J. Meseguer, and A. Verdejo. Towards a strategy language for Maude. In Martí-Oliet [228], pages 417–441. <http://www.sciencedirect.com/science/journal/15710661>.
235. N. Martí-Oliet, M. Palomino, and A. Verdejo. A tutorial on specifying data structures in Maude. In S. Lucas, editor, *Proceedings Fourth Spanish Conference on Programming and Computer Languages, PROLE 2004, Málaga, Spain, 10-12 November 2004*, volume 137(1) of *Electronic Notes in Theoretical Computer Science*, pages 105–132. Elsevier, 2005. <http://www.sciencedirect.com/science/journal/15710661>.
236. P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.
237. I. A. Mason and C. L. Talcott. A semantics preserving actor translation. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *Automata, Languages and Programming, 24th International Colloquium, ICALP'97, Bologna, Italy, July 1997, Proceedings*, volume 1256 of *Lecture Notes in Computer Science*, pages 369–378. Springer, 1997.
238. I. A. Mason and C. L. Talcott. Actor languages: Their syntax, semantics, translation, and equivalence. *Theoretical Computer Science*, 228(1), 1999.
239. I. A. Mason and C. L. Talcott. Simple network protocol simulation within Maude. In Futatsugi [164], pages 277–294. <http://www.sciencedirect.com/science/journal/15710661>.
240. I. A. Mason and C. L. Talcott. IOP: The InterOperability Platform & IMaude: An interactive extension of Maude. In Martí-Oliet [228], pages 315–333. <http://www.sciencedirect.com/science/journal/15710661>.
241. I. A. Mason and C. L. Talcott. The InterOperability Platform Manual, 2006. <http://mcs.une.edu.au/~iop/Data/Manual/manual.pdf>.
242. C. Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.
243. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
244. J. Meseguer. Multiparadigm logic programming. In H. Kirchner and G. Levi, editors, *Algebraic and Logic Programming, Third International Conference, Volterra, Italy, September 2-4, 1992, Proceedings*, volume 632 of *Lecture Notes in Computer Science*, pages 158–200. Springer, 1992.
245. J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. The MIT Press, 1993.
246. J. Meseguer, editor. *Proceedings First International Workshop on Rewriting Logic and its Applications, WRLA'96, Asilomar, California, September 3-6, 1996*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. <http://www.sciencedirect.com/science/journal/15710661>.
247. J. Meseguer. Rewriting logic as a semantic framework for concurrency: A progress report. In U. Montanari and V. Sassone, editors, *CONCUR'96: Concurrency Theory, 7th International Conference, Pisa, Italy, August 26-29, 1996, Proceedings*, volume 1119 of *Lecture Notes in Computer Science*, pages 331–372. Springer, 1996.
248. J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT'97, Tarquinia, Italy, June*

- 3–7, 1997, *Selected Papers*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1998.
249. J. Meseguer. Research directions in rewriting logic. In U. Berger and H. Schwichtenberg, editors, *Computational Logic, Proceedings of the NATO Advanced Study Institute on Computational Logic held in Marktoberdorf, Germany, July 29 – August 6, 1997*, volume 165 of *NATO ASI Series F: Computer and Systems Sciences*, pages 347–398. Springer, 1998.
250. J. Meseguer. A rewriting logic sampler. In D. V. Hung and M. Wirsing, editors, *Second International Colloquium on Theoretical Aspects of Computing (ICTAC 2005), Hanoi, Vietnam, October 2005*, volume 3722 of *Lecture Notes in Computer Science*, pages 1–28. Springer, 2005.
251. J. Meseguer and C. Braga. Modular rewriting semantics of programming languages. In C. Rattray, S. Maharaj, and C. Shankland, editors, *Algebraic Methodology and Software Technology: 10th International Conference, AMAST 2004, Stirling, Scotland, UK, July 12–16, 2004. Proceedings*, volume 3116 of *Lecture Notes in Computer Science*, pages 364–378. Springer, 2004.
252. J. Meseguer, K. Futatsugi, and T. Winkler. Using rewriting logic to specify, program, integrate, and reuse open concurrent systems of cooperating agents. In *Proceedings of the 1992 International Symposium on New Models for Software Architecture*, pages 61–106, Tokyo, Japan, November 1992. Research Institute of Software Engineering.
253. J. Meseguer and J. Goguen. Initiality, induction and computability. In M. Nivat and J. Reynolds, editors, *Algebraic Methods in Semantics*, pages 459–541. Cambridge University Press, 1985.
254. J. Meseguer, J. A. Goguen, and G. Smolka. Order-sorted unification. *J. Symbolic Computation*, 8(4):383–413, 1989.
255. J. Meseguer and U. Montanari. Mapping tile logic into rewriting logic. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT'97, Tarquinia, Italy, June 3–7, 1997, Selected Papers*, volume 1376 of *Lecture Notes in Computer Science*, pages 62–91. Springer, 1998.
256. J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational abstractions. In F. Baader, editor, *Automated Deduction - CADE-19. 19th International Conference on Automated Deduction, Miami Beach, FL, USA, July 28 - August 2, 2003, Proceedings*, volume 2741 of *Lecture Notes in Computer Science*, pages 2–16. Springer, 2003.
257. J. Meseguer, M. Palomino, and N. Martí-Oliet. Algebraic simulations. Submitted for publication. <http://maude.sip.ucm.es/~miguelpt/bibliography>, 2007.
258. J. Meseguer and G. Roşu. Rewriting logic semantics: From language specifications to formal analysis tools. In D. Basin and M. Rusinowitch, editors, *Automated Reasoning, Second International Joint Conference, IJCAR 2004, Cork, Ireland, July 4–8, 2004, Proceedings*, volume 3097 of *Lecture Notes in Artificial Intelligence*, pages 1–44. Springer, 2004.
259. J. Meseguer and G. Roşu. The rewriting logic semantics project. In P. D. Mosses and I. Ulidowski, editors, *Proceedings Second Workshop on Structural Operational Semantics, SOS 2005, Lisbon, Portugal, July 10, 2005*, volume 156(1) of *Electronic Notes in Theoretical Computer Science*, pages 27–56. Elsevier, 2006. <http://www.sciencedirect.com/science/journal/15710661>.

260. J. Meseguer and R. Sharykin. Specification and analysis of distributed object-based stochastic hybrid systems. In J. P. Hespanha and A. Tiwari, editors, *Hybrid Systems: Computation and Control, 9th International Workshop, HSCC 2006, Santa Barbara, CA, USA, March 29-31, 2006, Proceedings*, volume 3927 of *Lecture Notes in Computer Science*, pages 460–475. Springer, 2006.
261. J. Meseguer, M.-O. Stehr, and C. L. Talcott. Specifying the PLAN language in Maude. Slides available at <http://www-formal.stanford.edu/clt/Talks/00sep-utokyo-talk.ps.gz>, 2000.
262. J. Meseguer and C. Talcott. Semantic models for distributed object reflection. In B. Magnusson, editor, *ECOOP 2002 - Object-Oriented Programming: 16th European Conference Malaga, Spain, June 10-14, 2002. Proceedings*, volume 2374 of *Lecture Notes in Computer Science*, pages 1–36. Springer, 2002.
263. J. Meseguer and C. L. Talcott. Using rewriting logic to interoperate architectural description languages (I and II). Lectures at the Santa Fe and Seattle *DARPA-EDCS Workshops*, March and July 1997. <http://www-formal.stanford.edu/clt/ArpaNsf/adl-interop.html>.
264. J. Meseguer and C. L. Talcott. A partial order event model for concurrent objects. In J. C. M. Baeten and S. Mauw, editors, *CONCUR'99, Concurrency Theory, 10th International Conference Eindhoven, The Netherlands, August 24-27, 1999, Proceedings*, volume 1664 of *Lecture Notes in Computer Science*, pages 415–430. Springer, 1999.
265. J. Meseguer and P. Thati. Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols. In Martí-Oliet [228], pages 153–182. <http://www.sciencedirect.com/science/journal/15710661>.
266. J. Meseguer and P. Thati. Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols. *Higher-Order and Symbolic Computation*, 20(1-2):123–160, 2007.
267. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
268. P. D. Mosses. Semantics, modularity, and rewriting logic. In Kirchner and Kirchner [208], pages 404–421. <http://www.sciencedirect.com/science/journal/15710661>.
269. P. D. Mosses. Logical specification of operational semantics. In J. Flum and M. Rodríguez-Artalejo, editors, *Computer Science Logic, 13th International Workshop, CSL'99, 8th Annual Conference of the EACSL, Madrid, Spain, September 20-25, 1999, Proceedings*, volume 1683 of *Lecture Notes in Computer Science*, pages 32–49. Springer, 1999.
270. P. D. Mosses. Modular structural operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:195–228, 2004.
271. E. Najm and J.-B. Stefani. A formal operational semantics for the ODP computational model with signals, explicit binding, and reactive objects. Manuscript, ENST, Paris, France, 1994.
272. E. Najm and J.-B. Stefani. A formal semantics for the ODP computational model. *Computer Networks and ISDN Systems*, 27:1305–1329, 1995.
273. E. Najm and J.-B. Stefani. Computational models for open distributed systems. In H. Bowman and J. Derrick, editors, *Proceedings Second IFIP Conference on Formal Methods for Open Object-Based Distributed Systems, FMOODS'97, July 21-23, 1997, Canterbury, Kent, UK*, pages 157–176. Chapman & Hall, 1997.

274. S. Nakajima. Encoding mobility in CafeOBJ: An exercise of describing mobile code-based software architecture. In *Proceedings of the CafeOBJ Symposium '98, Numazu, Japan*. CafeOBJ Project, April 1998.
275. S. Nakajima and K. Futatsugi. An object-oriented modeling method for algebraic specifications in CafeOBJ. In *Proceedings, 19th International Conference on Software Engineering*, pages 34–44, Boston, Massachusetts, May 1997. IEEE Computer Society Press.
276. P. Naumov, M.-O. Stehr, and J. Meseguer. The HOL/NuPRL proof translator — A practical approach to formal interoperability. In R. J. Boulton and P. B. Jackson, editors, *Theorem Proving in Higher Order Logics, 14th International Conference, TPHOLs'2001, Edinburgh, Scotland, UK, September 3–6, 2001, Proceedings*, volume 2152 of *Lecture Notes in Computer Science*, pages 329–345. Springer, 2001.
277. R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 371–443. Elsevier and MIT Press, 2001.
278. Object Management Group. Object Constraint Language specification, 2004. http://www.omg.org/technology/documents/modeling_spec_catalog.htm#OCL.
279. Object Management Group. Unified Modeling Language specification, 2004. http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML.
280. Object Management Group. Mof 2.0 query/views/transformations final adopted specification, 2005. http://www.omg.org/technology/documents/modeling_spec_catalog.htm#MOF_QVT.
281. H. Ohsaki. Beyond regularity: Equational tree automata for associative and commutative theories. In L. Fribourg, editor, *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings*, volume 2142 of *Lecture Notes in Computer Science*, pages 539–553. Springer, 2001.
282. C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.
283. P. C. Ölveczky. *Specification and Analysis of Real-Time and Hybrid Systems in Rewriting Logic*. PhD thesis, University of Bergen, Norway, 2000. <http://maude.cs.uiuc.edu/papers/>.
284. P. C. Ölveczky and M. Caccamo. Formal simulation and analysis of the CASH scheduling algorithm in Real-Time Maude. In L. Baresi and R. Heckel, editors, *Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2006, Held as Part of ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings*, volume 3922 of *Lecture Notes in Computer Science*, pages 357–372. Springer, 2006.
285. P. C. Ölveczky, M. Keaton, J. Meseguer, C. L. Talcott, and S. Zabele. Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001, Held as Part of ETAPS 2001, Genova, Italy, April 2001, Proceedings*, volume 2029 of *Lecture Notes in Computer Science*, pages 333–347. Springer, 2001.
286. P. C. Ölveczky, P. Kosiuczenko, and M. Wirsing. An object-oriented algebraic steam-boiler control specification. In J.-R. Abrial, E. Börger, and H. Langmaack, editors, *Formal Methods for Industrial Applications: Specifying and*

- Programming the Steam Boiler Control*, volume 1165 of *Lecture Notes in Computer Science*, pages 379–402. Springer, 1996.
287. P. C. Ölveczky and J. Meseguer. Real-Time Maude: A tool for simulating and analyzing real-time and hybrid systems. In Futatsugi [164], pages 361–382. <http://www.sciencedirect.com/science/journal/15710661>.
288. P. C. Ölveczky and J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science*, 285(2):359–405, 2002.
289. P. C. Ölveczky and J. Meseguer. Specification and analysis of real-time systems using Real-Time Maude. In T. Margaria and M. Wermelinger, editors, *Fundamental Approaches to Software Engineering, 7th International Conference, FASE 2004, Held as Part of ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2984 of *Lecture Notes in Computer Science*, pages 354–358. Springer, 2004.
290. P. C. Ölveczky and J. Meseguer. Abstraction and completeness for Real-Time Maude. In Denker and Talcott [97], pages 128–153. <http://www.sciencedirect.com/science/journal/15710661>.
291. P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20(1-2):161–196, 2007.
292. P. C. Ölveczky, J. Meseguer, and C. L. Talcott. Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. *Formal Methods in System Design*, 29(3):253–293, 2006.
293. P. C. Ölveczky and S. Thorvaldsen. Formal modeling and analysis of wireless sensor network algorithms in Real-Time Maude. In *20th International Parallel and Distributed Processing Symposium, IPDPS 2006, Rhodes Island, Greece, April 2006*. IEEE Computer Society Press, 2006.
294. M. Palomino. *Reflexión, Abstracción y Simulación en la Lógica de Reescritura*. PhD thesis, Universidad Complutense de Madrid, Spain, Mar. 2005.
295. M. Palomino, N. Martí-Oliet, and A. Verdejo. Playing with Maude. In S. Abdennadher and C. Ringeissen, editors, *Proceedings Fifth International Workshop on Rule-Based Programming, RULE 2004, Aachen, Germany*, volume 124(1) of *Electronic Notes in Theoretical Computer Science*, pages 3–23. Elsevier, 2005. <http://www.sciencedirect.com/science/journal/15710661>.
296. R. Peña. *Diseño de Programas. Formalismo y Abstracción. Tercera Edición*. Prentice Hall, 2005.
297. G. E. Peterson and M. E. Stickel. Complete sets of reductions for some equational theories. *Journal of the ACM*, 28(2):233–264, 1981.
298. K. Petersson, J. Smith, and B. Nordström. *Programming in Martin-Löf's Type Theory. An Introduction*. International Series of Monographs on Computer Science. Oxford: Clarendon Press, 1990.
299. I. Pita and N. Martí-Oliet. A Maude specification of an object oriented database model for telecommunication networks. In Meseguer [246], pages 404–422. <http://www.sciencedirect.com/science/journal/15710661>.
300. I. Pita and N. Martí-Oliet. Using reflection to specify transaction sequences in rewriting logic. In J. L. Fiadeiro, editor, *Recent Trends in Algebraic Development Techniques, 13th International Workshop, WADT'98, Lisbon, Portugal, April 2–4, 1998, Selected Papers*, volume 1589 of *Lecture Notes in Computer Science*, pages 261–276. Springer, 1999.
301. I. Pita and N. Martí-Oliet. A Maude specification of an object-oriented model for telecommunication networks. *Theoretical Computer Science*, 285(2):407–439, 2002.

302. G. Plotkin. Building-in equational theories. In B. Meltzer and D. Michie, editors, *Machine Intelligence 7, Proceedings of the Seventh Annual Machine Intelligence Workshop, Edinburgh, 1971*, pages 73–90. Edinburgh University Press, 1972.
303. G. D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004. Previously published as technical report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.
304. R. Pollack. Implicit syntax. In G. Huet and G. Plotkin, editors, *Proceedings of the First Workshop on Logical Frameworks*, May 1990.
305. R. Pollack. Closure under alpha-conversion. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs: International Workshop TYPES'93, Nijmegen, May 1993, Selected Papers*, volume 806 of *Lecture Notes in Computer Science*, pages 313–332. Springer, 1993.
306. D. Porter. An interpreter for **JLambda**. A thesis submitted for the degree of BCompSci (Honours), University of New England, 2004. <http://mcs.une.edu.au/~iop/Data/Papers/dporter-honours-thesis.pdf>.
307. L. Pottier. Minimal solutions of linear diophantine systems: bounds and algorithms. In R. V. Book, editor, *Rewriting Techniques and Applications, 4th International Conference, RTA-91, Como, Italy, April 10-12, 1991, Proceedings*, volume 488 of *Lecture Notes in Computer Science*, pages 162–173. Springer, 1991.
308. M. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley and Sons, 1994.
309. J. F. Quesada. *The SCP Parsing Algorithm Based on Syntactic Constraint Propagation*. PhD thesis, Universidad de Sevilla, Spain, 1997.
310. J. F. Quesada. The Maude parser: Parsing and meta-parsing β -extended context-free grammars. Technical Report, SRI International, Computer Science Laboratory, 1999.
311. F. Rabhi and G. Lapalme. *Algorithms. A Functional Programming Approach*. Addison-Wesley, 1999.
312. U. S. Reddy. Narrowing as the operational semantics of functional languages. In *Proceedings of the 1985 Second Symposium on Logic Programming, Boston, Massachusetts, July 15-18, 1985*, pages 138–151. IEEE Computer Society Press, 1985.
313. S. Reich. Implementing and extending the MSR crypto-protocol specification language. Diplomarbeit. Universität Hamburg, Fachbereich Informatik, April 2006.
314. W. Reisig. *Petri Nets*. Springer, 1985.
315. J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12(1):23–41, 1965.
316. C. Rocha and J. Meseguer. Theorem proving modulo based on boolean equational procedures. In R. Berghammer, B. Möller, and G. Struth, editors, *Relations and Kleene Algebra in Computer Science, 10th International Conference on Relational Methods in Computer Science, and 5th International Conference on Applications of Kleene Algebra, RelMiCS/AKA 2008, Frauenwörth, Germany, April 7-11, 2008. Proceedings*, volume 4988 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 2008.

317. D. E. Rodríguez. Case studies in the specification and analysis of protocols in Maude. In Futatsugi [164], pages 257–275. <http://www.sciencedirect.com/science/journal/15710661>.
318. F. Rosa-Velardo, C. Segura, and A. Verdejo. Typed mobile ambients in Maude. In H. Cirstea and N. Martí-Oliet, editors, *Proceedings 6th International Workshop on Rule-Based Programming, RULE 2005, Nara, Japan*, volume 147 of *Electronic Notes in Theoretical Computer Science*, pages 135–161. Elsevier, 2006. <http://www.sciencedirect.com/science/journal/15710661>.
319. G. Roşu and K. Havelund. Generating optimal monitors from temporal formulae. Technical report, Research Institute for Advanced Computer Science, 2001.
320. G. Rosu, R. P. Venkatesan, J. Whittle, and L. Leustean. Certifying optimality of state estimation programs. In W. A. H. Jr. and F. Somenzi, editors, *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *Lecture Notes in Computer Science*, pages 301–314. Springer, 2003.
321. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual, Second Edition*. Addison Wesley, 2004.
322. R. Sasse. Taclets vs. rewriting logic – relating semantics of Java. Master’s thesis, Fakultät für Informatik, Universität Karlsruhe, Germany, May 2005. Technical Report in Computing Science No. 2005-16, <http://www.ubka.uni-karlsruhe.de/cgi-bin/psview?document=ira/2005/16>.
323. R. Sasse and J. Meseguer. Java+ITP: A verification tool based on Hoare logic and algebraic semantics. Technical Report UIUCDCS-R-2006-2685, Department of Computer Science, University of Illinois at Urbana-Champaign, USA, February 2006.
324. R. Sasse and J. Meseguer. Java+ITP: A verification tool based on Hoare logic and algebraic semantics. In Denker and Talcott [97], pages 202–222. <http://www.sciencedirect.com/science/journal/15710661>.
325. K. Schmidt. LoLA: Low Level Petri net Analyzer, 2004. <http://www.informatik.hu-berlin.de/~kschmidt/lola.html>.
326. M. Schmidt-Schauss. *Computational Aspects of Order-Sorted Logic with Term Declarations*, volume 395 of *Lecture Notes in Computer Science*. Springer, 1989.
327. R. Segala. *Modelling and Verification of Randomized Distributed Real Time Systems*. PhD thesis, Massachusetts Institute of Technology, 1995.
328. K. Sen, M. Viswanathan, and G. Agha. On statistical model checking of stochastic systems. In K. Etessami and S. K. Rajamani, editors, *Computer Aided Verification: 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005. Proceedings*, volume 3576 of *Lecture Notes in Computer Science*, pages 266–280. Springer, 2005.
329. D. O. Shklarsky, N. N. Chentzov, and I. M. Yaglom. *The USSR Olympiad Problem Book*. Dover, 1993.
330. J. Siekmann and P. Szabó. A Noetherian and confluent rewrite system for idempotent semigroups. *Semigroup Forum*, 25:83–110, 1982.
331. G. Smolka, W. Nutt, J. Goguen, and J. Meseguer. Order-sorted equational computation. In M. Nivat and H. Ait-Kaci, editors, *Resolution of Equations in Algebraic Structures*, volume 2, pages 297–367. Academic Press, 1989.
332. L. J. Steggle. Rewriting logic and Elan: Prototyping tools for Petri nets with time. In J.-M. Colom and M. Koutny, editors, *Applications and Theory*

- of *Petri Nets 2001, 22nd International Conference, ICATPN 2001, Newcastle upon Tyne, UK, June 25–29, 2001, Proceedings*, volume 2075 of *Lecture Notes in Computer Science*, pages 363–381. Springer, 2001.
333. L. J. Steggle and P. Kosiuczenko. A timed rewriting logic semantics for SDL: a case study of the alternating bit protocol. In Kirchner and Kirchner [208], pages 295–316. <http://www.sciencedirect.com/science/journal/15710661>.
334. M.-O. Stehr. CINNI — A generic calculus of explicit substitutions and its application to λ -, ζ - and π -calculi. In Futatsugi [164], pages 71–92. <http://www.sciencedirect.com/science/journal/15710661>.
335. M.-O. Stehr. A rewriting semantics for algebraic nets. In C. Girault and R. Valk, editors, *Petri Nets for System Engineering – A Guide to Modelling, Verification, and Applications*, pages 318–338. Springer, 2001.
336. M.-O. Stehr. *Programming, Specification, and Interactive Theorem Proving — Towards a Unified Language based on Equational Logic, Rewriting Logic, and Type Theory*. PhD thesis, Universität Hamburg, Fachbereich Informatik, Germany, 2002. <http://www.sub.uni-hamburg.de/disse/810/>.
337. M.-O. Stehr. The open calculus of constructions (Part I): An equational type theory with dependent types for programming, specification, and interactive theorem proving. *Fundamenta Informaticae*, 68(1-2):131–174, 2005.
338. M.-O. Stehr. The open calculus of constructions (Part II): An equational type theory with dependent types for programming, specification, and interactive theorem proving. *Fundamenta Informaticae*, 68(3):249–288, 2005.
339. M.-O. Stehr, I. Cervesato, and S. Reich. An execution environment for the MSR cryptoprotocol specification language. <http://formal.cs.uiuc.edu/stehr/msr.html>, 2004.
340. M.-O. Stehr and J. Meseguer. Pure type systems in rewriting logic. In *Proceedings of LFM'99: Workshop on Logical Frameworks and Meta-languages*, Paris, France, September 28, 1999. <http://www.site.uottawa.ca/~afelty/LFM99/index.html>.
341. M.-O. Stehr and J. Meseguer. Pure type systems in rewriting logic: Specifying typed higher-order languages in a first-order logical framework. In O. Owe, S. Krogdahl, and T. Lyche, editors, *From Object-Oriented to Formal Methods: Essays in Memory of Ole-Johan Dahl*, volume 2635 of *Lecture Notes in Computer Science*, pages 334–375. Springer, 2004.
342. M.-O. Stehr, J. Meseguer, and P. C. Ölveczky. Rewriting logic as a unifying framework for Petri nets. In H. Ehrig, G. Juhas, J. Padberg, and G. Rozenberg, editors, *Unifying Petri Nets*, volume 2128 of *Lecture Notes in Computer Science*, pages 250–303. Springer, 2001.
343. M.-O. Stehr, P. Naumov, and J. Meseguer. A proof-theoretic approach to the HOL-Nuprl connection with applications to proof translation. In M. Cerioli, P. D. Mosses, and G. Reggio, editors, *Preliminary Proceedings WADT/CoFI'01, 15th International Workshop on Algebraic Development Techniques and General Workshop of the CoFI WG*, 2001. <http://formal.cs.uiuc.edu/stehr/holnuprl-ext.ps.gz>.
344. M.-O. Stehr and C. Talcott. PLAN in Maude: Specifying an active network programming language. In Gadducci and Montanari [169], pages 221–241. <http://www.sciencedirect.com/science/journal/15710661>.
345. M.-O. Stehr and C. L. Talcott. Practical techniques for language design and prototyping. In J. L. Fiadeiro, U. Montanari, and M. Wirsing, editors, *Foundations of Global Computing*, number 05081 in Dagstuhl Seminar Proceedings.

- Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2006. <http://drops.dagstuhl.de/opus/volltexte/2006/300>.
346. W. J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton, 1994.
347. M. E. Stickel. A complete unification algorithm for associative-commutative functions. In *Advanced Papers of the Fourth International Joint Conference on Artificial Intelligence, Tbilisi, Georgia, USSR, 3-8 September 1975*, pages 71–76, 1975.
348. C. L. Talcott. An actor rewriting theory. In Meseguer [246], pages 360–383. <http://www.sciencedirect.com/science/journal/15710661>.
349. C. L. Talcott. Composable semantic models for actor theories. In M. Abadi and T. Ito, editors, *Theoretical Aspects of Computer Science, Third International Symposium, TACS'97, Sendai, Japan, September 23–26, 1997, Proceedings*, volume 1281 of *Lecture Notes in Computer Science*, pages 321–364. Springer, 1997.
350. C. L. Talcott. Interaction semantics for components of distributed systems. In E. Najm and J.-B. Stefani, editors, *Proceedings IFIP Conference on Formal Methods for Open Object-Based Distributed Systems, FMOODS'96*, pages 154–169. Chapman & Hall, 1997.
351. C. L. Talcott. Composable semantic models for actor theories. *Higher-Order and Symbolic Computation*, 11(3):281–343, 1998.
352. C. L. Talcott. Towards a toolkit for actor system specification. In T. Rus, editor, *Algebraic Methodology and Software Technology, 8th International Conference, AMAST 2000, Iowa City, Iowa, USA, May 20–27, 2000, Proceedings*, volume 1816 of *Lecture Notes in Computer Science*, pages 391–406. Springer, 2000.
353. C. L. Talcott. Actor theories in rewriting logic. *Theoretical Computer Science*, 285(2):441–485, 2002.
354. C. L. Talcott and D. L. Dill. The pathway logic assistant. In G. Plotkin, editor, *Third International Workshop on Computational Methods in Systems Biology*, pages 228–239, 2005.
355. C. L. Talcott, S. Eker, M. Knapp, P. Lincoln, and K. Laderoute. Pathway logic modeling of protein functional domains in signal transduction. In R. B. Altman, A. K. Dunker, L. Hunter, and T. E. Klein, editors, *Proceedings of the 9th Pacific Symposium on Biocomputing (PSB 2004), Fairmont Orchid, Hawaii, USA, January 6-10, 2004*, pages 568–580, January 2004. <http://helix-web.stanford.edu/psb04/>.
356. P. Thati and J. Meseguer. Complete symbolic reachability analysis using back-and-forth narrowing. In J. L. Fiadeiro, N. Harman, M. Roggenbach, and J. J. M. M. Rutten, editors, *Algebra and Coalgebra in Computer Science: First International Conference, CALCO 2005, Swansea, UK, September 3-6, 2005, Proceedings*, volume 3629 of *Lecture Notes in Computer Science*, pages 379–394. Springer, 2005.
357. P. Thati, K. Sen, and N. Martí-Oliet. An executable specification of asynchronous pi-calculus semantics and may testing in Maude 2.0. In Gadducci and Montanari [169], pages 242–262. <http://www.sciencedirect.com/science/journal/15710661>.
358. The Open Group. The Single UNIX Specification Version 3 Homepage, 2004. <http://www.unix-systems.org>.

359. S. Thorvaldsen and P. C. Ölveczky. Formal modeling and analysis of the OGDC wireless sensor network algorithm in Real-Time Maude. <http://www.ifi.uio.no/RealTimeMaude/OGDC/>, 2005.
360. A. P. Tomás. *On Solving Linear Diophantine Constraints*. PhD thesis, Universidade do Porto, 1997.
361. A. Toval and J. L. Fernández. Formally modeling UML and its evolution: A holistic approach. In S. F. Smith and C. L. Talcott, editors, *Proceedings IFIP Conference on Formal Methods for Open Object-Based Distributed Systems IV, FMOODS 2000, September 6–8, 2000, Stanford, California, USA*, pages 183–206. Kluwer Academic Publishers, 2000.
362. A. Toval and J. L. Fernández. Improving system reliability via rigorous software modeling: The UML case. In *Proceedings IEEE Aerospace Conference*, volume 6, pages 6–17. IEEE Press, 2001.
363. E. Tushkanova, C. Ringeissen, A. Giorgetti, and O. Kouchnarenko. Automatic decidability: A schematic calculus for theories with counting operators. In F. van Raamsdonk, editor, *24th International Conference on Rewriting Techniques and Applications, RTA 2013, June 24–26, 2013, Eindhoven, The Netherlands*, volume 21 of *LIPICs*, pages 303–318. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.
364. A. Verdejo. Building tools for LOTOS symbolic semantics in Maude. In D. Peled and M. Vardi, editors, *Formal Techniques for Networked and Distributed Systems — FORTE 2002, 22nd IFIP WG 6.1 International Conference Houston, Texas, USA, November 2002 Proceedings*, volume 2529 of *Lecture Notes in Computer Science*, pages 292–397. Springer, 2002.
365. A. Verdejo. *Maude como Marco Semántico Ejecutable*. PhD thesis, Universidad Complutense de Madrid, Spain, 2003.
366. A. Verdejo and N. Martí-Oliet. Executing and verifying CCS in Maude. Technical Report 99-00, Departamento de Sistemas Informáticos y Programación, Universidad Complutense de Madrid, February 2000. <http://maude.cs1.sri.com/casestudies/ccs>.
367. A. Verdejo and N. Martí-Oliet. Executing E-LOTOS processes in Maude. In H. Ehrig, M. Grosse-Rhode, and F. Orejas, editors, *INT 2000, Integration of Specification Techniques with Applications in Engineering, Extended Abstracts*, pages 49–53, 2000. Technical report 2000/04, Technische Universität Berlin.
368. A. Verdejo and N. Martí-Oliet. Implementing CCS in Maude. In T. Bolognesi and D. Latella, editors, *Formal Methods For Distributed System Development. FORTE/PSTV 2000 IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XIII) and Protocol Specification, Testing and Verification (PSTV XX) October 10–13, 2000, Pisa, Italy*, volume 183 of *International Federation for Information Processing*, pages 351–366. Kluwer Academic Publishers, 2000.
369. A. Verdejo and N. Martí-Oliet. Implementing CCS in Maude 2. In Gadducci and Montanari [169], pages 263–281. <http://www.sciencedirect.com/science/journal/15710661>.
370. A. Verdejo and N. Martí-Oliet. Two case studies of semantics execution in Maude: CCS and LOTOS. *Formal Methods in System Design*, 27:113–172, 2005.
371. A. Verdejo and N. Martí-Oliet. Executable structural operational semantics in Maude. *Journal of Logic and Algebraic Programming*, 67(1-2):226–293, 2006.

372. A. Verdejo, I. Pita, and N. Martí-Oliet. The leader election protocol of IEEE 1394 in Maude. In Futatsugi [164], pages 385–406. <http://www.sciencedirect.com/science/journal/15710661>.
373. E. Viola. E-unifiability via narrowing. In A. Restivo, S. R. D. Rocca, and L. Roversi, editors, *Theoretical Computer Science, 7th Italian Conference, ICTCS 2001, Torino, Italy, October 4-6, 2001, Proceedings*, volume 2202 of *Lecture Notes in Computer Science*, pages 426–438. Springer, 2001.
374. P. Viry. Rewriting: An effective model of concurrency. In C. Halatsis, D. Maritsas, G. Philokyprou, and S. Theodoridis, editors, *PARLE'94 Parallel Architectures and Languages Europe, 6th International PARLE Conference, Athens, Greece, July 4-8, 1994, Proceedings*, volume 817 of *Lecture Notes in Computer Science*, pages 648–660. Springer, 1994.
375. P. Viry. Input/output for ELAN. In Meseguer [246], pages 51–64. <http://www.sciencedirect.com/science/journal/15710661>.
376. P. Viry. Adventures in sequent calculus modulo equations. In Kirchner and Kirchner [208], pages 367–378. <http://www.sciencedirect.com/science/journal/15710661>.
377. P. Viry. Equational rules for rewriting logic. *Theoretical Computer Science*, 285(2):487–517, 2002.
378. M. Vittek. *ELAN: Un Cadre Logique pour le Prototypage de Langages de Programmation avec Contraintes*. PhD thesis, Université Henri Poincaré – Nancy I, November 1994.
379. B.-Y. Wang, J. Meseguer, and C. A. Gunter. Specification and formal analysis of a PLAN algorithm in Maude. In P.-A. Hsiung, editor, *Proceedings International Workshop on Distributed System Validation and Verification, Taipei, Taiwan*, pages 49–56, April 2000.
380. J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA, Second Edition*. Addison Wesley, 2003.
381. M. A. Weiss. *Data Structures and Problem Solving Using Java*. Addison-Wesley, 1998.
382. M. Wirsing and A. Knapp. A formal approach to object-oriented software engineering. In Meseguer [246], pages 321–359. <http://www.sciencedirect.com/science/journal/15710661>.
383. M. Wirsing and A. Knapp. A formal approach to object-oriented software engineering. *Theoretical Computer Science*, 285(2):519–560, 2002.
384. H. Zhang, D. Kapur, and M. S. Krishnamoorthy. A mechanizable induction principle for equational specifications. In E. L. Lusk and R. A. Overbeek, editors, *9th International Conference on Automated Deduction, Argonne, Illinois, USA, May 23-26, 1988, Proceedings*, volume 310 of *Lecture Notes in Computer Science*, pages 162–181. Springer, 1988.

Subject Index

- abort 152 (debugger), 774 (debugger),
814 (debugger)
- abstraction *see* model checking
- ansi-color 803
- AProVe 386
- array 303–304
- ASF+SDF 67n, 92n, 188, 578
- assoc 65, 71, 213, 789, 791, 792, 795,
797, 805
- associative *see* assoc
- attribute 70–90
 - equational 70–72
 - statement 91–97

- batch 803
- Boolean value 236–241
- break point 774, 788
- break select 774, 813
- bubble 577

- CafeOBJ X
- CCS 749
- cd 815
- ceq 66
- ChC 17, 387, 405, 487, 630, 738
- CHR 188
- Church-Rosser 101
 - context-sensitive 103
 - ground 101
 - modulo 102
- class 667 (Full Maude)
 - inheritance 668 (Full Maude)
 - multiple 668 (Full Maude)
- Clear 189

- cmb 66
- coherence 140, 181, 466
 - checking 141
 - completion 629 (Full Maude)
 - ground 141
- collapse theory 489, 794
- comm 65, 71, 793
- comment 37, 823
 - multiline 823
- commutative *see* comm
- config 344, 349, 354, 355
- configuration *see* config
- confluence 100
- connected component *see* subsort
relation
- constant 47
 - metarepresentation 426
 - qualified 50
- constructor 73–77
 - non-free 74
- constructor *see* ctor
- cont *see* continue
- continue 146, 805
- core dumped 790
- Core Maude 20
- counter 245–248
- cq *see* ceq
- CRC 17, 385, 405, 409, 487, 630, 736
- crl 138
- ctor 73, 81

- deadlock freedom 380, 403, 410,
526–531, 642 (Full Maude)

- debug **erewrite** 814
- debug **frewrite** 814
- debug **reduce** 814
- debug **rewrite** 814
- debugger 152, 248, 774–777
- debugging 612 (Full Maude), 763, *see*
 - tracing, term coloring
- dependent type 255
- descent function 424, 433–461
- design 1–11
- dictionary 328
- Diophantine equation solver 305
- distributed dataset 357
- ditto 82
- do **clear memo** 89

- ELAN X, 67n, 188
- eof 816
- eq 64
- equation 64–65
 - executable 64, 98
 - metarepresentation 427
- equational condition 66–70
 - abbreviated Boolean equation 66, 68
 - matching equation 67, 68
 - ordinary equation 68
 - satisfaction 68
- equational simplification 100
 - modulo 65, 71, 72
 - sharing 113, 113n
- erew** *see* **erewrite**
- erewrite** 246, 365, 805
- error
 - constant 121
 - propagation 120
 - supersort 120
- evaluation strategy *see* strategy
- ex** *see* **extending**
- explicit substitution calculus 228
- expressiveness 5–9
- extending** 190, 192
- external object 365–376

- finite variant property 493
 - checking 494
- fmod** 43, 63
- forest 326
- format** 78, 198, 774

- foundation 11–14
- frew** *see* **frewrite**
- frewrite** 142, 146, 147, 246, 804
 - metarepresentation *see* descent
 - function
- frozen argument *see* strategy
- frozen** 89
- fth** 202
- Full Maude 20, 210n, 216n, 219n, 343, 410, 422, 607–648
 - differences with Core Maude 632
 - extending Full Maude 634

- gather** 55, 198
- get variants** 495–501, 808

- help** 803

- id** 71, 794
- idem** 71, 789, 794, 795
- idempotent** *see* **idem**
- identifier 41–42, 44, 47, 48, 51
 - escape character 41
 - nonprinting characters 42
 - quoted *see* quoted identifier
 - special 41
- identity** *see* **id**
- IMaude *see* Interactive Maude
- in** 35–37, 815
- inc** *see* **including**
- including** 190, 193, 194, 204
- initial algebra 70
- interaction 33–37, 608 (Full Maude)
 - interrupt 152
 - metarepresentation *see* read-eval-print loop
- interactive** 804
- Interactive Maude 585–603
- invariant 377–378
 - model checking of 378–382
 - bounded 382–384
 - violation 486
- iter** 73, 105, 109, 213, 241, 797, 805
- iterated** *see* **iter**
- ITP 342, 407, 733
- ITP/OCL 756

- Java+ITP 754
- JavaFAN 753

- kind 48–50
 - canonical representation 49
 - metarepresentation 425
- Kripke structure 391
 - associated to a module 392
- label 91, 136
- lambda calculus 228–234
- left id 71, 795
- linear temporal logic 389
 - model checking *see* model checking
 - satisfiability 412
- linear.maude 305
- list
 - circular 169
 - from set 284–285
 - generalized 285–287
 - non-empty 167
 - of sets 219
 - parameterized 278–280, 317
 - sorted 319
 - sortable 291–298
 - strict weak order 291
 - total preorder 295
 - sorted 226
- load 35, 37, 816
- loop 576, 806
- LOTOS 749
- ls 815
- LTL *see* linear temporal logic
- machine-int.maude 235
- map 301–303
- match 111, 807
 - metarepresentation *see* descent function
- matching 99
 - modulo 102–109, 140, 795
 - with extension 104, 796
- Maude-NPA 485
- MAUDE.LIB 36, 37
- mb 65
- membership 65–66, 797
 - metarepresentation 427
- membership equational logic 63, 319
- memo 86
- memoization 86–89
 - table size 87
- message 666 (Full Maude)
- message 354, 356
- metadata 82, 91, 136
- MMT 748
- Mobile Maude 376, 533–570
- mod 43, 135
- model checker
 - implementation 397
 - procedure 396
- model checking 396–403
 - abstraction 384–388, 403–412
 - tool support 743
 - logical 486, 651
- model-checker.maude 389, 393, 397, 399, 412
- module 42–43
 - algebra 189
 - database 613 (Full Maude)
 - expression 189, 203, *see* module operation
 - functional 42, 63–121
 - admissible 98–99
 - mathematical semantics 63, 65, 69, 70
 - operational semantics 63, 65, 103
 - hierarchy *see* module importation
 - importation 190–196, 203, 204, 223
 - extending 192–193
 - implicit 191
 - including 193–194
 - protecting 191–192
 - metarepresentation 427
 - object-based 344–355
 - asynchronous 344
 - configuration 344
 - fairness 351
 - synchronous 344
 - uniqueness 351
 - object-oriented 666 (Full Maude)
 - as system module 704 (Full Maude)
 - operation 688 (Full Maude)
 - parameterized 685 (Full Maude)
 - operation 189
 - deadlock freedom 642 (Full Maude)
 - instantiation 189, 202, 218–224
 - metarepresentation 431
 - power 618 (Full Maude)
 - renaming 189, 198–201, 224

- summation 189, 197–198, 224
- tuple 617 (Full Maude)
- parameterized 202, 214–218
 - bound parameter 223
 - free parameter 223
 - interface 214
 - metarepresentation 430
 - parameter 202
 - parameter label 214
 - parameter theory 214, 216, 217
- predefined 235–304
- signature 42
 - extended *see* parsing
- system 43, 135–162
 - admissible 139
 - mathematical semantics 135, 142
- monoid 203
 - commutative 204
- monomial 220
- MSCP IX, 53
- msg** *see* **message**
- MSR 750
- MTT 17, 385, 386, 410, 630, 735
- multiset 164, 172, 173, 176, 177, 184
 - parameterized 322
- narrowing 483, 502, 649–662
 - based unification 483–484
 - completeness 652–653
 - folding variant 484, 503, 651
 - modulo axioms 483
 - search 654 (Full Maude)
 - with extra variables 657 (Full Maude)
 - with rules 485
 - with simplification 653
- no-advise** 804
- no-ansi-color** 803
- no-banner** 804
- no-mixfix** 803
- no-prelude** 803
- no-tecla** 803
- no-wrap** 804
- nonexec** 64, 92, 136
- number
 - floating-point 260–264
 - integer 248–251
 - machine 252–255
 - natural 241–245
 - random 245–248
 - rational 255–260
 - string conversion 268–270
- OBJ 189, 202
- obj** *see* **object**
- OBJ3 IX, 5n, 7, 83, 189, 215
- object 345, *see* module object-based, 666 (Full Maude)
 - communication 542 (Mobile Maude)
 - mobile 536 (Mobile Maude)
- object** 345, 354, 356
- op** 46, 49
- open calculus of constructions 743–748
- operation
 - metalevel *see* descent function
 - partial 49, 118–121
 - using error supersort 120
 - using subsort 119
 - total 49
- operator 46–48, 51
 - arity 46
 - at the kind level 49, 799
 - at the sort level 797
 - built-in 90, 213
 - coarity 46
 - derived *see* view
 - domain sort *see* operator arity
 - gathering *see* parsing
 - iterated *see* **iter**
 - mapping 209, *see* view, 213
 - metarepresentation 427
 - name
 - empty syntax 47
 - mixfix form 47
 - prefix form 47, 52
 - several identifiers 47
 - overloaded 50–51, 791
 - ad-hoc 50
 - subsort 50, 81
 - polymorphic 77, 213
 - precedence *see* parsing
 - range sort *see* operator coarity
- ops** 47
- optimizing 763, *see* debugger, profiler
- otherwise** 68, 92–96, 111
- overloading *see* operator overloaded
- owise** *see* **otherwise**, 185, 186

- paramodulation 662–663
- parse** 58, 814
 - metarepresentation *see* descent
 - function
- parsing 53–61
 - extended grammar 57
 - gathering 54
 - default pattern 56
 - precedence 54
 - default value 55
 - overridden 54
- Pathway logic 758
- pattern 68
- performance 9–11, 788
- PMaude 726
- poly** *see* polymorphic
- polymorphic** 77, 201, 237, 241
- polynomial 220, 222
- popd** 815
- pr** *see* protecting
- prec** *see* precedence
- precedence** 54, 198
- prelude.maude** 36, 235, 271, 272, 365, 423, 426, 427, 429, 458
- preregularity 52, 791
 - modulo 53
- print** 97, 136, 801
- print conceal** 811
- print reveal** 811
- printing
 - format 78
 - color 79
 - space 78
 - metarepresentation *see* descent
 - function
- probabilistic models 247, 724–727
- process 534 (Mobile Maude)
- profiler 777–788
- profiling 612 (Full Maude), 788
- protecting** 190, 191
- pushd** 815
- pwd** 815

- q** *see* quit
- queue 314
 - priority 315
- quit** 35, 816
- quoted identifier 270–271

- random-seed** 246, 804
- reachability problem 484
- read-eval-print loop 571
- Real-Time Maude 17, 724, 741
- record 329
- red** *see* reduce
- reduce** 35, 70, 110, 804
 - metarepresentation *see* descent
 - function
- reflection 423
 - moving between levels *see* descent
 - function, 626 (Full Maude)
 - tower of 432
- resume** 774 (debugger), 814 (debugger)
- rew** *see* rewrite, 232, 233
- rewrite condition 138–139
 - rewrite expression 138
 - satisfaction 140
- rewrite rule 135
 - executable 139
 - meaning
 - computational 135
 - logical 135
 - metarepresentation 427
 - object-oriented 669 (Full Maude)
 - probabilistic 725
 - tick 723
- rewrite** 142, 143, 147, 165, 246, 804
 - metarepresentation *see* descent
 - function
- rewriting
 - modulo 141
 - sharing 113n
- rewriting logic
 - proof equivalence 142
 - reflective 423
 - rewrite proof 142
 - timed 723
- right id** 71, 795
- ring 204
- r1** 136

- SCC 17, 75, 385, 387, 409, 739
- search** 148, 165, 166, 168, 172, 175, 177, 807
 - metarepresentation *see* descent
 - function
 - object-oriented 672 (Full Maude)

- such that 174, 177, 185, 187
- searching *see* search
- segmentation fault 790
- select 35, 814
- semiring 204
- set
 - from list 284–285
 - generalized 288–291
 - parameterized 281–283
 - partially ordered 205
 - totally ordered 205
- set clear memo 89
- set break 774, 813
- set clear memo 815
- set clear profile 778, 813
- set clear rules 806
- set extend 240, 815
- set include 815
- set print 773
- set print attribute 810
- set print attribute newline 810
- set print color 811
- set print conceal 811
- set print flattened 811
- set print format 811
- set print graph 810
- set print mixfix 810
- set print number 242, 811
- set print parentheses 811
- set print rat 256
- set print rational 811
- set print with aliases 811
- set profile 777, 813
- set protect 239, 814
- set show advisories 812
- set show breakdown 812
- set show command 812
- set show gc 812
- set show loop stats 811
- set show loop timing 812
- set show stats 811
- set show timing 811
- set trace 112, 763, 809
- set trace body 810
- set trace builtin 810
- set trace condition 809
- set trace eq 809
- set trace mb 809
- set trace rewrite 810
- set trace rl 809
- set trace select 112, 763, 809
- set trace substitution 809
- set trace whole 809
- set verbose 815
- set print format 79
- show 113
- show all 812
- show components 114, 813
- show eqs 812
- show mbs 812
- show module 812
- show modules 812
- show ops 812
- show path 161, 168, 173, 808
- show path labels 161, 169, 808
- show profile 779, 813
- show rls 143, 813
- show search graph 149, 808
- show sorts 113, 812
- show summary 813
- show vars 812
- show view 813
- show views 813
- simplicity 2–5
- socket 365–372
 - buffered 373–376
- socket.maude 365
- sort 43–45
 - error supersort *see* kind
 - least sort 52, 99
 - mapping 209, *see* view
 - metarepresentation 425
 - name collision 216
 - parameterized 215
 - structured 44
- sort constraint 319n
- sort decreasingness 101
- sort 43
- sorts 43
- special 90, 235
- stack 312, 685 (Full Maude)
- step 774 (debugger), 814 (debugger)
- strat 83
- strategy
 - internal 86, 461–464
 - object-message fair 354–357
 - operator 82–86
 - bottom-up *see* eager

- default 83
- eager 82
- frozen 86, 89–90, 142
- lazy 83
- operator-by-operator 83
- strategy** *see* **strat**
- string 264–268
 - number conversion 268–270
- structural axiom *see* attribute
 - equational
- submodule 190, *see* module importation
- subsort relation 45–46
 - connected component 46
 - metarepresentation 427
 - partial order 46
- subsort** 45
- subsorts** 45
- substitution 99
 - well-sorted 99
- sufficient completeness 75
- supermodule 190, *see* module importation
- symbolic reachability analysis 484–486, 651

- tautology checker 412
- tecla** 803
- term 51–53
 - canonical form 63, 100
 - relative to strategy 84
 - coloring 772–774
 - error 48
 - flattened 105
 - ground 53, 63
 - metarepresentation 426
 - qualified 50
 - undefined 48
- termination 100
 - context-sensitive 103
 - ground 101
 - modulo 102
- th** 202
- theory 202–208
 - flat 204, 205n
 - functional 202
 - admissible 202
 - mathematical semantics 202
 - operational semantics 203
 - importation 203
 - including 204, 205
 - metarepresentation 427
 - object-oriented 684 (Full Maude)
 - predefined 271–277
 - structured *see* theory importation
 - system 202
 - admissible 203
 - mathematical semantics 203
 - operational semantics 203
- token 578, 823
- trace deselect** 809
- trace exclude** 763
- trace include** 809
- trace select** 112, 763, 809
- tracing 612 (Full Maude), 763–772, 788
- tree
 - 2-3-4 337
 - AVL 333
 - binary 324
 - general 326
 - parameterized
 - leftist 622 (Full Maude)
 - red-black 340
 - search 328
- unification 465–492, 508–520, 634 (Full Maude)
 - E*-unification 465
 - algorithm 467
 - endogenous 487
 - exogenous 487
 - order-sorted 487–492
 - combination 489–490
 - equational 465
 - finitary 467
 - implementation 489–492
 - narrowing-based 483–484
 - order-sorted modulo axioms 466–468
 - problem 467
 - semantic 465
 - syntactic 465
 - unitary 467
 - variant-based 502–506
- unifier 465
 - E*-unifier 467
 - complete set 467

- most general 465
- semantic 465
- unify** 469–479, 808
- universal theory 423
- Universal** 77, 237, 241

- var** 51
 - in views 211
- variable 51
 - fresh 471, 480, 496, 501
 - in a module 51
 - metarepresentation 426
 - on-the-fly 51
- variant 492
 - based unification 502–506
 - complete set 492
 - finite variant property 493
 - generation 495–502
- variant** 492

- variant unify** 503–505, 808
- vars** 51
- vector 306
- version** 803
- view 202, 208–213
 - between theories 212
 - metarepresentation 432
 - object-oriented 685 (Full Maude)
 - parameterized 619 (Full Maude)
 - predefined 271–277
- view** 209

- where** 774 (debugger), 814 (debugger)

- xmatch** 107, 111, 112, 807
 - metarepresentation *see* descent
 - function
- xml-log** 804

Index of Maude Modules

234TREES 337
234TREES-TEST 339
3*NAT 65
8-PUZZLE 673

ABELIAN-GROUP 499
ABSTRACT-BAKERY 405
ABSTRACT-BAKERY-PREDS 409
ACCOUNT 670, 705
ACTOR-CONF 349
ACTOR-O-CONF 354
AGENT-TEST 362
ARRAY 303
ASSOC-ID-EX 795
ASSOC-MB-EX1 797
ASSOC-MB-EX2 798
ASSOCIATIVE-EX1 793
ASSOCIATIVE-EX2 793
AVL 333
AVL-TEST 336

BAG 217
BAKERY 404
BAKERY-PREDS 408
BANK-ACCOUNT 345
BANK-ACCOUNT-TEST 196, 347
BANK-MANAGER 348
BANK-MANAGER-TEST 348
BASIC-NAT 124
BASIC-NAT-LIST 127
BASIC-NAT-MSET 128
BASIC-NAT-NE-LIST 126
BASIC-NAT-SET 130
BASIC-NAT-TREE 124

BASIC-SET 215
BB-TEST 147
BETA-ETA 230
BIN-TREE 325
BIN-TREE-TEST 326
BLACKBOARD 166
BLOCKS-WORLD 158
BOOL 239
BOOL-OPS 239
BUFFERED-SOCKET 373
BUYER 559
BUYING-STRATS 461

CHECK-RROBIN 702
CHESS-COVER 184
CHIPS 176
CLOCK 247
COLLAPSE-ID-EX 794
COLLAPSE-IDEM-EX 795
COLLAPSE-NAT-EX 794
COLOR-TEST 79
COMM-ID-UNIFICATION-EX 478
COMM-IDEM-EX 789
CONFIGURATION 344
CONFIGURATION+ 706
CONVERSION 268
COUNTER 246

DATA-AGENTS 359
DATA-AGENTS-CONF 196, 358
DATA-AGENTS-INTERFACE 358
DEADLOCK-FREEDOM 527
DEKKER 416
DEKKER-CHECK 418

DF-ABSTRACT-BAKERY 410
 DF-ABSTRACT-BAKERY-CHECK 411
 DF-ABSTRACT-BAKERY-PREDS 411
 DF-BAKERY 410
 DF-BAKERY-PREDS 410
 DIE-HARD 172
 DIOPHANTINE 306

 EXCLUSIVE-OR 493
 EXCLUSIVE-OR-MB 496
 EXCLUSIVE-OR-NOT-COHERENT 631
 EXT-BOOL 83, 240
 EXTENDED-RENT-A-CAR-
 STORE-TEST 699

 FACTORIAL 241
 FACTORIAL-CLIENT 372
 FACTORIAL-SERVER 371
 FIBONACCI 87, 113, 778
 FLOAT 260
 FLOAT-STRING 197
 FORMAT-DEMO 80
 FULL-MAUDE 639

 GEN-TREE 326
 GEN-TREE-TEST 328
 GENERIC-SET-LIST 619

 HET-LIST 77
 HTTP/1.0-CLIENT 368

 ID-UNIFICATION-EX 477
 IDEM-SEMIGROUP 133
 INDEX-PAIR 305
 INSTRUMENTATION 521
 INSTRUMENTATION-
 INFRASTRUCTURE 520
 INSTRUMENTATION-TEST 523
 INT 249
 INT-GT-3 251
 INT-LIST 280
 INT-LIST* 287
 INT-LIST-AND-SET 285
 INT-MATRIX 305
 INT-SET 283
 INT-SET-MAX 222
 INT-SORTABLE-LIST-AND-SET 299
 INT-SORTABLE-LIST-AND-SET' 300
 INT-VECTOR 306

 ITER-EXAMPLE 473
 ITER-MB-EX1 800
 ITER-MB-EX2 801

 JOSEPHUS 170
 JOSEPHUS-GENERALIZED 171

 KHUN-PHAN 178

 LAMBDA 229
 LAST-APPEND 658
 LEFTID-UNIFICATION-EX 476
 LEFTIST-TREES 623
 LEFTIST-TREES-TEST 624
 LEFTIST-TREES-TEST-PAIR 625
 LEGAL-INST 225
 LEX-PAIR 217, 221
 LIBRARY 618
 LIBRARY-PETRI-NET 154
 LIST 278
 LIST* 285
 LIST-AND-SET 284
 LIST-CONS 317
 LIST-CONS-TEST 319
 LIST-KIND 226
 LOOP-MODE 571
 LTL 390
 LTL-SIMPLIFIER 397

 MACHINE-INT 253
 MACHINE-INT-TEST 254
 MAP 301
 MATRIX 305
 MAYBE 216
 MEMBERSHIP 659
 MEMORY 414
 META-LEVEL 433
 META-MODULE 427
 META-TERM 425
 META-VIEW 432
 METADATA-EX 91
 METAXMATCH-EX 447
 MINI-MAUDE 582
 MINI-MAUDE-SYNTAX 579
 MOBILE-OBJECT-INTERFACE 540
 MOBILE-PRINTERS-PREDS 569
 MODEL-CHECK-BAD-EX 402
 MODEL-CHECKER 399
 MONOMIAL 220

MULTISSET 322
 MULTISSET-TEST 324
 MUTEX 394
 MUTEX-CHECK 400
 MUTEX-PREDS 394
 MY-QID-SET-LIST 224
 MY-SET-LIST 224

 NAIVE-IDEM-SEMIGROUP 132
 NAIVE-NAT-LIST-MIXFIX-
 MAX 199
 NAIVE-SORTED-NAT-LIST 799
 NARROWING-VM-NOTOP 662
 NAT 241
 NAT-LIST 280
 NAT-LIST-GENERATOR 780
 NAT-LIST-KIND 799
 NAT-LIST-MAX 198
 NAT-LIST-MIXFIX-MAX 199
 NAT-MSET-MIN 773
 NAT-NARROWING 650
 NAT-PLUS 626
 NAT-PRED-KIND 118
 NAT-PRED-SUB 119
 NAT-PRED-SUPER 120
 NAT-SORTED-LIST-KIND 227
 NAT-VARIANT 493
 NAT/ 700
 NAT3 578
 NON-ASSOCIATIVE-EX 792
 NUMBERS 75, 109

 O-TICKER 354
 O-TICKER-CUSTOMER 354
 O-TICKER-FACTORY 354
 OO-BLOCKS-WORLD 678
 OO-BLOCKS-WORLD-COLOR 679
 OO-STACK 685
 OO-STACK2 687
 OVER-ASSOC-EX1 791
 OVER-ASSOC-EX2 791
 OWISE-TEST1 95
 OWISE-TEST2 95
 OWISE-TEST2-TRANSFORMED 96

 PAIR 217
 PARALLEL 415
 PARSING-EX1 59
 PARSING-EX2 59

 PARSING-EX3 60
 PARSING-EX4 60
 PATH 609
 PEANO-INT 116
 PEANO-NAT 115
 PEANO-RAT 117
 PERSON-RECORD 618
 PFUN 621
 POLYNOMIAL 220
 POWER[5] 618
 PRELIM-SET 214
 PRINT-ATTR-AMBIGUOUS 802
 PRINT-ATTRIBUTE-EX 97
 PRINTERS-PREDS 568
 PRIORITY-PAIR 316
 PRIORITY-QUEUE 315
 PRIORITY-QUEUE-TEST 316
 PRIORITY-QUEUE-TEST-PAIR 317
 PROCS-RESOURCES 379
 PROCS-RESOURCES-ENABLED 381

 QID 270
 QID-LIST 280
 QID-RAT-POLY 223
 QID-SET 283
 QID-SET* 290
 QUEUE 314
 QUEUE-TEST 314
 QVAL 588

 RABBIT-HOP 168
 RANDOM 245
 RAT 256
 RAT-POLY 222
 RB-TREES 340
 RB-TREES-TEST 341
 READERS-WRITERS 382
 READERS-WRITERS-ABS 386
 READERS-WRITERS-PREDS 385
 RECOLORING 186
 RECORD 329
 RENAMED-INT 252
 RENAMING-EX-A 200
 RENAMING-EX-B 200
 RENAMING-EX-C 200
 RENAMING-EX-D 201
 RENAMING-EX-E 201
 RENAMING-EX-F 201
 RENAMING-PAR-MOD-A 224

RENAMING-PAR-MOD-B 225
 RENAMING-PAR-MOD-C 225
 RENT-A-CAR-STORE-TEST 697
 REW-SEQ 696
 REW-SEQ-TEST 697
 RIGHTID-UNIFICATION-EX 477
 RIVER-CROSSING 182
 RIVER-CROSSING-2 420
 RIVER-CROSSING-2-PROP 421
 ROBIN 701

 SAMPLER 246
 SAT-SOLVER 412
 SAT-SOLVER-TEST 412
 SATISFACTION 393
 SAVING-ACCOUNT 671, 707
 SEARCH-TREE 330
 SEARCH-TREE-TEST 332
 SELLER 558
 SEQUENTIAL 415
 SET 281
 SET* 288
 SET-KIND 620
 SET-LIST 219
 SET-MAX 221
 SIEVE 85, 195
 SIMPLE-CLOCK 378
 SIMPLE-NAT 35
 SIMPLE-NAT-LIST 799
 SIMPLE-VENDING-MACHINE 145
 SOCKET 365
 SORTABLE-LIST 294
 SORTABLE-LIST' 297
 SORTABLE-LIST-AND-SET 298
 SORTABLE-LIST-AND-SET' 300
 SORTED-LIST 320
 SORTED-LIST-KIND 226
 SORTED-LIST-TEST 322
 SORTED-NAT-LIST-KIND 800
 SPREADSHEET 676
 SPREADSHEET-ASYNCH 676
 STACK 313
 STACK-TEST 313
 STRAT-EX1 84
 STRAT-EX2 84
 STRING 265
 STRING-NAT-ARRAY 304
 STRING-NAT-MAP 302
 STRING-OPS 367

 STRING-SET-MAX 222
 STRING-SORTABLE-LIST 294
 STRING-SORTABLE-LIST' 297
 STRING-SORTED-LIST-KIND 227

 TESTS 414
 TICKER 349
 TICKER-CUSTOMER 351
 TICKER-FACTORY 351
 TICKER-FACTORY-TEST 352
 TICKER-TEST 350
 TREE-NODE 622
 TRUTH 237
 TRUTH-VALUE 236
 TUPLE[2] 617

 U2 174
 UNIF-VENDING-MACHINE 474
 UNIF-VENDING-MACHINE-MB 475
 UNIFICATION 513
 UNIFICATION-AUX-OPS 511
 UNIFICATION-BANNER 639
 UNIFICATION-COMMAND-PROCESSING 637
 UNIFICATION-CYCLE 490
 UNIFICATION-DATABASE-HANDLING 638
 UNIFICATION-EX1 470
 UNIFICATION-EX3 472
 UNIFICATION-META-SIGN 636
 UNIFICATION-SIGN 635
 UNTYPED-LAMBDA-CALCULUS 231
 UP-DOWN-TEST 437

 VARIANT-VENDING-MACHINE 497, 655
 VECTOR 306
 VENDING-MACHINE 137, 143, 195, 594
 VENDING-MACHINE-GRAMMAR 572
 VENDING-MACHINE-INTERFACE 573
 VENDING-MACHINE-QUERY 594
 VENDING-MACHINE-SIGNATURE 136
 VENDING-MACHINE-TOP 144

 WEAKLY-SORTABLE-LIST 292
 WEAKLY-SORTABLE-LIST' 295
 WRONG-NAT-SET 789

 XMATCH-TEST 107

Index of Maude Theories

+MONOID 204
BIT-WIDTH 252
CELL 685
CHOICE 207
CONTENTS 329
DEFAULT 273
MONOID 203
NSPOSET 206
NSTOSET 207
NZNAT# 700
POSET 206
RING 204
SEMIRING 204
SPOSET 205
STOSET 207, 312
STRICT-TOTAL-ORDER 274
STRICT-WEAK-ORDER 274
TAOSET 205
TOSET 207, 315
TOTAL-ORDER 276
TOTAL-PREORDER 276
TRIV 203, 272
UNIFICATION-EX2 471
VAR 228

Index of Maude Views

32-BIT 253
64-BIT 253

5 703

Account 689

Bool 272

Contents 330

DEFAULT 273

Float 272
Float0 273
Float< 275
Float<= 277

IndexPair 305
Int 212, 272, 312
Int0 273
Int< 275
Int<= 277
IntAsStoset 211, 332
IntAsToset 212, 316, 624
IntStringAsToset 317, 625
IntVector 306

Nat 272
Nat0 273
Nat< 275
Nat<= 227, 277
NatAsToset 230, 322
Node 622
NSTOSET 226

POSET 225
PosetToToset 212

Qid 223, 272
Qid0 274

Rat 272
Rat0 273
Rat< 275
Rat<= 277
Record 329
RING 225
RingToRat 210

Set 619
SPosetToInt 211
STOSET 221
STRICT-TOTAL-ORDER 275
STRICT-WEAK-ORDER 275
String 272
String0 273
String< 275
String<= 227, 277
StringAsContents 332
StringAsToset 210
Substitution 696

TOSET 213, 320
TOTAL-ORDER 277
TOTAL-PREORDER 277
Tuple 620
VarNat 231