Twenty Years of Rewriting Logic

José Meseguer

Computer Science Department
University of Illinois at Urbana-Champaign, IL 61801, USA

Abstract

Rewriting logic is a simple computational logic that can naturally express both concurrent computation and logical deduction with great generality. This paper provides a gentle, intuitive introduction to its main ideas, as well as a survey of the work that many researchers have carried out over the last twenty years in advancing: (i) its foundations; (ii) its semantic framework and logical framework uses; (iii) its language implementations and its formal tools; and (iv) its many applications to automated deduction, software and hardware specification and verification, security, real-time and cyber-physical systems, probabilistic systems, bioinformatics and chemical systems.

Key words: rewriting logic, concurrency, logical frameworks, temporal logics, formal specification and verification, programming language semantics, networks and distributed systems, real-time systems, probabilistic systems, security, bioinformatics.

To the loving memory of my mother, Fuensanta Guaita Sánchez

Contents

1. Introduction

The first three papers on rewriting logic were published in 1990 [312, 311, 314]; they were then expanded in [315, 316]. Since that time, many researchers around the world have made important contributions to its foundations, tools, and applications. Since 1996, the *Workshop on Rewriting Logic and its Applications* has met biennially, with the 2010 Paphos meeting being its eighth edition, the *Workshop on Rewriting Techniques for Real-Time Systems* held its first edition in Spitsbergen in March 2010, and many hundreds of papers have been published on the subject ([301] contains a bibliography up to 2002, and this journal issue contains an up-to-date bibliography). This growth makes it desirable to reflect from time to time upon the advances made, survey such advances, and perhaps get some glimpses and make some guesses about future directions. It is somewhat like taking a snapshot of a person at age twenty. I have taken some similar, total or partial pictures at earlier ages, as a child [318, 321, 320], and as a teenager [301] (with Narciso Martí-Oliet) and [324]. It seems appropriate to attempt taking a coming-of-age picture, and to ask some questions about rewriting logic such as the following:

- How well-developed are its mathematical foundations?
- To what extent have its goals as a semantic framework for concurrency, and as a logical framework, been achieved?
- Which languages and tools supporting rewriting logic programming, specification, and verification have been developed?
- In which application areas has it been shown useful?
- What do its future prospects look like?

This paper is both a survey of the work that has been done, and my own attempt to answer the above questions.

I am grateful to the many gifted researchers who have contributed to the rewriting logic research program. I will explicitly mention some of them and some of their contributions. But I cannot really do justice to either all of them or all their contributions. This is due, in part, to my own limitations in keeping up with a vast and fast-growing literature; and to the impossibility, within the scope of this survey, of discussing, even summarily, the many hundreds of publications on the subject. The compilers of the detailed bibliography contained in this issue have gathered and organized by topic all the contributions that seem to have been made to date. I refer to this bibliography for a more complete picture of the different research directions that here I can only describe in broad outlines.

1.1. How to Read this Survey

This survey can be read in various ways, depending on the research interests, time, and degree of previous acquaintance with the overall area. For somebody unfamiliar with the area, not particularly interested in the mathematical foundations, and trying to gain a first overview of it, I would suggest reading first Sections 2, 4, and 7, and then looking at the other sections as needed. For a reader with a formal methods background, I would instead suggest reading first Sections 2, 3, 4, 5, and 6, and then looking at applications in Section 7 as needed. More specialized readings are also possible. For example, somebody only interested in security (resp. bioinformatics) applications could probably jump from Section 2 directly into Section 7.3 (resp. 7.6.1). Of course, for somebody trying to get an *in-depth* understandig of the whole area, I would recommend reading the entire survey from beginning to end.

2. Rewriting Logic in a Nutshell

Since the main goal of this paper is to facilitate access to a large body of research ideas to readers who may not be familiar with rewriting logic, it does not seem out of place to explain and illustrate, in an informal and impressionistic way, what rewriting logic is, and how it can be used.

Rewriting logic is like a coin with two sides: a computational side and a logical side. These two sides are complementary viewpoints on the same reality. Some applications fall more obviously into one of these sides, but when viewed as rewrite theories their other side is always present.

Computationally, rewriting logic is a *semantic framework* in which many different models of concurrency, distributed algorithms, programming languages, and software and hardware modeling languages can be naturally represented, executed and analyzed as *rewrite theories* (see Sections 4.2–4.4). Logically, it is a *logical framework* within which many different logics, and automated deduction procedures can likewise be represented, mechanized, and reasoned about (see Section 4.1).

Whenever anybody is selling you a semantic or logical framework you should be wary. A key reason for waryness is that such a framework may work *in principle*, but it may create a *big gap* between what is represented and its representation. I call this the *representational distance* imposed by the framework. For example, Turing machines provide an, in principle unobjectionable, semantic framework for sequential programming languages; but nobody uses them to define a language's semantics, except perhaps in the sense that a compiler for a language closely resembles a Turing machine semantics for it. There is just too much distance between a high-level programming language and a Turing machine, and much, including all the language's features, is lost in translation. In this regard, the evidence accumulated over the last twenty years strongly supports the claim that rewriting logic can rightfully be said to have "ϵ representational distance" as a semantic and logical framework. That is, what is represented and its representation are often isomorphic structures, typically differing only because of the slightly different notations used, but agreeing on all the main features.[1]

Why is this so? Whenever you represent a concurrent system or a logic, there are two key aspects about such a representation, which could be called the *static* and the *dynamic* aspects, and rewriting logic happens to be very well-suited for naturally representing both. Representing the static aspect of a concurrent system means representing its *distributed states*, while representing that of a logic means representing its *formulas*. Instead, representing the dynamic aspect of a concurrent system means representing its *concurrent transitions*, while representing that of a logic means representing its *inferences*.

The reason why rewriting logic's representational distance is typically ϵ is that a *rewrite theory* $\mathcal{R} = (\Sigma, E, R)$ consists of an equational theory (Σ, E) and a set of (possibly conditional) rewrite rules R, where (Σ, E) specifies the statics and R specifies the dynamics. If we are using (Σ, E, R) to represent a concurrent system (resp. a logic), then the distributed states (resp. formulas) of such a system are specified by the equational theory (Σ, E), where Σ is a collection of typed operators which includes the *state constructors* that build up a distributed state out of simpler state components (resp. the *logical and non-logical symbols* that build up a formula), and where E specifies the algebraic identities that such distributed states (resp. formulas) enjoy. That is, distributed states (resp. formulas) are specified as elements of an *algebraic data type*, namely, the initial algebra of the equational theory (Σ, E). Concretely, this means that a distributed state (resp. a formula) is mathematically represented as an E-equivalence class $[t]_E$ of terms (i.e., algebraic expressions) built up with the operators declared in Σ, *modulo* provable equality using the equations E, so that two state (resp. formula) representations t and t' describe the *same* state (resp. formula) if and only if one can prove the equality $t = t'$ using the equations E. The great generality with which algebraic data types can faithfully represent any data structures such as states or formulas (including binding operators such as quantifiers, λ-abstraction, and so on, which have a natural algebraic specification using a calculus of explicit substitutions such as CINNI [430]) is the reason why the static aspect can typically be represented with an ϵ representational distance.

The dynamic aspect of a system or logic represented as a rewrite theory

[1]When even the notation is identical, I speak of "0 representational distance," but the key point in either case is the isomorphic way in which a formalism is faithfully represented within a framework.

$\mathcal{R} = (\Sigma, E, R)$ is specified by its set R of *rewrite rules*. Why are they likewise so flexible? I focus first on concurrent systems specified with unconditional rewrite rules; the case of logics is discussed afterwards. What the rules R then represent are the system's *local concurrent transitions*. Each rewrite rule in R has the form $t \to t'$, where t and t' are algebraic expressions in the syntax of Σ. The lefthand side t describes a *local firing pattern*, and the righthand side t' describes a corresponding *replacement pattern*. That is, any fragment of a distributed state which is an instance of the firing pattern t can perform a local concurrent transition in which it is replaced by the corresponding instance of the replacement pattern t'. Both t and t' are typically *parametric* patterns, describing not single states, but parametric families of states. The parameters appearing in t and t' are precisely the *mathematical variables* that t and t' have, which can be instantiated to different concrete expressions by a mapping θ, called a *substitution*, sending each variable x to a term $\theta(x)$. The instance of t by θ is then denoted $\theta(t)$.

The most basic *logical deduction steps* in a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ are precisely atomic concurrent transitions, corresponding to applying a rewrite rule $t \to t'$ in R to a state fragment which is an instance of the firing pattern t by some substitution θ. That is, up to E-equivalence, the state is of the form $C[\theta(t)]$, where C is the rest of the state not affected by this atomic transition. Then, the resulting state is precisely $C[\theta(t')]$, so that the atomic transition has the form $C[\theta(t)] \to C[\theta(t')]$. Rewriting is *intrinsically concurrent*, because many other atomic rewrites can potentially take place in the rest of the state C (and in the substitution θ), at the same time that the local atomic transition $\theta(t) \to \theta(t')$ happens. That is, in general one may have complex concurrent transitions of the form $C[\theta(t)] \to C'[\theta'(t')]$, where the rest of the state C has evolved to C' and the substitution θ has evolved to θ' by other (possibly many) atomic rewrites simultaneous with the atomic rewrite $\theta(t) \to \theta(t')$. The rules of deduction of rewriting logic [315, 80] (which in general allow rules in R to be *conditional*) precisely describe all the possible, complex concurrent transitions that a system can perform, so that concurrent computation and logical deduction *coincide*. Such inference rules are discussed in Section 3.1.

If instead we adopt a logical point of view, so that the rewrite theory $\mathcal{R} = (\Sigma, E, R)$ represents a logic, then the rewrite rules R exactly specify the *inference rules* of the logic. What the rules rewrite may be formulas, or other formula-based data structures such as sets or lists of formulas, sequents, and so on. In the simplest case of an unconditional rewrite rule $t \to t'$, we describe an *inference step* in which we pass from a formula or formula-based structure which is an instance of the pattern t to another such formula or structure which is the corresponding instance of t', perhaps in a context C. That is, such atomic inference steps again take the form $C[\theta(t)] \to C[\theta(t')]$, for θ the substitution instantiating the patterns t and t'. Often, however, logical inference steps are *conditional*, and this may happen in two different ways. First, an inference step $t \to t'$ may only be allowed if we can previously show that other related steps, say, $u_1 \to v_1, \ldots, u_n \to v_n$ can be taken. Second, the inference step may be further constrained by a so-called *side condition* such as, for example, that a certain variable involved in the step is not a free variable in a given formula. Algebraically, such side conditions can be represented as equational constraints of the form $w_1 = q_1 \wedge \ldots \wedge w_m = q_m$. The ϵ representational distance of rewriting logic as a logical framework is due to the fact that such conditional inference rules can be exactly represented in R as *conditional rewrite rules* of the form

$$t \to t' \;\; \mathit{if} \;\; u_1 \to v_1 \wedge \ldots \wedge u_n \to v_n \wedge w_1 = q_1 \wedge \ldots \wedge w_m = q_m.$$

Of course, what we regard as concurrent computation or as logical deduction may, like beauty, be just in the eyes of the beholder. For example, we may regard any rewrite theory (Σ, E, R) where Σ has just a binary operator $\otimes$ and some constants, including a unit element I, E has associativity and commutativity axioms for $\otimes$ and an axiom for I as identity of $\otimes$, and R is a collection of unconditional ground rewrite rules, as *either* a Petri net, *or* as a theory in the linear conjunctive ($\otimes$) fragment of propositional linear logic [299]. But since both structures are mathematically isomorphic, there is no fact of the matter about which viewpoint should be adopted: this is just a pragmatic issue depending on what applications one has in mind.

I illustrate below all the ideas just discussed by means of two simple examples, one of a concurrent object system and another of an automated deduction procedure. For concreteness I give the specifications in Maude [105, 106], a language and system implementation directly based on rewriting logic (rewriting logic languages are discussed in Section 5). This emphasizes that rewriting logic is a *computational* logical and semantic framework, so that systems and logics can not only be mathematically represented: they can also be efficiently executed if they satisfy some minimum requirements (see Section 3.2).

2.1. Semantic Framework Uses: A Communication Protocol Example

I present a concurrent object-based system —namely, a simple communication protocol— specified in Maude. Maude's syntax is user-definable: operators can be declared with any desired "mixfix" syntax. A concurrent state made up of objects and messages can be thought of as a "soup" in which objects and messages are freely floating and can come into contact with each other in communication events. Mathematically, this means that the concurrent state, called a *configuration*, is modeled as a *multiset* or *bag* built up by a multiset union operator which satisfies the axioms of *associativity and commutativity*, with the empty multiset as its *identity* element. We can, for example, denote multiset union with *empty syntax*, that is, just by *juxtaposition* by declaring the type (called a *sort*) `Configuration` of configurations, which contains the sorts `Object` and `Msg` as *subsorts*, the empty configuration `none`, and the configuration union operator as follows:

```
sorts Object Msg Configuration .
subsorts Object Msg < Configuration .
op none : -> Configuration [ctor] .
op __ : Configuration Configuration -> Configuration
                     [ctor config assoc comm id: none] .
```

Each operator is declared with the `op` keyword, followed by its syntax, the list of its argument sorts, an arrow `->`, and its result sort. The configuration union operator has two argument positions, which are marked by underbars. Before and/or after such underbars, any desired syntax tokens can be declared. In this case an empty syntax (juxtaposition) has been chosen, so that no syntax tokens at all are declared. Note that constants like `none` are viewed as *operators with no arguments*. The keyword `config` declares that this is a union operator for configurations of objects and messages (the significance of this for fair execution is explained in Section 3.5). The `assoc comm id: none` attributes declare the associativity axiom $(x\ y)\ z = x\ (y\ z)$, the commutativity axiom $x\ y = y\ x$, and the identity axiom x `none` $= x$. Maude then supports rewriting *modulo* such axioms, so that a rule can be applied to a configuration regardless of parentheses, and regardless of the order of arguments. The `ctor` keyword declares that both `none` and `__` are state-building *constructors*, as opposed to functions defined on such constructors (see Section 3.7).

Consider an object-based system containing three classes of objects, namely, `Buffer`, `Sender`, and `Receiver` objects, so that a sender object sends to the corresponding receiver a sequence of values (say natural numbers) which it reads from its own buffer, while the receiver stores the values it gets from the sender in its own buffer. In Maude's Full Maude language extension (see Part II of [106]), such object classes can be declared as subsorts of the `Object` sort in *class declarations*, which specify the names and sorts of the *attributes* of objects in the class. The above three classes can be defined with class declarations:

```
class Buffer | q : NatList, owner : Oid .
class Sender | cell : Nat?, cnt : Nat, receiver : Oid .
class Receiver | cell : Nat?, cnt : Nat .
```

In general, if a class `Cl` has been declared with attributes `a1` of sort `A1`, ..., `an` of sort `An`, in a class declaration

```
class Cl | a1 : A1, ... , an : An .
```

then an object `o` of class `Cl` is a record-like structure of the form:

```
< o : Cl | a1 : v1, ... , an : vn >
```

where each `vi` is a term of sort `Ai`. For example, the sort `Oid` of object identifiers can use quoted identifiers as object names by importing the `QID` module, where quoted identifiers have sort `Qid`, and giving the subsort declaration `Qid < Oid`. Similarly, by importing the module `NAT`, where the natural numbers are the elements of sort `Nat`, one can then define the supersort `Nat?` of `Nat` containing an empty value `mt`, and the sort `NatList` of lists of natural numbers as follows:

```
sorts Nat? NatList .
subsorts Nat < Nat? NatList .
op mt : -> Nat? [ctor] .
op nil : -> NatList [ctor] .
op _._ : NatList NatList -> NatList [ctor assoc id: nil] .
```

then the following is an initial configuration of a sender and a receiver object, each with its own buffer, and each with its cell currently empty:

```
< 'a : Buffer | q : 1 . 2 . 3 , owner : 'b >
< 'b : Sender | cell : mt , cnt : 0 , receiver : 'd >
< 'c : Buffer | q : nil , owner : 'd >
< 'd : Receiver | cell : mt , cnt : 1 >
```

A sender object can send messages to its corresponding receiver object. The specifier has complete freedom to define the format of such messages by declaring operators of sort `Msg`, using the `msg` keyword instead of the more general `op` keyword to emphasize that the resulting terms are messages. For example, one can choose the following format:

```
msg to_::_from_cnt_ : Oid Nat Oid Nat -> Msg .
```

where a message, say, `to 'd :: 3 from 'b cnt 1`, means that `'b` sends to `'d` the data item 3, with counter 1, indicating that this is the *first* element transmitted. This last information is important, since message passing in a configuration is usually asynchronous, so that messages could be received out-of-order. Therefore, receiver objects need to use the counter information to properly reassemble a list of transmitted data. Of course, out-of-order communication is just *one* possible situation that can be modeled. If, instead, one wanted to model in-order communication, the distributed state could contain *channels*, similar for example to the buffer objects, so that axioms of associativity and identity are satisfied when inserting messages into a channel, but *not commutativity*, which is the axiom allowing out-of-order communication in a configuration of objects and messages. Up to now we have just defined the *distributed states* of our object-based system as the algebraic data type associated to the equational theory (Σ, E), where Σ is the signature whose sorts have been declared with the `sort` (and `class`) keywords, with subsort relations declared with the `subsort` keyword, and whose operators have been declared with the `op` (or `msg`) keywords; and where the equations E have been declared[2] as *equational axioms* of associativity and/or commutativity and/or identity associated to specific operators, declared with the `assoc`, `comm` and `id:` keywords.

What about the concurrent *transitions* for buffers, senders, and receivers? They are specified by rewrite rules R such as the following (note that, by convention, object attributes not changed by a rule need not be mentioned in its righthand side):

```
vars X Y Z : Oid .  vars N E : Nat .  vars L L' : NatList .

rl [read] : < X : Buffer | q : L . E, owner : Y >
            < Y : Sender | cell : mt, cnt : N, receiver: Z >
            => < X : Buffer | q : L > < Y : Sender | cell : E, cnt : N + 1 >  .
```

[2]In Maude one can also declare explicit equations with the `eq` and `ceq` keywords. See Section 2.2 for an example.

```
 rl [write] : < X : Buffer | q : L, owner : Y > < Y : Receiver | cell : E >
             => < X : Buffer | q : E . L > < Y : Receiver | cell : mt  >  .

 rl [send] : < Y : Sender | cell : E, cnt : N, receiver : Z >
            => < Y : Sender | cell : mt > (to Z :: E from Y cnt N) .

 rl [receive] : < Z : Receiver | cell : mt, cnt : N > (to Z :: E from Y cnt N)
               => < Z : Receiver | cell : E, cnt : N + 1 >  .
```

That is, senders can read data from the buffer they own and update their count; and receivers can write their received data in their own buffer. Also, each time a sender has a data element in its cell, it can send it to its corresponding receiver with the appropriate count; and a receiver with an empty cell can receive a data item from its sender, provided it has the correct counter. Note that rewriting is intrinsically concurrent; for example, `'b` could be sending the next data item to `'d` at the same time that `'d` is receiving the previous data item or is writing it into its own buffer; furthermore, there could be many different sender-receiver pairs executing concurrently in the same configuration. Note also that the rules `send` and `receive` describe the *asynchronous message passing* communication between senders and receivers typical of the Actor model [3]. Instead, the `read` and `write` rewrite rules describe *synchronization events*, in which a buffer and its owner object synchronously transfer data between each other. This illustrates the flexibility of rewriting logic as a semantic framework: no assumption of either synchrony or asynchrony is built into the logic. Instead, many different styles of concurrency and of in-order or out-of-order communication can be easily modeled.

Since the above rewrite theory is executable, we can use its rewrite rules not just as a formal specification, but also for simulation purposes. For example, from the initial state described above, where the sender's buffer had a list `1 . 2 . 3` and the receiver's buffer was empty, we would expect the above rewrite rules to achieve in-order communication, so that in the final state the sender's buffer is empty and the receiver's buffer has the list `1 . 2 . 3`. Maude achieves a rule-fair execution with the `rewrite` command. To support the object-oriented notation for classes, objects, and messages used in this example, we can declare the above sorts, subsorts, classes, and rules in an *object-oriented module* in Maude's Full Maude extension (see [106]). Then, to execute our system from the above-mentioned initial state we can give to Full Maude the following command (note that all Full Maude module declarations and commands must be enclosed in parentheses):

```
Maude> (rewrite < 'a : Buffer | q : 1 . 2 . 3 , owner : 'b >
                < 'b : Sender | cell : mt , cnt : 0 , receiver : 'd >
                < 'c : Buffer | q : nil , owner : 'd >
                < 'd : Receiver | cell : mt , cnt : 1 > .)

result Configuration :
                < 'a : Buffer | owner : 'b, q : nil >
                < 'b : Sender | cell : mt, cnt : 3, receiver : 'd >
                < 'c : Buffer | owner : 'd, q :(1 . 2 . 3) >
< 'd : Receiver | cell : mt, cnt : 4 >
```

2.2. Logical Framework Uses: A Propositional Satisfiability Example

Procedures for propositional satisfiability (SAT) are very useful in many applications, including SAT solving modulo decidable theories in first-order theorem proving. Sometimes, however, in the quest for performance the algorithmic details of a SAT solver may become so involved that it is unclear whether it is sound. In fact, this is not a theoretical possibility but a real concern in actual SAT solvers. What is needed is a clear *separation of concerns* between the SAT solver's *inference system* and its (typically quite sophisticated) *heuristics.* This separation of concerns has been advocated by Cesare Tinelli, who gave a precise sequent calculus specification of the Davis-Putnam-Logemann-Loveland (DPLL) SAT solving procedure, from which a proof of its correctness is quite direct, in [453]. I discuss in what follows a slightly enhanced version of Tinelli's inference system in [453], which Tinelli and I then used to develop the rewriting logic specification of the inference system executable in Maude discussed below. Tinelli's sequent-based formalization is as follows. To reason about the satisfiability of a propositional formula φ we first put it in conjuntive normal form as a conjunction of clauses $C_1 \wedge \ldots \wedge C_n$, where a clause C is a disjunction of literals, which is logically equivalent to the *set* of clauses $\Gamma = \{C_1, \ldots, C_n\}$. The DPLL procedure can then be formalized as a sequent-based inference system with sequents of the form $\Delta \vdash \Gamma$, where Δ is a set of *literals*, i.e., of atomic propositions p or negations $\neg p$ of such propositions, and where Γ is a set of clauses. A set of clauses Γ will be *satisfiable* iff from the initial sequent $\emptyset \vdash \Gamma$ we can derive a sequent of the form $\Delta \vdash \emptyset$ using the DPLL inference system, where Δ represents a satisfying assignment. As usual in sequent formulations, a set $\Gamma = \{C_1, \ldots, C_n\}$ is written without the enclosing parentheses as $\Gamma = C_1, \ldots, C_n$. Likewise, a set of literals $\Delta = \{l_1, \ldots, l_m\}$ is written $\Delta = l_1, \ldots, l_m$. The DPLL procedure can then be formalized as the following inference system:

$$(\textbf{subsume})\ \frac{\Delta \vdash \Gamma, l \vee C}{\Delta \vdash \Gamma}\ \textit{if}\ l \in \Delta \qquad (\textbf{resolve})\ \frac{\Delta \vdash \Gamma, l \vee C}{\Delta \vdash \Gamma, C}\ \textit{if}\ \neg l \in \Delta$$

$$(\textbf{assert})\ \frac{\Delta \vdash \Gamma, l}{\Delta, l \vdash \Gamma}\ \textit{if}\ l \notin \Delta, \neg l \notin \Delta \quad (\textbf{close})\ \frac{\Delta \vdash \Gamma, \Box}{\emptyset \vdash \Box}\ \textit{if}\ \Delta \neq \emptyset \vee \Gamma \neq \emptyset$$

$$(\textbf{split})\ \frac{\Delta \vdash \Gamma, l \vee C}{\Delta, l \vdash \Gamma \qquad \Delta, \neg l \vdash \Gamma, C}\ \ \textit{if}\ \ l \notin \Delta,\ \neg l \notin \Delta,\ C \neq \Box$$

where $\Box$ denotes the empty clause, C ranges over clauses, and for l any literal, $\neg\neg l = l$. The rewriting logic formalization of this inference system as a rewrite theory $\mathcal{R}_{DPLL} = (\Sigma_{DPLL}, E_{DPLL}, R_{DPLL})$ must axiomatize sequents as the algebraic data type of the equational theory $(\Sigma_{DPLL}, E_{DPLL})$, and then axiomatize the inference rules as rewrite rules in R_{DPLL}. We can, however, do better than that. Because of rewriting logic's *distinction between equations* *and rules*, we can *choose* to axiomatize as *equations* those inference rules that are *deterministic* (in the sense that their combined application will lead to a unique final result) and that should *always* be applied exhaustively. We only need to axiomatize as rules the truly nondeterministic rules. This makes the specification both more clever, since it makes explicit the implicit determinism, and much more efficient, because it can drastically reduce the amount of search required, given that search is now only needed for the nondeterministic rules. For the above DPLL inference system, only the **split** rule is nondeterministic: all other rules can be axiomatized equationally. The rewriting logic axiomatization $\mathcal{R}_{DPLL} = (\Sigma_{DPLL}, E_{DPLL}, R_{DPLL})$ is in fact executable in Maude as the `DPLL` module below and can be used as a prototype of the DPLL procedure.

Of course, the real smarts of a SAT solver are in its heuristics; but this is the whole point of Tinelli's proposal: we should cleanly separate between the inference system and its heuristics and not mix the two together in a confusion of pointers. Nevertheless, the rewrite theory $\mathcal{R}_{DPLL} = (\Sigma_{DPLL}, E_{DPLL}, R_{DPLL})$ captures in a declarative way a simple but important part of those heuristics, namely, it identifies those deterministic rules that should always be applied exhaustively; but it leaves unspecified the heuristics for applying the **split** rule. Heuristics or, more precisely, *strategies* are a separate and modular dimension of a rewrite theory that I discuss in Section 3.5. The same rewrite theory can be executed with many different strategies, which may be better or worse in various regards; but strategies, being now a particular way of applying intrinsically correct rules, can never affect correctness. For DPLL and DPLL(T) this completely agrees with Tinelli's approach in [453] and in his later joint work with Nieuwenhuis and Oliveras [356], where the issue of strategies is discussed in depth. Although the above DPLL calculus does not model fundamental features of modern SAT solvers such as back-jumping, conflict resolution, and clause learning, the Abstract DPLL framework of [356] —which could also be naturally specified as a rewrite theory— can express such features declaratively, so that a clean separation between heuristics and inference rules is maintained.

```
mod DPLL is protecting QID .
  sorts Literal Context Clause ClauseSet Sequent .
  subsorts Qid < Literal < Context Clause < ClauseSet .
  op ~ : Literal -> Literal .
  op null : -> Context .
  op _,_ : Context Context -> Context [assoc comm id: null] .
  op _,_ : ClauseSet ClauseSet -> ClauseSet [assoc comm id: null] .
  op [] : -> Clause .
  op _\/_ : Clause Clause -> Clause [assoc comm id: ([])] .
  op _|-_ : Context ClauseSet -> Sequent .
  op _in_ : Literal Context -> [Bool] .

  var p : Qid .
  var l : Literal .
  var CTX : Context .
  var C : Clause .
  var CS : ClauseSet .

  eq ~(~(l)) = l .
  eq l in l,CTX = true .

  eq [contraction] : C,C = C .
  eq [subsume] : l,CTX |- CS,(l \/ C) = l,CTX |- CS .
  eq [resolve1] : p,CTX |- CS,(~(p) \/ C) = p,CTX |- CS,C .
  eq [resolve2] : ~(p),CTX |- CS,(p \/ C) = ~(p),CTX |- CS,C .
  eq [close1] :  CTX |- C,CS,[] = null |- [] .
  eq [close2] :  CTX,l |- CS,[] = null |- [] .
  ceq [assert] : CTX |- CS,l = CTX,l |- CS
                     if (l in CTX) =/= true and (~(l) in CTX) =/= true .
  crl [split1] :  CTX |- CS,(l \/ C) => l,CTX |- CS
        if (l in CTX) =/= true and (~(l) in CTX) =/= true and C =/= [] .
  crl [split2] :  CTX |- CS,(l \/ C) => ~(l),CTX |- CS,C
        if (l in CTX) =/= true and (~(l) in CTX) =/= true and C =/= [] .
endm
```

Let me discuss the rewrite theory $\mathcal{R}_{DPLL} = (\Sigma_{DPLL}, E_{DPLL}, R_{DPLL})$ in more detail. The signature Σ_{DPLL} describes the sorts, subsorts, constructors, and auxiliary functions needed for sequents. Note that the order-sorted type structure in `DPLL` precisely captures the types of: (i) propositional symbols, represented here by the sort `Qid` of quoted identifiers, (ii) literals, (iii) sets of literals, called contexts, (iv) clauses, and (v) sets of clauses. Sequents are then pairs of a context and a set of clauses. Negation $\neg$ is represented by `~` in typewriter notation, set membership $\in$ by `in`, and the empty set $\emptyset$ by `null`. All other operators are typewriter analogues of their mathematical notation.

The equations E_{DPLL} are essentially of two kinds: those axiomatizing the basic properties of *sequents*, and those expressing the deterministic inference rules **subsume**, **resolve**, **assert**, and **close**. In any sequent calculus, the first order of business is to define the so-called *structural rules* enjoyed by sequents $\Delta \vdash \Gamma$. For propositional and first-order logic, sequents $\Delta \vdash \Gamma$ enjoy structural rules making Δ and Γ *sets* of formulas. This is captured above by the `assoc`, `comm` (corresponding to the so-called **exchange** structural rule of sequents), and `id:` attributes of the operator `_,_` of set union; but there is still one more structural rule, namely, the so-called **contraction** rule expressing the idempotency of set union, which is specified above as the `contraction` equation. Not all sequent calculi obey all these structural rules: linear logic drops **contraction**, and Lambek's logic drops both **contraction** and **exchange**. The general point is that, by choosing the right equations, we can capture any desired structural axioms. Furthermore, by declaring some of them as *axioms*, we can reason *modulo* such axioms without having to explicitly apply them as structural inference rules: the only exception here is the **contraction** rule, which is explicitly applied as a simplification equation modulo the built-in associativity, commutativity, and identity axioms for set union.

Since negations are restricted to literals in the above type structure, we only need the equation stating that the double negation of a literal is the literal itself. Set membership needs only be defined in the positive case by the obvious equation; since we are only defining the positive case, an expression like `'a in 'b,'c,'d`, where `'a` is not in the set `'b,'c,'d`, does not have a Boolean value: its value is the expression itself, which belongs to the supersort `[Bool]` of `Bool` automatically added by Maude. For simplicity and efficiency reasons, except for the **assert** rule, all deterministic inference rules that had *side conditions* in Tinelli's formulation are now specified as *unconditional* equations declared with the `eq` keyword. The simplicity of these unconditional equations is due to the expressiveness of pattern matching modulo associativity, commutativity and identity, which can capture the corresponding side conditions in the lefthand side patterns. Sometimes, as in the case of **resolve** and **close**, two equations are needed to specify one rule. This is done to express the conditions of the corresponding inference rules in the patterns of the unconditional equations, such as the disjunction of either Δ or Γ being nonempty in the side condition of **close**, and the side condition of the **resolve** inference rule. Finally, the two conditional rewrite *rules* in R_{DPLL}, declared with the `crl` keyword, exactly capture the two inference rules specified by the two different outcomes of the **split** rule. Note that we could have instead chosen to represent the DPLL inference rules *au pied de la lettre*. For example, using the `or` operator from the implicitly imported `BOOL` module, we could have represented the **close** rule by the single conditional equation

```
  ceq [close] :  CTX |- CS,[] = null |- [] if CTX =/= null or CS =/= null .
```

As already mentioned, the particular choice of equations and rules in `DPLL` is motivated by two reasons: first, to illustrate the high expressive power of matching modulo associativity, commutativity and identity, which allows expressing some conditions directly in the lefthand side pattern; and second, for efficiency reasons, since unconditional equations and rules can be executed more efficiently than condional ones. Again, the representational distance between the textbook formulation of the DPLL sequent calculus and its expression in an *executable* form in the rewriting logic framework, whether in the more literal way just alluded to or the freer one in the `DPLL` module, can be fairly described as an ϵ distance. Furthermore, rewriting logic's distinction between equations and rules gives a specifier additional expressive power to discriminate between deterministic and nondeterministic inference rules.

The above inference system, being an executable rewrite theory, provides a prototype implementation of a DPLL-style SAT solver. Of course, since the DPLL inference system is non-deterministic, using Maude's `rewrite` command is not enough, since the concrete sequence of inference steps followed by the default strategy of the `rewrite` command could result in an assignment not satisfied by the given formula, when the formula is actually satisfiable. One option is to specify a *strategy* that applies the DPLL rules in a way that guarantees that a satisfying assignment will be found if there is one; this could be done using Maude's strategy language [175]. A simpler option is to use Maude's `search` command, where we begin with an initial term t and search for a rewrite sequence reaching a term t' which is a substitution instance of a pattern (a term with variables) specified as the goal of the `search` command. For example, the satisfiability of a formula such as `('a \/ ~('b) \/ 'c), (~('a) \/ 'b \/ 'c), ('a \/ 'b)`, can be decided by giving to

Maude a `search` command to look for a satisfying assignment, which is represented as a sequent of the form `CTX |- null`. Therefore, we begin with the sequent `null |- ('a \/ ~('b) \/ 'c), (~('a) \/ 'b \/ 'c), ('a \/ 'b)` and search for a sequence of DPLL inference steps bringing us to a sequent which is an instance of the pattern `CTX |- null`. If we are interested in just one solution, we can qualify the `search` command with the `[1]` request for the first solution as follows:

```
Maude> search [1] null |- ('a \/ ~('b) \/ 'c), (~('a) \/ 'b \/ 'c), ('a \/ 'b)
                                                                  =>+ CTX |- null .

Solution 1 (state 4)
CTX --> 'a,'c,~('b)
```

which tells us that we can reach the satisfying assignment `'a,'c,~('b) |- null` by instantiating the pattern's variable `CTX` to the context `'a,'c,~('b)`. Instead, if we are interested in *all* satisfying assignments, we can give the unqualified `search` command (note that some satisfying assignments below are special cases of more general ones):

```
Maude> search null |- ('a \/ ~('b) \/ 'c), (~('a) \/ 'b \/ 'c), ('a \/ 'b)
                                                              =>+ CTX |- null .

Solution 1 (state 4)
CTX --> 'a,'c,~('b)

Solution 2 (state 5)
CTX --> 'b,'c,~('a)

Solution 3 (state 7)
CTX --> 'a,'b

Solution 4 (state 8)
CTX --> 'a,'c

Solution 5 (state 9)
CTX --> 'a,'b,~('c)

Solution 6 (state 10)
CTX --> 'b,'c

No more solutions.
```

3. Foundations

The foundations of rewriting logic begin of course with its proof theory and its model theory, but have various other aspects such as reflection, strategies, and executability properties. Furthermore, rewrite theories themselves can be extended to model real-time systems and probabilistic systems. Finally, the *properties* enjoyed by a rewrite theory need not be just those expressible in rewriting logic itself: they may also be expressible in other logics, such as temporal logics. Temporal logic properties can then be verified by model checking or deductive methods.

3.1. Rewriting Logic

A *rewrite theory*[3] is a tuple $\mathcal{R} = (\Sigma, E, R)$, with:

- (Σ, E) an equational theory with function symbols Σ and equations E; and
- R a set of *labeled rewrite rules* of the general form

  $$r : t \to t'$$

  with r a label and t, t' Σ-terms which may contain variables in a countable set X of variables which we assume fixed in what follows; that is, t and t' are elements of the term algebra $T_\Sigma(X)$. In particular, their corresponding sets of variables, $\mathit{vars}(t), \mathit{vars}(t')$ are both contained in X.

Given $\mathcal{R} = (\Sigma, E, R)$, the sentences that $\mathcal{R}$ proves are rewrites of the form, $t \to t'$, with $t, t' \in T_\Sigma(X)$, which are obtained by finite application of the following *rules of deduction*:

- **Reflexivity**. For each $t \in T_\Sigma(X)$, $\dfrac{}{t \to t}$
- **Equality**. $\dfrac{u \to v \quad E \vdash u = u' \quad E \vdash v = v'}{u' \to v'}$
- **Congruence**. For each $f : k_1 \dots k_n \to k$ in Σ, and $t_i, t'_i \in T_\Sigma(X)$, $1 \leq i \leq n$,

  $$\frac{t_1 \to t'_1 \quad \dots \quad t_n \to t'_n}{f(t_1, \dots, t_n) \to f(t'_1, \dots, t'_n)}$$

- **Replacement**. For each rule $r : t \to t'$ in R, with, say, $\mathit{vars}(t) \cup \mathit{vars}(t') = \{x_1, \dots, x_n\}$, and for each substitution $\theta : \{x_1, \dots, x_n\} \longrightarrow T_\Sigma(X)$, with $\theta(x_l) = p_l$, $1 \leq l \leq n$, then

  $$\frac{p_1 \to p'_1 \quad \dots \quad p_n \to p'_n}{\theta(t) \to \theta'(t')}$$

[3] As already mentioned in Section 2, rewrite rules can be conditional. To simplify the exposition I present here the simplest version of rewrite theories, namely, *unconditional* rewrite theories over an *unsorted* equational theory (Σ, E). In general, however, the equational theory (Σ, E) can be many-sorted, order-sorted, or even a membership equational theory [319]. And the rules can be *conditional*, where a rule's condition has a conjunction of rewrites, equalities, and even memberships, that is, rules have the general form

$$r : t \to t' \ \text{ if } \ (\bigwedge_i u_i = u'_i) \wedge (\bigwedge_j v_j : s_j) \wedge (\bigwedge_l w_l \to w'_l)$$

Furthermore, the theory may also specify an additional mapping $\phi : \Sigma \longrightarrow \mathcal{P}(\mathbb{N})$, assigning to each function symbol $f \in \Sigma$ (with, say, n arguments) a set $\phi(f) = \{i_1, \dots, i_k\}$, $1 \leq i_1 < \dots < i_k \leq n$ of *frozen argument positions* under which it is forbidden to perform any rewrites. Rewrite theories in this more general sense are studied in detail in [80]; they are clearly more expressive than the simpler unconditional and unsorted version presented here. This more general notion is the one supported by the Maude language [106]. I discuss further these generalized rewrite theories in Section 3.1.2.

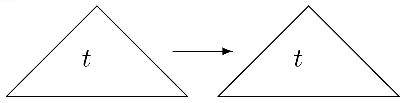
where for $1 \leq i \leq n$, $\theta'(x_i) = p'_i$.

- **Transitivity**

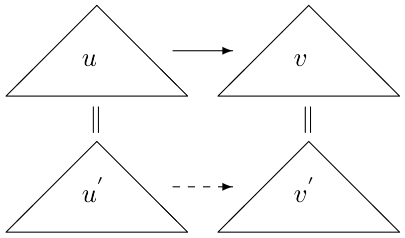
$$\frac{t_1 \rightarrow t_2 \qquad t_2 \rightarrow t_3}{t_1 \rightarrow t_3}$$

We can visualize the above inference rules as follows:

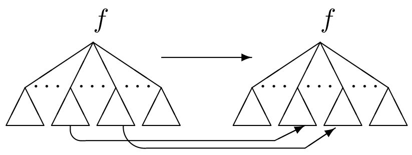
Reflexivity



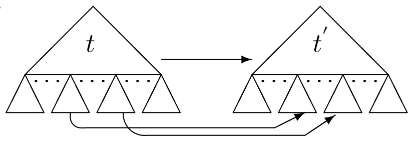
Equality



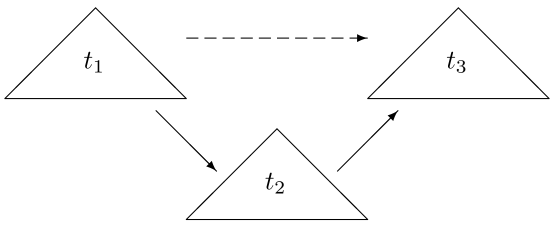
Congruence



Replacement



Transitivity



The notation $\mathcal{R} \vdash t \rightarrow t'$ states that the sequent $t \rightarrow t'$ is *provable* in the theory $\mathcal{R}$ using the above inference rules. Intuitively, we should think of the inference rules as different ways of *constructing* all the (finitary) *concurrent computations* of the concurrent system specified by $\mathcal{R}$. The **Reflexivity** rule says that for any state t there is an *idle transition* in which nothing changes. The **Equality** rule specifies that the states are in fact equivalence classes modulo the equations E. The **Congruence** rule is a very general form of "sideways parallelism," so that each operator f can be seen as a *parallel state constructor*, allowing its arguments to evolve in parallel. The **Replacement** rule supports a different form of parallelism, which I call "parallelism under one's feet," since besides rewriting an instance of a rule's lefthand side to the corresponding righthand side instance, the state fragments in the substitution of the rule's variables can also be rewritten. Finally, the **Transitivity** rule allows us to build longer concurrent computations by composing them sequentially.

3.1.1. Operational and Denotational Semantics of Rewrite Theories

A rewrite theory $\mathcal{R} = (\Sigma, E, R)$ has both a *deduction-based operational semantics*, and an *initial model denotational semantics*. Both semantics are defined naturally out of the proof theory just described. The deduction-based operational semantics of $\mathcal{R}$ is defined as the collection of *proof terms* [315] of the form $\alpha : t \rightarrow t'$. A proof term α is an algebraic description of a proof tree proving $\mathcal{R} \vdash t \rightarrow t'$ by means of the inference rules of rewriting logic. As already mentioned, such proof trees describe the different *finitary concurrent computations* of the concurrent system axiomatized by $\mathcal{R}$. When we specify $\mathcal{R}$ as a Maude module and rewrite a term t with the `rewrite` or `frewrite` commands, obtaining a term t' as a result, we can use Maude's `trace` mode to obtain a sequentialized version of a proof term $\alpha : t \rightarrow t'$ of the particular rewrite proof built by the Maude interpreter.

A rewrite theory $\mathcal{R} = (\Sigma, E, R)$ has also a model-theoretic semantics, so that the inference rules of rewriting logic are sound and complete with respect to satisfaction in the class of models of $\mathcal{R}$ [315]. Such models are *categories* with a (Σ, E)-algebra structure [315]. These are "true concurrency" denotational models of the concurrent system axiomatized by $\mathcal{R}$. That is, this model theory gives a precise mathematical answer to the question: when do two descriptions of two concurrent computations denote *the same* concurrent computation? The class of models of a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ has an *initial model* $\mathcal{T}_{\mathcal{R}}$ [315]. The initial model semantics is obtained as a *quotient* of the just-mentioned deduction-based operational semantics, precisely by axiomatizing algebraically when two proof terms $\alpha : t \rightarrow t'$ and $\beta : u \rightarrow u'$ denote the *same* concurrent computation. Of course, α and β should have identical beginning states and identical ending states. By the **Equality** rule this means that we should have $E \vdash t = u$, and $E \vdash t' = u'$. That is, the objects of the category $\mathcal{T}_{\mathcal{R}}$ are E-equivalence classes $[t]$ of ground Σ-terms, which denote the states of our system. The arrows or morphisms in $\mathcal{T}_{\mathcal{R}}$ are *equivalence classes of proof terms*, so that $[\alpha] = [\beta]$ iff both proof terms denote the same concurrent computation according to the "true concurrency" axioms. Such axioms are very natural. They express that the **Transitivity** rule behaves as an arrow composition and is therefore associative. Similarly, the **Reflexivity** rule provides an identity arrow for each object, satisfying the usual identity laws. Furthermore, they state that each f in the **Congruence** rule acts not only on states but also on arrows as a *functor*, i.e., preserving arrow compositions and identitites; this axiomatizes the true concurrency semantics of "sideways parallelism." Finally, the "parallelism under one's feet" semantics of the **Replacement** inference rule is axiomatized by giving equational axioms making each rewrite rule $r : t \to t'$ a *natural transformation* $r : t \Rightarrow t'$ between the functors t and t'.

Categorical models for rewrite theories go back to [312, 314, 315]. As pointed out in those papers and mentioned above, the models of a rewrite theory are (small) *categories* with an algebraic structure. They generalize ordinary algebras, which are *sets* with an algebraic structure. This means that the underlying universe in which these models and their morphisms should be considered is the *2-category* **Cat** of small categories [314, 315, 344], as opposed to the underlying universe of algebras, which is the *category* **Set** of sets. There is also a generalization of Lawvere's functorial semantics [279] for ordinary algebras: the models of a rewrite theory $\mathcal{R}$ have a functorial semantics as 2-product-preserving 2-functors into **Cat** from its associated Lawvere 2-theory $\mathcal{L}_{\mathcal{R}}$ [313, 322]. Such Lawvere 2-theories have been replaced by weaker sesqui-categories in [436, 123]; and in the context of tile logic (which I discuss further in Section 4.2) by Lawvere double theories in [328, 78, 81].

3.1.2. Generalized Rewrite Theories

Since rewriting logic is parameterized by its underlying equational logic, the more expressive its underlying equational part, the more expressive also the resulting rewriting logic. Increased expressivenes is not a theoretical luxury, but an eminently practical goal, since formal specification languages should describe as simply and naturally as possible the widest possible class of systems. As explained in [319], membership equational logic is indeed a very expressive equational logic generalizing order-sorted equational logic (which generalizes many-sorted equational logic, which, in turn, generalizes unsorted equational logic). It supports sorts, subsorts, partiality, and sorts defined by equational conditions through membership axioms. Its atomic formulas are either equalities $t = t'$, or memberships $t : s$ stating that t has sort s. Its sentences are universally quantified Horn clauses on such atoms. Therefore, as already pointed out in Footnote 3, a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, whose underlying equational theory (Σ, E) is a membership equational theory, may have conditional rules in R whose conditions can be conjunctions of equations, memberships, and rewrites.

In the quest for more expressive versions of rewriting logic, another feature, namely, *frozenness*, has proved to be very useful in many applications. The idea of frozenness is that some argument positions in a state constructor should be "frozen," in the sense that no rewrites are allowed below that position. For example, if $\_ \cdot \_$ is an action concatenation operator in a process calculus, then an expression like $a.P$, with a an action and P a process expression, should typically not be rewritten on the P part, that is, on its second argument. This can be simply captured by saying that $\_ \cdot \_$ is frozen on its second argument. More generally, given a signature Σ, its frozenness information is defined as a function $\phi : \Sigma \longrightarrow \mathcal{P}_{\mathit{fin}}(\mathbb{N})$, where $\phi(f)$ is the set of frozen argument positions. For example, $\phi(\_\cdot\_) = \{2\}$. In summary, a *generalized rewrite theory* is a 4-tuple $\mathcal{R} = (\Sigma, E, R, \phi)$ where: (i) (Σ, E) is a membership equational theory; (ii) the rules in R may be conditional, where conditions are conjunctions of equations, memberships and rewrites, and (iii) ϕ is the frozenness map. As shown in detail in [80], all the good properties of the proof theory and the model theory of rewriting logic, including the existence of initial and free models, extend naturally to the case of generalized rewrite theories.

A theme already developed in [315], which is extended to generalized rewrite theories in [80], is that of *reachability models*. For some purposes (for example, model checking or reachability analysis), we may not need the initial model of a rewrite theory $\mathcal{R}$ in its full glory as a category of truly concurrent computations: a much more abstract model, namely, its *reachability relation* may be sufficient for such purposes. It is well-known that any small category can be collapsed to a binary relation on its objects which is a *preorder*. In exactly this way, the initial model of $\mathcal{R} = (\Sigma, E, R, \phi)$ is collapsed to a preorder, namely, its *reachability initial model*, whose elements are E-equivalence classes $[t]$ of ground terms t; and where the reachabilty relation $[t] \rightarrow_{\mathcal{R}} [t']$ is defined by the equivalence:

$$[t] \rightarrow_{\mathcal{R}} [t'] \quad \Leftrightarrow \quad \mathcal{R} \vdash t \rightarrow t'.$$

It is also possible to distinguish in the initial reachability model between one-step transitions $[t] \rightarrow^{1}_{\mathcal{R}} [t']$, corresponding to the application of a single rewrite rule, and general transitions $[t] \rightarrow_{\mathcal{R}} [t']$, corresponding to zero, one, or more rewrite steps. This distinction is useful for various purposes, for example for giving semantics in the initial reachability model of $\mathcal{R}$ to the next operator $\bigcirc$ in temporal logic, a topic further discussed in Section 3.11.

3.2. Computability and Coherence

For execution purposes a rewrite theory $\mathcal{R} = (\Sigma, E, R, \phi)$ should satisfy some additional requirements. As already illustrated by the `DPLL` example in Section 2.2, the equations E may decompose as a union $E = E_0 \cup B$, where B is a (possibly empty) set of structural axioms, and E_0 is a set of equations used as simplification rules modulo B. We should require that matching modulo B is decidable, and that the equations E_0 are sort-decreasing, ground confluent and terminating modulo B and B-coherent.[4] This makes the initial algebra $\mathcal{T}_{\Sigma/E_0 \cup B}$, that is, the set of states of the system axiomatized by $\mathcal{R}$, *computable*; in fact, equality becomes obviously decidable, and the elements of the initial

[4]For B any combination of associativity and/or commutativity and/or identity axioms, B-coherence can be automatically guaranteed by a simple theory transformation, as done automatically in Maude (see [106, Section 4.8]). As explained in Footnote 5, the notion of coherence of an equational theory $(\Sigma, E_0 \cup B)$, though related, is different from that of coherence of a rewrite theory, which is the main topic discussed in this section.

algebra $\mathcal{T}_{\Sigma/E_0 \cup B}$ have a very simple description as the (irreducible) *canonical forms* $can_{E_0/B}(t)$ of ground terms t by the equations E_0 modulo the axioms B.

What about the computability of the one-step rewrite relation $\rightarrow^1_{\mathcal{R}}$ in $\mathcal{R} = (\Sigma, E, R, \phi)$? If we want the number of states reachable in one step from a given state to be finite, for unconditional rules R we should first of all require that for any rule $r : t \rightarrow t'$ in R we have $vars(t') \subseteq vars(t)$. But because of rewriting logic's **Equality** inference rule, computability is not at all obvious just by requiring $vars(t') \subseteq vars(t)$, or even by further requiring that $E = E_0 \cup B$ with the equations E_0 sort-decreasing, ground confluent and terminating modulo B. The problem is that the term t we rewrite need not be in canonical form, and there may easily be an infinite number of terms having the same canonical form. Otherwise put, model-theoretically the transitions in the initial model $\mathcal{T}_{\mathcal{R}}$, or in its collapse as an initial reachability model, are between states $[t]$ which are $E_0 \cup B$-equivalence classes of terms, and therefore possibly infinite sets. Finding a rewritable term in such a set is the proverbial search for a needle in a haystack and may be undecidable.

Of course, all would be easy if the existence of a one-step rewrite proof $\mathcal{R} \vdash t \rightarrow t'$ guarantees the existence of another such one-step rewrite proof of the form $\mathcal{R} \vdash can_{E_0/B}(t) \rightarrow t''$ such that $[t'] = [t'']$, since then, assuming R is finite, the one-step rewrite relation becomes easily computable: to rewrite $[t]$ what we can do is: (i) compute the canonical form $can_{E_0/B}(t)$ of t, and (ii) try to rewrite $can_{E_0/B}(t)$ with the rules R modulo B in all possible ways. By the assumptions on B and the finiteness of R there is only a *finite* set of such one-step rewrites that can be effectively computed, say, $can_{E_0/B}(t) \rightarrow t_1, \ldots, can_{E_0/B}(t) \rightarrow t_k$. Then the next states reachable from $[t]$ in one step are exactly $[t_1], \ldots, [t_k]$. Furthermore, we can conveniently *represent* such states by their unique canonical forms $can_{E_0/B}(t_1), \ldots, can_{E_0/B}(t_k)$. This is exactly how Maude computes with a rewrite theory: it reduces t to canonical form with E_0 modulo B, and then applies a rule in R modulo B, and keeps doing this until termination or until a user-given maximum number of rewrites with R, that is, of one-step transitions. Similarly, in reachability analysis or model checking, Maude stores the states in the state space as their canonical forms $can_{E_0/B}(t)$.

But is this complete? Couldn't we be *missing rewrite proofs*, and therefore transitions, by adopting this strategy? Completeness is guaranteed if we have the implication:

$$\mathcal{R} \vdash t \rightarrow^1 t' \quad \Rightarrow \quad (\exists t'')\, \mathcal{R} \vdash can_{E_0/B}(t) \rightarrow^1 t'' \wedge [t'] = [t'']$$

where $\mathcal{R} \vdash t \rightarrow^1 t'$ denotes a *one-step* rewrite proof. This property is called the *ground coherence of* R *with* E_0 *modulo* B. If we do not require t to be a ground term, we talk instead of the *coherence*[5] *of* R *with* E_0 *modulo* B.

[5] The notion of coherence of a rewrite theory is related to, but different from, that of coherence of an equational theory $(\Sigma, E_0 \cup B)$. In both cases the issue is to ensure an appropriate notion of completeness of a rewrite relation. For equational theories the relation is that of rewriting with equations E_0 modulo axioms B. Instead, for rewrite theories it is a matter of

This coherence property was first axiomatized by Viry [463, 464]. A similar but weaker property, what Viry calls "weak coherence," was independently identified in [316]. For the case of rewrite theories $\mathcal{R} = (\Sigma, E_0 \cup B, R)$ where $(\Sigma, E_0 \cup B)$ is an untyped equational theory, E_0 is confluent and terminating modulo B, and the axioms B consist of the associativity or the associativity-commutativity of some binary function symbols in Σ, a detailed study of critical pair criteria for checking coherence of R with E_0 modulo B was given by Viry in [467]. Since coherence is such a fundamental property to ensure the computability and efficient executability of rewrite theories, coherence needed to be generalized to support more expressive rewrite theories $\mathcal{R} = (\Sigma, E_0 \cup B, R, \phi)$ with: (i) an order-sorted signature Σ with sorts and subsorts; (ii) possibly conditional equations E_0; (iii) more general axioms B such as any axioms whose equations are unconditional, linear and regular and have a finitary unification algorithm; (iv) conditional rules R which can have a conjunction of equations in their condition; and (v) a frozenness map ϕ. Furthermore, proof methods and tools not only for coherence (the case studied by Viry) but also for *ground* coherence had to be developed. This has been done recently in [161], where the Maude Coherence Checker tool is also described (I further discuss this tool in Section 6.1.1). But of course, to check coherence or ground coherence under such general conditions is only possible if we can first check the confluence and termination of the underlying order-sorted conditional specification $(\Sigma, E_0 \cup B)$. Proof methods for checking confluence of equational theories under such general conditions and a tool (the Maude Church-Rosser Checker (CRC)) are presented in [161] (I discuss the CRC tool in Section 6.1.1). I postpone discussion of the termination methods until Section 3.8, and of termination tools until Section 6.1.

To summarize, equality of states, operations on states, and the one-step rewrite relation are all effectively computable in a finitary rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R, \phi)$ such that: (i) the (possibly conditional) equations E are sort-decreasing, ground confluent and terminating modulo B and B-coherent, and there is a B-matching algorithm; and (ii) the rules in R are coherent with the equations E modulo B and have only equalities and memberships in their conditions, and if they have extra variables in their righthand side or condition which do not appear in the lefthand side, then they are admissible rules in the sense of [106, Section 6.3].

An interesting question to ask is: how expressive is rewriting logic to specify computable transition systems and computable Kripke structures (for more on Kripke structures see Section 3.11)? For equational logic the same question was asked and answered by Bergstra and Tucker in [54]: any computable algebra, i.e., any computable data type, can be specified by a finitary equational theory (Σ, E), where the equations E are confluent and terminating. For rewriting logic the same question has been asked and answered in [332]: any computable tran-

the coherence between *two* rewrite relations modulo B, namely, one with equations E_0, and another with rules R. Early work on the coherence of a set of equations E_0 modulo axioms B includes, e.g., [244, 383, 248].

sition system, resp., computable Kripke structure, is isomorphic to one specified by a *finitary* rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R, \phi)$ satisfying conditions (i)–(ii) and with a chosen kind [*State*] of states, so that the transition system's set of states is the algebraic data type $T_{\Sigma/E \cup B_{[State]}}$, and its transition relation is $\rightarrow^1_{\mathcal{R}}$.

3.3. Unification, Generalization, Narrowing, and Symbolic Reachability

The rewrite rules of a rewrite theory $\mathcal{R}$, and the rewrite sequents we can deduce from it using the inference rules discussed in Section 3.1, are all (implicitly) *universally quantified.* But what about *existential* formulas of the form

$$\exists \overline{x} : t(\overline{x}) \rightarrow t'(\overline{x})$$

with $\overline{x}$ some variables; what do such formulas mean? and how can we reason formally about them? An existential formula $\exists \overline{x}.\ t(\overline{x}) \rightarrow t'(\overline{x})$ is of course a *reachability property.* It says that there is some instance of the state pattern t from which we can reach, by some possibly complex computation, another state which is an instance of the state pattern t'. A negated existential formula $\neg\exists \overline{x}.\ t(\overline{x}) \rightarrow t'(\overline{x})$, which is of course equivalent to the universal formula $\forall \overline{x}.\ \neg(t(\overline{x}) \rightarrow t'(\overline{x}))$, is then an *unreachability property.* Reachability and unreachability properties are among the most useful properties of rewrite theories. Typically, an unreachability property expresses a *safety property* such as an *invariant* (invariants are further discussed in Section 3.11.3). An invariant says that for all the states reachable from a specified set of initial states something bad can never happen. By describing our, possibly infinite, set of initial states as the ground instances of the state pattern t, and likewise describing the *bad* states as the ground instances of the state pattern t', the unreachability property $\forall \overline{x}.\ \neg(t(\overline{x}) \rightarrow t'(\overline{x}))$ says that bad states in t' are never reachable from the initial states in t or, equivalently, that the *complement* of the set of bad states which are ground instances of t' is an *invariant*, relative to the initial states in t. Understood this way, proving the formula $\exists \overline{x}.\ t(\overline{x}) \rightarrow t'(\overline{x})$ means proving that such a supposed invariant can be violated.

So the question now is: how can we *prove* existential formulas of the form $\exists \overline{x}.\ t(\overline{x}) \rightarrow t'(\overline{x})$ for a rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R, \phi)$ (where we assume the good executability properties already discussed in Section 3.2, i.e., that E is confluent and terminating modulo B, and R is coherent with E modulo B)? Prasanna Thati and I studied this question in [340, 450] and gave several conditions on $\mathcal{R}$ and several forms of *narrowing* modulo $E \cup B$ providing complete proof methods for formulas of the form $\exists \overline{x}.\ t(\overline{x}) \rightarrow t'(\overline{x})$. Let me summarize the simplest condition that can be given on $\mathcal{R}$, namely, the frequently occurring case of *topmost* rewrite theories. These are theories having a kind k (a topmost sort in some connected component in the poset of sorts) such that: (i) no operator has k as sort for any of its arguments; and (ii) the terms in all rewrite rules in R are of kind k. For example, the `DPLL` module satisfies these two conditions with $k =$ `Sequent`. Our object-based example in Section 2.1 does not quite satisfy requirements (i) and (ii) because the constructor for configurations `__` has the sort `Configuration` as an argument, but can be easily transformed into a semantically equivalent rewrite theory which does: we can just add a new sort, say, `State`, and declare an operator embracing a whole configuration to make a global distributed state:

```
  op {_} : Configuration -> State .
```

Then, to satisfy condition (ii) we can just place all the rules in our object-based example in the bigger context of a state by adding an extra variable `C` of sort `Configuration` to represent "the rest of the state" (which could be empty). For example, rule `send` now becomes:

```
rl [send] : { < Y : Sender | cell : E , cnt : N , receiver : Z > C }
          => { < Y : Sender | cell : mt, cnt : N > (to Z :: E from Y cnt N) C } .
```

As shown in [340], under conditions (i)–(ii), narrowing with R modulo $E \cup B$ is a complete method for proving formulas of the form $\exists \overline{x}.\ t(\overline{x}) \rightarrow t'(\overline{x})$, that is, for *symbolic reachability analysis*. Specifically, under such conditions $\exists \overline{x} : t(\overline{x}) \rightarrow t'(\overline{x})$ holds for $\mathcal{R}$ iff there is a narrowing sequence $t \rightsquigarrow^{*}_{R,E\cup B} u$ such that u and t' have a $E \cup B$-unifier. Narrowing is just like rewriting, but replacing matching modulo an equational theory by (semantic) unification modulo such a theory. That is, the one-step $(R, E \cup B)$-*narrowing* relation is defined as $t \rightsquigarrow_{R,E\cup B} t'$ iff there is a non-variable position[6] p of t, a (possibly renamed) rule $l \rightarrow r$ in R, and a unifier $\sigma \in \mathit{Unif}_{E\cup B}(t|_p, l)$ such that $t' = \sigma(t[r]_p)$, where $\mathit{Unif}_{E\cup B}(t|_p, l)$ denotes a complete set of *unifiers* of the equation $t|_p = l$, that is, of substitutions θ solving such an equation in the equational theory $E \cup B$, in the sense that $\theta(t|_p) =_{E\cup B} \theta(l)$. This has many applications to automated deduction, verification of safety properties, model checking, and security. Some of these applications were discussed in [340, 188]. I discuss some of the applications to model checking in Section 3.11.2, and to the analysis of cryptographic protocols in Section 7.3.

There is, however, a nontrivial problem, namely, how to obtain practical unification algorithms to compute $\mathit{Unif}_{E\cup B}(t|_p, l)$. If $E = \emptyset$, and B is a set of axioms for which a unification algorithm exists, then things are easy. For example, for the object-based system of sender and receiver objects with buffers in Section 2.1, $E = \emptyset$ and B consists of the axioms of associativity, commutativity and identity for the operators $\_\,\_$ and $\_,\_$ for which there is a *finitary* unification algorithm generating a finite set of solutions. There is, however, the remaining problem that the signature of the above example is *order-sorted* (indeed, the operators $\_\,\_$ and $\_,\_$ have different sorts), whereas the standard unification algorithms modulo associativity, commutativity and identity are unsorted. The paper [235] gives an algorithm, under very general conditions on B, by which one can use an unsorted B-unification algorithm to obtain a complete set of

[6]By viewing a term as a tree, we can represent a positions p in it by a string of natural numbers. For example, in the term $f(a, g(b, c))$, a is at position 1, $g(b, c)$ at postion 2, b at position 2.1, and c at position 2.2. The subterm of t at position p is then denoted $t|_p$. A position p is non-variable, iff $t|_p$ is not a variable.

order-sorted B-unifiers. Currently, Maude supports order-sorted unification for B any combination of: (i) free function symbols; (ii) commutativity axioms; (iii) associativity-commutativity axioms; and (iv) associativity, commutativity and identity axioms [152].

When E is nonempty, the matter of finding a $E \cup B$-unification algorithm is more complex. In principle, one can assume good properties about E such as confluence, termination, and coherence modulo B and use the results in [249] to compute $E \cup B$-unifiers by (E, B)-*narrowing*.[7] But there are two main problems: (i) in general the number of $E \cup B$-unifiers is not finite; and (ii) for $B \neq \emptyset$ unrestricted narrowing can be horribly inefficient in the sense of leading to huge search spaces, and known strategies making narrowing efficient such as basic narrowing can be incomplete. For example, basic narrowing is incomplete when B is the theory of associativity-commutativity (AC) [121]. To make things even worse, it is very easy to give examples of narrowing modulo, e.g., AC such that there is a *finite* set of most general narrowing solutions to a unification problem, but the narrowing algorithm modulo AC will *loop* forever looking for more solutions.

In fact, narrowing with (oriented) equations E modulo axioms B when $B \neq \emptyset$ has been for a long time a *terra incognita*, where little was known about any practical methods to deal with these problems. Using the idea of *variants*[8] of a term proposed by Comon and Delaune in [121], Santiago Escobar, Ralf Sasse and I have defined a complete narrowing strategy with equations E modulo B called *folding variant narrowing*[9] [189] (see also the longer paper [190] in this issue), that is *optimally terminating*, that is, if any complete narrowing strategy terminates on an input term, then folding variant narrowing will terminate on that term. Furthermore, if $E \cup B$ has the so-called *finite variant property* [121], folding variant narrowing will terminate on *all* input terms. For $E \cup B$-unification purposes this means that, if $E \cup B$ has the finite variant property, folding variant narrowing then provides a *finitary* $E \cup B$-unification algorithm.[10] Escobar, Sasse and I have also given methods to check the finite variant property of a theory in [187]. It turns out that many cryptographic

[7] This reduces the problem of computing $E \cup B$-unifiers to a symbolic reachability problem. Specifically, we add a new binary operator $\approx$ and a fresh constant *true* to our syntax, and add a new rule $x \approx x \rightarrow \mathit{true}$ to our equations E oriented as rewrite rules. Then the $E \cup B$-unification problem $\exists \overline{x}.\ t(\overline{x}) = t'(\overline{x})$ is transformed into the symbolic reachability problem $\exists \overline{x} : t(\overline{x}) \approx t'(\overline{x}) \rightarrow \mathit{true}$ for the rewrite theory with equations B and rules $E \cup \{x \approx x \rightarrow \mathit{true}\}$, which is solved by narrowing with rules $E \cup \{x \approx x \rightarrow \mathit{true}\}$ modulo B.

[8] The $E \cup B$-variants of a term t are pairs (u, θ) with $u = \mathit{can}_{E/B}(\theta(t))$ and θ some substitution. Therefore, the variants of t are essentially the irreducible *patterns* to which any instance of t may evaluate.

[9] Variant narrowing is a narrowing strategy which, given an input term t, computes a complete set of $E \cup B$-variants of t. The folding version of this strategy uses subsumption modulo B to avoid computing any variant which is a substitution instance modulo B of a more general variant.

[10] Using the ideas in Footnote 7, computing the $E \cup B$-unifiers of the equation $u = v$ by folding variant narrowing amounts to computing (a complete set among) those $E' \cup B$-variants of the term $u \approx v$ which are of the form (true, θ), for $E' = E \cup \{x \approx x \rightarrow \mathit{true}\}$.

theories of interest have the finite variant property [121]. I explain in Section 7.3 how —using folding variant narrowing to compute $E \cup B$-unifiers *and* narrowing with protocol rules R modulo $E \cup B$ to perform symbolic reachability analysis— this has been exploited in the Maude-NPA protocol analyzer [183] to provide complete formal analysis for security protocols modulo a variety of cryptographic theories. More generally, Maude 2.6 supports variant narrowing, and symbolic reachability analysis of topmost rewrite theories, modulo a large class of equational theories $E \cup B$ having the finite variant property [152].

Generalization is the dual of unification. Given two terms t and t', a set of most general B-unifiers for the equation $t = t'$ is, as already mentioned, a set $Unif_B(t, t')$ giving us a set of *most general instances* $\{\theta(t) \mid \theta \in Unif_B(t, t')\}$, which are *common instances* of t and t' up to B-equivalence, i.e., $\theta(t) =_B \theta(t')$. But we can ask the dual question: given terms t and t', can we compute a set $Gral_B(t, t')$ of *least general patterns* of which t and t' are instances modulo B, i.e., least general terms u such that there are substitutions θ, ρ with $\theta(u) =_B t$ and $\rho(u) =_B t'$? For example, for $B = \emptyset$ and Σ untyped, the terms $f(f(a, a), b)$ and $f(f(b, b), c)$ have a least general generalization in the pattern $f(f(x, x), y)$. Generalization has many useful applications, for example, to automated deduction, machine learning, testing, and partial evaluation. María Alpuente, Santiago Escobar, Pedro Ojeda and I have developed generalization algorithms for two cases that are important for rewriting logic, namely, order-sorted generalization [17], and generalization modulo B, for B any combination of associativity and/or commutativity and/or identity axioms [16].

3.4. Reflection

Reflection is a very important property of rewriting logic [113, 102, 115, 116]. Intuitively, a logic is reflective if it can faithfully represent its metalevel at the object level. Specifically, rewriting logic can faithfully represent its own theories and their deductions by having a finitely presented rewrite theory $\mathcal{U}$ that is *universal*, in the sense that for any finitely presented rewrite theory $\mathcal{R}$ (including $\mathcal{U}$ itself) we have the following equivalence

$$\mathcal{R} \vdash t \to t' \;\; \Leftrightarrow \;\; \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \to \langle \overline{\mathcal{R}}, \overline{t'} \rangle,$$

where $\overline{\mathcal{R}}$ and $\overline{t}$ are terms representing $\mathcal{R}$ and t as data elements of $\mathcal{U}$. Since $\mathcal{U}$ is representable in itself, we can achieve a "reflective tower" with an arbitrary number of levels of reflection [113, 102, 115], since we have

$$\mathcal{R} \vdash t \to t' \; \Leftrightarrow \; \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \to \langle \overline{\mathcal{R}}, \overline{t'} \rangle \; \Leftrightarrow \; \mathcal{U} \vdash \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \overline{t} \rangle} \rangle \to \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \overline{t'} \rangle} \rangle \ldots$$

Reflection is a very powerful property: (i) it allows defining rewriting strategies by means of metalevel theories that extend $\mathcal{U}$ and guide the application of the rules in a given object-level theory $\mathcal{R}$ (this is further discussed in Section 3.5); (ii) it is efficiently supported in the Maude implementation by means of *descent functions* [104] in the `META-LEVEL` module; (iii) it can be used to build a variety of theorem proving and theory transformation tools (this is further discussed in Sections 4.1 and 6.1); (iv) it can endow a rewriting logic language like Maude with powerful theory composition operations [159, 150, 151, 160]; (v) it can be used to prove metalogical properties about families of theories in rewriting logic, and about other logics represented in the rewriting logic metalogical framework [50, 109] (this is further discussed in Section 4.1); and (vi) has important connections with distributed object-based reflection and adaptation [338].

3.5. Strategies

Recall the `DPLL` rewrite theory in Section 2.2. The most complex aspect of a SAT solver is precisely its heuristics or strategy. In the case of the rewrite theory specified in `DPLL` this means that performance will crucially depend on the strategies used to apply the `split1` and `split2` rewrite rules. In a more sophisticated SAT solver supporting back-jumping, conflict resolution and clause learning, the situation is similar: performance will crucially depend on the strategies guiding the application of the Abstract DPLL inference rules in [356]. Of course, this is a general issue that applies not just to SAT solving but to any rewrite theory; and that involves not only performance but also any *goal-oriented* use of a rewrite theory. The key issue is the potential *nondeterminism* of rules, as opposed to the *determinism* of confluent and terminating equations.

Strategies are still relevant for equations for performance and termination reasons, even when the equations are confluent and terminating, or to ensure their termination as in the case of context-sensitive rewriting for equations (see, e.g., [290] and references there). Context-sensitive rewriting of equational specifications is supported by OBJ, CafeOBJ, and Maude. Note that the addition of a frozenness map ϕ to a generalized rewrite theory, as explained in Section 3.1.2, provides a similar form of context-sensitive rewriting at the rule level, as opposed to the equation level.[11] But for nondeterministic rules, strategies become a much more essential issue, because such rules, depending on when and where they are applied, can yield totally different outcomes. Frozenness provides a very simple form of strategic rewriting with rules, but more than frozenness is needed.

The role of strategies is to tame the potentially wild nondeterminism of rules for various purposes, which may include: (i) *realistic modeling* of the behavior of a truly nondeterministic system, whose nondeterminism we cannot or we do not intend to control, but where some behaviors may be utterly unrealistic; and (ii) goal-oriented (and perhaps performance-oriented) *control* of the nondeterminism in a system's execution. It is of course possible to mix purposes (i) and (ii): for example, we may have an asynchronous object system where the asynchronous behavior is only restricted by a few fairness assumptions, but where the objects are intelligent and use sophisticated game-theoretic strategies when interacting with each other. In all cases, what strategies do is to *restrict* the

[11] Maude supports both forms of context-sensitive rewriting: with equations using the `strat` attribute, and with rules using the `frozen` attribute.

set of all possible dynamic behaviors of the system axiomatized by the given rewrite theory. That is, roughly speaking a strategy determines a *subset* of the set of all the possible computations of a system specified by a rewrite theory $\mathcal{R}$, where those computations need not be just the finite ones but may also include infinite computations.

If we are modeling a concurrent, asynchronous system whose nondeterminism is an intrinsic fact of life which cannot really be controlled, and we want to *simulate* such a system, strategies may still be relevant, not so much to control the outcome of system executions as to observe the behavior of the system under *realistic assumptions* about its execution. Recall the example of sender, receiver, and buffer objects in Section 2.1. It is easy to extend such a system to one where there are also sensor objects that are periodically writing numerical data observations into the sender's buffer. In this way the system immediately becomes a nonterminating reactive system. Such a system can have executions that are totally unrealistic. For example, a sensor can be regularly writing new data into the sender's buffer, the sender object can be sending this potentially infinite stream of data to the receiver, but the receiver *never* receives anything! Intuitively, such a behavior is *unfair*. Therefore, *fair strategies*, which restrict the set of behaviors to those were starvations such as this are ruled out, are very important to model a system's behavior realistically, and to reason formally about system properties such as termination or satisfaction of temporal logic formulas (I further discuss fair termination in Section 3.8, and model checking of temporal logic formulas under fairness assumptions in Section 3.11). As explained in [323], fair rewriting is not just a matter of *rule fairness*, that is, of making sure that all rewrite rules are given a chance to be executed. For example, in the above concurrent object system with sensor, buffer, sender and receiver objects, if we have two different sensors hooked up to two different senders through their respective buffers and two corresponding receiver objects with their own buffers, we can be rule fair by making sure that the `receive` and `write` rules are executed infinitely often; but we can still starve one of the receivers, just by only executing `receive` and `write` rules for the other. That is, we here need not only rule fairness but also *object* fairness: each object should be treated fairly. The general notion is that of *localized fairness* in rule applications [323]. This is of course important to obtain realistic simulations. For example, Maude provides rule fair executions through its `rewrite` command; and rule and position fair executions through its `frewrite` command, which becomes also object fair for object-based concurrent systems specified with a multiset union operator using the `config` keyword, as illustrated in the example of Section 2.1. But what can be done if we want to obtain fair behaviors besides the ones provided by a language implementation? Fairness is just a particular kind of temporal logic property. More generally, we can view a temporal logic formula as a strategy expression which defines a corresponding class of behaviors. In Section 3.11.2, I explain how an expressive temporal logic such as *TLR* can be used as a strategy language, which is then implemented by a model checker.

If instead our purpose is to *control* the nondeterministic behavior of a rewrite theory $\mathcal{R}$ for goal-oriented and perhaps performance-oriented purposes, an appropriate way to achieve that end is to provide a *strategy language* that can be used to guide and control the way in which the rules of $\mathcal{R}$ are applied. To give a logical example, $\mathcal{R}$ can be the inference system of a theorem prover or of a SAT solver, and then the strategies correspond to proof *tactics* or to solving *heuristics*. In concurrent system applications the relevant strategies may have other purposes, such as, for example, having a winning strategy in a game-theoretic interaction between agents. Given all these useful purposes, different rule-based languages such as, for example, ELAN [71, 70], Maude [113, 114, 303], and Stratego [468], provide strategy languages to guide and control rule executions. The ELAN researchers deserve much credit as pioneers in this area for having made key contributions to rewriting strategy ideas from the beginning of the ELAN language.

For modularity and reasoning purposes it is very useful to keep a clear separation between the rewrite theory $\mathcal{R}$ and the strategies used to control it. As discussed in Section 2.2, this was one of the key motivations of Tinelli in seeking formal specifications of SAT solvers by inference systems, so that the proof of correctness of a SAT solver is completely decoupled from its, possibly quite complex, heuristics. Following this point of view, a strategy language SL is understood in [303] as a *theory transformation* of the form:

$$(\mathcal{R}, SM) \mapsto SL(\mathcal{R}, SM)$$

where SM is a *strategy module* completely separated from the rewrite theory $\mathcal{R}$, and $SL(\mathcal{R}, SM)$ is a transformed rewrite theory which executes the rules in $\mathcal{R}$ using the strategy expressions of SM. Modularity and separation of concerns are thus achieved, because we can have different strategy modules, say, $SM_1, \ldots, SM_n$, to control the executions of the same rewrite theory $\mathcal{R}$ in different ways for different purposes. The fact that $SL(\mathcal{R}, SM)$ is *another rewrite theory* means that the *operational semantics* of the strategy language SL is also defined by rewriting, as done, for example, in [71, 70, 114, 303]. But what is now rewritten is not just a term t in $\mathcal{R}$, but a pair $s \mathbin{@} t$, consisting of a strategy expression s in SM which is *applied* to a term t in $\mathcal{R}$. What the term $s \mathbin{@} t$ rewrites to are *solutions* (plus possibly pending strategy tasks); that is, terms t' in $\mathcal{R}$ that are *reachable* from t when the rules in $\mathcal{R}$ are applied according to the strategy s. Therefore, one can also give to SL a more abstract *set-theoretic semantics* that assigns to $s \mathbin{@} t$ the *set* of all its solutions, as done, for example, in [71, 70, 303].

Of course, the theory $SL(\mathcal{R}, SM)$ manipulates or controls the theory $\mathcal{R}$. It needs to know and handle notions such as term, subterm, rule, position, matching substitution, and so on. This makes an explicit use of reflection in the definition of $SL(\mathcal{R}, SM)$ very natural, in the sense that $SL(\mathcal{R}, SM)$ can be viewed as a rewrite theory that extends the universal theory $\mathcal{U}$ with special combinators aimed at controlling the execution of $\mathcal{R}$ at the metalevel. This has been the approach taken in Maude since its first strategy languages until its current one [113, 114, 303]. In this way, strategies are made *internal* to rewriting logic itself. There are of course various requirements that one would like a strategy language to satisfy, the most basic one being its *soundness*, i.e., only terms reachable from t in $\mathcal{R}$ should be among the solutions of $s \,@\, t$. The paper [303] discusses several such requirements, emphasizing the fact that the *determinism* of $SL(\mathcal{R}, SM)$ is a highly desirable feature: since we want to control the nondeterminism of $\mathcal{R}$, once we fix a strategy s, the solutions of $s \,@\, t$ should not depend on how $s \,@\, t$ is executed in $SL(\mathcal{R}, SM)$, in the sense that any possible solution not yet seen should always be obtainable by further rewriting.

An important area where more advances are needed is that of formal reasoning about rewriting with strategies. Useful formal reasoning techniques and tools already exist for proving *termination* under some notion of strategy: I discuss work on termination under fairness, context-sensitive termination, and termination under ELAN strategies in Section 3.8. However, other formal reasoning methods are less developed; for example, the paper [289] studies conditions for context-sensitive confluence, but the conditions are quite strong.

3.6. The ρ-Calculus

One of the attractive aspects of the λ-calculus is that it is very simple, both in its syntax and its rules, yet all of higher-order functional programming can be encoded in it, or in some variant of it such as a typed version. Couldn't there be a similar calculus for rewriting? And could such a calculus be general enough as to naturally embed the λ-calculus as a subcalculus? Horatiu Cirstea and Claude Kirchner both posed these intriguing questions and gave an elegant positive answer to them in their ρ-calculus [95, 96, 97]. The key idea is to replace the λ-abstraction operator $\lambda x.u$ by a ρ-abstraction $t \rightharpoonup u$, where the role of the bound variable x in $\lambda x.u$ is now played by the bound *term* t in $t \rightharpoonup u$. As in the λ-calculus, there is also an *application operator* $[\_]\_$. The intended meaning of an application $[t \rightharpoonup u](v)$ is to rewrite the term v at the top with the rewrite rule $t \rightarrow u$. The λ-calculus is then naturally encoded in the ρ-calculus as a special case. For example, the λ-term $\lambda x.(y\ x)$ is encoded as the ρ-term $x \rightharpoonup [y](x)$. The entire ρ-calculus is then described by a small set of evaluation rules; furthermore, such evaluation rules, particularly the **Fire** rule, can be made parametric on the *matching algorithm* employed, i.e., the ρ-calculus can express not only syntactic rewriting, but also rewriting modulo axioms such as associativity-commutativity. In similarity to the λ-calculus, there are also typed versions of the ρ-calculus [99, 287], and even a "ρ-cube" [98].

From the point of view of reflection, the ρ-calculus can be understood as a convenient simple calculus specifying a *universal theory* (modulo using an explicit substitution calculus such as, e.g., CINNI [430] to turn the ρ-calculus itself into a first-order rewrite theory). Indeed, it is shown in [96, 97] that the ρ-calculus can faithfully simulate at the metalevel the rewriting behavior of any other rewrite theory. Since, as pointed out in Section 3.5, from a reflective point of view a strategy language SL can be understood as the addition of appropriate strategy combinators to a universal theory $\mathcal{U}$, it is entirely natural to see that one of the important uses of the ρ-calculus has been to give a rewriting semantics at the metalevel to strategy languages such as ELAN, and that the ρ-calculus itself has been extended with such strategy combinators to become in effect a powerful strategy language [100].

3.7. Sufficient Completeness

Given a rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R, \phi)$, with good executability conditions such as E being ground confluent and terminating modulo B, and R being coherent with E modulo B, we can represent its states uniquely up to B-equality as canonical forms $can_{E/B}(t)$ with t a ground term. The equations E may define various *auxiliary functions* (for example, numerical functions), which operate on some parts of the state, that is, that manipulate elements of the initial algebra $\mathcal{T}_{\Sigma/E\cup B}$. Therefore in $can_{E/B}(t)$ all such auxiliary functions should have already disappeared and only *state constructors* should remain. This is the (equational) *sufficient completeness* problem: given a subsignature $\Omega \subseteq \Sigma$ of operators called *constructors*, is it the case that for any ground Σ-term t, the term $can_{E/B}(t)$ is an Ω-term? If this holds, $(\Sigma, E \cup B)$ is called *sufficiently complete* with respect to the constructor subsignature Ω; if it fails to hold, this is clear indication that we have *not given enough equations* to define some auxiliary function $f \in \Sigma - \Omega$, so that there is something wrong with the specification. For a rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R)$ this means that there are *extra states* that we had not intended to have in our system and which are not built by the state constructors Ω alone.

It is therefore important to check that an equational theory $(\Sigma, E \cup B)$, or the equational part of a rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R, \phi)$, is sufficiently complete. When $B = \emptyset$, Σ is unsorted, and the equations E are unconditional, several algorithms to check sufficient completeness are known (see, e.g., [120] and references there). An attractive possibility is to further assume that the equations E are *left-linear* (i.e., if $(t = t') \in E$, then each variable x in t occurs at a *single position* p of t), because then the problem can be reduced to an emptiness problem for *tree automata* (see [120]). In general, however, one would like to have sufficient completeness proof methods that can apply more broadly to: (i) order-sorted or even membership-equational signatures; (ii) modulo axioms B; and (iii) with E containing conditional equations and even conditional memberships. In such a broad generality the problem becomes undecidable, but proof obligations can be generated. For example, the tool described in [232] addresses (i) and (iii) by providing a decision procedure to check the sufficient completeneess of unconditional order-sorted equational theories without requiring left linearity, and generates proof obligations which are sent to the Maude Inductive Theorem Prover (ITP) (see Section 6.1.5), to prove sufficient completeness of order-sorted and membership-equational conditional specifications. Instead, the Maude Sufficient Completeness Checker tool (SCC) [236, 234] addresses (i) and (ii) by providing a decision procedure which can check sufficient completeness of order-sorted equational specifications modulo combinations of associativity and/or commutativity and/or identity axioms when the equations E are unconditional and left-linear. The SCC tool reduces the problem to an emptiness problem for *propositional tree automata* [238], and uses the CETA library that efficiently implements tree automata operations for propositional tree automata [231]. As already mentioned, sufficient completeness for membership equational logic (MEL) is in general undecidable, but proof obligations can be generated. The MEL sufficient completeness problem has been studied in [72, 237, 231].

For a rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R, \phi)$ there are actually *two* different sufficient completeness problems. The first, of course, is the *equational* sufficient completeness of its equational part $(\Sigma, E \cup B)$ relative to a constructor subsignature Ω described above. The second problem is the sufficient completeness of the *rules* R. But what does that mean? If $(\Sigma, E \cup B)$ is sufficiently complete in the equational sense, are not all states of $\mathcal{R}$ already representable as Ω-constructor terms of the form $can_{E/B}(t)$? Yes indeed, but what about the set of *final* states, that is, states for which it is not possible to perform any further transitions with R? They are in general a subset of all ground Ω-terms, so that they may be describable by an even smaller constructor subsignature $\Lambda \subseteq \Omega \subseteq \Sigma$. By specifying Λ, a user makes clear a set of state constructors that is enough to generate all such final states. What is then a failure of sufficient completeness for the rules R? What does it mean? It means exactly a violation of *deadlock freedom.* A *deadlock* is an unintended and unwanted final state. Lack of sufficient completeness for R means that there is a final state of $\mathcal{R}$ which is not a Λ-term, that is, $\mathcal{R}$ has a deadlock. Therefore, checking sufficient completeness of R means checking deadlock-freedom. This has been proposed by Camilo Rocha and me in [397], where we show that the same propositional tree automata techniques used to verify sufficient completeness for order-sorted equational specifications modulo axioms can be extended to check sufficient completeness of the rules R in $\mathcal{R}$ under the assumption that they are unconditional, left-linear, and weakly terminating; we also extend the Maude SCC tool to also support such checking. For the case of rewrite theories of the form $\mathcal{R} = (\Sigma, \emptyset, R)$, with Σ unsorted and R unconditional, a different method to check the sufficient completeness of R using narrowing techniques has been proposed by I. Gnaedig and H. Kirchner in [213].

3.8. Termination

Termination of a rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R, \phi)$ is a very important problem, and there is a rich body of termination techniques for term rewriting systems that can be used. However, the standard termination proof methods address the much simpler case of *untyped* rewrite theories of either the form $\mathcal{R} = (\Sigma, \emptyset, R)$, or the form $\mathcal{R} = (\Sigma, B, R)$ for some restricted set B of axioms. These standard methods are clearly insufficient for rewrite theories and need to be substantially generalized in several dimensions such as: (i) support for sorts, subsorts, and memberships; (ii) support for conditional rules with extra variables in their conditions[12] in both E and R; (iii) the existence, when

[12]The use of extra variables in conditions, which are instantiated incrementally, greatly increases the expressive power of specifications. See [106, Sections 4.6 and 6.3] for the executability conditions required in Maude for such specifications.

E and R are conditional, of *two* separate rewrite relations $\to_E$ and $\to_R$ that cannot be easily combined into a single one; (iv) the need to support a wide range of equational axioms B containing at the very least any combination of associativity and/or commutativity and/or identity axioms; and (v) support for context-sensitive rewriting. Furthermore, standard termination methods were developed in the context of equational logic and automated deduction and do not address important kinds of termination relevant for rewriting logic applications such as: (a) termination under fairness assumptions; (b) termination under strategies; and (c) probabilistic termination.

To address problems (i)–(v) in the context of generalized rewrite theories $\mathcal{R} = (\Sigma, E \cup B, R, \phi)$ whose equational part is a (possibly conditional) membership equational theory (Σ, E), the first thing to observe is that the "vanilla flavored" description of the computations by a single rewrite relation $\to_R$, or even by two relations $\to_E$ and $\to_R$, is utterly inadequate, because the computation of the membership relations $t : s$ is just as important and is entwined with that of rewrites using $\to_E$ and $\to_R$. What one needs to make explicit is an *inference system* involving both rewrites (with R and E) and memberships. This, in turn, poses the problem of *conditional termination* not in terms of a rewrite relation $\to_R$, but in terms of different logics with different inference systems. This has led to proposing the notion of *operational termination* in [155], not only for membership rewriting, but for logical inference systems in general. Although very general, this notion is also very practical, because it captures the idea of an interpreter carrying out the inference steps, so that operational termination means that such an interpreter will never loop. Even for the vanilla-flavored case of untyped conditional rewrite theories $\mathcal{R} = (\Sigma, \emptyset, R)$ this notion provides useful insights: as shown in [291], operational termination coincides there with the notion of quasi-decreasing conditional term rewriting systems, making it clear that other conditional rewrite systems, which are *soi disant* terminating, such as those enjoying "effective termination," are not effective at all, since interpreters can loop on such systems [155]. The relations of operational termination with other notions of conditional termination for untyped conditional term rewriting systems have been futher investigated in [414].

Although the approach to the operational termination of membership rewrite theories in [155] already dealt with rewriting modulo axioms B, and was extended in [157] to deal simultaneously with the relations $\to_E$ and $\to_R$ plus the memberships $t : s$, there is great practical interest in being able to use existing state-of-the-art termination tools for term rewriting systems to prove the termination of generalized rewrite theories $\mathcal{R} = (\Sigma, E \cup B, R, \phi)$ beyond their scope. To bridge this gap, several important problems need to be solved. First, the rewrite theories $\mathcal{R} = (\Sigma, E \cup B, R, \phi)$, or even the membership equational theories (Σ, E) need to be transformed into *untyped* vanilla-flavored term rewriting systems, eliminating features such as sorts, subsorts, memberships, and even conditions. This is accomplished in [155, 157] by appropriate *non-termination preserving* theory transformations. The second problem is that the sets of axioms B for which proofs of termination modulo B are supported in existing tools are quite restricted. To solve this problem, semantics-preserving theory transformations based on the notion of variant (see Section 3.3) that transform a rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R, \phi)$ into a semanticaly equivalent one $\widehat{\mathcal{R}} = (\Sigma, \widehat{E} \cup \widehat{D} \cup B_0, \widehat{R}, \phi)$ with simpler axioms B_0, where $B = B_0 \cup D$, are presented in [158]. However, transformational methods come at a nontrivial cost, since the transformed theories are usually more complex. Therefore, more intrinsic proof methods to handle the above two problems are also of great interest. For example, in [294] the transformations in [155] are replaced by transformations into *order-sorted* rewrite theories, which still keep a lot of sort information, and in [292] dependency-pair-based methods are generalized from the unsorted to the order-sorted level. Similarly, in [10] intrinsic methods to prove termination modulo useful combinations of equational axioms by dependency pair techniques are proposed. The advantages of intrinsic methods over transformational ones are also clear in proofs of context-sensitive termination (see, e.g., [8, 225]). Many of the above-mentioned techniques for proving termination of rewrite theories are already supported by the Maude Termination Tool (MTT), which I discuss in Section 6.1.3.

My current view is that the class of *order-sorted* rewrite theories of the form $\widehat{\mathcal{R}} = (\Sigma, B_0, R, \phi)$, where: (i) B_0 is the widest possible class of axioms for which dependency pair proof methods are available; and (ii) the rules R are unconditional, is a good target class for which intrinsic methods should be further developed, since the transformations of general rewrite theories into that class become much simpler than the transformations into untyped rewrite theories, and therefore the proof methods will become considerably more effective in practice.

Another, orthogonal set of techniques that need to be further developed in order for termination proofs to scale up to large rewrite theories are *modularity* techniques that work at the richer level of at least order-sorted rewrite theories modulo axioms B_0. At the vanilla-flavored level of untyped rewrite theories of the form $\mathcal{R} = (\Sigma, \emptyset, R)$, there is already a substantial body of such techniques available (see, e.g., [358, 456]), and even some very useful work for untyped rewrite theories of the form $\mathcal{R} = (\Sigma, AC, R)$, with AC associative-commutative axioms [296]. Felix Schernhammer and I have initiated the study of modularity techniques for the termination of unconditional order-sorted specifications modulo combinations of associativity and/or commutativity and/or identity axioms in [415].

All the termination techniques described above provide an important necessary core. However, this core is not sufficient to cover important applications. Suppose that our rewrite theory $\mathcal{R}$ specifies a communication protocol whose termination we want to prove. Very often $\mathcal{R}$ will *not* terminate in the standard sense, but will terminate *under appropriate fairness assumptions*. That is, infinite rewrite sequences do exist, but all such sequences are *unfair* and therefore unrealistic. For example, the simple communication protocol in Section 2.1 can be easily extended to a fault-tolerant one that can operate in a lossy medium by: (i) modeling the lossy medium by a rewrite rule which can destroy a message (rewrite it to the `none` configuration); (ii) modifying the `receive` rule, so that an acknowledgment is sent back to the sender; and (iii) modifying the `send` rule so that the sender keeps resending the n-th item without emptying its cell until an acknowledgment for it is received. Since now any message can be destroyed before it is received, plus a sender can keep resending a message forever, the system is no longer terminating. However, under fairness assumptions about how each receiver object will apply the `receive` rule, and each sender object will receive acknowledgments and clear its cell, the fault-tolerant system is indeed fairly terminating. Proof techniques for termination of rewrite theories under fairness assumptions have been studied in [293], substantially extending prior work in [386, 387]. Another way in which termination techniques need to be extended is to reason about termination of $\mathcal{R}$ when executed under a given strategy (see Section 3.5). This extension has been carried out in [199, 214] and is supported by the CARIBOO tool, which I discuss in Section 6.1.2. Yet another topic requiring a substantial extension of standard termination techniques is the termination of probabilistic rewriting, a topic investigated in [212] (for a discussion of probabilistic rewriting and the different notions that have been proposed see Section 3.10).

3.9. Real-Time Rewrite Theories

In many reactive and distributed systems, including, for example, schedulers, networks, and so-called cyber-physical 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 related perspectives. On the one hand, an extension of rewriting logic called *timed rewriting logic* has been investigated, and has been applied to several examples and specification languages [273, 366, 274, 429]. On the other hand, Peter Ölveczky and I found a simple way to express real-time and hybrid system specifications *directly* in rewriting logic [367, 359, 368, 371]. Such specifications are called *real-time rewrite theories* and have rules of the form

$$\{t\} \xrightarrow{r} \{t'\} \;\; \mathit{if} \;\; C$$

with r a term denoting the *duration* of the transition (where the time can be chosen to be either discrete or continuous), $\{t\}$ representing the *whole* state of a system, and C an equational condition. Peter Ölveczky and I showed that, by making the clock an explicit part of the state, these theories can be *desugared* into semantically equivalent ordinary rewrite theories [367, 359, 368]. That is, in the desugared version we can model the state of a real-time or hybrid system as a pair $(\{t\}, r_0)$, with $\{t\}$ the current state and r_0 the current global clock time. Then the above rule becomes desugared as

$$(\{t\}, r_0) \rightarrow (\{t'\}, r_0 + r) \;\; \mathit{if} \;\; C$$

Rewrite rules can then be either *instantaneous rules*, that take no time and only change some part of the state t, or *tick rules*, that advance the global time of the system according to some time expression r and may also change the global state[13] t. By characterizing equationally the enabledness of each rule and using conditional rules and *frozen* operators [79], 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 [369]. When time is continuous, tick rules may be *nondeterministic*, in the sense that the time r advanced by the rule is not uniquely determined, but is instead a parametric expression (however, this time parameter is typically subjected to some equational condition C). In such cases, tick rules need a *time sampling strategy* to choose suitable values for time advance.

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 [368]), an important advantage of the above approach is that one can use an existing implementation of rewriting logic to execute and formally analyze real-time specifications. Because of some technical subtleties, this seems difficult for the alternative of timed rewriting logic, although a mapping into the above framework does exist [368].

Of course, one would like to simulate and formally analyze real-time systems specified as real-time rewrite theories. The Real-Time Maude tool [359, 371] has been developed for this purpose (I further discuss Real-Time Maude in Section 6.1.8). In this way, a wide range of applications, including schedulers, networks, cyber-physical systems, and real-time programming and modeling languages, have been specified (I discuss such applications in Section 7.4), and have been formally analyzed by model checking their temporal logic properties (I discuss the model checking of temporal logic properties, including the model checking of such properties for real-time systems in Section 3.11.2).

3.10. Probabilistic Rewrite Theories

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 [276, 277, 5]. Rules in such theories are *probabilistic rewrite rules* of the form

$$l : t(\vec{x}) \to t'(\vec{x}, \vec{y}) \;\; \textit{if} \;\; \textit{cond}(\vec{x}) \;\; \textit{with probability} \;\; \vec{y} := \pi_r(\vec{x})$$

[13]Instantaneous rules need not involve the global state: they can be local (for example, local to a give object, which receives a message) and can be applied concurrently; only tick rules, which change the global time and must reflect the effects of time elapse everywhere (for example, in all timers) need to be global and must rewrite the entire state.

where the first thing to observe is that the term t' has new variables $\vec{y}$ disjoint from the variables $\vec{x}$ appearing in t. Therefore, such a rule is *nondeterministic*; that is, the fact that we have a matching substitution θ such that $\theta(cond)$ 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 $\vec{y}$ in t'. In fact, we can denote the different such next states by expressions of the form $t'(\theta(\vec{x}), \rho(\vec{y}))$, where θ is fixed as the given matching substitution, but ρ ranges along all the possible substitutions for the new variables $\vec{y}$. The probabilistic nature of the rule is expressed by the notation: *with probability* $\vec{y} := \pi_r(\vec{x})$, where $\pi_r(\vec{x})$ is a probability distribution *which may depend on the matching substitution* θ. We then choose the values for $\vec{y}$, that is, the substitution ρ, probabilistically according to the distribution $\pi_r(\theta(\vec{x}))$.

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 rational 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 in PMaude notation (see Section 6.1.9) by means of the probabilistic rewrite rule

```
rl [tick]: clock(T,C) => if B then clock(T + 1,C - (C / 1000))
                              else broken(T,C - (C / 1000))
                         fi
                 with probability B := BERNOULLI(C / 1000) .
```

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 [276], probabilistic rewrite theories can express a wide range of models of probabilistic systems, including continuous-time Markov chains [437], probabilistic nondeterministic systems [388, 418], and generalized semi-Markov processes [211]; they can also naturally express probabilistic object-based distributed systems [277, 5], 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 [336].

A completely different notion of probabilistic rewriting has been proposed in [76, 74]. The key idea in both of these papers is that the rewrite rules themselves, $r : t \to t'$, are still *deterministic* (the lefthand side t' has no extra variables); what is probabilistic is the *choice* of which rule to apply and where. In [76] it is shown how such choices can be defined in quite sophisticated ways by probabilistic ELAN strategies to model, for example, probabilistic algorithms; and in [74] ordinary deterministic rewrite rules are endowed with weights to achieve a notion of probabilistic rewrite system. A good way to understand how the ideas in [76, 74] are different from those in [276, 277, 5] is to observe that in a rewrite theory $\mathcal{R}$ there are two completely different potential sources of nondeterminism: (i) the choice of which rule to apply at any given moment and where to apply it; and (ii) once a choice of rule, term position and matching substitution has been made, if the rule $r : t(\vec{x}) \to t'(\vec{x}, \vec{y})$ has extra variables $\vec{y}$ on its righthand side, the choice of a ground substitution ρ to instantiate the variables $\vec{y}$. The semantics in [76, 74] makes the choice (i) probabilistic while keeping the rules themselves deterministic; while the semantics in [276, 277, 5] keeps the choice (i) nondeterministic while making the instantiation of nondeterministic rewrite rules governed by probability distributions that are parametric on the lefhand side's matching substitution. A final observation to make is that the existence of nondeterminism in the choice (i) of which transition to fire and where, with the transitions themselves being probabilistic in their outcome, is well-known in the modeling of probabilistic systems, e.g., in probabilistic nondeterministic systems [388, 418]; and in the probabilistic model checking of such systems, which introduces the notion of a *scheduler* to eliminate the nondeterminism in the choice of transitions, and then model checks the system considering all such possible schedulers.

It is highly desirable to be able to specify, simulate and analyze probabilistic systems specified as probabilistic rewrite theories. The PMaude language design [5] has exactly this purpose; I further discuss PMaude in Section 6.1.9. The kinds of possible formal analyses go beyond simulations and include *statistical model checking* with respect to properties expressed in either a probabilistic temporal logic or even a *quantitative* probabilistic temporal logic where the result of evaluating a formula on a path is a real number corresponding to some quantity associated to a system behavior. I discuss probabilistic temporal logics and model checking of probabilistic properties in Section 3.11. Many applications to probabilistic systems are thus made possible; I discuss some of them in Section 7.5.

3.11. Temporal Logic Properties

As already observed at the end of Section 3.1.2, the reachability initial model of a rewrite theory $\mathcal{R} = (\Sigma, E, R, \phi)$ has an associated one-step rewrite relation $[t] \to^1_{\mathcal{R}} [t']$ relating the states, i.e., the E-equivalence classes $[t]$ of ground Σ-terms t. Since $\mathcal{R}$ can have different sorts and kinds, we should furthermore specify which is the *preferred kind of states*, so that terms of other kinds describe state fragments, or data components of the state, but not an entire state of our system. Let $[State]$ be such a kind. Then we can associate to $\mathcal{R}$ a *transition system*, namely, the pair $(T_{\Sigma/E_{[State]}}, \to^1_{\mathcal{R}})$ where $T_{\Sigma/E_{[State]}}$ denotes the set of E-equivalence classes $[t]$ of ground Σ-terms t of kind $[State]$. Without loss of generality we may also assume that the equations E already define a desired collection of *state predicates* (if they do not, we can just add new function symbols and equations defining such state predicates as Boolean-valued functions). That is to say, we can associate to $\mathcal{R}$ not just a transition system

$(T_{\Sigma/E_{[State]}}, \to^1_{\mathcal{R}})$, but in fact a *Kripke structure*[14] $(T_{\Sigma/E_{[State]}}, \to^1_{\mathcal{R}}, L_{\mathcal{R}})$, where $L_{\mathcal{R}}$ is a *labeling function*, associating to each state predicate p the set of all states where p holds.

All this means that, since rewrite theories model concurrent systems and we can naturally associate to them Kripke structures, their *temporal logic properties* can then be defined semantically in terms of such Kripke structures (or for real-time or probabilistic rewrite theories the analogous real-time or probabilistic transition systems). For expressing such properties, suitable temporal logics can be used. Then, both model checking, or theorem proving, or a combination of both approaches, can be used to *verify* that a rewrite theory (more precisely, its reachability initial model) satisfies some desired temporal logic properties.

3.11.1. Temporal Logics

Which temporal logic is best suited for specifying which properties of a rewrite theory is itself a very good question. Here are several choices with specific advantages.

State-Based Logics. There are many choices. The most common is *CTL** [101], or one of its subsets such as *CTL* or *LTL*. These logics are well suited for properties based on state predicates; but not well suited for properties based on *events*, which need to be encoded unnaturally in the state itself to be expressible.

TLR and Parameterized Fairness. To avoid the limitations of state-based logics in expressing events, while keeping all their good state-based features; and to take advantage of the expressive power of rewrite theories in expressing *parameterized events* by rewrite rules, and *spatial information* by term patterns, the *temporal logic of rewriting TLR* [325] can be used. *TLR* is a simple extension of *CTL** where just one more construct is added to the syntax of formulas, namely, *spatial action patterns.* The simplest such patterns are just labels of rewrite rules, stating that a transition event with a rule having that label has taken place. For example, for the object-based system of Section 2.1, we can state the liveness property that each message `send` is always eventually followed by a `receive` event by the (implicitly universally path quantified) *TLR* formula $\Box$(`send` $\to$ $\Diamond$`receive`). However, more complex patterns are possible taking advantage of both the parametric nature of rewrite rules (whose parameters are the mathematical variables of each rule) and the context where the rewrite takes place. For example, we can *localize* the above property both to sender object `'b` and its associated receiver object `'d` by the formula $\Box$(`send('b)`$\to$ $\Diamond$ `receive('d)`). It is also very easy to express *localized* (that is, parameterized) fairness conditions as universally quantified *TLR* properties. For example, the (weak) object fairness of the `receive` and `write` actions needed for a realistic modeling of the object-based system of Section 2.1 when sensor

[14]For technical reasons, in some approaches, e.g., [101], the transition relation of a Kripke structure is assumed to be a *total* relation; there is no problem in extending the relation $\to^1_{\mathcal{R}}$ to a total relation for this purpose.

objects are added, as explained in Section 3.5, can be succinctly captured by the *TLR* formulas $(\forall x : Oid)\ \diamond\Box\,\texttt{receive}.enabled(x) \to \Box\diamond\ \texttt{receive}(x)$, and $(\forall x : Oid)\ \diamond\Box\,\texttt{write}.enabled(x) \to \Box\diamond\ \texttt{write}(x)$, where $\texttt{receive}.enabled(x)$ and $\texttt{write}.enabled(x)$ are the obvious state predicates stating that the object x can perform the `receive`, resp., `write` action. Of course, the reachability initial model of a rewrite theory $\mathcal{R}$ and its associated Kripke structure $(T_{\Sigma/E_{[State]}}, \to^1_{\mathcal{R}}, L_{\mathcal{R}})$ throw away all information about actions and therefore cannot be used to give semantics to *TLR*. We need to use the initial model $\mathcal{T}_{\mathcal{R}}$ of $\mathcal{R}$ and its associated *labeled Kripke structure*, where labeled transitions are of the form $[t] \xrightarrow{[\alpha]}_{\mathcal{R}} [t']$, with α a one-step proof term [325].

Metric Temporal Logic and TCTL. For real-time systems, standard temporal logics, although able to express many useful properties (particularly when the state predicates refer to timers or even to the global clock), are not expressive enough: one often wants to express the requirement that a certain property must hold within certain time bounds. Various temporal logics for real-time systems can be used. A simple possibility is to use the *metric temporal logic MTL* [275], which extends *LTL* to timed paths by qualifying *LTL*'s until operator $\mathcal{U}$ with a time interval $[t, r]$. The meaning of a formula $\varphi\,\mathcal{U}_{[t,r]}\,\psi$ is then that $\varphi\,\mathcal{U}\,\psi$ holds in the standard *LTL* sense and, furthermore, ψ must hold at a time $t' \in [t, r]$, and φ must continuously hold until time t'. Instead, *Timed CTL* (*TCTL* [25]) extends *CTL* by qualifying the until operator $\mathcal{U}$ with a time bound t plus an indication of whether the second formula must hold before, after, or exactly at time t, that is, we have formulas of the form $\varphi\,\mathcal{U}_{\bowtie t}\,\psi$, where $\bowtie \in \{\geq, >, \leq, <, =\}$, with the expected meaning. For example, $\varphi\,\mathcal{U}_{\geq t}\,\psi$ is equivalent to $\varphi\,\mathcal{U}_{[t,+\infty)}\,\psi$ in an interval formulation.

PCTL, CSL, and QuaTEx. For probabilistic systems, temporal logics that extend standard ones are also needed. One well-known such logic is *Probabilistic CTL* (*PCTL*) [227]. The basic idea is that sets of computation paths in a probabilistic system have probability measures associated to them, and we can qualify temporal logic formulas by requiring that the set of paths satisfying a certain formula has a probability greater (resp., smaller) than or equal to a certain $p \in [0, 1]$. For example, the *PCTL* formula $\mathcal{P}_{\geq 0.7}(\varphi\,\mathcal{U}\,\psi)$ states that the set of paths where $\varphi\,\mathcal{U}\,\psi$ holds has a probability measure greater than or equal to 0.7.

Since many probabilistic systems are also real-time systems, for such systems there is also a need to have temporal logics which combine both probabilistic and time-bounded features. *Continuous Stochastic Logic* (*CSL*) [1, 43] is one such logic extending *PCTL* by qualifying temporal logic operators by a time bound. For example, the formula $\mathcal{P}_{\geq 0.7}(\varphi\,\mathcal{U}^{\leq 3.2}\,\psi)$ states that the set of paths where $\varphi\,\mathcal{U}\,\psi$ holds and, furthermore, ψ holds at a time $t \in [0, 3.2]$, and φ holds continuously until time t, has a probability measure greater than or equal to 0.7.

In the analysis of probabilistic systems we are often interested not just in the probabilities associated to the satisfaction of certain temporal logic formulas, but in *quantitative properties* such as, for example, the expected latency of a communication protocol when hardened against DoS attacks under specific assumptions about the attacker and the network. Such a latency is not a probability but a real number. To be able to express such quantitative properties, *PCTL* and *CSL* have been generalized to a logic of *Quantitative Temporal Expressions* (QuaTEx) in [5]. The key idea is to generalize state formulas and path formulas to *real-valued state expressions* and *path expressions*, where the appropriate real-valued functions can be defined by the user, just as the appropriate state predicates are defined by the user in standard temporal logics. Boolean-valued and probability-valued formulas are now regarded as special cases of real-valued QuaTEx formulas by using the subset containments $\{0, 1\} \subset [0, 1] \subset \mathbb{R}$. For example, Boolean-valued *CSL* formulas such as $\mathcal{P}_{\geq 0.7}(\varphi\ \mathcal{U}^{\leq 3.2}\ \psi)$ are also expressible in QuaTEx, but QuaTEx can express properties beyond *CSL* [5].

3.11.2. Model-Checking Verification of Rewrite Theories

Model Checking of State-Based Temporal Properties. The simplest, yet very useful, form of model-checking analysis of rewrite theories is the verification of *invariants*. As usual in model checking, what we search for is the *violation* of a property, in this case the invariant. An invariant I is a Boolean-valued state predicate, so we can express a search for its violation as a search for a proof of the existential formula

$$(\exists x : [\mathit{State}])\ (\mathit{init} \rightarrow x\ \wedge\ I(x) = \mathit{false})$$

where *init* is the initial state, and [*State*] is our chosen kind of states. If the number of states reachable from *init* is finite, breadth first search is a complete model-checking procedure to verify the invariant. If the number of states reachable from *init* is infinite, breadth first search still gives us a *semidecision procedure* to check the failure of the invariant: if I fails, we are guaranteed to find a counterexample in finite time.

More generally, we can model check properties in state-based temporal logics such as *CTL*, *LTL*, or *CTL** using the model-checking algorithms described in [101] by using the Kripke structure $(T_{\Sigma/E_{[State]}}, \rightarrow^1_{\mathcal{R}}, L_{\mathcal{R}})$ associated to the given rewrite theory $\mathcal{R}$, provided the number of states reachable from the given initial state *init* is finite.

Model Checking of TLR Properties. To verify *TLR* properties on a rewrite theory $\mathcal{R}$, assuming again that the number of states reachable from the given initial state *init* is finite, we have two different possibilities: (i) to transform $\mathcal{R}$ and the property φ into a new rewrite theory $\widetilde{\mathcal{R}}$ and a *CTL** formula $\widetilde{\varphi}$ and then model check $\widetilde{\mathcal{R}}, \widetilde{\mathit{init}} \models \widetilde{\varphi}$ as described in [325] and implemented in Maude in [37] for the linear time temporal logic fragment *LTLR*; or (ii) to use a more efficient algorithm that can directly verify *LTLR* formulas on a rewrite theory $\mathcal{R}$ on the fly, as the one developed and implemented in the Maude system in [38]. One of the good features of *TLR* is that it is very easy to express fairness assumptions in it [325], so a first approach to the verification of a *TLR* property ψ under fairness assumptions φ is to verify the implication $\varphi \rightarrow \psi$. However, this suffers from two major drawbacks: (i) in a logic like *LTL* the Büchi automaton associated to $\varphi \to \psi$ grows exponentially with the size of the formula; and since φ typically contains several fairness formulas and can be relatively complex, we can easily hit severe performance barriers; and (ii) to make things worse, the approach of model checking $\varphi \to \psi$ has no reasonable way of dealing with localized fairness formulas which are *parametric*, i.e., what we have is not a propositional formula φ, but a universally quantified first-order formula $(\forall x)\ \varphi(x)$. For example, $(\forall x)\ \varphi(x)$ may express an object fairness assumption in a system with dynamic object creation. Even if we could predict the set O of all such objects, which may not be possible unless we explore the entire state space, the only way to encode this directly at the propositional level would be as a conjunction $\bigwedge_{o \in O} \varphi(o)$, something quite unfeasible to model check in practice because of the typically huge size of the corresponding Büchi automaton. For these reasons, Kyungmin Bae and I have developed a completely new model-checking algorithm for *LTLR* which can model check *LTLR* formulas under parametric fairness assumptions of the form $(\forall x)\ \varphi(x)$. The algorithm and its Maude implementation are described in [39].

An interesting, additional aspect of *LTLR* model checking is its use as a *strategy language*. Since *TLR* formulas contain action patterns corresponding to how rules are applied, with which substitutions, and where in the state, and describe complex behaviors involving such elementary actions and tests expressed by state predicates, a *TLR* path formula φ can be naturally understood as a *strategy expression*, which defines a corresponding set of computations in the given rewrite theory $\mathcal{R}$. Assuming that φ does not contain any path quantifiers, we can use an *LTLR* model checker to generate a behavior for the strategy expression φ by giving to the model checker the *LTLR* state formula $\forall\neg\varphi$. If the strategy expression φ can be realized by a concrete behavior, the *LTLR* model checker will provide such a behavior as a counterexample for $\forall\neg\varphi$, that is, as a constructive *proof* of the existentially path quantified *TLR* state formula $\exists\varphi$.

Narrowing-Based Symbolic Model Checking of Rewrite Theories. One important limitation of standard model-checking algorithms such as those described in [101] is that they work under the assumption that the set of states reachable from the initial state is finite. There are several ways to avoid this limitation: (i) to use deductive methods such as those I discuss in Section 3.11.3; (ii) to use some kind of abstraction or simulation that transforms the system into a finite-state one (I discuss this in Section 3.12); and (iii) to use a model-checking approach that does not require the system to be finite-state. Regarding approaches of type (iii), Section 3.3 has explained how narrowing can be used as a complete symbolic reachability analysis method to model check the failure of an invariant for a possibly infinite-state rewrite theory $\mathcal{R}$. This is of course a very different notion of "symbolic model checking" than the usual one based on BDDs, which uses the representation of a *finite* set of states as a propositional formula assuming a *finite* state space. But Section 3.3 dealt only with reachability and invariants. What about other temporal logic properties? In [186] Santiago Escobar and I show how the same narrowing approach can be extended to model check $ACTL^*$ properties of a possibly infinite system specified as a topmost rewrite theory $\mathcal{R}$, where $ACTL^*$ denotes the universal fragment of CTL^*.

Model Checking of Real-Time Rewrite Theories. The simplest models of real-time systems are *timed automata* [26], whose $TCTL$ properties are decidable by model checking [25]. The paper [52] shows how timed automata model checking can be expressed as a symbolic procedure using appropriate strategies in the ELAN rewriting logic language. Timed automata can be seen as very simple real-time rewrite theories [368], but their simplicity also involves a severe limitation: they are finite-state systems. Even a relatively simple system such as a scheduler whose state includes unbounded queues cannot be modeled by a timed automaton [364]. What real-time rewrite theories offer is a more expressive high-level way of specifying many real-time systems of interest, such as network protocols and distributed object systems, whose states are in principle unbounded and often contain complex data structures. The challenge is to identify temporal logic properties and conditions on the real-time rewrite theory that make the verification of such properties decidable by model checking. A very broad class of real-time rewrite theories (whose time may be continuous) has been identified in [370], where it is shown that the following temporal logic properties are decidable for such systems: (i) time-bounded $LTL\backslash\bigcirc$ formulas[15] of the form φ *in time* r, where φ is an $LTL\backslash\bigcirc$ formula and r is a time bound (for a detailed explanation of the semantics of such formulas see [371]); and (ii) $LTL\backslash\bigcirc$ formulas whose state predicates do not refer to the global clock, provided the set of discrete states reachable from the initial state is finite. Recall that a state of a system specified by a real-time rewrite theory is a pair $(\{t\}, r)$, with $\{t\}$ a ground term describing the global state and r a (possibly continuous) clock value. By the "discrete state" I mean the global state $\{t\}$. Formulas of types (i) or (ii) can already express many properties of practical interest, but formalisms such as MTL and $TCTL$ are obviously more expressive. More recent work has developed two new model-checking algorithms for real-time rewrite theories. In [283], a model-checking algorithm to verify properties in a subset of MTL for object-oriented real-time rewrite theories whose state is a multiset of objects and messages is presented; and [282] presents an algorithm to model check real-time rewrite theories for the satisfaction of $TCTL$ formulas, except for formulas of the form $\varphi\, \mathcal{U}_{=t}\, \psi$. In Section 6.1.8 I discuss the Real-Time Maude tool, which supports all the model-checking procedures mentioned above; and in Section 7.4 I discuss many real-time system applications that have been specified and analyzed in Real-Time Maude.

Statistical Model Checking of Probabilistic Rewrite Theories. Temporal logic properties of a probabilistic system can be model checked either by exact model-checking algorithms, or in an approximate, but more scalable and more widely

[15] $LTL\backslash\bigcirc$ is the sublogic of LTL obtained by not using the $\bigcirc$ operator.

applicable way, by *statistical model checking* (see, e.g., [419, 475, 5]). The idea of statistical model checking is to verify the satisfaction of a temporal logic property by statistical methods up to a user-specified level of statistical confidence. For this, a large enough number of Monte-Carlo simulations of the system are performed, and the formula is evaluated on each of the simulations.

Recall the discussion in Section 3.10 about how a probabilistic rewrite theory in general has a nondeterministic aspect corresponding to the choice of which probabilistic transition to fire. One important requirement of statistical model-checking algorithms is that they assume that the system is purely probabilistic: there is no nondeterminism in the choice of transitions. This seems like a strong requirement. However, using the methodology presented in [5], a wide class of object-oriented probabilistic real-time rewrite theories specifying many concurrent, actor-based systems of interest can be expressed so that no nondeterminism is involved in the application of rewrite rules. The key idea is to take advantage of three facts: (i) time is continuous; (ii) the probability distributions governing message arrival latencies are also continuous; and (iii) since the message arrival latency distributions are continuous, the probability that two messages will arrive at the same time to any two objects (or to the same object) is then zero. Since the rewrite rules specify how an actor changes state when it receives a message, and at each instant in time at most one message has arrived to at most one object, there is at most one rewrite rule that can be applied at each continuous instant and all nondeterminism disappears.

Properties expressed in either *CSL* or QuaTEx can then be statistically model checked for such probabilistic real-time rewrite theories, using the algorithms presented in, respectively, [419] and [5]. Furthermore, as shown in [23], the above algorithms are naturally parallelizable and can scale up very well using such paralelization. A related algorithm for statistical model checking of quantitative properties is presented in [261]. In Section 6.1.10 I discuss how the VeStA and PVeStA tools support the statistical model checking of *CSL* and QuaTEx properties for the above-mentioned class of probabilistic rewrite theories; and in Section 7.5 I discuss various applications that have been specified and analyzed this way.

3.11.3. Deductive Verification of Rewrite Theories

Model checking, while extremely useful, is not sufficient for all verification purposes. This is clear from the fact that satisfaction of properties is in general undecidable, from the infinite-state nature of many systems, and, even when a system is finite-state for each initial state, from the fact that in general there may be an infinite number of initial states. Furthermore, even if we succeed in reducing the verification problem to a finite-state model-checking problem by the use of an abstraction as discussed in Section 3.12, deduction still plays a fundamental role in verifying the correctness of such an abstraction. The late Amir Pnueli expressed the situation succinctly in his motto "deduction is forever" [385].

Given a rewrite theory $\mathcal{R}$ (resp. a parameterized[16] rewrite theory $\mathcal{R}[\mathcal{P}]$ with $\mathcal{P}$ its parameter theory), there are different kinds of properties that one may want to verify deductively about its initial model $\mathcal{T}_{\mathcal{R}}$, or the Kripke structure associated to its initial reachability model (resp. the free models of $\mathcal{R}[\mathcal{P}]$ or their associated Kripke structures). Properties we may want to verify include: (i) temporal logic properties; (ii) inductive properties about the rewrite relation itself; and (iii) inductive equational properties about the states of $\mathcal{R}$. The termination methods for rewrite theories discussed in Section 3.8 can be naturally regarded as proof methods for a particular kind of type (i) property.

Regarding deductive verification of temporal logic (type (i)) properties, the general idea is to use a sound and relatively complete proof system for a temporal logic to get rid of the temporal logic operators as much as possible and try to reduce the proof task to the verification of proof obligations of type (iii). The term "relatively complete" expresses the fact that the original temporal logic property holds for the given model iff the proof obligations of type (iii) generated by the inference system do; but since these are *inductive* proof obligations, a complete proof system for properties of type (iii) does not exist in general. A good example of a sound and relatively complete deductive proof system for CTL^* is the one proposed by Gabbay and Pnueli in [204]. An important remaining problem in using a deductive system of this kind is how to deal with the resulting proof obligations of type (iii). In this regard, rewrite theories are particularly attractive, because there is a rich body of inductive proof methods for equational logic which can then be used to discharge such proof obligations. For example, for Maude specifications one can use various formal tools described in Section 6.1 for this purpose.

For rewrite theories, this approach to the verification of type (i) properties has so far focused mostly on *safety properties*, including *invariants*. For the deductive proof of invariants there is a rich body of work, including several substantial case studies, using proof scores in CafeOBJ to verify invariants of observational transition systems (OTSs) (see, e.g., [357, 202]). The CafeOBJ researchers have also shown how deductive verification of invariants for an OTS can be combined with model-checking verification of the rewrite theory associated to the OTS, or an abstraction of it [476, 202]. Another approach to invariant and temporal logic verification which can be viewed as *both* deductive and algorithmic is the narrowing-based reachability analysis method already discussed in Sections 3.3 and 3.11.2. Rusu and Clavel [410], and Rusu [409], present a different approach to invariant verification that reduces the problem to a type (iii) proof task by associating to a rewrite theory $\mathcal{R}$ a corresponding

[16]A parameterized rewrite theory $\mathcal{R}[\mathcal{P}]$ can be understood as a theory inclusion $\mathcal{P} \hookrightarrow \mathcal{R}$ of the parameter theory $\mathcal{P}$ into the "body" $\mathcal{R}$ and specifies a parametric family of concurrent systems. $\mathcal{R}[\mathcal{P}]$ can then be instantiated by *views*, i.e., theory interpretations $V : \mathcal{P} \longrightarrow \mathcal{Q}$, by the usual "pushout construction." Semantically, what is used is the fact that rewriting logic is a "liberal institution," i.e., that it has not only initial models, but also free models along theory interpretations. For the treatment of parameterized rewrite theories in Maude see [106, Section 8.3].

membership equational theory $\mathcal{M}(\mathcal{R})$ with a sort *Reachable* of reachable states characterized by appropriate membership predicates. In a sense, this can be seen as using an enrichment of the characterization of the initial reachability model of $\mathcal{R}$ as the initial model of a membership equational theory given in [80] and discussed in Section 3.1.2. Camilo Rocha and I have presented a different approach to the verification of safety properties in [398]. The basic idea is to use narrowing-based proof methods to reduce the proof of: (a) invariants, (b) stability properties of the form $P \Rightarrow \Box P$, and (c) strengthenings of invariants, to proof obligations of type (iii); and to then discharge many such proof obligations automatically, so that a considerably smaller set of proof obligations is left for an inductive theorem prover.

Finally, Camilo Rocha and I have initiated a study of constructor-based proof methods for inductive properties about the rewrite relation of the initial reachability model of a rewrite theory $\mathcal{R}$ (type (ii) properties) in [397]. That is, we want to prove that the initial reachability model of $\mathcal{R}$ satisfies some property of the form $(\forall \vec{x})\ t \to t'$, which is equivalent to proving $\mathcal{R} \vdash \theta(t) \to \theta(t')$ for all ground substitutions θ. A related task is to prove that the initial reachability model of $\mathcal{R}$ satisfies *inductive joinability* properties of the form $(\forall \vec{x})\ t \downarrow t'$, stating that all ground instances of t and t' can be rewritten to a common term. The key idea is that, the same way that equational constructors are crucial for proving inductive equalities $t = t'$, *both* equational constructors for $(\Sigma, E \cup B)$, and constructors for R associated to final states (see Section 3.7) are crucial for proving inductive properties of the form $(\forall \vec{x})\ t \to t'$ for a rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R, \phi)$.

3.12. Simulation and Abstraction

As already mentioned, the application of standard model checking methods to the verification of a temporal logic property φ by (the initial model of) a rewrite theory $\mathcal{R}$ may be hindered by $\mathcal{R}$ being infinite-state. Even if $\mathcal{R}$ is finite-state, the huge size of its state space may still make it unfeasible to model check such a property. Under such circumstances a very useful approach is to find a different rewrite theory $\widehat{\mathcal{R}}$ which has a much smaller (and finite) state space than $\mathcal{R}$, to verify φ for $\widehat{\mathcal{R}}$, and to show that we have an implication

$$\widehat{\mathcal{R}}, \widehat{init} \models \varphi \ \Rightarrow\ \mathcal{R}, init \models \varphi.$$

As shown in, e.g., [101, 295, 332], this can be done if we can relate the sets of states of $\mathcal{R}$ and $\widehat{\mathcal{R}}$ and the initial states *init* and $\widehat{init}$ by a binary relation H such that either: (i) H is a simulation and $\varphi \in ACTL^*$; or (ii) H is a stuttering simulation and $\varphi \in ACTL^* \backslash \bigcirc$ (i.e., φ is an $ACTL^*$ formula which does not contain the operator $\bigcirc$). In addition, the above implication can be turned into an equivalence if H is a bisimulation (resp. stuttering bisimulation).

Given a rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R, \phi)$, a very simple, yet powerful, approach to obtaining such a theory $\widehat{\mathcal{R}}$ is to realize that rewriting logic comes with a built-in "abstraction dial" which allows us to turn some rewrite rules in R into equations that can be removed from R and added to E. That is, we can decompose R into a disjoint union $R = G \cup R_0$ and define $\widehat{\mathcal{R}} = (\Sigma, E \cup \widehat{G} \cup B, R_0, \phi)$, where $\widehat{G}$ denotes the set of equations associated to the rules G. A good example of the use of such an abstraction dial is the `DPLL` module in Section 2.2, where G consisted of the **subsume**, **resolve**, **assert**, and **close** rules. Of course, for the use of this abstraction dial to be natural, the rules G should be deterministic in nature, so that the equations $E \cup \widehat{G}$ are still ground confluent and terminating modulo B. But in order for the Kripke structure associated to $\widehat{\mathcal{R}}$ to be computable (an essential requirement for model checking it) we also need R_0 to be coherent with $E \cup \widehat{G}$ modulo B. If these two conditions are satisfied, and, furthermore, the rules G preserve all the state predicates in φ, Azadeh Farzan and I proved in [192] that the quotient Σ-homomorphism $q : \mathcal{T}_{\Sigma/E\cup B} \longrightarrow \mathcal{T}_{\Sigma/E\cup\widehat{G}\cup B}$ defines a stuttering bisimulation, so that for any $\varphi \in ACTL^{*}\backslash\bigcirc$ we have the equivalence $\widehat{\mathcal{R}}, \widehat{init} \models \varphi \;\Leftrightarrow\; \mathcal{R}, init \models \varphi$, where $\widehat{init} = q(init)$.

If the theory $\widehat{\mathcal{R}}$ thus obtained by turning the abstraction dial as much as possible is still too big to be model checked, a second, also very useful approach is to further collapse the set of states by an *equational abstraction.* Given a rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R, \phi)$ and a set $G \cup B'$ of Σ-equations, we can collapse $\mathcal{R}$ into the rewrite theory $\mathcal{R}/G \cup B' = (\Sigma, E \cup G \cup B \cup B', R, \phi)$ which typically has a much smaller state space than $\mathcal{R}$. Again, we need the equations $G \cup B'$ to preserve the state predicates appearing in the formula φ we want to model check; and we need $\mathcal{R}/G \cup B'$ itself to yield a computable Kripke structure, i.e., $E \cup G$ should be ground confluent and terminating modulo $B \cup B'$, and R should be coherent with $E \cup G$ modulo $B \cup B'$. Under these conditions, Miguel Palomino, Narciso Martí-Oliet and I proved in [331] that the quotient Σ-homomorphism $q : \mathcal{T}_{\Sigma/E\cup B} \longrightarrow \mathcal{T}_{\Sigma/E\cup G\cup B\cup B'}$ defines a simulation, so that for any $\varphi \in ACTL^{*}$ we have the implication $\mathcal{R}/G \cup B', q(init) \models \varphi \;\Rightarrow\; \mathcal{R}, init \models \varphi$.

In the two methods just discussed for collapsing the state space of a rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R, \phi)$, the signature Σ did not change at all: we either changed some rules into equations or added some more equations to the equational part. But this is not a necessary requirement: our more abstract rewrite theory $\widehat{\mathcal{R}}$ may be based on a different signature Σ', so that it is of the form $\widehat{\mathcal{R}} = (\Sigma', E' \cup B', R', \phi')$. All we need is to find an appropriate simulation relation H between $\mathcal{R}$ and $\widehat{\mathcal{R}}$. Several methods for finding such simulation, or stuttering simulation, relations are presented in [302, 332] under the general banner of "algebraic simulations." The general idea is to use algebraic and/or rewriting logic methods to define such an H as either a function or a relation. Another idea explored in depth in [377, 332] is that simulations and stuttering simulations are arrows in appropriate *categories*, so that they can be *composed*, i.e., the entire approach is compositional, so that we can combine several of the above-mentioned abstraction methods to arrive at the desired abstraction. A general emphasis common to all the abstraction methods presented in [192, 331, 302, 332] is on the inductive proof obligations that need to be discharged in order to prove that the proposed simulation H is *correct.* That is, although H is used to verify a property by model checking, the correctness of the verification requires the interplay between model checking and inductive theorem proving: deduction is forever!

Another stuttering-simulation-based method frequently used to reduce the state space is *partial order reduction* (POR). The general idea is that a concurrent system can have a huge number of states due to the many different interleavings involved; however, many concurrent transitions are *independent*, in the sense that they can be interleaved with each other in arbitrary order without affecting the resulting state. This leads to the idea of cutting down the number of interleavings by only considering a subset of the computations involving independent transitions (see [101] for a detailed discussion). To support POR at the level of rewrite theories, Azadeh Farzan and I proposed in [193] a general theory transformation mapping rewrite theories of a certain type into their corresponding POR versions. In particular we showed how this transformation can be applied as a generic method to model check programs much more efficiently in a wide range of concurrent programming languages whose semantics has been defined in rewriting logic by the methods outlined in Section 4.3.

In Section 3.11.2 I explained how for topmost rewrite theories $ACTL^*$ properties can be model checked symbolically by narrowing by the methods presented in [186]. The reason why the CTL^* property must be in the universal fragment $ACTL^*$ is precisely that what is used is a simulation relating the ground term instances of a term to such a term. That is, the original system we want to verify is the Kripke structure associated to the given rewrite theory $\mathcal{R}$, whose states are $E \cup B$-equivalence classes of ground terms; but we simulate it symbolically by another Kripke structure where the states are terms with variables. The abstraction relation H is precisely the "being an instance of modulo $E \cup B$" relation, denoted $\preceq_{E \cup B}$, where $E \cup B$ are the equations in $\mathcal{R}$. Given a ground term t and a term t', $t \preceq_{E \cup B} t'$ holds iff there is a substitution σ such that $t =_{E \cup B} \sigma(t')$. Since the transition system defined on terms with variables by the narrowing relation $\rightsquigarrow_{R,E \cup B}$ in general has still an infinite number of states reachable from a symbolic initial state, a further abstraction can be obtained by adding a *folding* relation[17] between terms with variables. This gives rise to an even more abstract simulation relation, where now the symbolic transition system can in some cases become finite-state. In order for the number of $E \cup B$-unifiers to be finite, the finite variant property is required of $E \cup B$ [186].

I have already mentioned in Section 3.11.2 that, under very general conditions, time-bounded $LTL \backslash \bigcirc$ properties and standard $LTL \backslash \bigcirc$ properties of a real-time rewrite theory can be effectively verified by model checking, even when time is continuous. The reason for this is also related to simulations and abstractions. Specifically, we show in [370] that there is a stuttering bisimulation between the fair timed computations of a "time-robust" real-time rewrite theory and the much smaller set of computations obtained by always advancing the

[17]Several relations can be used to "fold" the state space whose states are terms with variables. One is the already-mentioned relation $\preceq_{E \cup B}$; another, the relation of one term being equal (up to $E \cup B$-equality) to a term obtained by renaming the variables of another term.

clock as much as possible until the next zero-time transition becomes enabled. For continuous time rewrite theories and time-bound $LTL\backslash\bigcirc$ properties, this provides an abstraction from an infinite-state system to a finite-state one; but even for discrete time rewrite theories this provides a huge abstraction, marking in practice the difference between feasible and unfeasible model checking. One can further prove that when the state predicates in $\varphi \in LTL\backslash\bigcirc$ do not depend on the value r of the global clock, but only on the global state $\{t\}$, the projection map $(\{t\}, r) \mapsto \{t\}$ provides a further abstraction allowing the model checking of *time unbounded* properties in $LTL\backslash\bigcirc$ when the set of discrete states (of the form $\{t\}$) reachable from the initial state is finite.

4. Rewriting Logic as a Logical and Semantic Framework

I further discuss here the logical and semantic framework uses already illustrated by means of simple examples in Section 2.

4.1. Representing Logics

Using rewriting logic as a logical framework can be best understood within a metatheory of logics such as the theory of *general logics* [310], which provides an axiomatic framework to formalize the proof theory and model theory of a logic, and which also provides adequate notions of *mapping* between logics, that is, of logic translations. This theory contains Goguen and Burstall's theory of institutions [216] as its model-theoretic component.

The theory of general logics allows us to define the space of logics as a *category*, in which the objects are the different logics, and the morphisms are the different mappings translating one logic into another. We can therefore axiomatize a translation Φ from a logic $\mathcal{L}$ to a logic $\mathcal{L}'$ as a morphism

$$(\dagger)\ \ \Phi : \mathcal{L} \longrightarrow \mathcal{L}'$$

in the category of logics. A *logical framework* is then a logic $\mathcal{F}$ such that a very wide class of logics can be mapped to it by maps of logics

$$(\ddagger)\ \ \Psi : \mathcal{L} \longrightarrow \mathcal{F}$$

called *representation maps*, that have particularly good properties such as conservativity.[18]

A number of logics, particularly higher-order logics based on typed lambda calculi, have been proposed as logical frameworks, including the Edinburgh logical framework LF [230, 34, 206], generic theorem provers such as Isabelle [381], λProlog [348, 196], and Elf [384], and the work of Basin and Constable [51] on

[18]A map of logics is *conservative* [310] if the translation of a sentence is a theorem if and only if the sentence was a theorem in the original logic. Conservative maps are sometimes said to be *adequate* and *faithful* by some authors.

metalogical frameworks. Other approaches, such as Feferman's logical framework FS_0 [195] (that has been used in the work of Matthews, Smaill, and Basin [305]), earlier work by Smullyan [425], and the 2OBJ generic theorem prover of Goguen, Stevens, Hobley, and Hilberdink [219] are instead first-order. The role of rewriting logic as a logical framework should of course be placed within the context of the above related work, and of experiments carried out in different frameworks to prototype formal systems (for more discussion see the survey [327]).

As I have already pointed out in Section 2, one key property by which the practicality of a logical framework should be judged is by how short its representational distance is, and of course by how general it is in representing other logics. Regarding generality, since various typed lambda calculi have been extensively used as logical frameworks, a logical framework that can represent them with 0 representational distance can *a fortiori* represent anything they can represent, and possibly better. As Mark-Oliver Stehr and I have shown in [434], rewriting logic can represent with 0 representational distance not just some particular typed lambda calculus, but the parametric family of typed lambda calculi called *pure type systems* [53], which generalize the λ-cube and therefore contain virtually all typed lambda calculi of interest. The reverse is not at all the case: there is no representation of rewriting logic, or even of equational logic, into such calculi which could be said to have ϵ representational distance. The obvious reason for this is the well-known difficulty of lambda calculi in dealing with equational reasoning, since the only equational reasoning native to such calculi is that between lambda expressions by β-reduction. Furthermore, in LF there is no adequate representation for linear logic in a precise technical sense of "adequate" [206, Corollary 5.1.8]. Instead, linear logic can be faithfully represented in rewriting logic with 0 representational distance [300].

All these representations of logics are easily *mechanizable* using a rewriting logic language like Maude, leading to useful prototypes supporting formal reasoning for the logic in question. The nontrivial matter of quantifiers and substitutions is elegantly supported by Stehr's CINNI calculus of explicit substitutions [430]. In particular, using CINNI pure type systems can not only be represented: they can also be efficiently executed in a rewriting logic language like Maude. This trivial representation in one direction, and the serious difficulties for lambda calculi to deal with equality in the converse direction, were seen by Stehr as an opportunity to generalize the Coquand-Huet Calculus of Constructions (CC) [122] into his own Open Calculus of Constructions (OCC) [431, 432, 433] within rewriting logic (implemented in Maude as a theorem prover) to naturally support both CC reasoning *and* equational reasoning in a seamless way.

The above remarks make it obvious that rewriting logic has very good properties as a logical framework. Several other examples of well-known logics which can be represented in rewriting logic with ϵ representational distance are given in [298, 300], and a more detailed discussion of logical framework applications is given in Section 7.1. An additional good feature of rewriting logic as a logical framework is its ability to deal naturally with state changes, and therefore to solve in a straightforward way the thorny "frame problem," which has plagued for decades AI researchers using first-order logic as a knowledge representation formalism; this is explained in detail in [299].

Yet another very useful representational feature is rewriting logic's "abstraction dial" (see Section 3.12). This was already obvious in the `DPLL` example of Section 2.2 and is systematically exploited for model-checking purposes as explained in Section 3.12. For logical framework uses the general point is that: (a) there is a very useful distinction to be made between (i) *computation*, which is deterministic and can be blindly and exhaustively applied with high efficiency, and (ii) *deduction*, which is nondeterministic, requires search, and can be very inefficient; and (b) this computation vs. deduction distinction is naturally supported by a rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R, \phi)$ as the distinction between its deterministic equations $E \cup B$ and its nondeterministic rules R. The practical meaning of all this is that one can make the implementation of a logic much more efficient, and the level at which a user interacts with a tool much higher, if millions of trivial computations are automatically performed, so that the strategic thinking about proofs can be focused at a much higher level. This was emphasized since the early papers on logical framework uses of rewriting logic [327, 298, 300], has been later dubbed "deduction modulo" by some researchers [148], and has been illustrated with interesting examples of rewrite theories representing logics such as those in [466, 396].

All the above remarks are fine and well, but even with all those good features a mapping $\Psi : \mathcal{L} \longrightarrow \mathcal{F}$ of a logic $\mathcal{L}$ into a logical framework $\mathcal{F}$ is still a complex *metalevel* entity: how can Ψ itself be represented? It is neither in $\mathcal{L}$ nor in the framework $\mathcal{F}$ but hovers abstractly above both. More generally, how can a map of logics $\Phi : \mathcal{L} \longrightarrow \mathcal{L}'$ be represented? This is not a theoretical question but an eminently practical one: how are Ψ or Φ going to be *implemented*? And how can we *reason* about them? Here is where rewriting logic's *reflective* features play a key role, so that it is not just a good logical framework, but a *reflective metalogical framework* in the precise, technical sense given to the term in [50].

The key advantage of having a reflective logical framework such as rewriting logic is that we can represent—or as it is said *reify*—within the logic in a computable way maps of the form (†) and (‡). We can do so by extending the universal theory $\mathcal{U}$ (see Section 3.4) of our reflective framework logic $\mathcal{F}$ (namely, rewriting logic), which has a sort *Theory* representing rewrite theories $\mathcal{R}$ as terms $\overline{\mathcal{R}}$ of sort *Theory*, with equational abstract data type definitions for the data type of theories $\mathit{Theory}_{\mathcal{L}}$ for each logic $\mathcal{L}$ of interest. Then, a map $\Phi : \mathcal{L} \longrightarrow \mathcal{L}'$ can be reified as an equationally-defined function

$$\overline{\Phi} : \mathit{Theory}_{\mathcal{L}} \longrightarrow \mathit{Theory}_{\mathcal{L}'}.$$

And, similarly, a representation map $\Psi : \mathcal{L} \longrightarrow \mathcal{F}$, with $\mathcal{F}$ rewriting logic, can be reified as an equationally-defined function

$$\overline{\Psi} : \mathit{Theory}_{\mathcal{L}} \longrightarrow \mathit{Theory}.$$

If the maps Φ and Ψ are computable, then, by a metatheorem of Bergstra and Tucker [54] it is possible to define the functions $\overline{\Phi}$ and $\overline{\Psi}$ by means of corresponding *finite* sets of confluent and terminating equations. That is, such functions can be effectively defined and executed *within* rewriting logic.

The point worth emphasizing again is that all this is not a theoretical *divertimento* but an enormously practical feature. For example, Pavel Naumov, Mark-Oliver Stehr and I used exactly the above approach to represent the logics of the HOL and NuPrl theorem provers within rewriting logic, define a conservative map of logics between them, prove its correctness, make such a formal definition executable in Maude, and automatically translate several megabytes of HOL theories into correct-by-construction NuPrl theories in [353] (a *mechanical* proof of correctness of such a map of logics was later given in [417]). Many more examples of how reflection is enormously useful to define and implement within $\mathcal{F}$ itself maps of logics, particularly maps of the form $\Psi : \mathcal{F} \longrightarrow \mathcal{F}$ mapping the reflective framework to itself and corresponding to *theory transformations* are discussed in Sections 3.4, 6.1 and 7.1.

The last point worth making is that rewriting logic is not just a logical framework but a *metalogical* one. As explained in [51], what a metalogical framework adds to a logical framework is the capacity to reason formally within itself about the metalogical properties of the logics represented in it. Typically such reasoning requires induction. As explained in [50], the reflective features of membership equational logic and of rewriting logic, combined with the fact that both logics have initial models supporting inductive reasoning principles, and with the fact that, in particular, their universal theories do come with their own induction principles, is what makes them into reflective metalogical framewoks. For several practical applications of rewriting logic's metalogical reasoning capabilities see [50, 112, 109].

4.2. Representing Models of Concurrency

Since rewriting logic is a coin with two sides, a logical side and a computational one, the exact same reasons making it a very flexible logical framework with 0 or ϵ representational distance make it also a very flexible semantic framework. Since this is one of the main uses of rewriting logic since the beginning [315], so much work has been done that it is hardly possible to survey it all. But perhaps what is most important is for me to explain the philosophical distinction between a *model* and a *logic*, and why that distinction is crucial for representing concurrency models within rewriting logic.

The way concurrency models have been traditionally compared is by building *encodings* from one model into another. For example, some researchers encoded the CCS process calculus into Petri nets; and others encoded the lambda calculus and some variants of the actor model into the π-calculus. These are Turing-machine-like representations, where in principle one can show that some model can be *simulated* by another model by some kind of compilation process, but in general there is a substantial representational distance and much is lost in translation. If rewriting logic were to be one more such model into which other models are similarly compiled, there would be little point in such a futile representational exercise. The key observation is that *rewriting logic is not a model at all.* It is instead a *logic* within which widely different models can be *specified* *as rewrite theories without any encoding*. One can think of it as an "ecumenical movement" with no sectarian ax to grind: it makes no commitments to specific concurrency mechanisms. Is it better to be synchronous or asynchronous? Is message-passing the best communication mechanism? Should channels be conceived as names, or as communication links containing messages? Should the order of messages be preserved or not? Should processes have unique names? All these are questions for each specific model, that is, each specific rewrite theory, to address or ignore. Rewriting logic remains politely silent about the choices made in each model, but tries to be as flexible as possible in representing different choices. My own opinion is that concurrency is such a motley phenomenon (much more so than, say, functional computation) that the question "what is the *best* model of concurrency?" is both meaningless and unwise. Chivalrous quests for the *Holy Grail of Concurrency*, while commendable and probably quite useful in their side effects, are likely to remain inconclusive. The point is that *any* model must make some commitments about what concurrency mechanisms to favor; and this will automatically create a representational distance between it and other models making other equally valid commitments, perhaps for different purposes and reasons.

Just to give some feeling for the vast amount of work which has been done in defining different models of concurrency as rewrite theories, typically with 0 or ϵ representational distance, I mention first some well-known models not involving real time or probabilities, and then discuss real-time and probabilistic models. Next to each model I mention some references for illustration purposes, without any attempt to cover them all (see the bibliography in this issue for a hopefully complete list of references).

1. *Actors and Concurrent Objects* [316, 440].
2. *CCS* [128, 460, 77].
3. *LOTOS* [460, 458].
4. *Dataflow* [318].
5. *Gamma and the CHAM* [315].
6. *Graph Rewriting* [318, 421].
7. *Neural Networks* [318, 411].
8. *Parallel λ-Calculus* [278].
9. *Petri Nets* [315, 435].
10. *π-Calculus* [465, 430, 451].
11. *Tile Logic* [328, 83, 78, 82].
12. *The UNITY Model* [315].

An important point not made explicit by the above list is that the *initial model semantics* of rewriting logic (see Section 3.1.1) plays also a crucial role, because it unifies within a single semantics very different *denotational models* that have been independently proposed for various models of concurrency. For example, rewriting logic's initial model semantics specializes to: (i) for Actors to the *event diagram* partial order of events model of [44, 119], as shown in [337]; (ii) for Petri nets to the Best-Devillers commutative process model [57], as shown in [129, 435]; (iii) for the parallel lambda calculus to its traditional model, shown to be a simple quotient of the initial model of the corresponding rewrite theory in [278]; and (iv) for CCS to the proved transition causal model of Degano and Priami [130], shown to be a simple quotient of the initial model of the corresponding rewrite theory in [84].

For real-time models, real-time rewrite theories also provide a very general and flexible semantic framework. For example, the following models of real time can all be naturally specified as real-time rewrite theories:

1. *Hybrid Automata* [368].
2. *Timed Petri Nets* [368, 435].
3. *Timed Automata* [368].
4. *Timed Transition Systems* [368].
5. *Object-Oriented Real-Time Systems* [368].
6. *The Orc Model of Concurrent Real-Time Computation* [20, 21].
7. *Phase Transition Systems* [368].

Probabilistic rewrite theories can also be used as a semantic framework for a wide range of probabilistic systems, including:

1. *Continuous Time Markov Chains* [276].
2. *Generalized Semi-Markov Processes* [276].
3. *Object-Oriented Probabilistic Systems* [277, 5].
4. *Object-Oriented Stochastic Hybrid Systems* [336].
5. *Probabilistic Nondeterminisitc Systems* [276].

4.3. Rewriting Logic Semantics of Programming Languages

The flexibility of rewriting logic to naturally express many different models of concurrency can be exploited not just at the theoretical level, for expressing such models both deductively, and denotationally in the model theory of rewriting logic [315, 318]: it can also be applied to give *formal definitions of concurrent programming languages* by specifying the concurrent model of a language $\mathcal{L}$ as a rewrite theory $(\Sigma_{\mathcal{L}}, E_{\mathcal{L}}, R_{\mathcal{L}})$, where: (i) the signature $\Sigma_{\mathcal{L}}$ specifies both the syntax of $\mathcal{L}$ and the types and operators needed to specify semantic entities such as the store, the environment, input-output, and so on; (ii) the equations $E_{\mathcal{L}}$ can be used to give semantic definitions for the *deterministic* features of $\mathcal{L}$ (a sequential language typically has only deterministic features and can be specified just equationally as $(\Sigma_{\mathcal{L}}, E_{\mathcal{L}})$); and (iii) the rewrite rules $R_{\mathcal{L}}$ are used to give semantic definitions for the concurrent features of $\mathcal{L}$ such as, for example, the semantics of threads. By specifying the rewrite theory $(\Sigma_{\mathcal{L}}, E_{\mathcal{L}}, R_{\mathcal{L}})$ in a rewriting logic language like Maude, it becomes not just a mathematical definition but an *executable* one, that is, an *interpreter* for $\mathcal{L}$. Furthermore, one can leverage Maude's generic `search` and LTL model-checking features to automatically endow $\mathcal{L}$ with powerful *program analysis capabilities*. For example, the `search` command can be used in the module $(\Sigma_{\mathcal{L}}, E_{\mathcal{L}}, R_{\mathcal{L}})$ to detect any violations of invariants, e.g., a deadlock or some other undesired state, of a program in $\mathcal{L}$. Likewise, for terminating concurrent programs in $\mathcal{L}$ one can model check any desired LTL property. All this can be effectively done not just for toy languages, but for real ones such as Java and the JVM, Scheme, and C (see Section 7.2 for a discussion of such "real language" applications), and with performance that compares favorably with state-of-the-art model-checking tools for real languages.

There are essentially three reasons for this surprisingly good performance. First, rewriting logic's distinction between equations $E_{\mathcal{L}}$, used to give semantics to deterministic features of $\mathcal{L}$, and rules $R_{\mathcal{L}}$, used to specify the semantics of concurrent features, provides in practice an enormous *state space reduction*. Note that a state of $(\Sigma_{\mathcal{L}}, E_{\mathcal{L}}, R_{\mathcal{L}})$ is, by definition, an $E_{\mathcal{L}}$-equivalence class $[t]_{E_{\mathcal{L}}}$, which in practice is represented as the state of the program's execution after all deterministic execution steps possible at a given stage have been taken. That is, the equations $E_{\mathcal{L}}$ have the effect of "fast forwarding" such an execution by skipping all intermediate deterministic steps until the next truly concurrent interaction is reached. For example, for $\mathcal{L} = \mathit{Java}$, E_{Java} has hundreds of equations, but R_{Java} has just 5 rules. The second reason is of course the high performance of rewriting logic languages such as Maude, which can reach millions of rewrite steps per second. The third reason is that the intrinsic flexibility of rewriting logic means that it does not prescribe a fixed style for giving semantic definitions. Instead, *many different styles* such as, for example, small-step or big-step semantics, reduction semantics, CHAM-style semantics, modular structural operational semantics, or continuation semantics, can all be naturally supported [423]. But not all styles are equally efficient; for example, small-step semantics makes heavy use of conditional rewrite rules, insists on modeling every single computation step as a rule in $R_{\mathcal{L}}$, and is in practice horribly inefficient. Instead, the continuation semantics style described in [423] and used in, e.g., [191] is very efficient.

As for models of concurrency, the general idea for SOS definitions is that rewriting logic provides a general framework for such definitions, but has no ax to grind regarding specification style choices. From its early stages rewriting logic has been recognized as ideally suited for SOS definitions [326, 300], and has been used to give SOS definitions of programming languages in quite different styles, e.g., [459, 77, 460, 461, 191, 194]. What the paper [423] makes explicit is both the wide range of SOS styles supported, and the possibility of defining new styles that may have specific advantages over traditional ones. Where the "abstraction dial" is placed in such choices is of course crucial for the efficiency of model-checking analyses: traditional styles will tend to force the least abstract choices that specify all computation steps with rules; but many more choices are available when the underlying logic supports a distinction between equations and rules.

The good theoretical and practical advantages of using rewriting logic to give semantic definitions to programming languages have stimulated an international research effort called the *rewriting logic semantics project* (see [333, 334, 423, 335] for some overview papers). Not only have semantic definitions allowing effective program analyses been given for many real languages such as Java, the JVM, Scheme, and C, and for hardware description languages such as ABEL and Verilog: it has also been possible to build a host of sophisticated program analysis tools for real languages based on different kinds of *abstract semantics*. The point is that instead of a "concrete semantics" $(\Sigma_{\mathcal{L}}, E_{\mathcal{L}}, R_{\mathcal{L}})$, describing the actual execution of programs in a language $\mathcal{L}$, one can just as easily define an "abstract semantics" $(\Sigma_{\mathcal{L}}^{A}, E_{\mathcal{L}}^{A}, R_{\mathcal{L}}^{A})$ describing any desired abstraction A of $\mathcal{L}$. A good example is type checking, where the values manipulated by the abstract semantics are the types. All this means that many different forms of program analysis, much more scalable than the kind of search and model checking based on a language's concrete semantics, become available essentially for free by using Maude to execute and analyze one's desired abstract semantics $(\Sigma_{\mathcal{L}}^{A}, E_{\mathcal{L}}^{A}, R_{\mathcal{L}}^{A})$. I further discuss different applications of both concrete and abstract rewriting semantics of programming languages in Section 7.

Two further developments of the rewriting logic semantics project, both pioneered by Grigore Roşu with several collaborators, are worth mentioning. One is the K *semantic framework* for programming language definitions [408], which provides a very concise and highly modular notation for such definitions. The K-Maude tool then automatically translates language definitions in K into their corresponding rewrite theories in Maude for execution and program analysis purposes (I further discuss K and the K-Maude tool in Section 6.2.2). Another is *matching logic* [406, 404], a program verification logic, with substantial advantages over both Hoare logic and separation logic, which uses a language's rewriting logic semantics, including the possibility of using patterns to symbolically characterize sets of states, to mechanize the formal verification of programs, including programs that manipulate complex data structures using pointers (I further discuss matching logic and the MatchC tool in Section 6.2.3).

4.4. Representing Distributed Systems, Software Architectures, and Models

It is well known that the most expensive errors in system development are design errors. They are not coding errors having to do with some mistake in the details of a program: they happened much earlier, when the system was designed and no programs yet existed. Because design errors affect the overall structure of a system and are often discovered quite late in the development cycle, they can be enormously expensive to fix. All this is uncontroversial: there is widely-held agreement that, to develop systems, designs themselves should be made machine-representable, and that tools are needed to keep such designs consistent and to uncover design errors as early as possible. This has led to the development of many software modeling languages and of architectural notations to describe software designs.

There are however two main limitations at present. The first is that some of these notations lack a formal semantics: they can and do mean different things to different people. The second is that this lack of semantics manifests itself at the practical level as a lack of *analytic power*, that is, as an incapacity to uncover expensive design errors which could have been caught by better analysis. It is of course virtually impossible to solve the second problem without solving the first: without a precise mathematical semantics any analytic claims about satisfaction of formal requirements are meaningless.

The practical upshot of all this is that a semantic framework such as rewriting logic can play an important role in: (i) giving a precise semantics to modeling languages and architectural notations; and in (ii) endowing such languages and notations with powerful formal analysis capabilities. Essentially the approach is the same as for programming languages. If, say, $\mathcal{M}$ is a modeling language, then its formal semantics will be a rewrite theory of the form $(\Sigma_{\mathcal{M}}, E_{\mathcal{M}}, R_{\mathcal{M}})$. If the modeling language $\mathcal{M}$ provides enough information about the dynamic behavior of models, the equations $E_{\mathcal{M}}$ and the rules $R_{\mathcal{M}}$ will make $\mathcal{M}$ *executable*, that is, it will be possible to *simulate* models in $\mathcal{M}$ before they are realized by concrete programs, and of course such models thus become amenable to various forms of *formal analysis*. There is a large body of research in rewriting logic that has done just this, including:

1. giving formal semantics to various *object-oriented design notations, architectural notations, and software modeling languages*, e.g., [197, 268, 269, 474, 33, 110, 154, 61, 346, 60, 42, 393, 394, 169, 345, 62, 65, 347, 66, 40, 363], and

2. giving formal semantics to various *middleware and distributed coordination mechanisms*, e.g., [350, 13, 14, 167, 168, 402, 153].

I discuss all this work in more detail in Section 7, and the MOMENT-2 tool in Section 6.2.

Since many of the software architectures needed in practice are *distributed* architectures, the flexibility of rewriting logic to naturally represent a wide range of distributed communication and interaction mechanisms has proved very useful in all the applications mentioned above. But the medium of a modeling language or an architectural description language is not a necessary requirement. It is also possible to specify and analyze a wide range of distributed system designs and algorithms directly in rewriting logic. In practice this has been often the case for many:

1. *network algorithms*, e.g., [131, 133, 462, 221, 373, 375, 258, 391], and

2. *middleware designs and distributed reflective architectures*, e.g., [134, 338, 441, 163].

I further discuss all this work in Section 7.

5. Rewriting Logic Languages

In this section I discuss CafeOBJ, ELAN, and Maude, three languages that implement rewriting logic and whose researchers, through their language design and implementation work and through a host of important new techniques and applications, have made fundamental contributions to the rewriting logic research program. These are not the only rule-based languages that I could discuss. For example, OBJ [218], ASF+SDF [457], Tom [46], and Stratego [468] are other important rule-based languages; but they are somewhat more specialized in nature: OBJ and ASF+SDF deal with equational specifications; Tom enriches Java with rewriting capabilities; and Stratego is a rewriting strategy language aimed particularly at program transformation applications.

5.1. CafeOBJ

CafeOBJ `http://www.ldl.jaist.ac.jp/cafeobj/` [141] is a language containing in essence OBJ [218] as a functional sublanguage but extending substantially order-sorted equational logic in two orthogonal and complementary directions: (i) it supports behavioral specifications and their execution by behavioral rewriting in behavioral equational logic; and (ii) it also supports rewriting logic specifications. Furthermore, these orthogonal logical features are combined in the "CafeOBJ Cube" [141]. As OBJ, CafeOBJ has powerful module composition features through module hierarchies, parameterization, and module expressions. Two additional important features are CafeOBJ's support for object-oriented modules, and its support for observational transition systems (OTS), a special type of behavioral specifications ideally suited to specify transition systems such as network protocols and other distributed systems. CafeOBJ specifications can be formally analyzed in various ways. An important theme is the use of *proof scores* [201, 202] which reduce the proof of inductive properties about a CafeOBJ specification to rewriting on the underlying CafeOBJ engine. Of particular interest from the rewriting logic point of view is CafeOBJ's search feature, which supports breadth-first search *modulo* a user-specified equality predicate [202], a very useful form of abstraction-based model checking. Also interesting in this direction is the synergistic way, already mentioned in Section 3.11.3, in which CafeOBJ and Maude can be used together to analyze OTS specifications by model checking [476]. I discuss some CafeOBJ applications in Section 7.

5.2. ELAN

I have already mentioned in Section 3.5 the importance of strategies for controlling the rewriting process when the rules can be highly nondeterministic, and the key contributions that the ELAN researchers have made in this area. ELAN `http://elan.loria.fr/` [70, 69] supports the specification of sophisticated strategies that can guide the rewriting process to achieve complex tasks. This has applications in many areas that have been developed by the ELAN researchers; I discuss some of them in Section 7. In particular, from the beginning of the language the ELAN researchers have developed many applications of rewriting logic as a *logical framework* which greatly benefit from the use of strategies. The key idea is that the logical inference system used in a theorem prover or in some other logical procedure is typically nondeterministic. Therefore *search*, as opposed to deterministic *computation*, is essential. ELAN supports a corresponding distinction at the language level between computation rules, which are applied exhaustively without using strategies, and strategy-guided rules. At the language implementation level, besides the contributions to efficiently support strategies, an important additional contribution has been the development of novel compilation techniques for efficient rewriting modulo associativity-commutativity [265].

5.3. Maude

Maude `http://maude.cs.uiuc.edu/` [105, 106] supports both membership equational logic (its functional sublanguage of *functional modules*), and rewriting logic (*system modules*) in the fullest possible generality: equations and rules can be conditional and can have extra variables in their righthand sides and conditions, and rewriting modulo any combination of associativity and/or commutativiy and/or identity axioms is supported [106]. All this is achieved without sacrificing high performance thanks to Maude's use of advanced semi-compilation techniques and novel matching modulo algorithms [111, 171, 105, 172]. Maude has also powerful module composition operations and support for parameterized modules, theories and views. A key feature is its efficient support for *reflection* (see Section 3.4) through its `META-LEVEL` module. Besides providing powerful higher-order metaprogramming features (functions can take not just other functions as arguments, but entire *modules* as arguments), this makes the Maude module composition operations *extensible* [160], which is exploited in the Full Maude language extension [106] to support, for example, very convenient syntax for object-oriented specifications. Reflection is also exploited in an essential way in Maude's strategy language [175, 303]. A unique feature of Maude is its efficient built-in support for model checking. Reachability analysis and invariant verification are supported by its breadth-first `search` command; and *LTL* model checking by its `MODEL-CHECKER` module. Another important feature is its support for order-sorted unification modulo axioms, and for variant computations and symbolic reachability analysis modulo equational theories with the finite variant property [103, 152]. I discuss Maude's formal enviroment in Section 6.1, and some Maude applications in Section 7.

6. Tools

In Section 6.1 I discuss some tools supporting various kinds of formal reasoning about rewriting logic specifications. In Section 6.2 I discuss several more specialized tools that use rewriting logic and its reasoning methods to support formal analysis in various application domains.

6.1. Formal Tools for Rewriting Logic

In Section 3 I discussed in detail various formal properties that one often wants to verify about a rewrite theory. Tools supporting verification of such properties are very important. I discuss some of them here with the exception of the search and model-checking capabilities already native to rewriting logic languages: CafeOBJ, ELAN, and Maude support various forms of search analysis, and Maude also supports *LTL* model checking. Some of these formal tools, particularly the Maude-based ones, systematically use *reflection* (see Section 3.4) in their design: since formal analysis tools manipulate and transform theories, a reflective approach making such theories data structures manipulable within rewriting logic is very useful in practice. Indeed, several of the Maude formal tools use the Full Maude reflective extension of Maude [106, Part II] as their basis, and then use the general methodology outlined in [162] to add tool-specific reflective features. The tools mentioned below are an incomplete set of tools; see the rewriting logic bibliography in this issue for references to other tools.

6.1.1. The Maude Church-Rosser Checker and Coherence Checker (CRChC)

These two tools `http://maude.cs.uiuc.edu/maude-tools.html` are combined into one tool [161]. The *CRC* tool checks the confluence and sort-decreasingness of conditional order-sorted speciications modulo axioms, assuming they are operationally terminating (see Section 3.8). Instead, the *ChC* tool checks the coherence, or ground coherence, of a rewrite theory's rules R with respect to their equations E modulo axioms B, assuming that the equations themselves are (ground) confluent, sort-decreasing and operationally terminating.

6.1.2. The CARIBOO Termination Tool

CARIBOO [198, 214] `http://cariboo.loria.fr/index.html` is a termination tool written in ELAN which can prove ground termination of rewrite theories written in ELAN with respect to a given strategy (see Section 3.5). Based on an induction principle, it uses an abstraction mechanism to represent sets of terms symbolically with abstraction variables, and narrowing controlled by abstraction and ordering constraints. Orderings need not be chosen in advance but can be partially and incrementally determined by means of constraints.

6.1.3. The Maude Termination Tool (MTT) and μ-Term

MTT `http://maude.cs.uiuc.edu/maude-tools.html`, the Maude termination tool, [156, 157] supports termination proofs for generalized rewrite theories and for membership equational theories, which can both be conditional and have axioms such as associativity, commutativity and identity. As already explained in Section 3.8, the main technique used by *MTT* is that of non-termination-preserving theory transformations that transform such theories to either order-sorted or unsorted context-sensitive unconditional specifications modulo axioms. Termination tools such as μ-Term [9] or AProVE [210] can then be invoked by the user to try to prove the transformed theory terminating. μ-Term is in some ways closer to *MTT* because of its unrivaled support for context-sensitive termination and its support for order-sorted termination.

6.1.4. The Maude Sufficient Completeness Checker (SCC)

SCC `http://maude.cs.uiuc.edu/maude-tools.html`, the Maude Sufficient Completeness Checker [236] can check the sufficient completeness (see Section 3.7) of context-sensitive unconditional left-linear order-sorted equational theories modulo axioms [234], and in its most recent version also the sufficient completeness of both equations and rules in unconditional order-sorted left-linear theories modulo axioms [397]. *SCC* uses the CETA library of propositional tree automata operations developed by Joe Hendrix as part of his Ph.D. dissertation [231] to reduce all the above sufficient completeness problems to tree automata emptiness problems.

6.1.5. The Maude Inductive Theorem Prover (ITP)

ITP, `http://maude.cs.uiuc.edu/maude-tools.html`, the Maude Inductive Theorem Prover, was originally developed by Manuel Clavel and has been substantially extended by Joe Hendrix [107, 117, 231]. It supports inductive reasoning about membership equational theories in Maude and has been applied to a wide range of problems and also to build more specialized theorem provers for imperative programming languages and for modeling languages [118, 413, 110]. Its original support for structural induction has been more recently extended to also support coverset induction [231, 233]. An important feature of the *ITP* is its natural support for partiality, which is nicely demonstrated by the extended powerlist case study developed by Joe Hendrix as part of his Ph.D. thesis [231, 233].

6.1.6. The Maude Formal Environment (MFE)

Often a verification task requires interoperating different tools. For example, the proof of an inductive theorem using the *ITP* may be based on a structural induction scheme using constructors whose sufficient completeness proof is provided by the *SCC* tool, but the sufficient completeness proof relies on a weak termination assumption for which the *MTT* tool may be invoked. Similarly, a proof of ground coherence using the *ChC* tool may generate inductive proof obligations for the *ITP*, and requires a proof of confluence of the equations using the *CRC*, which itself relies on a proof of operational termination of those equations using the *MTT*. To support the seamless interoperation of formal tools for rewriting logic within a single formal environment, the Maude Formal Environment (*MFE*) [164, 165] has been developed as an extensible framework to which different Maude-based tools can be added. Besides allowing the user to ship proof tasks from one tool to another, *MFE* keeps track of the overall proof effort, and stores a record of the tool interactions and subproof invocations involved in such an overall proof, so that proof scripts can be stored and reused. *MFE* already exists as a prototype, and will be released as a Maude tool in the near future.

6.1.7. The Declarative Maude Debugger

In addition to the debugging capabilities already provided by Maude [106], the Declarative Maude Debugger `http://maude.sip.ucm.es/debugging/` [392] can interact with a user to find the causes of *wrong answers* in a Maude program execution and also of *missing answers*, which are particularly important for nondeterministic programs such as rewrite theories (system modules), but are also meaningful for deterministic ones (functional modules) because of sort information. The debugger traverses an *abbreviated proof tree*, which stores an abbreviated declarative summary of the computation, and interacts with the user asking questions until the cause of the bug is found.

6.1.8. Real-Time Maude

Real-Time Maude `http://heim.ifi.uio.no/peterol/RealTimeMaude/` [371] is a specification language and a formal tool built as an extension of Full Maude by reflection. It provides special syntax to specify real-time systems, including distributed object-oriented ones, where the time can be either discrete or continuous. It offers a range of formal analysis capabilities, including simulation, reachability analysis, and model checking. Real-Time Maude systematically exploits the underlying Maude efficient rewriting, search, and LTL model-checking capabilities to both execute and formally analyze real-time specifications, which are internally desugared into ordinary Maude specifications and Maude search and model-checking queries using reflection [371]. It furthermore supports model checking in a subset of *MTL* [283], and in *TCTL* [282] (see Section 3.11). Real-Time Maude has been applied in a wide range of industrial applications, including networks, embedded car software, and scheduling algorithms. It has also been used to give formal semantics to, and provide formal analysis for, several real-time programming languages and software modeling languages. I further discuss these applications in Section 7.4.

6.1.9. The PMaude Language Design

The PMaude language [277, 5] is an experimental specification language whose modules are probabilistic rewrite theories. It is still a language design, since it has not yet passed the prototyping level. However, since its methodology has already been successfully applied to a wide range of applications such as sensor networks, defenses agains Denial of Service (DoS) attacks, and stochastic hybrid systems (I further discuss these applications in Section 7.5), it seems appropriate to discuss it here. Recall from Section 3.10 that, due to their nondeterminism, 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 (see Section 3.3 in [324] for a detailed explanation). Using the transformed Maude module one can perform *Monte-Carlo simulations* of the given PMaude module. Using the methodology presented in [5] and discussed in Section 3.11.2, one can then use the VeStA and PVeStA tools discussed below to perform *statistical model-checking verification* of temporal logic properties of a real-time PMaude module expressed in either *CSL* or QuaTEx (see Section 3.11.1).

6.1.10. VeStA and PVeStA

The VeStA tool [420, 5] supports statistical model checking (see Section 3.11.2) of probabilistic real-time systems specified as either: (i) discrete or continuous Markov chains; or (ii) probabilistic rewrite theories in Maude. Furthermore, the properties that it can model check can be expressed in either: (i) *CSL/PCTL*, or (ii) the QuaTEx quantitative temporal logic (see Section 3.11.1). One important practical issue for any model-checking analysis is *scalability*. Since statistical model checking is parametric on a user-specified level of statistical confidence, if such a level is high, the number of Monte-Carlo simulations that have to be performed before VeStA can return an answer to a model-checking query can be very large. Fortunately, Monte-Carlo simulations can be run in parallel on different processors. This has led to the design and implementation of PVeStA `http://maude.cs.uiuc.edu/maude-tools.html` [23], which parallelizes the statistical model-checking analysis of probabilistic rewrite theories, making it highly efficient and scalable. For example, a realistic model-checking problem can be sped up by a factor of 46 on a 60-node parallel machine using PVeStA, compared with the time required for VeStA to perform the same task on a single node [23].

6.2. Some Domain-Specific Tools

This section is much more of a *random sample* than Section 6.1: there are many more domain-specific tools based on a rewriting logic semantics than the ones mentioned below, and discussing them all is out of the question. For example, any rewriting logic semantics of a programming language or of a modeling language expressed in Maude or Real-Time Maude automatically provides a tool supporting simulation, reachability analysis, and *LTL* model checking for such a language. A number of other tools are discussed much more briefly in Section 7, and the bibliography in this issue gives a more comprehensive picture. My main goal here is to give the reader a feeling through concrete examples for some of the advanced applications that can be supported by tools of this kind.

6.2.1. JavaFAN

JavaFAN `http://fsl.cs.uiuc.edu/index.php/JavaFAN`, the Java Formal Analyzer [191, 194] is a tool supporting the execution and analysis of the source code and the JVM code of Java programs. It is based on rewriting logic semantic definitions in Maude at both the Java and the JVM levels. The entire language, except for the libraries, is supported. Such definitions provide interpreters for Java and for the JVM. Also, multithreaded Java and JVM programs can be formally analyzed to detect violations of invariants using Maude's breadth-first search command; and terminating multithreaded programs can likewise be model checked with respect to *LTL* properties using Maude's *LTL* model checker. To facilitate the use of the tool and make knowledge of the underlying semantics unnecessary for users, Java and JVM code can be directly entered into JavaFAN and is then automatically translated into Maude. Similarly, JavaFAN provides an intuitive Java-like syntax for defining atomic predicates which makes it easy for users to define search commands and *LTL* queries only in terms of their programs. The performance of JavaFAN compares favorably with other state-of-the-art tools such as Java PathFinder on various benchmarks [191, 194], which is encouraging since JavaFAN is just a formal semantic definition of Java. One of the reasons for this is rewriting logic's distinction between equations and rules (the "abstraction dial" mentioned in Section 3.12), which, while still faithfully capturing the concrete semantics, allows a huge equational abstraction of the state space by expressing all deterministic features equationally and reserving rules for the nondeterministic, concurrent features.

6.2.2. K-Maude

As mentioned in Section 4.3, one of the important recent contributions to the rewriting logic semantics project is the K framework [408], which provides a concise and *highly modular* notation for programming language definitions. K is a new *definitional style* offering specific advantages over SOS-based styles such as those discussed in [423]. Furthermore, the relation between a K definition and its corresponding rewriting logic semantics is essentially one of *desugaring*, where what is conveniently implicit in the more compact K notation is made fully explicit in its rewriting logic counterpart. The K-Maude tool [422, 421] `http://fsl.cs.uiuc.edu/index.php/K-Maude` allows a user to define the semantics of a programming language in K and provides two main features. The first one is the automatic generation of a LaTeX rendering of the given K definition for ease of readability in two different styles, one more textual and another more graphical and intuitive. The second and main feature is that the rewriting logic semantics of K is supported by the tool, so that the rewrite theory corresponding to a language definition in K is automatically generated as a Maude module. In this way, K definitions can be executed as interpreters, and programs can be formally analyzed by reachability analysis and *LTL* model checking. K-Maude has already been applied to give K definitions for entire languages such as, for example, Scheme and C.

6.2.3. The MatchC Tool

As mentioned in Section 4.3, *matching logic* [406, 404] is another key contribution to the rewriting logic semantics project. It is a logic of programs with clear advantages over Hoare logic and separation logic. The key idea is to leverage a programming language's rewriting logic definition as the mathematical basis for the matching logic inference system. What matching logic essentially does is to extend such a definition into a full-fledged first-order reasoning system which manipulates *symbolic descriptions* (with existential and universal variables) of programs and their properties, and uses the term matching (modulo axioms) native to rewriting logic to express both properties about program configurations, and the application of semantic rules to such configurations. This accomplishes at a simpler, *structural* level all the separation properties achieved by separation logic at the *logical* level. In this way, programs involving pointers and complex data structures on the heap can be easily reasoned about. A very appealing feature of matching logic is that there is essentially no gap between the level of a language's semantic definition and that of its logic, whereas proving soundness and relative completeness of a Hoare logic with respect to an operational semantics is a highly nontrivial task. Although the matching logic ideas are very general, the current MatchC tool [404] realizes them for the C language with a remarkable level of automation and with very high efficiency `http://fsl.cs.uiuc.edu/index.php/Matching_Logic`. An impressive web-accessible collection of benchmarks has already been assembled [404].

6.2.4. The Maude-NPA

The Maude-NPA `http://maude.cs.uiuc.edu/maude-tools.html` [183] is a tool to verify security properties of cryptographic protocols *modulo* the algebraic properties of their cryptographic functions. The point is that one can "verify" that a protocol is correct with respect to the traditional Dolev-Yao model which treats the cryptography as a "black box," but an attacker can sometimes break such a protocol by making use of algebraic properties. For example, if the protocol uses an exclusive or operation $\oplus$, and the attacker has already seen a message m, then it can get message m' from the message $m \oplus m'$ just by performing the operation $m \oplus m' \oplus m$, since $\oplus$ is associative and commutative, and satisfies the equations $x \oplus x = 0$ and $x \oplus 0 = x$. All this means that reasoning modulo such axioms is an essential feature of security proofs, since attacks can be mounted using them. The Maude-NPA does exactly this by: (i) axiomatizing a protocol $\mathcal{P}$ as a (topmost) rewrite theory $(\Sigma_{\mathcal{P}}, E_{\mathcal{P}} \cup B, R_{\mathcal{P}})$, where $\mathcal{P}$'s equational properties are axiomatized by the equations $E_{\mathcal{P}} \cup B$, and $\mathcal{P}$'s transitions are axiomatized by the rules $R_{\mathcal{P}}$; (ii) characterizing *attack patterns* as terms with variables describing a possibly infinite set of concrete attack states; and (iii) using the rules $R_{\mathcal{P}}$ *in reverse*[19] to search for an *initial state* from the given attack pattern p. This is accomplished by narrowing p with the reversed rules $R_{\mathcal{P}}^{-1}$ modulo $E_{\mathcal{P}} \cup B$, which, as explained in Section 3.3 and in [340], is a complete reachability analysis method for topmost rewrite theories. Of course this still leaves the problem of computing $E_{\mathcal{P}} \cup B$-unifiers. Fortunately, many equational theories $E_{\mathcal{P}} \cup B$ of interest satisfy the finite variant property (see Section 3.3), so that the Maude-NPA uses narrowing at *two* levels: with $R_{\mathcal{P}}^{-1}$ modulo $E_{\mathcal{P}} \cup B$ for reachability analysis; and with $E_{\mathcal{P}}$ modulo B to compute $E_{\mathcal{P}} \cup B$-unifiers. Since the narrowing tree generated by a search from an attack pattern p is typically

[19]That is, a rule $t \to t'$ is now viewed in reverse as a rule $t' \to t$.

infinite, an important additional feature of the Maude-NPA is the use of very powerful *state space reduction* techniques [182] that often make such a symbolic search space finite, so that not finding an attack is in fact a *proof* that the protocol is safe from the given attack modulo the algebraic properties $E_{\mathcal{P}} \cup B$. I further discuss applications of the Maude-NPA in Section 7.3.

6.2.5. MOMENT2

MOMENT2 `http://www.cs.le.ac.uk/people/aboronat/tools/moment2-gt/` is an algebraic model management framework and tool written in Maude and developed by Artur Boronat [60]. It permits manipulating software models in the Eclipse Modeling Framework (EMF). It uses OMG standards, such as Meta-Object Facility (MOF), Object Constraint Language (OCL) and Query/View/Transformation (QVT), as a clean interface between rewriting-logic-based formal methods and model-based industrial tools. Specifically, it supports formal analyses based on rewriting logic and graph transformations to endow model-driven software engineering with strong analytic capabilities. MOMENT2 supports not just one fixed modeling language, but any modeling language whose *metamodel* is specified in MOF. In more detail, a modeling language is specified as a pair $(\mathcal{M}, \mathcal{C})$, where $\mathcal{M}$ is its MOF-based metamodel, and $\mathcal{C}$ are the OCL constraints that $\mathcal{M}$ should satisfy. Using rewriting-logic-based reflection and its efficient support in Maude, MOMENT2 provides an *executable algebraic semantics* for such metamodel specifications $(\mathcal{M}, \mathcal{C})$ in the form of a theory in membership equational logic (MEL) $\mathbb{A}(\mathcal{M}, \mathcal{C})$, so that a model M conformant with the metamodel $(\mathcal{M}, \mathcal{C})$ is exactly a term of sort *Model* in $\mathbb{A}(\mathcal{M}, \mathcal{C})$, and so that satisfaction of OCL constraints is also decidable using the algebraic semantics [64, 66].

Due to the executability of MEL specifications in Maude, the realization of MOF metamodels as MEL theories enhances the formalization and prototyping of model-driven development processes, such as: (i) model transformations; (ii) model-driven roundtrip engineering; (iii) model traceability; and (iv) model management. These processes permit, for example, merging models, generating mappings between models, and computing differences between models; they can be used to solve complex scenarios such as the roundtrip problem. In MOMENT2 the formal semantics of *model transformations* is given by rewrite theories specified in a user-friendly QVT-based syntax [62]. Such model transformations can describe the dynamic evolution of systems at the level of their models. Using the search and *LTL* model checking features of Maude, properties about the dynamic evolution of a model M conformant with a metamodel specification $(\mathcal{M}, \mathcal{C})$ can then be formally analyzed by model checking [62]. Real-time modeling languages can likewise be supported and analyzed [67]; this is further discussed in Section 7.4.4.

7. Some Applications

I discuss applications in areas such as automated deduction, software and hardware specification and verification, security, real-time and cyber-physical systems, probabilistic systems, and bioinformatics. Neither the choice of areas nor the work discussed in each of them aim at any completeness: again, this is just a sample.

7.1. Automated Deduction Applications

Perhaps the most important automated deduction applications are *formal tools* for different logics and automated deduction procedures that use rewriting logic as a logical framework. As explained in Section 4.1, the systematic idea common to all such tools is the faithful representation of their underlying inference systems as rewrite theories. Furthermore, using reflection very sophisticated tools can be built this way for many logics and for rewriting logic itself [108]. All the rewriting-logic-based tools discussed in Section 6.1 exemplify this general approach. But many other tools or prototypes for different automated deduction procedures have likewise been developed this way using either ELAN or Maude, including, for example,

- Constraint solving [68, 266, 267, 472, 242].
- Higher-order logics, procedures, and provers, explicit substitution calculi, and translations between such logics [45, 56, 146, 430, 147, 353, 434, 432, 433].
- Proof certification [354, 405].
- Rule Completion [264].
- Timed automata verification [52].
- Other theorem proving systems and procedures [94, 140, 466, 148, 395, 396].

7.2. Software and Hardware Specification and Verification

Systems need to be specified and verified at various levels of abstraction. Rewriting logic has very good properties as a semantic framework to support such specification and verification at different levels: at the level of models in the early stages of software design; at the level of code written in different programming languages; and at the hardware level. Furthermore, specification and verification of different network systems, and of distributed architectures, middleware, and coordination and reflection mechanisms can likewise be supported. All this has been described in broad outlines in Sections 4.3 and 4.4. Here I discuss in more detail some of the concrete applications that have been developed at all these levels.

7.2.1. Modeling Languages

As explained in Section 4.4, software design notations and modeling languages are quite useful, but they can be made even more useful by substantially increasing their analytic power through formal analysis, since this can make it possible to catch expensive design errors very early. Formal analysis is impossible or fraudulent without a formal semantics. Early work in developing rewriting-logic-based formal semantics focused on object-oriented design notations and languages [473, 352, 351], and stimulated subsequent work on UML and UML-like notations, e.g., [197, 268, 269, 474, 33, 110, 346, 345, 169, 347].

A more ambitious question is: can we give semantics not just to a single modeling language, but to an entire *modeling framework* where different modeling languages can be defined? This question has been answered positively in [61, 60, 62, 65, 66, 394], and has led to the *MOMENT2* and the *e-Motions* tools (see Sections 6.2.5 and 7.4.4).

I further discuss the semantics of *real-time modeling languages* [42, 393, 394, 40, 363, 67, 41] in Section 7.4. Some recent work has also considered the semantics of *multi-modeling languages* [63], that is, languages that can combine different models describing various perspectives about the same system.

7.2.2. Programming Languages

I have already given an overview of the rewriting logic semantics project in Section 4.3. Here I discuss concrete applications within this project. Early work focused on SOS definitions of process calculi and of small programming languages [326, 300, 459, 77, 460, 461]. The first application to a "real" programming language showing that this approach could scale up to large languages and could be used to analyze programs with competitive performance was the semantics of Java and the JVM [191, 194] described in Section 6.2.1. Since then, many other languages have been partially or totally defined in rewriting logic, sometimes using the K notation. For example, Beta [239] and KOOL [241] have been so defined; all of Scheme has been defined in [307, 308], and the formal semantics of C in [176] is arguably the most complete ever and will soon cover the entire C language. Another real language whose rewriting semantics has been fully defined in Maude is PLEXIL, a synchronous language developed by NASA to support autonomous spacecraft operations. The Maude-based formal executable semantics of PLEXIL [149] has become the de facto PLEXIL standard at NASA, against which the correctness of PLEXIL implementations is judged, and is the basis of other PLEXIL tools [399].

As mentioned in Section 4.3, the rewriting semantics of a language can be extended and/or abstracted to provide other kinds of static and dynamic analysis, for example, for units of measurement [91, 240], type checking [177], and runtime verification [407, 421]. Two extensions of a programming language's rewriting logic semantics to model fault detection (resp. hang detection) have been developed by Pattabiraman et al. [379] (resp. Wang et al. [469]). In [379], the authors use rewriting logic to model both the semantics of an assembly language and the hardware on which it runs, as well as various hardware errors. The overall goal is to provide a formal semantic framework (called SymPLFIED) to analyze the effectiveness of error detection mechanisms. Maude's search command is used for complete reachability analysis. In [469], a Linux-like operating system, as well as the underlying hardware, are formally specified in Maude in order to verify the detection effectiveness of an operating system's hang detector. In order to exhaustively explore all the possible hanging behaviors, Maude's search command is used (up to a specified depth) to explore all behaviors. It is also possible to use a language's rewriting logic semantics as the basis for program refactoring, as shown for C in [208] and for Java in [207].

Regarding tools supporting rewriting-logic-based language definitions, besides the direct use of rewriting logic languages for this purpose and the K-Maude tool discussed in Section 6.2.2, the Maude MSOS tool [89] supports definition, execution and analysis of language definitions on the MSOS style. Also, tools to simulate and analyze CCS processes and LOTOS specifications based on their rewriting semantics are discussed in [106, Section 21.2.3]. Deductive tools based on rewriting logic semantic definitions include the MatchC tool discussed in Section 6.2.3, and two Hoare logic provers built on top of the Maude ITP [118, 413]. Furthermore, the rewriting logic semantics of Java was used in [7] to automatically validate the semantics of a Java verification tool.

7.2.3. Hardware Specification and Verification

Prior to the use of rewriting logic, its equational logic subset (plus inductive principles) has been used for hardware specification and verification by various researchers, e.g., [215, 428, 250]. The earliest work I know on hardware specification and verification using Maude is by Neil Harman [228, 229]. Subsequent work has focused mostly on extending the rewriting logic semantics project from the level of programming languages to that of *hardware description languages* (HDLs). In this way, hardware designs written in an HDL can be both simulated and analyzed using the executable rewriting semantics of the HDL and tools like ELAN, CafeOBJ, or Maude. The first HDL to be given a rewriting logic semantics in Maude was ABEL [254]; this semantics was used not only for hardware designs, but also for hardware/software co-designs. An important new development has been the use of the rewriting logic semantics of an HDL for *generating sophisticated test inputs for hardware designs*. The point is that random testing can catch a good number of design errors, but uncovering deeper errors after random testing is hard and costly and requires a good understanding of the design to exercise complex computation sequences. The key insight, due to Michael Katelman, is that the rewriting semantics can be used *symbolically* to generate desired test inputs, not on a device's concrete states, but on states that are partly symbolic (contain logical variables) and partly concrete. Broadly speaking, this is an instance of the symbolic reachability analysis of rewrite theories I have discussed in Section 3.3; but for hardware verification the approach, first outlined in [257] and more fully developed in [256], has a number of unique features including: (i) the use of SAT solvers to symbolically solve Boolean constraints; (ii) support for user-guided random generation of partial instantiations; and (iii) a flexible *strategy language*, in which a hardware designer can specify in a declarative, high-level way the kind of test that needs to be generated. The effectiveness of this approach for generating sophisticated tests on real hardware designs, and for finding unknown bugs in such designs, has been demonstrated for medium-sized Verilog designs, including the I^2C-Bus Master Controller, and a microprocessor design [256, 251].

But the value of the rewriting semantics of an HDL is not restricted to testing. For example, the recent Maude-based rewriting logic semantics of Verilog in [309] is arguably the most complete formal semantics to date, both in the sense of covering the largest subset of the language and in its faithful modeling of nondeteministic features. Besides being executable and supporting formal analysis, this semantics has uncovered several nontrivial bugs in various mature Verilog tools, and can serve as a practical and rigorous standard to ascertain what the correct behavior of such tools should be in complex cases.

A more exotic application of rewriting logic semantics, for which it is ideally suited due to its intrinsically concurrent nature, is that of *asynchronous hardware designs*. These are digital designs which do not have a global clock, so that different gates in a device can fire at different times. Such devices can behave correctly in much harsher environments (e.g., a satellite in outer space) and with much wider ranges of physical operating conditions, than clocked devices. Asynchronous designs can be specified with the notation of *production rules*, which roughly speaking describe how each gate behaves when inputs to its wires are available. In [252] a rewriting logic semantics of asynchronous digital devices specified as sets of production rules is given and is realized in Maude (see also the longer paper [253] in this issue). This is the first executable formal semantics of such devices I am aware of. It can be used both for simulation purposes and for model-checking verification of small-sized devices (about 100 gates). An interesting challenge is how to scale up model checking for larger devices; this is nontrivial due to the large state space explosion caused by their asynchronous behavior.

7.2.4. Networks, Distributed Architectures, Middleware and Coordination

Networks and network protocols are among the most basic distributed systems, on top of which other systems communicate. There is a long history of work on formal specification and verification of network protocols. Early work using rewriting logic in this area includes [131, 133, 304, 462]. What rewriting logic seems to be particularly good at is its support for distributed objects, which naturally describes network nodes, and its flexibility in handling many different network and communication models: in-order or out-of-order, link-based communication, broadcast, multicast and unicast, active networks, wireless communication, and so on; and to also handle naturally real time and probabilistic features. For example, to faithfully model wireless communication in a sensor network the *geometry* of the network, the varying *power* at each node, the *time* required for transmission, and the *radius* that a wireless message broadcast can travel without being lost depending on the power with which it is transmitted, all need to be modeled as done in [375]; likewise, probabilistic algorithms for sensor networks, modeling of packet contention, clock synchronization, and formal analysis by statistical model checking are all naturally handled in [258]. Network specifications and analyses have tackled not just single protocols, but composable collections of them in actual active network systems, where important design problems not revealed by standard testing have been uncovered [373].

In some cases, e.g., [373], the network protocols specified and analyzed in rewriting logic had already been implemented before the formal analysis was done; but the most useful application of these methods is *before* a protocol is implemented. The reason is obvious, although not always perceived by the unenlightened: it is much easier to debug a design expressed as a formal executable specification which can be very quickly specified and can then be subjected to exhaustive formal analysis, than it is to adopt the standard alternative of *testing* successive prototypes written in, say, C. Also, using formal executable specifications one can much more easily explore *different* design alternatives and get a better understanding of the design choices. Everybody knows that debugging distributed code is notoriously hard to do, but the brute force approach still remains a widespread, wasteful and unreliable way to develop protocols. One of the key contributions of [221] was to make exactly this point in a very thorough way by taking to heart the idea of using formal specification and model-checking analysis in Maude to design a completely new protocol (L3A) and using this as a method to make the right choices between design alternatives and to fully debug the design. The beauty of it was that the subsequent implementation of L3A (reported in [222]) was essentially a transcription of the executable Maude specification into imperative code, which was accomplished much faster and in a much more reliable way than if the formal analysis had not been done. In the words of one of the authors [220],

> [the Maude modeling and analysis] gave us a complete story of a model with proofs *and* an implementation that was really done from the Maude model. In essence, the debugging was done in Maude and we could focus on implementation and performance issues and not the correctness of the protocol.

For a similar detailed case study of using Maude to fully explore a protocol design (in this case one that was not implemented, precisely because of the complexities uncovered by the formal analysis) see [223]. Some of the above protocols, e.g., [131, 133, 221, 223], are security protocols. I discuss them from a security perspective, as well as other security protocols, in Section 7.3.

Besides networks themselves, different distributed architectures and middleware systems, and various distributed coordination and reflection mechanisms, have also been modeled and formally analyzed in rewriting logic. For example, there is work on formalizing different aspects of ODP [350, 167, 168, 154, 166, 402], SOAP [13], CORBA [14], and the SMEPP P2P middleware [153]. Similarly, work on formal models of coordination includes [85, 86, 441, 444]. Closely related to coordination models is work on formal models of distributed object reflection and adaptation [134, 338, 441, 261, 88]. For work on formal analysis of web applications and services using rewriting logic specifications see [15, 163].

7.3. Security

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 and authenticity*: malicious attackers should not be able to get secret information or to falsely impersonate honest agents; on the other, we are also concerned with properties such as *availability*, which may be destroyed by a (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. Furthermore, security concerns span many different levels and subsystems, such as network protocols, programming languages, browsers, web applications, operating systems, and hardware.

Rewriting logic has been successfully applied to analyze various security properties for a wide range of systems and at different levels of abstraction. Research in this general area includes: (i) work on cryptographic protocols; (ii) work on network security; (iii) work on browser security; (iv) work on access control, and (v) work on code security.

7.3.1. Cryptographic Protocol Specification and Analysis

The earliest work on the formal specification and analysis of cryptographic protocols in rewriting logic is by Denker, Meseguer, and Talcott [132, 133]. This stimulated further work by Rodriguez [400, 401], and inspired Millen and Denker to use Maude to give a formal semantics to their cryptographic protocol specification language CAPSL, and to endow CAPSL with an execution and formal analysis environment [135, 136, 137, 138]. In a similar vein, Cervesato, Stehr, and Reich gave a rewriting logic semantics to the MSR security specification formalism, leading to the first executable environment for MSR [87, 390].

An important breakthrough was the realization that, by specifying a crypto protocol as a rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R)$, where $E \cup B$ describes the algebraic properties of the protocol's cryptographic functions, and R are the protocol rules, one could use narrowing with R modulo the equations $E \cup B$ as a complete reachability analysis method (see Section 3.3). This was first pointed out in [339, 340]. This advance was crucial for two main reasons: (i) protocols could be analyzed *modulo* their algebraic properties $E \cup B$; it is well known (as already pointed in Section 6.2.4), that the traditional Dolev-Yao analysis treating cryptography as a "black box" is too weak, since protocols proved secure under the black box assumption can sometimes be broken by an attacker using the properties $E \cup B$; and (ii) by adopting a narrowing-based *symbolic* model-checking approach, the fact that the number of protocol states, and even the number of protocol *sessions*, is unbounded does not preclude performing a complete analysis. Based on these ideas and on the rich experience about symbolic reachability methods in the NRL Protocol Analyzer [306], Santiago Escobar, Catherine Meadows and I have developed the Maude-NPA protocol analysis tool, discussed in Section 6.2.4, and its foundations [180, 183]. To the best of my knowledge the Maude-NPA is the most advanced analysis tool to date for analyzing cryptographic protocols modulo algebraic properties with an active intruder and an unbounded number of sessions in a complete way and without using any abstractions or approximations. For many protocols, Maude-NPA can exploit the fact that $E \cup B$ happens to enjoy the finite variant property to obtain a finitary $E \cup B$-unification algorithm by variant narrowing (see [190] and Section 3.3). But finitary algorithms for theories $E \cup B$ not having the finite variant property, e.g., homomorphic encryption, are also supported by Maude-NPA. In this way, we have formally analyzed protocols of the form $\mathcal{R} = (\Sigma, E \cup B, R)$, where $E \cup B$ can be a cryptographic theory involving a combination of functionalities such as: (i) encryption-decryption; (ii) bounded associativity; (iii) Diffie-Hellman exponentiation; (iv) exclusive or; and (v) homomorphic encryption [178, 183, 181, 412, 179]. In general, of course, protocol analysis with an unbounded number of sessions is undecidable. However, thanks to Maude-NPA's use of grammars [180] and of other state space reduction techniques [182, 184], a protocol's symbolic state space can often become finite while remaining complete. This means that one can not only be sure to find attacks if they exist, but that one can often *prove* that the specified attacks are not possible modulo the algebraic properties $E \cup B$. Protocols are often compositions of smaller protocols, so that, even when the subprotocols are secure, unforeseen insecure interactions may take place in a composition. To support compositional reasoning in Maude-NPA, new composition constructs and associated analysis methods have been developed in [185].

7.3.2. Network Security

I have already discussed in Section 7.2.4 the usefulness for protocol design of the formal specifications in [221] and [223]. Since both specify network security protocols, I briefly discuss them here from a security perspective. The work in [221] describes in detail the design steps, using Maude and its model-checking formal analysis, to arrive at the design of L3A, an accounting protocol built on top of IPsec (using IPsec tunnels) to support billing which was subsequently implemented in [222]. One of the unique features of L3A is that it is resilient under *cramming attacks*, where a malicious attacker can direct traffic to a client for the purpose of having the client billed for the spurious traffic. The work in [223] uses Maude and its model-checking features to explore and analyze a new protocol design called Sectrace. The problem addressed by Sectrace is the setting up of associations and policies assumed, but not provided, by the IPSec protocol in order to provide encryption and authentication services. Due to the presence of nested channels and concatenated channels involving several security gateways, setting up such security associations and policies is highly nontrivial. Indeed, the formal analysis uncovered quite complex issues, such as the fact that certain possibilities to set up correct security associations could be missed; and that concurrent runs of the protocol could cause undesirable interference effects. The design of Sectrace was not further advanced to resolve these issues, but the lessons learned were very valuable and could not have been learned without such kind of formal specification and analysis.

The work by Gutierrez-Nolasco et al. in [226] uses formal specification and verification in Maude to address a very important and real problem: how can the security requirements of a protocol be balanced with other equally important requirements such as timeliness or other QoS requirements? And how can a design be made *adaptive*, so that such a balancing can take place at runtime? This problem was addressed in the context of the Secure Spread group communication protocol [427], for which a formal specification in Maude had been previously developed. One problem with Secure Spread was its assumption of virtual synchrony (VS), which is more restrictive and expensive than the extended virtual synchrony (EVS) semantics. What the work in [226] accomplished was to extend the formal Maude specification of Secure Spread to a considerably more flexible and dynamically adaptive secure group communication protocol with two simultaneous dimensions of adaptation: (i) synchrony, which could be chosen to have the VS or EVS semantics; and (ii) group key security, where various levels of laziness in the key establishment protocol could be specified.

Regarding *availability* properties, a big problem in network security is Denial of Service (DoS) attacks, which are often distributed (DDoS) and employ many "bots," i.e., large numbers of compromised machines from which a simultaneous attack is mounted. Two key questions are how to make network protocols resilient to DoS attacks, and how to formally analyze such resilience. A probabilistic approach to the formal specification and verification of DoS-resilient protocols is very natural for two reasons: (i) both the attacker models and the defense algorithms may be probabilistic; and (ii) the answers from a formal analysis will typically not be "true" or "false" answers; they will instead be numerical *quality of service* (QoS) answers, such as the expected latency for a client to get a response from a server during an attack of given intensity. This means that probabilistic rewrite theories (see Section 3.10) and statistical model checking of qualitative properties in QuaTEx (see Section 3.11.1) are ideally suited for specification and analysis of DoS-resilient protocols. This is exactly the approach taken by Agha et al. in [4] to analyze the DoS-resilience of a hardening of the TCP/IP protocol by means of the Selective Verification (SV) probabilistic DoS-defense mechanism. This work has been later extended by AlTurki et al. [24] to the formal specification and verification of a more sophisticated DoS-defense protocol, namely, Adaptive Selective Verification (ASV), where both clients and servers ramp up or slow down their response to a DoS attack based on its perceived intensity. In his recent Ph.D. thesis [18], Musab AlTurki has modularized ASV as a meta-object wrapper that can be added to the objects of a distributed application without changing the application code; he has also extended the study of DoS defense mechanisms from simple client-server architectures to complex orchestrations of web services in Orc: he has shown how combinations of web services can be secured against DoS attacks by wrapping its distributed objects with ASV wrappers (for Orc and its rewriting logic semantics see Section 7.4.3).

However, neither the analysis nor the DoS-defense mechanisms need to be probabilistic. For example, in [424], Shankesi et al. give a formal specification in Maude of the VoIP Session Initiation Protocol, and of defense mechanisms against DoS amplification attacks, and use LTL model checking in Maude with parametric predicates, which can actually measure performance metrics, to formally analyze the effectiveness of the specified defenses. Another DoS defense mechanism not involving probabilities is that of *cookies*. In [88], Chadha et al. propose a formal specification of the cookie-based DoS defense mechanisms as a modular *wrapper*, which can be composed with an underlying communication protocol without any modifications to the protocol's code; and they prove that this modular approach preserves all the *safety properties*, for example secrecy properties, enjoyed by the underlying protocol. That is, the addition of this DoS defense can be made modular both at the code level and at the level of verifying safety properties, which need not be re-verified when the cookie wrappers are added.

7.3.3. Browser Security

To achieve end-to-end security, traditional machine-to-machine security measures are insufficient if the human-computer interface is compromised. This is particularly the case for browsers, where visual spoofing attacks that exploit GUI logic flaws can lure even security-conscious users to perform unintended actions. In [92], Shuo Chen, Ralf Sasse, Helen Wang, Yi-Min Wang and I called the preventing of such visual spoofing attacks "securing the last 20 inches." That is, all the machine-to-machine protocols, code and hardware may be secure, but these visual attacks take place in the last 20 inches separating a user's eyes from the screen where he/she is interacting with a browser using the browser's GUI. Before we performed a rewriting-logic based formal analysis of Microsoft's Internet Explorer (IE), it seems fair to say that the approach to IE security was basically *reactive*, i.e., each new attack was patched up, but there was no systematic way to predict and prevent future attacks. Based on an in-depth study of IE's code, we developed a formal specification of IE (including a model of the user) in Maude as a rewrite theory. We then characterized status bar and address bar spoofing attacks as violations of *visual invariants*, where the web site that the user *assumes* he/she is interacting with is different from the real web site: what you see is *not* what you get. Our model-checking-based formal analysis uncovered nine status bar types of spoofing attacks and four address bar spoofing attack types that had not been previously mounted against IE. For each attack type, a malicious web page producing an actual attack could be built. The IE team then confirmed all these attack scenarios and proceeded to make IE secure for these new types of attacks.

This work stimulated new research by C. Grier, S. Tang and S. King at the University of Illinois at Urbana-Champaign. They asked the question: can we use rewriting logic not to uncover browser security flaws a posteriori, but to design a browser that is secure by construction? This question was answered in their paper [224], where they presented the design and implementation of the OP secure browser, whose design was specified in Maude and was subjected to model-checking analysis to uncover design flaws. A more advanced browser variant of OP, OP2, as well as IBOS, a design and system implementation which integrates into a single architecture a secure browser and a secure operating system, are described in detail in Shuo Tang's thesis [449]. This design is being submitted to detailed formal analysis by Ralf Sasse, who has already verified by model checking the same origin policy; this and other verification results will be reported in Sasse's upcoming doctoral dissertation at UIUC.

The security of web browsers is part of a bigger problem, namely, the security of web applications. In [15] Alpuente et al. give a rewriting semantics of web applications which formalizes the interactions between multiple browsers and a web server through a request/response protocol that supports the main features of HTTP and models browsers actions such as refresh, forward/backward navigation, and window/tab openings. Their formal model also supports a scripting language which abstracts the main common features (e.g. session data manipulation, data base interactions) of the most popular web scripting languages and formalizes adaptive navigation, where page transitions may depend on users data or previous computation states of the web application. They also show how the temporal logic of rewriting *LTLR* and its Maude-based model checker (see Section 3.11.1) are ideally suited to express and verify various safety and security properties of web applications specified this way.

7.3.4. Access Control

Access control policies specify the conditions under which access to information is permitted or should be denied in a system. They are a key security feature of many systems and apply way beyond the original setting of operating systems: enterprise systems, web-based systems, and even cloud computing applications all need and use access control policies. Such a policies are typically specified as collections of access control rules. Several authors, e.g., [48, 126, 145], have formalized access control rules as rewrite rules. To further increase the expressive power of access control rules, the corresponding rewrite rules may be conditional, and they may be controlled by some given strategy. This leads to the notion of rewrite-based access control policies, and to a corresponding notion of policy composition [145]. One important advantage of this rewriting-based formalization is that sophisticated forms of formal analysis about an access control policy become possible. C. Kirchner, H. Kirchner, and A. Santana de Oliveira show in [263] how narrowing-based analysis (see Section 3.3) with the rewrite rules formalizing an access control policy and following given strategies can provide an in-depth understanding of policies and their dynamic behavior to policy designers. Furthermore, since the rewrite rule formalization is directly executable and, using a language like Tom, can be automatically translated into Java code, the paper [127] shows how rewrite-based access control policies can be used to generate Java monitoring code for such policies. The monitoring code can then be automatically "weaved" with the application code it monitors using aspect-oriented methods.

7.3.5. Code Security

Many security attacks such as format string, heap corruption and buffer overflow involve malicious code performing pointer manipulations. The insight of Shuo Chen and his collaborators in [93] is that all these problems have a common cause that they call *pointer taintedness*, where a pointer is tainted whenever a user input can directly or indirectly be used as a pointer value. The formal approach taken in [93] is a good example of the general way of giving a rewriting logic semantics to a programming language already described in Sections 4.3 and 7.2.2. Indeed, what it is done in [93] is to give a rewriting semantics to a sequential programming language (since the language used is deterministic, only equations are needed) which includes a *memory model*. This formal model is then used to reason formally about pointer taintedness. This reasoning is applied to several library functions to extract security preconditions which guarantee the absence of pointer taintedness. In this way, various commonly occurring security vulnerabilities, such as format string, heap corruption and buffer overflow vulnerabilities can be both detected and prevented.

The topic of *application level insider attacks*, where a malicious insider tries to overwrite one or more data items in an application, has been systematically studied by Pattabiraman et al. in [380]. The application code is modeled at the assembly level by defining the rewriting logic semantics of assembly code. An insider attack is then represented as a corruption of data values at specific points in the program's execution (called attack points). The behavior of an application code subjected to security attacks in the specified attack points is then formally modeled by replacing concrete values by appropriate symbolic values when attack points are reached; and by systematically modeling with rewrite rules the behaviors that such symbolic values can generate. Given the application code and its inputs, a set of attack points, and a goal state that the attacker intends to achieve, Maude is then used to generate a comprehensive set of insider attacks that lead to the goal state.

A very elegant application of a programming language's *abstract* rewriting logic semantics (see Section 4.3) to Java code security is presented by Alba-Castro et al. in [11, 12] as part of their rewriting-logic-semantics-based approach to proof carrying code. The key idea is to use an abstract rewriting logic semantics of Java that correctly approximates security properties such as *noninterference* (that is, the specification of what objects should not have any effects on other objects according to a stated security policy [217]) and *erasure* (a security policy that mandates that secret data should be removed after its intended use). Since the abstract rewriting logic semantics is finite-state, it supports the automatic creation of certificates for noninterference and erasure properties of Java programs that are independently checkable and small enough to be used in practice.

Yet another code security application is M. LeMay and C. Gunter's verification of the security and fault-tolerance requirements of their cumulative attestation kernel (CAK). This kernel runs on a flash microcontroler unit (MCU) as part of an advanced metering infrastructure for utilities in the Power Grid. For example, a meter for electricity consumption in a household or business will use such an MCU, connected to a communications network, to automatically gather and send power consumption data. Security threats include the installation of malware in the MCU to send false data. The CAK code protects the MCU against such attacks and also provides fault tolerance. The CAK's behavior has been fully specified as a rewrite theory in Maude, and Maude's LTL model checker has been used to verify that key security and fault tolerance requirements of the CAK are satisfied [281].

7.4. Real-Time and Cyber-Physical Systems

I have already mentioned in Section 3.11.2 that ELAN has been used to model check timed automata in [52]. Here I focus on the more general issue of specification and formal analysis of real-time and cyber-physical systems which, by having arbitrary data structures in their discrete states, may not be specifiable at all as timed automata but have a natural specification as real-time rewrite theories (see Section 3.9). The best tool currently available to specify and analyze systems as real-time rewrite theories is Real-Time Maude (see Section 6.1.8). A wide range of applications have been specified and analyzed in Real-Time Maude including: (i) network protocols; (ii) middleware for distributed real-time systems; (iii) real-time programming languages; (iv) real-time modeling languages; (v) scheduling algorithms; and (vi) cyber-physical systems. Furthermore, in some cases the Real-Time Maude specifications have been used to easily derive actual system prototypes operating in physical time.

7.4.1. Real-Time Network Protocols

Because of their frequent use of timers, timeouts, roundtrip times, and so on, many network protocols (discussed already in Section 7.2.4) are in fact real-time systems. This means that their rewriting logic specification naturally takes the form of a real-time rewrite theory, and that their model-checking analysis can best be performed by the kind of real-time model checking supported by Real-Time Maude. Important network protocols that have been specified and have been thoroughly analyzed in Real-Time Maude include: (i) the AER/NCA suite of active network protocols [359, 365, 373] already mentioned in Section 7.2.4; (ii) the NORM multicast protocol standard [285, 286]; and (iii) the OGDC wireless sensor network algorithm [452, 376]. This last work is quite unique, because it seems to be the first time that a sensor network was fully formally modeled in all its main aspects, such as geometry, power, transmission times, effective broadcast radius for each node, and so on; and because the formal analysis turned out to be more accurate than (and to uncover flaws in) prior simulation-based analyses of OGCD. It is also noteworthy in terms of scalability, since a network of up to 600 nodes was modeled and analyzed. In fact, a sensor network is more than a network: it is a cyber-physical system, which in this work was fully modeled as such.

7.4.2. Middleware for Distributed Real-Time Systems

Many distributed real-time systems (DRTS), such as integrated modular avionics systems and distributed control systems in motor vehicles, are made up of a collection of components that communicate asynchronously and that must change their state and respond to environment inputs within hard real-time bounds. Such systems are often safety-critical and need to be certified; but their certification is currently very hard due to their distributed nature. The Physically Asynchronous Logically Synchronous (PALS) architectural pattern [341] can greatly reduce the design and verification complexities of achieving virtual synchrony in a DRTS. A key property that the PALS pattern should satisfy is to be provably *correct-by-construction.* This of course requires that the pattern itself should be formally specified as a parameterized construction. In [329, 330] Peter Ölveczky and I have used Real-Time Maude to specify PALS as a formal model transformation that maps a synchronous design, together with performance bounds of the underlying infrastructure, to a formal DRTS specification that is semantically equivalent to the synchronous design. This semantic equivalence is proved, showing that the formal verification of temporal logic properties of the DRTS can be reduced to their verification on the much simpler synchronous design. Furthermore, the PALS period is shown to be the shortest possible. The issue of how to mechanize PALS at the Maude metalevel, and an application of PALS to a wireless network protocol are discussed in [255].

7.4.3. Real-Time Programming Languages

How should the formal semantics of a *real-time* programming language be defined? And how can programs in such a language be formally analyzed? For an ordinary programming language, the rewriting logic semantics project answers the first question by saying: "with a rewrite theory," and the second by saying: "by model checking and/or deductive reasoning based on such a theory." The obvious answers for real-time programming languages are: (i) "with a real-time rewrite theory," and (ii) "by real-time model checking and/or deductive reasoning based on such a theory." Of course, the effectiveness of such answers has to be shown in actual languages. Three real-time programming languages have been given semantics in exactly this way, and their semantics have been used to verify their programs.

In [19], AlTurki et al. present a language for real-time concurrent programming for industrial use in DOCOMO Labs called L. The goal of L is to serve as a programming model for higher-level software specifications in SDL or UML. A related goal is to support formal analysis of L programs by both real-time model checking and static analysis, so that software design errors can be caught at design time. The way all this is accomplished is by giving a formal semantics to L in Real-Time Maude, which automatically provides an interpreter and a real-time model checker for L. Static analysis capabilities are added to L by using Maude to define an *abstract semantics* for L in rewriting logic, which is then used as the static analyzer.

As already mentioned in Section 4.2, the Orc model of real-time concurrent computation [342, 343, 470] has been given semantics in rewriting logic using real-time rewrite theories [20, 21, 18]. Although Orc is a very simple and elegant language, its real-time semantics is quite subtle for two reasons. First, in the evaluation of any Orc expression, internal computation always has higher priority than the handling of external events; this means that, even without modeling time, a vanilla-flavored SOS semantics is not expressive enough to capture these different priorities: two SOS relations are needed [343]. Second, Orc is by design a real-time language, where time is a crucial feature. Using real-time rewrite theories, this double subtlety of the Orc semantics was faithfully captured by Musab AlTurki and I in [20], and has been expressed in an even simpler way using subsorts and memberships in [18]. This semantics has yielded an Orc interpreter and a real-time model checker. But Orc is not just a model of computation: it is also a concurrent programming language. This suggested the following challenge question: can a correct-by-construction distributed Orc implementation be derived from its rewriting logic semantics? This question was answered in two stages. Since, as discussed in Section 4.3, a small-step SOS semantics is typically horribly inefficient and it was certainly so in the case of Orc, a much more efficient *reduction semantics* was first defined in [21], and was proved to be bisimilar to the small-step SOS semantics. This semantics provided a much more efficient interpreter and model checker. Furthermore, to explicitly model different Orc clients and various web sites, and their message passing communication, the Orc semantics was seamlessly extended in [21] to a distributed object-based Orc semantics, which modeled what a distributed implementation should look like. The only remaining step was to pass from this model of a distributed implementation to an actual Maude-based distributed real-time implementation. This was accomplished in [22] using three main ideas: (i) the use of sockets in Maude to actually deploy a distributed implementation; (ii) the systematic replacement of logical time by physical time, supported by ticker objects external to Maude, while retaining the rewriting semantics throughout; and (iii) the experimental estimation of the physical time required for "zero-time" Maude subcomputations, to ensure that the granularity of time ticks is such that all "instantaneous transitions" have already happened before the next tick. Ideas (i)–(iii) are of course much more widely applicable: they have subsequently been used to derive prototypes of real-time systems from their rewriting logic specifications for other applications such as medical devices, as explained in Section 7.4.6.

Creol is an object-oriented language supporting concurrent objects which communicate through asynchronous method calls. Its rewriting-logic-based operational semantics was defined in [245] without real-time features. However, to support applications such as sensor systems with wireless communication, where messages expire and may collide with each other, Creol's design and operational semantics have been extended in [58] to Timed Creol using rewriting logic. The notion of time used by Timed Creol is described as a "lightweight" one in [58]. Time is discrete and is represented by a time object. This approach does not require a full use of the features in Real-Time Maude (Maude itself is sufficient to define the real-time semantics). The effectiveness of Timed Creol in the modeling and analysis of applications such as sensor networks is illustrasted in [58] through a case study.

7.4.4. Real-Time Modeling Languages

The usefulness and importance of giving a formal rewriting logic semantics to software modeling languages has already been discussed in Sections 4.4 and 7.2.1. In particular, there is strong interest in modeling languages for real-time and embedded systems. The rewriting logic semantics for such modeling languages can be naturally based on real-time rewrite theories. Using a tool like Real-Time Maude, what this means in practice is that such models can then be simulated; and that their formal properties, in particular their safety requirements, can be model checked. Furthermore, the simulations and formal analysis capabilities added to the given modeling language can be offered as "plugins" to already existing modeling tools, so that much of the formal analysis happens "under the hood," and somebody already familiar with the given modeling notation can make use of such formal analysis without needing to have an in-depth understanding of the underlying formalism.

The Ptolemy II modeling language [170] supports design and simulation of concurrent, real-time, embedded systems expressed in several models of computation, such as state machines, data flow, and discrete-event models, that govern the interaction between concurrent components. A user can visually design and simulate hierarchical models, which may combine different models of computations. Furthermore, Ptolemy II has code generation capabilities to translate models into other modeling or programming languages such as C or Java. Discrete-event (DE) models are among the most central in Ptolemy II. Their semantics is defined by the *tagged signal model* [280]. The work by Bae et al. in [42] endows DE models in Ptolemy II with formal analysis capabilities by: (i) defining a semantics for them as real-time rewrite theories; (ii) automating such a formal semantics as a model transformation using Ptolemy II's code generation features; (iii) providing a Real-Time Maude plugin, so that Ptolemy II users obtain an extended GUI to define temporal logic properties of their models in an intuitive syntax and can invoke Real-Time Maude from the GUI to model check their models. This work has been further advanced in [40] to support not just flat DE models, but *hierarchical* ones. That is, above tasks (i)–(iii) have been extended to hierarchical DE models; this extension is nontrivial, because it requires combining synchronous fixpoint computations with hierarchical structure.

AADL is a standard for modeling embedded systems that is widely used in avionics and other safety-critical applications. However, AADL lacks a formal semantics, which severely limits both unambiguous communication among model developers and the formal analysis of AADL models. In [363] Ölveczky et al. define a formal object-based real-time concurrent semantics for a behavioral subset of AADL in rewriting logic, which includes the essential aspects of AADL's behavior annex. Such a semantics is directly executable in Real-Time Maude and provides an AADL simulator and LTL model-checking tool called *AADL2Maude*. *AADL2Maude* is integrated with the OSATE AADL tool, so that OSATE's code generation facility is used to automatically transform AADL models into their corresponding Real-Time Maude specifications. Such transformed models can then be executed and model checked by Real-Time Maude. One difficulty with AADL models is that, by being made up of various hierarchical components that communicate asynchronously with each other, their model-checking formal analysis can easily experience a state space explosion. However, many such models express designs of distributed embedded systems which, while being asynchronous, should behave in a virtually synchronous way. This suggests the possibility of using the PALS pattern (see Section 7.4.2) to pass from simple synchronous systems, which have much smaller state spaces and are much easier to model check, to semantically equivalent asynchronous systems, which often cannot be directly model checked but can be verified indirectly through their synchronous counterparts. This has led to the design of the Synchronous AADL sublanguage in [41], where the user can specify synchronous AADL models by using a sublanguage of AADL with some special keywords. A synchronous rewriting semantics for such models has also been defined in [41]. Using OSATEs code generation facility, synchronous AADL models can be transformed into their corresponding Real-Time Maude specifications in the *SynchAADL2Maude* tool, which is provided as a plugin to OSATE. Likewise, the user can define temporal logic properties of synchronous AADL models based on their features, without requiring knowledge of the underlying formalism, and can model check such models in Real-Time Maude.

A more ambitious goal is to provide a *framework*, where a wide range of real-time domain-specific visual languages (DSVLs), as well as their dynamic real-time behavior, can be specified with a rigorous semantics. This is precisely the goal of two frameworks and associated tools: (i) the *e-Motions* framework [394]; and (ii) *MOMENT2*'s support for real-time DSVLs [67].

- In *e-Motions*, DSVLs are specified by their corresponding metamodels, and dynamic behavior is specified by rules that define in-place model transformations. But the goals of *e-Motions* do not remain at the syntax/visual level: they also include giving a precise rewriting logic semantics in Real-Time Maude to the different real-time DSVLs that can be defined in *e-Motions*, and to automatically support simulation and formal analysis of models by using the underlying Real-Time Maude engine. The formal semantics translates the metamodel of a DSVL as an object class, the corresponding models as object configurations of that class, and the *e-Motions* rules as rewrite rules. Since all these translations are automatic and define a DSVL's formal semantics, a modeling language designer using *e-Motions* does not have to explicitly define the DSVL's formal semantics: it comes for free, together with the simulation and model-checking features, once the DSVL's metamodel and the dynamic behavior rules are specified.

- In [67], the *MOMENT2* framework (see Section 6.2.5) has been extended to support the formal specification and analysis of real-time model-based systems. This is achieved by means of a collection of built-in timed constructs for defining the timed behavior of such systems. Timed behavior is specified using in-place model transformations. Furthermore, the formal semantics of a *timed behavioral specification* in *MOMENT2* is given by a corresponding real-time rewrite theory. In this way, models can be simulated and model checked using *MOMENT2*'s Maude-based analysis tools. In addition, by using in-place multi-domain model transformations in *MOMENT2*, an existing model-based system can be extended with timed features in a non-intrusive way, in the sense that no modification is needed for the class diagram.

7.4.5. Resource Sharing Protocols

Real-time *resource sharing protocols* are protocols governing the way in which multiple tasks can share common resources such as a data structure, a memory area, a file, a set of registers in a peripheral device, and so on. The dynamic behavior of such protocols divides naturally into a *scheduling part*, and a *resource access part*. Although this is a very well-established area, the emergence of multicore machines has brought about new protocols and more sophisticated approaches, for which correctness is not obvious, so that formal modeling and analysis can be a valuable design methodology. The first work applying rewriting logic in this area was by P. Ölveczky and M. Caccamo, who modeled and analyzed in Real-Time Maude the CASH capacity sharing scheduling algorithm [364], corresponding to the scheduling part of a resource sharing protocol. Search analysis of CASH's Real-Time Maude specification uncovered a previously unknown behavior that led to missed deadlines. This was a subtle error that it would have been virtually impossible to detect by testing. Indeed, extensive Monte-Carlo simulation was utterly incapable of detecting the flaw. The CASH protocol furthermore illustrated a broad class of applications beyond the pale of (timed) automata-based analysis techniques. The point is that model-checking algorithms for such techniques work only for finite-state real-time systems, but the Real-Time Maude formal analysis showed that the queues in the state of the CASH protocol could grow in an unbounded manner.

A broader framework for formally modeling and analyzing real-time resource sharing protocols, in both their scheduling and resource access parts, is presented by P. Ölveczky, P. Prabhakar and X. Liu in [374]. In particular, [374] shows how crucial properties such as: (i) unbounded priority inversion; (ii) deadlocks; and (iii) schedulability, can be analyzed for such protocols when they are specified as real-time rewrite theories. The effectiveness of this framework is illustrated by means of the analysis of the priority inheritance protocol (PIP).

7.4.6. Cyber-Physical Systems

Cyber-physical systems are real-time systems, often distributed, which interact with the physical world by sensing and possibly by means of actuators. A number of such systems have been specified and modeled in Real-Time Maude. One example is the OGCD wireless sensor network algorithm in [452, 376] already described in Section 7.4.1. Another example is the family of traffic system designs specified and analyzed in [372], where one of the experiences gained was the ease with which the use of distributed objects and class inheritance provided a very high degree of genericity and extensibility of the different designs (including European and American light regimes, a special regime for emergency vehicles, and so on), and allowed for a distributed control without any need for a centralized controller. A third example is the modeling in Real-Time Maude of object-oriented real-time systems that follow the Actor model, and the application of this modeling style to the specification and analysis of the *simplex architecture* [142], a software architecture for fault-tolerant real-time control systems. Yet a fourth example is the use of Real-Time Maude to analyze embedded code in a Japanese car design; the analysis uncovered flaws in the embedded code but has not been published for proprietary reasons.

The safe interoperation of medical devices has been the topic of several research papers, which have formally modeled and analyzed various device configurations in Real-Time Maude. For example, in [360] P. Ölveczky describes the application of Real-Time Maude to the formal modeling and analysis of a network integrating an X-ray machine, a ventilator, and a controller. This configuration automates a similar manual interoperation between an X-ray machine and a ventilator for which an accidental death in an operating room was reported in the literature. As part of the formal specification and analysis, [360] introduces novel techniques for: (i) modeling nondeterministic transmission delays while maintaining completeness and reasonable performance of the analysis; (ii) modeling clock drifts; and (iii) analyzing bounded response properties. Subsequent work by M. Sun, J. Meseguer, and L. Sha in [439] has focused on the development of *patterns* for interoperation of medical devices (among themselves and with a patient) that are *safe by construction*, and *generic*, so that they can be instantiated for many different devices. Specifically, one such pattern, called the *Command-Shaper* pattern, is formally specified as a parameterized Real-Time Maude module and proved correct in [439]. The key idea of the Command-Shaper is to intercept the commands from external devices (possibly including the patient), so that the patient is never placed in a medically dangerous state, including states where the patient's medical constants may be stressed for a dangerously long time. Instances of the Command-Shaper pattern include a mechanism for enforcing that a sophisticated pacemaker, which can adapt to changes in the patient's activity, will never place the patient's heart in stressful situations, and a patient-operated infusion pump for morphine. As already pointed out for the Orc orchestration language in Section 7.4.3, Real-Time Maude specifications of distributed real-time systems can be easily transformed into distributed real-time implementations using Maude's socket mechanism. For the Command-Shaper pattern this has been done by Mu Sun and me in [438]. One attractive feature of this transformation is that formal specifications can be interoperated with actual physical devices in a system that emulates a final implementation.

Using the PALS pattern discussed in Section 7.4.2, Peter Ölveczky and I have specified in Real-Time Maude synchronous and asynchronous versions of an active standby avionics system [329, 330], and, using the synchronous version plus its bisimulation equivalence with the asynchronous one, have verified by model checking that it satisfies (appropriately enhanced versions of) all the informal requirements listed by the designers. This example underscores the power and usefulness of the PALS pattern, since the synchronous version had just a few hundred states and each property was model checked in less than 0.8 seconds, whereas the simplest possible asynchronous version (with no message delays) had over 3 million states.

7.5. Probabilistic Systems

Probabilisitic rewrite theories (see Section 3.10) can model a wide variety of probabilistic systems, including many cyber-physical systems. As already mentioned, both the environments in which such systems operate and the very algorithms they use are often probabilistic. Furthermore, the verification of their *quantitative properties* may be just as important as that of Boolean-valued properties such as safety requirements. For this purpose, one can use statistical model-checking methods (see Section 3.11.2) of quantitative properties expressed in a formalism such as QuaTEx (see Section 3.11.1). As the PVeStA tool demonstrates, such statistical model-checking analyses can be quite scalable (see Section 6.1.10).

Up to now, the probabilistic system applications that have been specified and analyzed using the just-mentioned methods fall into three areas: (i) DoS-resistant protocols; (ii) distributed embedded systems; and (iii) distributed stochastic hybrid systems. There are of course many other possibilities, including applications for the quite different notion of probabilistic rewriting proposed in [76, 74] and discussed in Section 3.10. Since DoS-resistant protocols have already been discussed in Section 7.3.2, I focus here on areas (ii) and (iii).

7.5.1. Distributed Embedded Systems

For many distributed embedded systems, particularly those including energy-constrained components such as hand-held devices, quality of service (QoS) properties are essential. For achieving such properties in an end-to-end manner, adaptive resource management policies across different layers of the system, such as the application, middleware, and OS layers, are needed. M. Kim, M.-O. Stehr, C. Talcott, N. Dutt, and N. Venkatasubramanian have used probabilistic rewrite theories specified in Maude, and statistical model-checking analysis of quantitative properties of such theories (using the algorithm described in [261]), to model and formally analyze various sophisticated adaptive designs of distributed embedded systems that can provide desired QoS guarantees. Their general methodology is presented in [261], where it is applied to a multi-mode multimedia case study. Furthermore, in [259] they show how these methods can be combined with direct observation of system executions to *refine the probabilistic models* of the system, and how this can be used to achieve system adaptation under timing constraints by iteratively tuning system parameters. This line of research is continued in [260], where they present a compositional method for cross-layer system optimization based on a constraint refinement technique which can be used to fine tune system parameters in a compositional manner, allowing coordinated interaction among sublayer optimizers to achieve cross-layer optimization. Experiments on a realistic multimedia application demonstrate that constraint refinement can generate robust and near optimal parameter settings.

An important class of energy-constrained distributed embedded systems is that of wireless sensor networks, since the power of the sensors must be used very carefully to ensure an acceptable network lifetime. In [258], Michael Katelman, the late Jennifer Hou and I used probabilistic rewrite theories and qualitative analysis in VeStA to study in depth and under realistic conditions the design of the local minimum spanning tree (LMST) topology control protocol, which tries to maintain connectivity in an ad-hoc wireless sensor network while minimizing power consumption and maximizing data bandwidth. Our starting point was an idealized LMST design with perfect clocks and perfect communication, which did in fact maintain connectivity at an abstract level. However, our formal analysis revealed that, as soon as more realistic implementation details such as clock synchronization and network contention were introduced, the idealized LMST design failed rather badly to maintain network connectivity. The problem we then addressed was how to use probabilistic modeling and statistical model checking to *redesign* LMST at a realistic level, so that it would meet its intended goals. For this purpose we developed a system redesign methodology supporting three mutually-reinforcing tasks: (i) to uncover flaws in a given design; (ii) to conjecture the causes of the various malfunctions and to confirm such conjectures by means of statistical correlations between further analyses; and (iii) to then use the confirmed conjectures of the hypothesized causes of flaws to redesign a system and verify by statistical model checking that the final design satisfies the desired requirements. Our application of this methodology to LMST resulted in a new, implementable design that satisfied all the desired requirements under realistic operating conditions.

7.5.2. Distributed Stochastic Hybrid Systems

Stochastic hybrid systems generalize ordinary hybrid systems by allowing continuous evolution to be governed by stochastic differential equations (SDEs) and/or by allowing instantaneous changes in system modes to be probabilistic. This fits well the intrinsic uncertainty of the environments in which many hybrid systems must operate, and is also very useful when some of the systems algorithms are probabilistic. Indeed, there is a wide range of application areas, including communication networks, air traffic control, economics, fault-tolerant control, and bioinformatics. However, in practice many stochastic hybrid systems are not autonomous: they are *distributed* as collections of objects that communicate with other objects by exchanging messages through an asynchronous medium such as a network. In [336], Raman Sharikin and I used probabilistic rewrite theories to investigate several open issues such as: (i) how to compositionally specify distributed object-based stochastic hybrid systems; (ii) how to formally model them, and (iii) how to verify their properties. Specifically, in [336] we addressed these issues by: (i) defining a mathematical model for such systems; (ii) proposing a formal specification language in which system transitions are specified in a modular way by probabilistic rewrite rules; and (iii) showing how these systems can be subjected to statistical model-checking analysis to verify their probabilistic temporal logic properties. Maude and VeStA were used to illustrate the approach with specific examples such as: (i) an international auction system in which bidders reside in different countries and their different currencies fluctuate according to an SDE; and (ii) a system consisting of N rooms, each equipped with a thermostat, plus a central server unit controlling them, where each thermostat can be in either heating, cooling, or idle mode, and the temperature in each room changes randomly according to an SDE.

In Section 7.6.1 I discuss another very useful application of probabilistic rewriting to the modeling of biological systems as stochastic hybrid systems [2].

7.6. Bioinformatics, Chemical Systems, and Membranes

I discuss here several related research strands where rewriting logic has been applied to bioinformatics, to modeling the dynamics of chemical systems, and to chemically and biologically inspired membrane systems.

7.6.1. Bioinformatics

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 (QED) 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 biologists' 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.

As first pointed out in [173], and vigorously developed in the subsequent Pathway Logic research which I discuss later, 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 $A\,B \to C\,D$ is *exactly* a rewriting notation. In this way we can develop *symbolic bioinformatics models* which we can then analyze in their dynamic behavior just as we would analyze any other rewrite theory.

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 .
  ops 14-3-3 cdc37 GTP Hsp90 Raf1 Ras : -> Protein .
```

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 phosphorilated 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 .
  op none : -> ModSet .
  op __ : ModSet ModSet -> ModSet [assoc comm id: none] .
  op __ : AminoAcid MachineInt -> Site .
  op [_\_] : Protein ModSet -> Protein [right id: none] .
```

Proteins can stick together to form *complexes*. This can be modeled by the following subsort and operator declarations

```
  sort Complex .
  subsort Protein < Complex .
  op _:_ : Complex Complex -> Complex [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 Complex < Soup .
  op __ : Soup Soup -> Soup [assoc comm] .
  op cell{_{_}} : Soup Soup -> Soup .
  op nucl{_{_}} : Soup Soup -> Soup .
```

that is, soups are made up out of complexes, including 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 then a *structured soup*, composed by the above `cell` operator out of two subsoups, namely the soup in the membrane, and that inside the membrane; but this second soup is itself also structured by the cytoplasm and the nucleus. Finally, the nucleus itself is made up of two soups, namely that in the nucleus membrane, and that inside the nucleus, which are composed using the above `nucl` operator. Then, the following expression gives a partial description of a cell:

```
cell{cm  (Ras : GTP) {cyto
      (([Raf1 \ phos(S 259)phos(S 621)] : (cdc37 : Hsp90)) : 14-3-3)
                                                          nucl{nm{n}}}}
```

where `cm` denotes the rest of the soup in the cell membrane, `cyto` 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 [271]:

> Raf-1 resides in the cytosol, tied into an inactive state by the binding of a 14-3-3 dimer to phosphoserine-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:

```
 rl[10]: {CM  (Ras : GTP) {CY
 (([Raf1 \ phos(S 259)phos(S 621)] : (cdc37 : Hsp90)) : 14-3-3) }}
=>
 {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 chains 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*.

Since the first paper in this direction [173], on which the above summary is based, this line of research has been vigorously advanced by the Pathway Logic (PL) team of computer scientists and molecular biologists at SRI led by Carolyn Talcott [174, 448, 446, 447, 443, 442, 2, 455, 445, 454] (for a good overview, see Talcott's tutorial [445]). The PL researchers have used rewriting logic to develop sophisticated analyses of cell behavior in biological pathways, and have built useful notations and visualization tools, such as the Pathway Logic Assistant [446], that can represent the Maude-based analyses in forms more familiar to biologists. The papers [448, 447] contain good discussions 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 show how cell behavior can be modeled with rewrite rules and can be analyzed *at different levels of abstraction*, and even across such levels. A very exciting more recent development is the use of several probabilistic rewriting methods to model cell behaviors as stochastic hybrid systems [2]. Yet another very exciting development is the use of rewriting logic in *neuroinformatics*, at a much higher level of abstraction than that of reactions in molecular biology. What are now modeled are neural systems, with neurons as objects, in the object-oriented sense, plus what might be called "wiring information" about neuron interconnections. Changes in neuron states due to firings are then described by rewrite rules. A Maude model of the neural system responsible for the feeding behavior of the marine mollusk *Aplysia* has been used to model quite accurately Aplysia's neural behavior in a way consistent with other studies [2]; furthermore, using symbolic model checking, more ambitious properties of Aplysia's neural behavior have been verified in [454]. In general, one of the important contributions of the PL project is the combination of various modeling and analysis techniques to model biological systems; in addition to all the already-mentioned techniques, SAT-solving is yet one more weapon in PL's arsenal [455].

The PL research has stimulated the use of rewriting logic and Maude by other bioinformatics researchers. For example, M.G. Sriram has used Maude to model protein functional domains in signal transduction, and to obtain testable hypotheses at various levels of abstraction [426], and, my UIUC colleague Thomas Anastasio has used Maude to analyze and obtain useful hypotheses about biological pathways whose malfunction is related to Alzheimer's disease [27].

Although the research by O. Andrei and H. Kirchner in [30] makes also valuable contributions to the bioinformatics applications of rewriting logic, I discuss it in the next section because of its similarities with other work on chemical systems.

7.6.2. Chemical Systems

The already-mentioned fact that the chemical notation for a reaction like $A\,B \to C\,D$ is a rewriting notation suggests that rewrite theories can be used to symbolically model not just cell biology but *any* chemical systems, with the reactions modeled as rewrite rules. This is exactly the research approach taken by O. Bournez et al. in [73], and further developed by O. Bournez, L. Ibanescu and H. Kirchner in [75], and by O. Andrei, L. Ibanescu and H. Kirchner in [29]. This research makes a number of novel contributions. First of all, it emphasizes the fact that chemical compounds are *graphs*, so that chemical reactions can be more properly modeled as graph rewrite rules. Second, it identifies an appropriate *term representation* for chemical graphs so that: (i) equivalent representations can be effectively identified; (ii) "soups" of different chemical compounds can be represented as multisets by an AC operator; and (iii) the graph rewriting modeling of chemical reactions can be faithfully represented as term rewriting modulo AC. In particular, the paper [75] provides a detailed study of this dual graph/term representation and proves the faithfulness of the associated term rewriting in capturing the desired graph rewriting. A third contribution is the use of *strategies* to characterize chemical processes, which do not correspond to arbitrary sequences of rewrites, but have to obey certain dynamic constraints. A fourth contribution is the implementation of all these ideas in the *GasEl* system, first implemented in ELAN in [73], but subsequently implemented in TOM for enhanced efficiency, as reported in [29].

The already-mentioned work by O. Andrei and H. Kirchner in [30], although belonging to the more specific area of biochemistry and bioinformatics applications—indeed, to the modeling of biochemical networks—has some similarities with the just-mentioned work on chemical modeling, but makes different contributions. It models the molecular complexes appearing in cell biology as *labeled multigraphs with ports*, with molecules represented as nodes, sites as ports, and bonds as edges. Biochemical reactions are then modeled as graph transformation rules and biochemical networks are finally modeled as strategies which express the appropriate control between the different reactions and the dynamic evolution of molecular complexes. In analogy with [75], careful attention is paid to finding a faithful *term representation*, that is, a faithful representation as an (order-sorted) rewrite theory of the corresponding graphs and graph transformation rules associated to a given biochemical network. A biochemical calculus where rules and strategies are port graphs has been defined and applied to autonomic computing in [31].

7.6.3. Membrane Systems

Transfer of ideas can sometimes go in both directions. Not only can rewriting logic provide formal models for cell biology and bioinformatics, but *chemical* *and biological metaphors* may suggest models of computation. Indeed, chemical metaphors understood as multiset rewriting—so that a multiset of entities is visualized as a chemical "soup," and atomic computation steps as chemical reactions—go back to the *Gamma* model of computation of J.-P. Banâtre and D. Le Mètayer [47], which inspired the Chemical Abstract Machine (CHAM) of G. Berry and G. Boudol [55]. A further development of this line of research has been the study of *membrane systems* in the sense of O. Andrei, G. Ciobanu, and D. Lucanu [28], who base their ideas on the cell-inspired proposal of membrane computing by Gh. Paun [382]. The basic idea is that membrane systems are hierarchical systems consisting of nested cells, each surrounded by a membrane enclosing a multiset of elements, which may include other cells. This bears some similarities to the Meseguer-Talcott "Russian dolls" model of distributed object-oriented reflection [338] already mentioned in Section 7.2.4. Another important idea is that rules describing local changes in a membrane system have *priorities*, and that *maximal parallelism* is the desired model of computation. A careful study of all these issues within the rewriting logic framework has been presented in [28]. The issue of maximal parallelism using the idea of "promoters and inhibitors" is further studied by O. Agrigoraei and G. Ciobanu in [6]. Of course, since rules in membrane systems have priorities and should fire with maximal parallelism, not all rewriting computations are desirable; this leads to the issue of characterizing membrane computations by appropriate rewriting strategies, a topic studied by O. Andrei and D. Lucanu in [32], and by D. Lucanu in [288].

8. Some Future Research Directions

Of course, all the research areas already discussed are promising future directions. The question is rather, which new or recent areas seem most in need of development and look particularly promising? Answers to such questions are necessarily subjective, and can only be *guesses*. In fact, the emergence of other areas which one has not anticipated should be a cause for rejoicement. With that said, here are some directions I think need development and are promising:

1. *Rewriting logic as a new paradigm for declarative concurrent programming*, as well as new multicore and distributed rewriting logic language implementations. Everybody agrees that concurrent and distributed programming are at present quite difficult and messy. What most people fail to realize is that this is not an *intrinsic* necessity: programming concurrent systems in a declarative way can be *simpler* than programming a sequential system in a conventional, imperative way. At the sequential implementation level, the great simplicity of rewrite rules as a programming paradigm has been amply demonstrated; what now is needed is to develop efficient concurrent implementations of rewriting languages that show in practice their intrinsic superiority over conventional concurrent programming languages.

2. *Advancing the rewriting logic semantics project*, including future advances in K, matching logic, and compiler generation from language definitions. The advances in this area have already been quite impressive: it has already been shown that this approach can scale up to produce full executable semantics for entire languages like C or Java, and that a wide range of *semantics-based tools* can then be derived from such formal definitions. But more ambitious goals lie ahead such as, for example: (i) language-generic program verifiers; (ii) language-generic static analysis tools; (iii) more efficient language-generic model checkers; and (iv) efficient language-generic compilers; where in all such cases those meta-tools would be instantiated to specific languages by providing a rewriting logic definition of the given language.

3. *Embedded and cyber-physical systems*, including safety verification and correct-by-construction code generation. Further research in formal patterns such as PALS that can greatly simplify the design and verification of safety properties for cyber-physical systems seems very promising to tame the many complexities involved. New formal verification methods are also needed. But this still leaves open the additional challenge of deriving correct-by-construction real-time implementations from formal rewriting logic specifications.

4. *Deductive and symbolic verification methods for rewrite theories*, including narrowing-based methods, their combination with SMT solving, deductive temporal logic verification, and inductive proof methods. Symbolic methods can bring theorem proving and model checking verification so close to each other that it will be difficult to classify some tools as either model checkers or theorem provers. Furthermore, they can be naturally combined with temporal logic and inductive reasoning. New proof techniques, new algorithms, and new tool implementations are needed to make all this happen. The great advantage of developing them for suitable classes of rewrite theories is that they will be highly generic, so that they can be amortized over many different instance languages and application domains.

5. *New verification methods and tools for probabilistic rewrite theories*, including languages, verification methods, and tools. This area is still relatively undeveloped, yet quite promising advances have already been made. A PMaude implementation should be developed in the near future. New, probabilisitc model checkers complementing the already exisiting statistical model checkers should also be developed. And a more intimate integration between probabilisitc and real-time systems, including stochastic hybrid systems, should be sought.

9. Conclusions

In the introduction I raised the following questions about rewriting logic:

- How well-developed are its mathematical foundations?
- To what extent have its goals as a semantic framework for concurrency, and as a logical framework, been achieved?
- Which languages and tools supporting rewriting logic programming, specification, and verification have been developed?
- In which application areas has it been shown useful?
- What do its future prospects look like?

I believe that I have given quite extensive answers to all these questions, except perhaps for a briefer treatment of the last one on future prospects. The foundations are in my mind rock-solid. At this point the wide range of models of concurrency and of logics that have been *naturally* expressed within the rewriting logic framework provides overwhelming evidence that it is a very suitable framework. The languages supporting rewriting logic are mature, provide many features, and are furthermore still growing. The spectrum of formal tools is quite adequate, although more advances are and will be happening. And the range of applications is quide wide and exciting. I think some of us will be busy pushing the envelope for years to come; and I hope this survey will encourage other researchers to use rewriting logic in their own work and to make new contributions.

Acknowledgements. I thank the organizers of WRLA 2010 for giving me the opportunity and the stimulus to do some reflecting, surveying, and guessing about rewriting logic at this point, when twenty years have passed since the first papers were published, for their further encouragement to turn my WRLA lecture into a survey paper, and for their patience in allowing me the necessary time to finish it. As already mentioned, I feel a debt of gratitude to the many gifted researchers who have made important contributions to the rewriting logic research program. I have benefitted from very helpful comments to improve the exposition by the referees and by Francisco Durán, Santiago Escobar, Maribel Fernández, Kokichi Futatsugi, Alwyn Goodloe, Hélène Kirchner, Alberto Lluch, Narciso Martí-Oliet, Peter Ölveczky, Miguel Palomino, Camilo Rocha, Carolyn Talcott, Cesare Tinelli and Alberto Verdejo (I apologize if I inadvertedly omitted anybody). This work has been supported in part by NSF Grants CNS 07-16638, CNS 08-34709, CNS 08-31064, CNS 09-04749, and CCF 09-05584, AFOSR Grant FA8750-11-2-0084, and the "Programa de Apoyo a la Investigación y Desarrollo" (PAID-02-11) of the Universitat Politècnica de València.

References

[1] A. Aziz, V. Singhal, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. It usually works: The temporal logic of stochastic systems. In P. Wolper, editor, *7th International Conference On Computer Aided Verification*, volume 939, pages 155–165, Liege, Belgium, 1995. Springer Verlag.

[2] A. Abate, Y. Bai, N. Sznajder, C. L. Talcott, and A. Tiwari. Quantitative and probabilistic modeling in pathway logic. In M. M. Zhu, Y. Zhang, H. R. Arabnia, and Y. Deng, editors, *Proceedings of the 7th IEEE International Conference on Bioinformatics and Bioengineering, BIBE 2007, Harvard Medical School, Boston, MA, USA, October 14-17, 2007*, pages 922–929. IEEE, 2007.

[3] G. Agha. *Actors.* MIT Press, 1986.

[4] G. A. Agha, M. Greenwald, C. A. Gunter, S. Khanna, J. Meseguer, K. Sen, and P. Thati. Formal modeling and analysis of DoS using probabilistic rewrite theories. In A. Sabelfeld, editor, *Proceedings of the Workshop on Foundations of Computer Security, FCS'05, (Affiliated with LICS'05), Chicago, IL, June 30-July 1, 2005*, pages 91–102, 2005.

[5] G. A. Agha, J. Meseguer, and K. Sen. PMaude: Rewrite-based specification language for probabilistic object systems. In A. Cerone and H. Wiklicky, editors, *Proceedings of the Third Workshop on Quantitative Aspects of Programming Languages, QAPL 2005, Edinburgh, UK, April 2-3, 2005*, volume 153(2) of *Electronic Notes in Theoretical Computer Science*, pages 213–239. Elsevier, 2006.

[6] O. Agrigoroaiei and G. Ciobanu. Rewriting logic specification of membrane systems with promoters and inhibitors. In Roşu [403], pages 5–22.

[7] 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 Computer Science*, pages 412–426. Springer, 2005.

[8] B. Alarcón, R. Gutiérrez, and S. Lucas. Context-sensitive dependency pairs. *Inf. Comput.*, 208(8):922–968, 2010.

[9] B. Alarcón, R. Gutiérrez, S. Lucas, and R. Navarro-Marset. Proving termination properties with MU-TERM. In Johnson and Pavlovic [246], pages 201–208.

[10] B. Alarcón, S. Lucas, and J. Meseguer. A dependency pair framework for $A \vee C$-termination. In Ölveczky [362], pages 35–51.

[11] M. Alba-Castro, M. Alpuente, and S. Escobar. Abstract certification of global non-interference in rewriting logic. In M. Leuschel, S. Hallerstede, F. de Boer, and M. Bonsangue, editors, *Formal Methods for Components and Objects, 8th International Symposium, FMCO 2009, Eindhoven, The Netherlands, November 4-6, 2009, Revised Selected Papers*, volume 6286 of *Lecture Notes in Computer Science*, pages 105–124. Springer, 2010.

[12] M. Alba-Castro, M. Alpuente, and S. Escobar. Approximating non-interference and erasure in rewriting logic. In T. Ida, editor, *Proceedings of the 12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2010, Timisoara, Romania, September 23-26, 2010*, pages 124–132. IEEE Computer Society, 2010.

[13] A. Albarrán, F. Durán, and A. Vallecillo. From Maude specifications to SOAP distributed implementations: A smooth transition. In O. Díaz, A. Illarramendi, and M. Piattini, editors, *Actas de las VI Jornadas de Ingeniería del Software y Bases de Datos, JISBD 2001, Almagro (Ciudad Real), España, Noviembre 21-23, 2001*, pages 419–434, 2001.

[14] A. Albarrán, F. Durán, and A. Vallecillo. Maude meets CORBA. In G. Fernandez and C. Pons, editors, *Proceedings of the Second Argentine Symposium on Software Engineering, ASSE 2001, Buenos Aires, Argentina, September 10-11, 2001*, 2001.

[15] M. Alpuente, D. Ballis, and D. Romero. Specification and verification of web applications in rewriting logic. In A. Cavalcanti and D. Dams, editors, *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, volume 5850 of *Lecture Notes in Computer Science*, pages 790–805. Springer, 2009.

[16] M. Alpuente, S. Escobar, J. Meseguer, and P. Ojeda. A modular equational generalization algorithm. In M. Hanus, editor, *Logic-Based Program Synthesis and Transformation, 18th International Symposium, LOPSTR 2008, Valencia, Spain, July 17-18, 2008, Revised Selected Papers*, volume 5438 of *Lecture Notes in Computer Science*, pages 24–39. Springer, 2009.

[17] M. Alpuente, S. Escobar, J. Meseguer, and P. Ojeda. Order-sorted generalization. In M. Falaschi, editor, *Proceedings of the 17th International Workshop on Functional and (Constraint) Logic Programming, WFLP 2008, Siena, Italy, July 3-4, 2008*, volume 246 of *Electronic Notes in Theoretical Computer Science*, pages 27–38. Elsevier, 2009.

[18] M. AlTurki. *Rewriting-based formal modeling, analysis and implementation of real-time distributed services.* PhD thesis, University of Illinois at Urbana-Champaign, 2011. `http://hdl.handle.net/2142/26231`.

[19] M. AlTurki, D. Dhurjati, D. Yu, A. Chander, and H. Inamura. Formal specification and analysis of timing properties in software systems. In Chechik and Wirsing [90], pages 262–277.

[20] M. AlTurki and J. Meseguer. Real-time rewriting semantics of Orc. In Leuschel and Podelski [284], pages 131–142.

[21] M. AlTurki and J. Meseguer. Reduction semantics and formal analysis of Orc programs. In D. Ballis, S. Escobar, and M. Marchiori, editors, *Proceedings of the 3rd International Workshop on Automated Specification and Verification of Web Systems, WWV 2007, Venice, Italy, December 14, 2007*, volume 200(3) of *Electronic Notes in Theoretical Computer Science*, pages 25–41. Elsevier, 2008.

[22] M. AlTurki and J. Meseguer. Dist-Orc: A rewriting-based distributed implementation of Orc with formal analysis. In Ölveczky [361], pages 26–45.

[23] M. AlTurki and J. Meseguer. PVeStA: A parallel statistical model checking and quantitative analysis tool. In Corradini et al. [124], pages 386–392.

[24] M. AlTurki, J. Meseguer, and C. A. Gunter. Probabilistic modeling and analysis of DoS protection for the ASV protocol. In Dougherty and Escobar [144], pages 3–18.

[25] R. Alur, C. Courcoubetis, and D. L. Dill. Model-checking for real-time systems. In *LICS'90*, pages 414–425. IEEE, 1990.

[26] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[27] T. J. Anastasio. Data-driven modeling of Alzheimer disease pathogenesis. *Journal of Theoretical Biology*, 290:60–72, 2011.

[28] O. Andrei, G. Ciobanu, and D. Lucanu. A rewriting logic framework for operational semantics of membrane systems. *Theoretical Computer Science*, 373(3):163–181, 2007.

[29] O. Andrei, L. Ibănescu, and H. Kirchner. Non-intrusive formal methods and strategic rewriting for a chemical application. In K. Futatsugi, J.-P. Jouannaud, and J. Meseguer, editors, *Algebra, Meaning, and Computation, Essays Dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday*, volume 4060 of *Lecture Notes in Computer Science*, pages 194–215. Springer, 2006.

[30] O. Andrei and H. Kirchner. Graph rewriting and strategies for modeling biochemical networks. In V. Negru, T. Jebelean, D. Petcu, and D. Zaharie, editors, *Proceedings of the Ninth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2007, Timisoara, Romania, September 26-29, 2007*, pages 407–414. IEEE Computer Society, 2007.

[31] O. Andrei and H. Kirchner. A port graph calculus for autonomic computing and invariant verification. *Electr. Notes Theor. Comput. Sci.*, 253(4):17–38, 2009.

[32] O. Andrei and D. Lucanu. Strategy-based proof calculus for membrane systems. In Roşu [403], pages 23–43.

[33] N. Aoumeur and G. Saake. Integrating and rapid-prototyping UML structural and behavioural diagrams using rewriting logic. In A. B. Pidduck, J. Mylopoulos, C. C. Woo, and M. T. Özsu, editors, *Advanced Information Systems Engineering, 14th International Conference, CAiSE 2002, Toronto, Canada, May 27-31, 2002, Proceedings*, volume 2348 of *Lecture Notes in Computer Science*, pages 296–310. Springer, 2002.

[34] A. Avron, F. Honsell, I. A. Mason, and R. Pollack. Using typed lambda calculus to implement formal systems on a machine. *Journal of Automated Reasoning*, 9(3):309–354, December 1992.

[35] 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*. Springer, 2007.

[36] M. Backes and P. Ning, editors. *Computer Security - ESORICS 2009, 14th European Symposium on Research in Computer Security, Saint-Malo, France, September 21-23, 2009. Proceedings*, volume 5789 of *Lecture Notes in Computer Science*. Springer, 2009.

[37] K. Bae and J. Meseguer. A rewriting-based model checker for the temporal logic of rewriting. In Kniesel and Pinto [270], pages 46–60.

[38] K. Bae and J. Meseguer. The linear temporal logic of rewriting Maude model checker. In Ölveczky [362], pages 208–225.

[39] K. Bae and J. Meseguer. State/event-based LTL model checking under parametric generalized fairness. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 132–148. Springer, 2011.

[40] K. Bae and P. C. Ölveczky. Extending the Real-Time Maude semantics of Ptolemy to hierarchical DE models. In Ölveczky [361], pages 46–66.

[41] K. Bae, P. C. Ölveczky, A. Al-Nayeem, and J. Meseguer. Synchronous AADL and its formal analysis in Real-Time Maude. Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign, 2011.

[42] K. Bae, P. C. Ölveczky, T. H. Feng, and S. Tripakis. Verifying Ptolemy II discrete-event models using Real-Time Maude. In K. Breitman and A. Cavalcanti, editors, *Formal Methods and Software Engineering, 11th International Conference on Formal Engineering Methods, ICFEM 2009, Rio de Janeiro, Brazil, December 9-12, 2009. Proceedings*, volume 5885 of *Lecture Notes in Computer Science*, pages 717–736. Springer, 2009.

[43] C. Baier, J.-P. Katoen, and H. Hermanns. Approximate symbolic model checking of continuous-time markov chains. In *CONCUR'99*, volume 1664 of *LNCS*, pages 146–161. Springer, 1999.

[44] H. Baker and C. Hewitt. Laws for communicating parallel processes. In *Proceedings of the 1977 IFIP Congress*, pages 987–992. IFIP Press, 1977.

[45] P. Baldan, C. Bertolissi, H. Cirstea, and C. Kirchner. A rewriting calculus for cyclic higher-order term graphs. *Mathematical Structures in Computer Science*, 17(3):363–406, 2007.

[46] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. Tom: Piggybacking rewriting on Java. In Baader [35], pages 36–47.

[47] J.-P. Banâtre and D. L. Mètayer. The Gamma model and its discipline of programming. *Science of Computer Programming*, 15:55–77, 1990.

[48] S. Barker and M. Fernández. Term rewriting for access control. In E. Damiani and P. Liu, editors, *DBSec*, volume 4127 of *Lecture Notes in Computer Science*, pages 179–193. Springer, 2006.

[49] G. Barthe and F. S. de Boer, editors. *Formal Methods for Open Object-Based Distributed Systems, 10th IFIP WG 6.1 International Conference, FMOODS 2008, Oslo, Norway, June 4-6, 2008, Proceedings*, volume 5051 of *Lecture Notes in Computer Science*. Springer, 2008.

[50] D. Basin, M. Clavel, and J. Meseguer. Reflective metalogical frameworks. *ACM Transactions on Computational Logic*, 5(3):528–576, 2004.

[51] D. A. Basin and R. L. Constable. Metalogical frameworks. In G. Huet and G. Plotkin, editors, *Logical Environments*, pages 1–29. Cambridge University Press, 1993.

[52] E. Beffara, O. Bournez, H. Kacem, and C. Kirchner. Verification of timed automata using rewrite rules and strategies. In N. Dershowitz and A. Frank, editors, *Proceedings of the Seventh Biennial Bar-Ilan International Symposium on the Foundations of Artificial Intelligence, BISFAI 2001, Ramat-Gan, Israel, June 25-27, 2001*. Computing Research Repository (CoRR), 2001.

[53] S. Berardi. Towards a mathematical analysis of the Coquand-Huet calculus of constructions and other systems in Barendregt's cube. Technical Report, Carnegie-Mellon University and Università di Torino, 1988.

[54] 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, Seventh Colloquium*, pages 76–90. Springer-Verlag, 1980. LNCS, Volume 81.

[55] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, 1992.

[56] C. Bertolissi, H. Cirstea, and C. Kirchner. Translating combinatory reduction systems into the rewriting calculus. In J.-L. Giavitto and P.-E. Moreau, editors, *Proceedings of the 4th International Workshop on Rule-Based Programming, RULE 2003, Valencia, Spain, June 9, 2003*, volume 86(2) of *Electronic Notes in Theoretical Computer Science*, pages 28–44. Elsevier, 2003.

[57] E. Best and R. Devillers. Sequential and concurrent behavior in Petri net theory. *Theoretical Computer Science*, 55:87–136, 1989.

[58] J. Bjørk, E. B. Johnsen, O. Owe, and R. Schlatte. Lightweight time modeling in timed Creol. In Ölveczky [361], pages 67–81.

[59] M. M. Bonsangue and E. B. Johnsen, editors. *Formal Methods for Open Object-Based Distributed Systems, 9th IFIP WG 6.1 International Conference, FMOODS 2007, Paphos, Cyprus, June 6-8, 2007, Proceedings*, volume 4468 of *Lecture Notes in Computer Science*. Springer, 2007.

[60] A. Boronat. *MOMENT: A Formal Framework for MOdel ManageMENT*. PhD thesis, Universitat Politècnica de València, Spain, 2007.

[61] A. Boronat, J. A. Carsí, and I. Ramos. Automatic reengineering in MDA using rewriting logic as transformation engine. In N. Gold and T. Systä, editors, *Proceedings of the 9th European Conference on Software Maintenance and Reengineering, CSMR 2005, Manchester, UK, March 21-23, 2005, Proceedings*, pages 228–231. IEEE Computer Society, 2005.

[62] A. Boronat, R. Heckel, and J. Meseguer. Rewriting logic semantics and verification of model transformations. In Chechik and Wirsing [90], pages 18–33.

[63] A. Boronat, A. Knapp, J. Meseguer, and M. Wirsing. What is a multi-modeling language? In Corradini and Montanari [125], pages 71–87.

[64] A. Boronat and J. Meseguer. Algebraic semantics of OCL-constrained metamodel specifications. In M. Oriol and B. Meyer, editors, *Objects, Components, Models and Patterns, 47th International Conference, TOOLS EUROPE 2009, Zurich, Switzerland, June 29-July 3, 2009. Proceedings*, volume 33 of *Lecture Notes in Business Information Processing*, pages 96–115. Springer, 2009.

[65] A. Boronat and J. Meseguer. MOMENT2: EMF model transformations in Maude. In A. Vallecillo and G. Sagardui, editors, *Actas de las XIV Jornadas de Ingeniería del Software y Bases de Datos, JISBD 2009, San Sebastián, España, Septiembre 8-11, 2009*, pages 178–179, 2009.

[66] A. Boronat and J. Meseguer. An algebraic semantics for MOF. *Formal Aspects of Computing*, 22(3-4):269–296, 2010.

[67] A. Boronat and P. C. Ölveczky. Formal real-time model transformations in MOMENT2. In D. S. Rosenblum and G. Taentzer, editors, *Fundamental Approaches to Software Engineering, 13th International Conference, FASE 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6013 of *Lecture Notes in Computer Science*, pages 29–43. Springer, 2010.

[68] P. Borovanský and C. Castro. Cooperation of constraint solvers: Using the new process control facilities of ELAN. In Kirchner and Kirchner [262], pages 1–20.

[69] P. Borovanský, H. Cirstea, H. Dubois, C. Kirchner, H. Kirchner, P.-E. Moreau, Q.-H. Nguyen, C. Ringeissen, and M. Vittek. ELAN v 3.6 user manual. Technical report, INRIA Lorraine & LORIA, Nancy, France, Feb. 2004.

[70] 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.

[71] P. Borovanský, C. Kirchner, H. Kirchner, and C. Ringeissen. Rewriting with strategies in ELAN: A functional semantics. *International Journal of Foundations of Computer Science*, 12(1):69–95, 2001.

[72] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236(1-2):35–132, 2000.

[73] O. Bournez, G.-M. Côme, V. Conraud, H. Kirchner, and L. Ibănescu. A rule-based approach for automated generation of kinetic chemical mechanisms. In Nieuwenhuis [355], pages 30–45.

[74] O. Bournez and M. Hoyrup. Rewriting logic and probabilities. In Nieuwenhuis [355], pages 61–75.

[75] O. Bournez, L. Ibănescu, and H. Kirchner. From chemical rules to term rewriting. In H. Cirstea and N. Martí-Oliet, editors, *Proceedings of the 6th International Workshop on Rule-Based Programming, RULE 2005, Nara, Japan, April 23, 2005*, volume 147(1) of *Electronic Notes in Theoretical Computer Science*, pages 113–134. Elsevier, 2006.

[76] O. Bournez and C. Kirchner. Probabilistic rewrite strategies. Applications to ELAN. In S. Tison, editor, *Rewriting Techniques and Applications, 13th International Conference, RTA 2002, Copenhagen, Denmark, July 22-24, 2002, Proceedings*, volume 2378 of *Lecture Notes in Computer Science*, pages 252–266. Springer, 2002.

[77] C. Braga and J. Meseguer. Modular rewriting semantics in practice. In Martí-Oliet [297], pages 393–416.

[78] R. Bruni. *Tile Logic for Synchronized Rewriting of Concurrent Systems*. PhD thesis, Dipartimento di Informatica, Università di Pisa, 1999. Technical Report TD-1/99.

[79] 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.

[80] R. Bruni and J. Meseguer. Semantic foundations for generalized rewrite theories. *Theoretical Computer Science*, 360(1-3):386–414, 2006.

[81] R. Bruni, J. Meseguer, and U. Montanari. Symmetric monoidal and cartesian double categories as a semantic framework for tile logic. *Mathematical Structures in Computer Science*, 12(1):53–90, 2002.

[82] R. Bruni, J. Meseguer, and U. Montanari. Tiling transactions in rewriting logic. In Gadducci and Montanari [205], pages 90–109.

[83] R. Bruni, U. Montanari, and J. Meseguer. Internal strategies in a rewriting implementation of tile systems. In Kirchner and Kirchner [262], pages 263–284.

[84] G. Carabetta, P. Degano, and F. Gadducci. CCS semantics via proved transition systems and rewriting logic. In Kirchner and Kirchner [262], pages 369–387.

[85] M. Casadei, L. Gardelli, and M. Viroli. Simulating emergent properties of coordination in Maude: the collective sort case. In C. Canal and M. Viroli, editors, *Proceedings of the Fifth International Workshop on the Foundations of Coordination Languages and Software Architectures, FOCLASA 2006, Bonn, Germany, August 31, 2006*, volume 175(2) of *Electronic Notes in Theoretical Computer Science*, pages 59–80. Elsevier, 2007.

[86] M. Casadei, A. Omicini, and M. Viroli. Prototyping A&A ReSpecT in Maude. In C. Canal, P. Poizat, and M. Viroli, editors, *Proceedings of the 6th International Workshop on the Foundations of Coordination Languages and Software Architectures, FOCLASA 2007, Lisbon, Portugal, September 8, 2007*, volume 194 of *Electronic Notes in Theoretical Computer Science*, pages 93–109. Elsevier, 2008.

[87] 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.

[88] R. Chadha, C. A. Gunter, J. Meseguer, R. Shankesi, and M. Viswanathan. Modular preservation of safety properties by cookie-based DoS-protection wrappers. In Barthe and de Boer [49], pages 39–58.

[89] F. Chalub and C. Braga. Maude MSOS tool. In Denker and Talcott [139], pages 133–146.

[90] M. Chechik and M. Wirsing, editors. *Fundamental Approaches to Software Engineering, 12th International Conference, FASE 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5503 of *Lecture Notes in Computer Science*. Springer, 2009.

[91] F. Chen, G. Roşu, and R. P. Venkatesan. Rule-based analysis of dimensional safety. In Nieuwenhuis [355], pages 197–207.

[92] S. Chen, J. Meseguer, R. Sasse, H. J. Wang, and Y.-M. Wang. A systematic approach to uncover security flaws in GUI logic. In B. Pfitzmann and P. McDaniel, editors, *Proceedings of the 2007 IEEE Symposium on Security and Privacy (S&P 2007), Oakland, California, USA, May 20-23, 2007*, pages 71–85. IEEE Computer Society, 2007.

[93] S. Chen, K. Pattabiraman, Z. Kalbarczyk, and R. K. Iyer. Formal reasoning of various categories of widely exploited security vulnerabilities by pointer taintedness semantics. In Y. Deswarte, F. Cuppens, S. Jajodia, and L. Wang, editors, *19th International Information Security Conference, SEC 2004, Toulouse, France, August 22-27, 2004, Proceedings*, pages 83–100. Kluwer, 2004.

[94] H. Cirstea and C. Kirchner. Theorem proving using computational systems: The case of the B predicate prover. Presented at *CCL'97 Workshop*, Schloss Dagstuhl, Germany, Sept. 1997.

[95] 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*, Research Studies, pages 95–120. Wiley, 1999.

[96] H. Cirstea and C. Kirchner. The rewriting calculus – Part I. *Logic Journal of the IGPL*, 9(3):363–399, 2001.

[97] H. Cirstea and C. Kirchner. The rewriting calculus – Part II. *Logic Journal of the IGPL*, 9(3):401–434, 2001.

[98] 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 the Joint European Conferences on Theory and Practice of Software, ETAPS 2001, Genova, Italy, April 2-6, 2001, Proceedings*, volume 2030 of *Lecture Notes in Computer Science*, pages 168–183. Springer, 2001.

[99] H. Cirstea, C. Kirchner, and L. Liquori. Rewriting calculus with(out) types. In Gadducci and Montanari [205], pages 3–19.

[100] H. Cirstea, C. Kirchner, L. Liquori, and B. Wack. Rewrite strategies in the rewriting calculus. In B. Gramlich and S. Lucas, editors, *Proceedings of the 3rd International Workshop on Reduction Strategies in Rewriting and Programming, WRS 2003, Valencia, Spain, June 8, 2003*, volume 86(4) of *Electronic Notes in Theoretical Computer Science*, pages 593–624. Elsevier, 2003.

[101] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2001.

[102] M. Clavel. *Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications*. CSLI Publications, 2000.

[103] M. Clavel, F. Durán, S. Eker, S. Escobar, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. Unification and narrowing in Maude 2.4. In R. Treinen, editor, *Rewriting Techniques and Applications, 20th International Conference, RTA 2009, Brasília, Brazil, June 29-July 1, 2009, Proceedings*, volume 5595 of *Lecture Notes in Computer Science*, pages 380–390. Springer, 2009.

[104] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, and J. Meseguer. Metalevel computation in Maude. In Kirchner and Kirchner [262], pages 331–352.

[105] 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.

[106] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[107] M. Clavel, F. Durán, S. Eker, and J. Meseguer. Building equational proving tools by reflection in rewriting logic. In Futatsugi et al. [203], pages 1–31.

[108] M. Clavel, F. Durán, S. Eker, J. Meseguer, and M.-O. Stehr. Maude as a formal meta-tool. In Wing et al. [471], pages 1684–1703.

[109] M. Clavel, F. Durán, and N. Martí-Oliet. Polytypic programming in Maude. In Futatsugi [200], pages 339–360.

[110] M. Clavel and M. Egea. ITP/OCL: A rewriting-based validation tool for UML+OCL static class diagrams. In Johnson and Vene [247], pages 368–373.

[111] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In Meseguer [317], pages 65–89.

[112] M. Clavel, N. Martí-Oliet, and M. Palomino. Formalizing and proving semantic relations between specifications by reflection. In Rattray et al. [389], pages 72–86.

[113] M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. In Meseguer [317], pages 126–148.

[114] M. Clavel and J. Meseguer. Internal strategies in a reflective logic. In B. Gramlich and H. Kirchner, editors, *Proceedings of the CADE-14 Workshop on Strategies in Automated Deduction*, pages 1–12, Townsville, Australia, 1997.

[115] M. Clavel and J. Meseguer. Reflection in conditional rewriting logic. *Theoretical Computer Science*, 285(2):245–288, 2002.

[116] M. Clavel, J. Meseguer, and M. Palomino. Reflection in membership equational logic, many-sorted equational logic, Horn logic with equality, and rewriting logic. *Theoretical Computer Science*, 373(1-2):70–91, 2007.

[117] M. Clavel, M. Palomino, and A. Riesco. Introducing the ITP tool: a tutorial. *Journal of Universal Computer Science*, 12(11):1618–1650, 2006.

[118] M. Clavel and J. Santa-Cruz. ASIP + ITP: A verification tool based on algebraic semantics. In F. J. López-Fraguas, editor, *Actas de las V Jornadas sobre Programación y Lenguajes, PROLE 2005, Granada, España, Septiembre 14-16, 2005*, pages 149–158. Thomson, 2005.

[119] W. Clinger. Foundations of actor semantics. Technical report AI-TR-633, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1981.

[120] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007. Release October, 12th 2007.

[121] H. Comon-Lundth and S. Delaune. The finite variant property: how to get rid of some algebraic properties. In Proc *RTA'05*, Springer LNCS 3467, 294–307, 2005.

[122] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[123] A. Corradini, F. Gadducci, and U. Montanari. Relating two categorial models of term rewriting. In Hsiang [243], pages 225–240.

[124] A. Corradini, B. Klin, and C. Cîrstea, editors. *Algebra and Coalgebra in Computer Science - 4th International Conference, CALCO 2011, Winchester, UK, August 30 - September 2, 2011. Proceedings*, volume 6859 of *Lecture Notes in Computer Science*. Springer, 2011.

[125] A. Corradini and U. Montanari, editors. *Recent Trends in Algebraic Development Techniques, 19th International Workshop, WADT 2008, Pisa, Italy, June 13-16, 2008, Revised Selected Papers*, volume 5486 of *Lecture Notes in Computer Science*. Springer, 2009.

[126] A. S. de Oliveira. Rewriting-based access control policies. *Electr. Notes Theor. Comput. Sci.*, 171(4):59–72, 2007.

[127] A. S. de Oliveira, E. K. Wang, C. Kirchner, and H. Kirchner. Weaving rewrite-based access control policies. In P. Ning, V. Atluri, V. D. Gligor, and H. Mantel, editors, *Proceedings of the 2007 ACM workshop on Formal methods in security engineering, FMSE 2007, Fairfax, VA, USA, November 2, 2007*, pages 71–80. ACM, 2007.

[128] P. Degano, F. Gadducci, and C. Priami. A causal semantics for CCS via rewriting logic. *Theoretical Computer Science*, 275(1-2):259–282, 2002.

[129] P. Degano, J. Meseguer, and U. Montanari. Axiomatizing the algebra of net computations and processes. *Acta Informatica*, 33:641–667, 1996.

[130] P. Degano and C. Priami. Proved trees. In *Proc. ICALP'92*, pages 629–640. Springer LNCS 623, 1992.

[131] 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 of the 37th Allerton Conference on Communication, Control and Computation*, pages 738–747. University of Illinois, 1999.

[132] G. Denker, J. Meseguer, and C. L. Talcott. Protocol specification and analysis in Maude. In N. Heintze and J. Wing, editors, *Proceedings of the Workshop on Formal Methods and Security Protocols, FMSP'98, Indianapolis, Indiana, June 25, 1998*, 1998.

[133] G. Denker, J. Meseguer, and C. L. Talcott. Formal specification and analysis of active networks and communication protocols: The Maude experience. In Koob et al. [272], pages 251–265.

[134] G. Denker, J. Meseguer, and C. L. Talcott. Rewriting semantics of meta-objects and composable distributed services. In Futatsugi [200], pages 405–425.

[135] 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.

[136] G. Denker and J. Millen. CAPSL intermediate language. In N. Heintze and E. Clarke, editors, *Proceedings of the Workshop on Formal Methods and Security Protocols, FMSP'99, Trento, Italy, July 5, 1999*, 1999.

[137] G. Denker and J. Millen. CAPSL integrated protocol environment. In Koob et al. [272], pages 207–222.

[138] G. Denker and J. Millen. The CAPSL integrated protocol environment. Technical Report SRI-CSL-2000-02, Computer Science Laboratory, SRI International, 2000.

[139] G. Denker and C. Talcott, editors. *Proceedings of the 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.

[140] E. Deplagne, C. Kirchner, H. Kirchner, and Q. H. Nguyen. Proof search and proof check for equational and inductive theorems. 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 297–316. Springer, 2003.

[141] R. Diaconescu and K. Futatsugi. *CafeOBJ Report. The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*, volume 6 of *AMAST Series in Computing*. World Scientific, 1998.

[142] H. Ding, C. Zheng, G. Agha, and L. Sha. Automated verification of the dependability of object-oriented real-time systems. In *Proceedings of the 9th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2003 Fall), Anacapri (Capri Island), Italy, October 1–3, 2003*, pages 171–178. IEEE Computer Society, 2004.

[143] J. S. Dong and H. Zhu, editors. *Formal Methods and Software Engineering - 12th International Conference on Formal Engineering Methods, ICFEM 2010, Shanghai, China, November 17-19, 2010, Proceedings*, volume 6447 of *Lecture Notes in Computer Science*. Springer, 2010.

[144] D. J. Dougherty and S. Escobar, editors. *Proceedings of the Third International Workshop on Security and Rewriting Techniques, SecReT 2008, Pittsburgh, PA, USA, June 22, 2008*, volume 234 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2009.

[145] D. J. Dougherty, C. Kirchner, H. Kirchner, and A. S. de Oliveira. Modular access control via strategic rewriting. In J. Biskup and J. Lopez, editors, *Computer Security - ESORICS 2007, 12th European Symposium On Research In Computer Security, Dresden, Germany, September 24-26, 2007, Proceedings*, volume 4734 of *Lecture Notes in Computer Science*, pages 578–593. Springer, 2007.

[146] G. Dowek, T. Hardin, and C. Kirchner. Higher order unification via explicit substitutions. *Information and Computation*, 157(1-2):183–235, 2000.

[147] 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.

[148] G. Dowek, T. Hardin, and C. Kirchner. Theorem proving modulo. *Journal of Automated Reasoning*, 31(1):33–72, 2003.

[149] G. Dowek, C. Muñoz, and C. Rocha. Rewriting logic semantics of a plan execution language. In B. Klin and P. Sobociński, editors, *Proceedings of the Sixth Workshop on Structural Operational Semantics, SOS 2009, Bologna, Italy, August 31, 2009*, volume 18 of *Electronic Proceedings in Theoretical Computer Science*, pages 77–91, 2010.

[150] F. Durán. *A Reflective Module Algebra with Applications to the Maude Language*. PhD thesis, Universidad de Málaga, Spain, June 1999.

[151] 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.

[152] F. Durán, S. Eker, S. Escobar, J. Meseguer, and C. L. Talcott. Variants, unification, narrowing, and symbolic reachability in Maude 2.6. In M. Schmidt-Schauß, editor, *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications, RTA 2011, Novi Sad, Serbia, May 30 - June 1, 2011*, volume 10 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 31–40. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

[153] F. Durán, F. Gutiérrez, P. López, and E. Pimentel. A formalization of the SMEPP model in Maude. In V. Cahill, editor, *Proceedings of the 5th Annual International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services, MobiQuitous 2008, July 21-25, 2008, Dublin, Ireland.* ACM, 2008.

[154] F. Durán, J. Herrador, and A. Vallecillo. Using UML and Maude for writing and reasoning about ODP policies. In J. Moffett and F. Garcia, editors, *Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks, POLICY 2003, Lake Como, Italy, June 4-6, 2003*, pages 15–25. IEEE Computer Society, 2003.

[155] 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.

[156] 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.

[157] F. Durán, S. Lucas, and J. Meseguer. Methods for proving termination of rewriting-based programming languages by transformation. In J. M. Almendros-Jiménez, editor, *Proceedings of the Eighth Spanish Conference on Programming and Computer Languages, PROLE 2008, Gijón, Spain, October 8-10, 2008*, volume 248 of *Electronic Notes in Theoretical Computer Science*, pages 93–113. Elsevier, 2009.

[158] F. Durán, S. Lucas, and J. Meseguer. Termination modulo combinations of equational theories. In S. Ghilardi and R. Sebastiani, editors, *Frontiers of Combining Systems, 7th International Symposium, FroCoS 2009, Trento, Italy, September 16-18, 2009. Proceedings*, volume 5749 of *Lecture Notes in Computer Science*, pages 246–262. Springer, 2009.

[159] F. Durán and J. Meseguer. An extensible module algebra for Maude. In Kirchner and Kirchner [262], pages 174–195.

[160] F. Durán and J. Meseguer. Maude's module algebra. *Science of Computer Programming*, 66(2):125–153, 2007.

[161] F. Durán and J. Meseguer. On the Church-Rosser and coherence properties of conditional order-sorted rewrite theories. *Journal of Logic and Algebraic Programming*, 2011. This volume.

[162] F. Durán and P. C. Ölveczky. A guide to extending Full Maude illustrated with the implementation of Real-Time Maude. In Roşu [403], pages 83–102.

[163] F. Durán, M. Ouederni, and G. Salaün. Checking protocol compatibility using Maude. In G. Salaün and M. Sirjani, editors, *Proceedings of the 8th International Workshop on the Foundations of Coordination Languages and Software Architectures, FOCLASA 2009, Rhodes, Greece, July 11, 2009*, volume 255 of *Electronic Notes in Theoretical Computer Science*, pages 65–81. Elsevier, 2009.

[164] F. Durán, C. Rocha, and J. M. Álvarez. Tool interoperability in the Maude formal environment. In Corradini et al. [124], pages 400–406.

[165] F. Durán, C. Rocha, and J. M. Álvarez. Towards a Maude formal environment. In G. Agha, O. Danvy, and J. Meseguer, editors, *Formal Modeling: Actors, Open Systems, Biological Systems - Essays Dedicated to Carolyn Talcott on the Occasion of Her 70th Birthday*, volume 7000 of *Lecture Notes in Computer Science*, pages 329–351. Springer, 2011.

[166] F. Durán, M. Roldán, and A. Vallecillo. Using Maude to write and execute ODP information viewpoint specifications. *Computer Standards & Interfaces*, 27(6):597–620, 2005.

[167] F. Durán and A. Vallecillo. Specifying the ODP information viewpoint using Maude. In H. Kilov and K. Baclawski, editors, *Proceedings of the Tenth OOPSLA Workshop on Behavioral Semantics, Tampa Bay, Florida*, pages 44–57, Oct. 2001.

[168] F. Durán and A. Vallecillo. Formalizing ODP enterprise specifications in Maude. *Computer Standards & Interfaces*, 25(2):83–102, 2003.

[169] M. Egea and V. Rusu. Formal executable semantics for conformance in the MDE framework. *Innovations in Systems and Software Engineering*, 6(1-2):73–81, 2009.

[170] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[171] S. Eker. Fast matching in combination of regular equational theories. In Meseguer [317], pages 90–109.

[172] S. Eker. Associative-commutative rewriting on large terms. In Nieuwenhuis [355], pages 14–29.

[173] S. Eker, M. Knapp, K. Laderoute, P. Lincoln, J. Meseguer, and M. K. Sönmez. Pathway logic: Symbolic analysis of biological signaling. In R. B. Altman, A. K. Dunker, L. Hunter, and T. E. Klein, editors, *Proceedings of the 7th Pacific Symposium on Biocomputing, PSB 2002, Lihue, Hawaii, USA, January 3-7, 2002*, pages 400–412, January 2002.

[174] S. Eker, M. Knapp, K. Laderoute, P. Lincoln, and C. Talcott. Pathway logic: Executable models of biological networks. In Gadducci and Montanari [205], pages 144–161.

[175] S. Eker, N. Martí-Oliet, J. Meseguer, and A. Verdejo. Deduction, strategies, and rewriting. In M. Archer, T. B. de la Tour, and C. Muñoz, editors, *Proceedings of the 6th International Workshop on Strategies in Automated Deduction, STRATEGIES 2006, Seattle, WA, USA, August 16, 2006*, volume 174(11) of *Electronic Notes in Theoretical Computer Science*, pages 3–25. Elsevier, 2007.

[176] C. Ellison and G. Roşu. A formal semantics of C with applications. Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign, 2010.

[177] C. Ellison, T. F. Şerbănuţă, and G. Roşu. A rewriting logic approach to type inference. In Corradini and Montanari [125], pages 135–151.

[178] 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 of the Second International Workshop on Security and Rewriting Techniques, SecReT 2007, Paris, France, June 29, 2007*, 2007.

[179] S. Escobar, D. Kapur, C. Lynch, C. Meadows, J. Meseguer, P. Narendran, and R. Sasse. Protocol analysis in Maude-NPA using unification modulo homomorphic encryption. In Schneider-Kamp and Hanus [416], pages 65–76.

[180] 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.

[181] S. Escobar, C. Meadows, and J. Meseguer. Equational cryptographic reasoning in the Maude-NRL Protocol Analyzer. In M. Fernández and C. Kirchner, editors, *Proceedings of the 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.

[182] S. Escobar, C. Meadows, and J. Meseguer. State space reduction in the Maude-NRL Protocol Analyzer. In S. Jajodia and J. López, editors, *Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings*, volume 5283 of *Lecture Notes in Computer Science*, pages 548–562. Springer, 2008.

[183] 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.

[184] S. Escobar, C. Meadows, and J. Meseguer. State space reduction in the Maude-NRL protocol analyzer, May 2011. `http://arxiv.org/abs/1105.5282`.

[185] S. Escobar, C. Meadows, J. Meseguer, and S. Santiago. Sequential protocol composition in Maude-NPA. In D. Gritzalis, B. Preneel, and M. Theoharidou, editors, *Computer Security - ESORICS 2010, 15th European Symposium on Research in Computer Security, Athens, Greece, September 20-22, 2010, Proceedings*, volume 6345 of *Lecture Notes in Computer Science*, pages 303–318. Springer, 2010.

[186] S. Escobar and J. Meseguer. Symbolic model checking of infinite-state systems using narrowing. In Baader [35], pages 153–168.

[187] 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.

[188] S. Escobar, J. Meseguer, and P. Thati. Narrowing and rewriting logic: from foundations to applications. In F. J. López-Fraguas, editor, *Proceedings of the 15th Workshop on Functional and (Constraint) Logic Programming, WFLP 2006, Madrid, Spain, November 16-17, 2006*, volume 177 of *Electronic Notes in Theoretical Computer Science*, pages 5–33. Elsevier, 2007.

[189] S. Escobar, R. Sasse, and J. Meseguer. Folding variant narrowing and optimal variant termination. In Ölveczky [362], pages 52–68.

[190] S. Escobar, R. Sasse, and J. Meseguer. Folding variant narrowing and optimal variant termination. *Journal of Logic and Algebraic Programming*, 2011. This volume.

[191] A. Farzan, F. Chen, J. Meseguer, and G. Roşu. Formal analysis of Java programs in JavaFAN. In R. Alur and D. 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. Springer, 2004.

[192] A. Farzan and J. Meseguer. State space reduction of rewrite theories using invisible transitions. In Johnson and Vene [247], pages 142–157.

[193] A. Farzan and J. Meseguer. Partial order reduction for rewriting semantics of programming languages. In Denker and Talcott [139], pages 61–78.

[194] A. Farzan, J. Meseguer, and G. Roşu. Formal JVM code analysis in JavaFAN. In Rattray et al. [389], pages 132–147.

[195] S. Feferman. Finitary inductively presented logics. In R. Ferro et al., editors, *Logic Colloquium'88*, pages 191–220. North-Holland, 1989.

[196] A. Felty and D. Miller. Encoding a dependent-type λ-calculus in a logic programming language. In M. Stickel, editor, *Proc. 10th. Int. Conf. on Automated Deduction, Kaiserslautern, Germany, July 1990*, volume 449 of *LNCS*, pages 221–235. Springer-Verlag, 1990.

[197] J. L. Fernández Alemán and J. A. Toval Álvarez. Can intuition become rigorous? Foundations for UML model verification tools. In F. M. Titsworth, editor, *Proceedings of the 11th International Symposium on Software Reliability Engineering, ISSRE 2000, San Jose, CA, USA, October 8-11, 2000*, pages 344–355. IEEE Computer Society, 2000.

[198] O. Fissore, I. Gnaedig, and H. Kirchner. System presentation – CARIBOO: An induction based proof tool for termination with strategies. In C. Kirchner, editor, *Proceedings of the 4th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, PPDP 2002, Pittsburgh, PA, USA, October 6-8, 2002 (Affiliated with PLI 2002)*, pages 62–73. ACM, 2002.

[199] O. Fissore, I. Gnaedig, and H. Kirchner. Simplification and termination of strategies in rule-based languages. In D. Miller and K. Sagonas, editors, *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, PPDP 2003, Uppsala, Sweden, August 27-29, 2003*, pages 124–135. ACM, 2003.

[200] K. Futatsugi, editor. *Proceedings of the 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.

[201] K. Futatsugi. Verifying specifications with proof scores in CafeOBJ. In S. Uchitel and S. Easterbrook, editors, *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, ASE 2006, Tokyo, Japan, September 18-22, 2006*, pages 3–10. IEEE Computer Society, 2006.

[202] K. Futatsugi. Fostering proof scores in CafeOBJ. In Dong and Zhu [143], pages 1–20.

[203] K. Futatsugi, A. T. Nakagawa, and T. Tamai, editors. *Cafe: An Industrial-Strength Algebraic Formal Method*. Elsevier, 2000.

[204] D. M. Gabbay and A. Pnueli. A sound and complete deductive system for CTL* verification. *Logic Journal of the IGPL*, 16(6):499–536, 2008.

[205] F. Gadducci and U. Montanari, editors. *Proceedings of the 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.

[206] P. Gardner. *Representing Logics in Type Theory*. PhD thesis, Technical Report CST-93-92, Department of Computer Science, University of Edinburgh, 1992.

[207] A. Garrido and J. Meseguer. Formal specification and verification of Java refactorings. In *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation, SCAM 2006, Philadelphia, Pennsylvania, September 27-29, 2006*, pages 165–174. IEEE, 2006.

[208] A. Garrido, J. Meseguer, and R. Johnson. Algebraic semantics of the C preprocessor and correctness of its refactorings. Technical Report UIUCDCS-R-2006-2688, Department of Computer Science, University of Illinois at Urbana-Champaign, February 2006.

[209] 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*. Springer, 2005.

[210] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Automated termination proofs with AProVE. In *RTA 2004*, volume 3091, pages 210–220. Springer LNCS, 2004.

[211] P. Glynn. The role of generalized semi-Markov processes in simulation output analysis, 1983.

[212] I. Gnaedig. Induction for positive almost sure termination. In Leuschel and Podelski [284], pages 167–178.

[213] I. Gnaedig and H. Kirchner. Computing constructor forms with non terminating rewrite programs. In A. Bossi and M. J. Maher, editors, *Proceedings of the 8th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, PPDP 2006, Venice, Italy, July 10-12, 2006*, pages 121–132. ACM, 2006.

[214] I. Gnaedig and H. Kirchner. Termination of rewriting under strategies. *ACM Transactions on Computational Logic*, 10(2), 2009.

[215] J. Goguen. OBJ as a theorem prover with application to hardware verification. In P. Subramanyam and G. Birtwistle, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 218–267. Springer-Verlag, 1989.

[216] J. Goguen and R. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the ACM*, 39(1):95–146, 1992.

[217] J. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the 1982 Symposium on Security and Privacy*, pages 11–20. IEEE, 1982.

[218] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In *Software Engineering with OBJ: Algebraic Specification in Action*, pages 3–167. Kluwer, 2000.

[219] J. A. Goguen, A. Stevens, K. Hobley, and H. Hilberdink. 2OBJ: A meta-logical framework based on equational logic. *Philosophical Transactions of the Royal Society, Series A*, 339:69–86, 1992.

[220] A. Goodloe. Private communication, May 25, 2011.

[221] A. Goodloe, C. A. Gunter, and M.-O. Stehr. Formal prototyping in early stages of protocol design. In C. Meadows, editor, *Proceedings of the POPL 2005 Workshop on Issues in the Theory of Security, WITS 2005, Long Beach, California, USA, January 10-11, 2005*, pages 67–80. ACM, 2005.

[222] A. Goodloe, M. Jacobs, G. Shah, and C. Gunter. L3A: A protocol for layer three accounting. In *Proceedings of the First Workshop on Secure Network Protocols, NPSEC 2005, Boston, Massachusetts, November 6, 2005*, pages 1–6. IEEE Computer Society, 2005.

[223] A. Goodloe, M. McDougall, C. A. Gunter, and M.-O. Stehr. Design and analysis of Sectrace: A protocol to set up security associations and policies in ipsec networks. Technical report, CIS Department, University of Pennsylvania, 2004. `http://seclab.web.cs.illinois.edu/penn-security-lab`.

[224] C. Grier, S. Tang, and S. T. King. Secure web browsing with the OP web browser. In *2008 IEEE Symposium on Security and Privacy (S&P 2008), Oakland, California, May 18-21, 2008*, pages 402–416. IEEE Computer Society, 2008.

[225] R. Gutiérrez and S. Lucas. Proving termination in the context-sensitive dependency pair framework. In Ölveczky [362], pages 18–34.

[226] S. Gutierrez-Nolasco, N. Venkatasubramanian, M.-O. Stehr, and C. L. Talcott. Exploring adaptability of secure group communication using formal prototyping techniques. In F. Kon, F. M. Costa, N. Wang, and R. Cerqueira, editors, *Proceedings of the 3rd Workshop on Adaptive and Reflective Middleware, Toronto, Ontario, Canada, October 19, 2004*, pages 232–237. ACM, 2004.

[227] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Asp. Comput.*, 6(5):512–535, 1994.

[228] N. A. Harman. Correctness and verification of hardware systems using Maude. Technical Report 3-2000, Department of Computer Science, University of Wales Swansea, 2000.

[229] 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.

[230] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association Computing Machinery*, 40(1):143–184, 1993.

[231] J. Hendrix. *Decision Procedures for Equationally Based Reasoning.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2008. `http://hdl.handle.net/2142/10967`.

[232] J. Hendrix, M. Clavel, and J. Meseguer. A sufficient completeness reasoning tool for partial specifications. In Giesl [209], pages 165–174.

[233] J. Hendrix, D. Kapur, and J. Meseguer. Coverset induction with partiality and subsorts: A powerlist case study. In M. Kaufmann and L. C. Paulson, editors, *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6172 of *Lecture Notes in Computer Science*, pages 275–290. Springer, 2010.

[234] J. Hendrix and J. Meseguer. On the completeness of context-sensitive order-sorted specifications. In Baader [35], pages 229–245.

[235] J. Hendrix and J. Meseguer. Order-sorted equational unification revisited. In Kniesel and Pinto [270], pages 16–29.

[236] 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, WA, USA, August 17-20, 2006, Proceedings*, volume 4130 of *Lecture Notes in Computer Science*, pages 151–155. Springer, 2006.

[237] J. Hendrix, H. Ohsaki, and J. Meseguer. Sufficient completeness checking with propositional tree automata. Technical Report UIUCDCS-R-2005-2635, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[238] J. Hendrix, H. Ohsaki, and M. Viswanathan. Propositional tree automata. In F. Pfenning, editor, *Term Rewriting and Applications, 17th International Conference, RTA 2006, Seattle, WA, USA, August 12-14, 2006, Proceedings*, volume 4098 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2006.

[239] M. Hills, T. B. Aktemur, and G. Roşu. An executable semantic definition of the Beta language using rewriting logic. Technical Report UIUCDCS-R-2005-2650, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[240] M. Hills, F. Chen, and G. Roşu. A rewriting logic approach to static checking of units of measurement in C. In Kniesel and Pinto [270], pages 76–91.

[241] M. Hills and G. Roşu. KOOL: An application of rewriting logic to language prototyping and analysis. In Baader [35], pages 246–256.

[242] M. M. Hölzl, M. Meier, and M. Wirsing. Which soft constraints do you prefer? In Roşu [403], pages 189–205.

[243] J. Hsiang, editor. *Rewriting Techniques and Applications, 6th International Conference, RTA-95, Kaiserslautern, Germany, April 5-7, 1995, Proceedings*, volume 914 of *Lecture Notes in Computer Science*. Springer, 1995.

[244] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the Association for Computing Machinery*, 27:797–821, 1980.

[245] E. B. Johnsen, O. Owe, and E. W. Axelsen. A run-time environment for concurrent objects with asynchronous method calls. In Martí-Oliet [297], pages 375–392.

[246] M. Johnson and D. Pavlovic, editors. *Algebraic Methodology and Software Technology, 13th International Conference, AMAST 2010, Lac-Beauport, QC, Canada, June 23-25, 2010, Revised Selected Papers*, volume 6486 of *Lecture Notes in Computer Science*. Springer, 2011.

[247] M. Johnson and V. Vene, editors. *Algebraic Methodology and Software Technology, 11th International Conference, AMAST 2006, Kuressaare, Estonia, July 5-8, 2006, Proceedings*, volume 4019 of *Lecture Notes in Computer Science*. Springer, 2006.

[248] J.-P. Jouannaud and HélèneKirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal of Computing*, 15:1155–1194, November 1986.

[249] J.-P. Jouannaud, C. Kirchner, and H. Kirchner. Incremental construction of unification algorithms in equational theories. In *Proc. ICALP'83*, pages 361–373. Springer LNCS 154, 1983.

[250] D. Kapur and M. Subramaniam. Mechanical verification of adder circuits using rewrite rule laboratory. *Formal Methods in System Design*, 13(2):127–158, 1998.

[251] M. Katelman. *A Meta-Language for Functional Verification*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2011. `http://hdl.handle.net/2142/29614`.

[252] M. Katelman, S. Keller, and J. Meseguer. Concurrent rewriting semantics and analysis of asynchronous digital circuits. In Ölveczky [362], pages 140–156.

[253] M. Katelman, S. Keller, and J. Meseguer. Rewriting semantics of production rule sets. *Journal of Logic and Algebraic Programming*, 2011. This volume.

[254] M. Katelman and J. Meseguer. A rewriting semantics for ABEL with applications to hardware/software co-design and analysis. In Denker and Talcott [139], pages 47–60.

[255] M. Katelman and J. Meseguer. Using the PALS architecture to verify a distributed topology control protocol for wireless multi-hop networks in the presence of node failures. In Ölveczky [361], pages 101–116.

[256] M. Katelman and J. Meseguer. vlogsl: A strategy language for simulation-based verification of hardware. In S. Barner, I. G. Harris, D. Kroening, and O. Raz, editors, *Hardware and Software: Verification and Testing - 6th International Haifa Verification Conference, HVC 2010, Haifa, Israel, October 4-7, 2010. Revised Selected Papers*, volume 6504 of *Lecture Notes in Computer Science*, pages 129–145. Springer, 2011.

[257] M. Katelman, J. Meseguer, and S. Escobar. Directed-logical testing for functional verification of microprocessors. In S. A. Edwards and K. Schneider, editors, *Proceedings of the 6th ACM & IEEE International Conference on Formal Methods and Models for Co-Design, MEMOCODE 2008, Anaheim, CA, USA, June 5-7, 2008*, pages 89–100. IEEE Computer Society, 2008.

[258] M. Katelman, J. Meseguer, and J. C. Hou. Redesign of the LMST wireless sensor protocol through formal modeling and statistical model checking. In Barthe and de Boer [49], pages 150–169.

[259] M. Kim, M.-O. Stehr, C. L. Talcott, N. Dutt, and N. Venkatasubramanian. Combining formal verification with observed system execution behavior to tune system parameters. In J.-F. Raskin and P. S. Thiagarajan, editors, *Formal Modeling and Analysis of Timed Systems, 5th International Conference, FORMATS 2007, Salzburg, Austria, October 3-5, 2007, Proceedings*, volume 4763 of *Lecture Notes in Computer Science*, pages 257–273. Springer, 2007.

[260] M. Kim, M.-O. Stehr, C. L. Talcott, N. Dutt, and N. Venkatasubramanian. Constraint refinement for online verifiable cross-layer system adaptation. In *Design, Automation and Test in Europe, DATE 2008, Munich, Germany, March 10-14, 2008*, pages 646–651. IEEE, 2008.

[261] M. Kim, M.-O. Stehr, C. L. Talcott, N. D. Dutt, and N. Venkatasubramanian. A probabilistic formal analysis approach to cross layer optimization in distributed embedded systems. In Bonsangue and Johnsen [59], pages 285–300.

[262] C. Kirchner and H. Kirchner, editors. *Proceedings of the 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.

[263] C. Kirchner, H. Kirchner, and A. S. de Oliveira. Analysis of rewrite-based access control policies. In Dougherty and Escobar [144], pages 55–75.

[264] H. Kirchner and P.-E. Moreau. Prototyping completion with constraints using computational systems. In Hsiang [243], pages 438–443.

[265] H. Kirchner and P.-E. Moreau. Promoting rewriting to a programming language: a compiler for non-deterministic rewrite programs in associative-commutative theories. *Journal of Functional Programming*, 11(2):207–251, 2001.

[266] H. Kirchner and C. Ringeissen. Combining symbolic constraint solvers on algebraic domains. *Journal of Symbolic Computation*, 18(2):113–155, 1994.

[267] H. Kirchner and C. Ringeissen. Constraint solving by narrowing in combined algebraic domains. In *Proceedings of the 11th International Conference on Logic Programming*, pages 617–631. The MIT Press, 1994.

[268] A. Knapp. Generating rewrite theories from UML collaborations. In Futatsugi et al. [203], pages 97–120.

[269] A. Knapp. *A Formal Approach to Object-Oriented Software Engineering*. Shaker Verlag, Aachen, Germany, 2001. PhD thesis, Institut für Informatik, Universität München, 2000.

[270] G. Kniesel and J. S. Pinto, editors. *Preliminary Proceedings of the Ninth International Workshop on Rule-Based Programming, RULE 2008, Hagenberg Castle, Austria, June 18, 2008*, 2008. Technical Report IAI-TR-08-02, Institut für Informatik III, Rheinische Friedrich-Wilhelm-Universität Bonn.

[271] W. Kolch. Meaningful relationships: The regulation of the Ras/Raf/MEK/ERK pathway by protein interactions. *Biochem. J.*, 351:289–305, 2000.

[272] G. Koob, D. Maughan, and S. Saydjari, editors. *Proceedings of the DARPA Information Survivability Conference and Exposition, DISCEX 2000, Hilton Head Island, South Carolina, January 25-27, 2000*. IEEE Computer Society Press, 2000.

[273] P. Kosiuczenko and M. Wirsing. Timed rewriting logic for the specification of time-sensitive systems. In H. Schwichtenberg, editor, *Proceedings of the NATO Advanced Study Institute on Logic of Computation, Held in Marktoberdorf, Germany, July 25-August 6, 1997*, volume 157 of *NATO ASI Series F: Computer and Systems Sciences*, pages 229–264. Springer, 1997.

[274] P. Kosiuczenko and M. Wirsing. Timed rewriting logic with an application to object-based specification. *Science of Computer Programming*, 28(2-3):225–246, 1997.

[275] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990.

[276] N. Kumar, K. Sen, J. Meseguer, and G. Agha. Probabilistic rewrite theories: Unifying models, logics and tools. Technical Report UIUCDCS-R-2003-2347, Department of Computer Science, University of Illinois at Urbana-Champaign, May 2003.

[277] N. Kumar, K. Sen, J. Meseguer, and G. Agha. A rewriting based model for probabilistic distributed object systems. In Najm et al. [349], pages 32–46.

[278] C. Laneve and U. Montanari. Axiomatizing permutation equivalence. *Mathematical Structures in Computer Science*, 6(3):219–249, 1996.

[279] F. W. Lawvere. Functorial semantics of algebraic theories. *Proceedings, National Academy of Sciences*, 50:869–873, 1963. Summary of Ph.D. Thesis, Columbia University.

[280] E. A. Lee. Modeling concurrent real-time processes using discrete events. *Ann. Software Eng.*, 7:25–45, 1999.

[281] M. LeMay and C. A. Gunter. Cumulative attestation kernels for embedded systems. In Backes and Ning [36], pages 655–670.

[282] D. Lepri, P. Ölveczky, and E. Ábrahám. Timed CTL model checking in Real-Time Maude. Submitted for publication, 2011.

[283] D. Lepri, P. C. Ölveczky, and E. Ábrahám. Model checking classes of metric LTL properties of object-oriented Real-Time Maude specifications. In Ölveczky [361], pages 117–136.

[284] M. Leuschel and A. Podelski, editors. *Proceedings of the 9th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, PPDP 2007, Wroclaw, Poland, July 14-16, 2007*. ACM, 2007.

[285] E. Lien. Formal modelling and analysis of the NORM multicast protocol using Real-Time Maude. Master's thesis, Department of Linguistics, University of Oslo, April 2004.

[286] E. Lien and P. C. Ölveczky. Formal modeling and analysis of an IETF multicast protocol. In D. V. Hung and P. Krishnan, editors, *Proceedings of the Seventh IEEE International Conference on Software Engineering and Formal Methods, SEFM 2009, Hanoi, Vietnam, November 23-27, 2009*, pages 273–282. IEEE Computer Society, 2009.

[287] L. Liquori and B. Wack. The polymorphic rewriting-calculus: [type checking vs. type inference]. In Martí-Oliet [297], pages 89–111.

[288] D. Lucanu. Strategy-based rewrite semantics for membrane systems preserves maximal concurrency of evolution rule actions. In A. Middeldorp, editor, *Proceedings of the Eighth International Workshop on Reduction Strategies in Rewriting and Programming, WRS 2008, Castle of Hagenberg, Austria, July 14, 2008*, volume 237 of *Electronic Notes in Theoretical Computer Science*, pages 107–125. Elsevier, 2009.

[289] S. Lucas. Context-sensitive computations in functional and functional logic programs. *Journal of Functional and Logic Programming*, 1998(1), 1998.

[290] S. Lucas. Context-sensitive rewriting strategies. *Information and Computation*, 178(1):294–343, 2002.

[291] S. Lucas, C. Marché, and J. Meseguer. Operational termination of conditional term rewriting systems. *Information Processing Letters*, 95(4):446–453, 2005.

[292] S. Lucas and J. Meseguer. Order-sorted dependency pairs. In S. Antoy and E. Albert, editors, *Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, PPDP 2008, Valencia, Spain, July 15-17, 2008*, pages 108–119. ACM, 2008.

[293] S. Lucas and J. Meseguer. Termination of just/fair computations in term rewriting. *Information and Computation*, 206(5):652–675, 2008.

[294] S. Lucas and J. Meseguer. Operational termination of membership equational programs: the order-sorted way. In Roşu [403], pages 207–225.

[295] P. Manolios. A compositional theory of refinement for branching time. In *CHARME 2003*, volume 2860 of *Lecture Notes in Computer Science*, pages 304–318. Springer, 2003.

[296] C. Marché and X. Urbain. Modular and incremental proofs of AC-termination. *J. Symb. Comput.*, 38(1):873–897, 2004.

[297] N. Martí-Oliet, editor. *Proceedings of the Fifth International Workshop on Rewriting Logic and its Applications, WRLA 2004, Barcelona, Spain, March 27-April 4, 2004*, volume 117 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2004.

[298] 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.

[299] 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.

[300] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In D. M. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, Second Edition, Volume 9*, pages 1–87. Kluwer Academic Publishers, 2002.

[301] N. Martí-Oliet and J. Meseguer. Rewriting logic: roadmap and bibliography. *Theoretical Computer Science*, 285(2):121–154, 2002.

[302] N. Martí-Oliet, J. Meseguer, and M. Palomino. Theoroidal maps as algebraic simulations. In J. L. Fiadeiro, P. D. Mosses, and F. Orejas, editors, *Recent Trends in Algebraic Development Techniques, 17th International Workshop, WADT 2004, Barcelona, Spain, March 27-29, 2004, Revised Selected Papers*, volume 3423 of *Lecture Notes in Computer Science*, pages 126–143. Springer, 2004.

[303] N. Martí-Oliet, J. Meseguer, and A. Verdejo. A rewriting semantics for Maude strategies. In Roşu [403], pages 227–247.

[304] I. A. Mason and C. L. Talcott. Simple network protocol simulation within Maude. In Futatsugi [200], pages 274–291.

[305] S. Matthews, A. Smaill, and D. Basin. Experience with FS_0 as a framework theory. In G. Huet and G. Plotkin, editors, *Logical Environments*, pages 61–82. Cambridge University Press, 1993.

[306] C. Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.

[307] P. Meredith, M. Hills, and G. Roşu. An executable rewriting logic semantics of K-Scheme. In D. Dube, editor, *Proceedings of the 2007 Workshop on Scheme and Functional Programming, SCHEME 2007, Freiburg, Germany, September 30, 2007*, pages 91–103. Laval University, 2007.

[308] P. Meredith, M. Hills, and G. Roşu. A K definition of Scheme. Technical Report UIUCDCS-R-2007-2907, Department of Computer Science, University of Illinois at Urbana-Champaign, 2007.

[309] P. Meredith, M. Katelman, J. Meseguer, and G. Roşu. A formal executable semantics of Verilog. In B. Jobstmann and L. Carloni, editors, *Proceedings of the Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE 2010, Grenoble, France, July 26-28, 2010*, pages 179–188. IEEE Computer Society, 2010.

[310] J. Meseguer. General logics. In H.-D. E. et al., editor, *Logic Colloquium'87*, pages 275–329. North-Holland, 1989.

[311] J. Meseguer. A logical theory of concurrent objects. In N. Meyrowitz, editor, *Proceedings of the ECOOP-OOPSLA'90 Conference on Object-Oriented Programming, Ottawa, Canada, October 21-25, 1990*, pages 101–115. ACM Press, 1990.

[312] J. Meseguer. Rewriting as a unified model of concurrency. In J. C. M. Baeten and J. W. Klop, editors, *CONCUR '90, Theories of Concurrency: Unification and Extension, Amsterdam, The Netherlands, August 27-30, 1990, Proceedings*, volume 458 of *Lecture Notes in Computer Science*, pages 384–400. Springer, 1990.

[313] J. Meseguer. Rewriting as a unified model of concurrency. Technical Report SRI-CSL-90-02, SRI International, Computer Science Laboratory, February 1990. Revised June 1990.

[314] J. Meseguer. Conditional rewriting logic: Deduction, models and concurrency. In S. Kaplan and M. Okada, editors, *Conditional and Typed Rewriting Systems, 2nd International CTRS Workshop, Montreal, Canada, June 11-14, 1990, Proceedings*, volume 516 of *Lecture Notes in Computer Science*, pages 64–91. Springer, 1991.

[315] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[316] 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.

[317] J. Meseguer, editor. *Proceedings of the 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.

[318] 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.

[319] J. Meseguer. Membership algebra as a logical framework for equational specification. In Parisi-Presicce [378], pages 18–61.

[320] J. Meseguer. Rewriting logic and Maude: a wide-spectrum semantic framework for object-based distributed systems. In S. F. Smith and C. L. Talcott, editors, *Formal Methods for Open Object-Based Distributed Systems IV, IFIP TC6/WG6.1 Fourth International Conference on Formal Methods for Open Object-Based Distributed Systems, FMOODS 2000, Stanford, California, USA, September 6-8, 2000, Proceedings*, volume 177 of *IFIP Conference Proceedings*, pages 89–117. Kluwer, 2000.

[321] J. Meseguer. Rewriting logic and Maude: Concepts and applications. In L. Bachmair, editor, *Rewriting Techniques and Applications, 11th International Conference, RTA 2000, Norwich, UK, July 10-12, 2000, Proceedings*, volume 1833 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2000.

[322] J. Meseguer. Functorial semantics of rewrite theories. In H.-J. Kreowski, U. Montanari, F. Orejas, G. Rozenberg, and G. Taentzer, editors, *Formal Methods in Software and Systems Modeling, Essays Dedicated to Hartmut Ehrig on the Occasion of His 60th Birthday*, volume 3393 of *Lecture Notes in Computer Science*, pages 220–235. Springer, 2005.

[323] J. Meseguer. Localized fairness: A rewriting semantics. In Giesl [209], pages 250–263.

[324] J. Meseguer. A rewriting logic sampler. In D. V. Hung and M. Wirsing, editors, *Theoretical Aspects of Computing - ICTAC 2005, Second International Colloquium, Hanoi, Vietnam, October 17-21, 2005, Proceedings*, volume 3722 of *Lecture Notes in Computer Science*, pages 1–28. Springer, 2005.

[325] J. Meseguer. The temporal logic of rewriting: A gentle introduction. In P. Degano, R. D. Nicola, and J. Meseguer, editors, *Concurrency, Graphs and Models, Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*, volume 5065 of *Lecture Notes in Computer Science*, pages 354–382. Springer, 2008.

[326] 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, Tokyo, Japan, November 1992*, pages 61–106. Research Institute of Software Engineering, 1992.

[327] J. Meseguer and N. Martí-Oliet. From abstract data types to logical frameworks. In E. Astesiano, G. Reggio, and A. Tarlecki, editors, *Recent Trends in Data Type Specification, 10th Workshop on Specification of Abstract Data Types Joint with the 5th COMPASS Workshop, S. Margherita, Italy, May 30-June 3, 1994,*

Selected Papers, volume 906 of *Lecture Notes in Computer Science*, pages 48–80. Springer, 1995.

[328] J. Meseguer and U. Montanari. Mapping tile logic into rewriting logic. In Parisi-Presicce [378], pages 62–91.

[329] J. Meseguer and P. C. Ölveczky. Formalization and correctness of the PALS architectural pattern for distributed real-time systems. In Dong and Zhu [143], pages 303–320.

[330] J. Meseguer and P. C. Ölveczky. Formalization and correctness of the PALS architectural pattern for distributed real-time systems. Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign, 2010.

[331] J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational abstractions. *Theoretical Computer Science*, 403(2-3):239–264, 2008.

[332] J. Meseguer, M. Palomino, and N. Martí-Oliet. Algebraic simulations. *Journal of Logic and Algebraic Programming*, 79(2):103–143, 2010.

[333] 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 Computer Science*, pages 1–44. Springer, 2004.

[334] J. Meseguer and G. Roşu. The rewriting logic semantics project. *Theoretical Computer Science*, 373(3):213–237, 2007.

[335] J. Meseguer and G. Roşu. The rewriting logic semantics project: A progress report. In O. Owe, M. Steffen, and J. A. Telle, editors, *Fundamentals of Computation Theory - 18th International Symposium, FCT 2011, Oslo, Norway, August 22-25, 2011. Proceedings*, volume 6914 of *Lecture Notes in Computer Science*, pages 1–37. Springer, 2011.

[336] 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.

[337] 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.

[338] J. Meseguer and C. L. 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.

[339] J. Meseguer and P. Thati. Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols. In Martí-Oliet [297], pages 153–182.

[340] 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.

[341] S. Miller, D. Cofer, L. Sha, J. Meseguer, and A. Al-Nayeem. Implementing logical synchrony in integrated modular avionics. In *Proc. 28th Digital Avionics Systems Conference*. IEEE, 2009.

[342] J. Misra. Computation orchestration: A basis for wide-area computing. In M. Broy, editor, *Proc. of the NATO Advanced Study Institute, Engineering Theories of Software Intensive Systems Marktoberdorf, Germany, 2004*. NATO ASI Series, 2004.

[343] J. Misra and W. R. Cook. Computation orchestration. *Software and System Modeling*, 6(1):83–110, 2007.

[344] H. Miyoshi. Modelling conditional rewriting logic in structured categories. In Meseguer [317], pages 20–34.

[345] F. Mokhati and M. Badri. Generating Maude specifications from UML use case diagrams. *Journal of Object Technology*, 8(2):319–136, 2009.

[346] F. Mokhati, P. Gagnon, and M. Badri. Verifying UML diagrams with model checking: A rewriting logic based approach. In A. Mathur and W. E. Wong, editors, *Proceedings of the Seventh International Conference on Quality Software, QSIC 2007, Portland, Oregon, USA, October 11-12, 2007*, pages 356–362. IEEE Computer Society, 2007.

[347] F. Mokhati, B. Sahraoui, S. Bouzaher, and M. T. Kimour. A tool for specifying and validating agents' interaction protocols: From Agent UML to Maude. *Journal of Object Technology*, 9(3):59–77, 2010.

[348] G. Nadathur and D. Miller. An overview of λProlog. In K. Bowen and R. Kowalski, editors, *Fifth Int. Joint Conf. and Symp. on Logic Programming*, pages 810–827. The MIT Press, 1988.

[349] 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*. Springer, 2003.

[350] E. Najm and J.-B. Stefani. A formal semantics for the ODP computational model. *Computer Networks and ISDN Systems*, 27(8):1305–1329, 1995.

[351] S. Nakajima. Using algebraic specification techniques in development of object-oriented frameworks. In Wing et al. [471], pages 1664–1683.

[352] S. Nakajima and K. Futatsugi. An object-oriented modeling method for algebraic specifications in CafeOBJ. In *Proceedings of the 19th International Conference on Software Engineering, ICSE'97, Boston, Massachussets, May 17-23, 1997*. ACM Press, 1997.

[353] 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.

[354] Q. H. Nguyen, C. Kirchner, and H. Kirchner. External rewriting for skeptical proof assistants. *Journal of Automated Reasoning*, 29(3-4):309–336, 2002.

[355] 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*. Springer, 2003.

[356] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, Nov. 2006.

[357] K. Ogata and K. Futatsugi. Proof scores in the OTS/CafeOBJ method. In Najm et al. [349], pages 170–184.

[358] E. Ohlebusch. *Advanced Topics in Term Rewriting*. Springer Verlag, 2002.

[359] P. C. Ölveczky. *Specification and Analysis of Real-Time and Hybrid Systems in Rewriting Logic*. PhD thesis, University of Bergen, Norway, 2000.

[360] P. C. Ölveczky. Towards formal modeling and analysis of networks of embedded medical devices in Real-Time Maude. In P. Muenchaisri, editor, *Proceedings of the Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, SNPD 2008, Phuket, Thailand, August 6-8, 2008*, pages 241–248. IEEE Computer Society, 2008.

[361] P. C. Ölveczky, editor. *Proceedings of the First International Workshop on Rewriting Techniques for Real-Time Systems, RTRTS 2010, Longyearbyen, Spitsbergen, Norway, April 6-9, 2010*, volume 36 of *Electronic Proceedings in Theoretical Computer Science*. Computing Research Repository (CoRR), 2010.

[362] P. C. Ölveczky, editor. *Rewriting Logic and its Applications. 8th International Workshop, WRLA 2010, Held as a Satellite Event of ETAPS 2010, Paphos, Cyprus, March 20-21, 2010, Revised Selected Papers*, volume 6381 of *Lecture Notes in Computer Science*. Springer, 2010.

[363] P. C. Ölveczky, A. Boronat, and J. Meseguer. Formal semantics and analysis of behavioral AADL models in Real-Time Maude. In J. Hatcliff and E. Zucca, editors, *Formal Techniques for Distributed Systems, Joint 12th IFIP WG 6.1 International Conference, FMOODS 2010 and 30th IFIP WG 6.1 International Conference, FORTE 2010, Amsterdam, The Netherlands, June 7-9, 2010. Proceedings*, volume 6117 of *Lecture Notes in Computer Science*, pages 47–62. Springer, 2010.

[364] 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 the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings*, volume 3922 of *Lecture Notes in Computer Science*, pages 357–372. Springer, 2006.

[365] 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 the Joint European Conferences on Theory and Practice of Software, ETAPS 2001, Genova, Italy, April 2-6, 2001, Proceedings*, volume 2029 of *Lecture Notes in Computer Science*, pages 333–348. Springer, 2001.

[366] 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.

[367] P. C. Ölveczky and J. Meseguer. Specifying real-time systems in rewriting logic. In Meseguer [317], pages 284–309.

[368] P. C. Ölveczky and J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science*, 285(2):359–405, 2002.

[369] P. C. Ölveczky and J. Meseguer. Real-Time Maude 2.1. In Martí-Oliet [297], pages 285–314.

[370] P. C. Ölveczky and J. Meseguer. Abstraction and completeness for Real-Time Maude. In Denker and Talcott [139], pages 5–27.

[371] P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20(1-2):161–196, 2007.

[372] P. C. Ölveczky and J. Meseguer. Specification and verification of distributed embedded systems: A traffic intersection product family. In Ölveczky [361], pages 137–157.

[373] 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.

[374] P. C. Ölveczky, P. Prabhakar, and X. Liu. Formal modeling and analysis of real-time resource-sharing protocols in Real-Time Maude. In Y. Robert, editor, *22nd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, Miami, Florida USA, April 14-18, 2008*, pages 1–8. IEEE, 2008.

[375] P. C. Ölveczky and S. Thorvaldsen. Formal modeling and analysis of the OGDC wireless sensor network algorithm in Real-Time Maude. In Bonsangue and Johnsen [59], pages 122–140.

[376] P. C. Ölveczky and S. Thorvaldsen. Formal modeling, performance estimation, and model checking of wireless sensor network algorithms in Real-Time Maude. *Theor. Comput. Sci.*, 410(2-3):254–280, 2009.

[377] M. Palomino, J. Meseguer, and N. Martí-Oliet. A categorical approach to simulations. 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 313–330. Springer, 2005.

[378] 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*. Springer, 1997.

[379] K. Pattabiraman, N. Nakka, Z. Kalbarczyk, and R. K. Iyer. SymPLFIED: Symbolic program-level fault injection and error detection framework. In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2008, Anchorage, Alaska, USA, June 24-27, 2008*, pages 472–481. IEEE Computer Society, 2008.

[380] K. Pattabiraman, N. Nakka, Z. Kalbarczyk, and R. K. Iyer. Discovering application-level insider attacks using symbolic execution. In D. Gritzalis and J. Lopez, editors, *Emerging Challenges for Security, Privacy and Trust, 24th IFIP TC 11 International Information Security Conference, SEC 2009, Pafos, Cyprus, May 18-20, 2009. Proceedings*, volume 297 of *IFIP Advances in Information and Communication Technology*, pages 63–75. Springer, 2009.

[381] L. C. Paulson. *Isabelle*, volume 828 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.

[382] G. Paun, editor. *Membrane Computing. An Introduction.* Springer, 2002.

[383] G. E. Peterson and M. E. Stickel. Complete sets of reductions for some equational theories. *Journal of the Association for Computing Machinery*, 28(2):233–264, 1981.

[384] F. Pfenning. Elf: A language for logic definition and verified metaprogramming. In *Proc. Fourth Annual IEEE Symp. on Logic in Computer Science*, pages 313–322, Asilomar, California, June 1989.

[385] A. Pnueli. Deduction is forever. Invited talk at FM'99 avaliable online at `cs.nyu.edu/pnueli/fm99.ps`, 1999.

[386] S. Porat and N. Francez. Fairness in term rewriting systems. In *RTA'85*, volume 202, pages 287–300. Springer LNCS, 1985.

[387] S. Porat and N. Francez. Full-commutation and fair-termination in equational (and combined) term-rewriting systems. In *CADE'86*, volume 230, pages 21–41. Springer LNCS, 1986.

[388] M. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming.* John Wiley and Sons, 1994.

[389] 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*. Springer, 2004.

[390] S. Reich. Implementing and extending the MSR crypto-protocol specification language. Master's thesis, Fachbereich Informatik, Universität Hamburg, April 2006.

[391] A. Riesco and A. Verdejo. Implementing and analyzing in Maude the Enhanced Interior Gateway Routing Protocol. In Roşu [403], pages 249–266.

[392] A. Riesco, A. Verdejo, N. Martí-Oliet, and R. Caballero. Declarative debugging of rewriting logic specifications. *Journal of Logic and Algebraic Programming*, 2011. This volume.

[393] J. E. Rivera, F. Durán, and A. Vallecillo. A graphical approach for modeling time-dependent behavior of DSLs. In R. DeLine and M. Minas, editors, *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2009, Corvallis, OR, USA, September 20-24, 2009*, pages 51–55. IEEE, 2009.

[394] J. E. Rivera, F. Durán, and A. Vallecillo. On the behavioral semantics of real-time domain specific visual languages. In Ölveczky [362], pages 174–190.

[395] C. Rocha and J. Meseguer. A rewriting decision procedure for Dijkstra-Scholten's syllogistic logic with complements. *Revista Colombiana de Computación*, 8(2), Dec. 2007.

[396] 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.

[397] C. Rocha and J. Meseguer. Constructors, sufficient completeness and deadlock freedom of rewrite theories. In C. G. Fermüller and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010, Proceedings*, volume 6397 of *Lecture Notes in Computer Science*, pages 594–609. Springer, 2010.

[398] C. Rocha and J. Meseguer. Proving safety properties of rewrite theories. In Corradini et al. [124], pages 314–328.

[399] C. Rocha, C. Muñoz, and H. Cadavid. A graphical environment for the semantic validation of a plan execution language. In S. Grenander and L. Bergman, editors, *Proceedings of the Third IEEE International Conference on Space Mission Challenges for Information Technology, SMC-IT 2009, Pasadena, California, USA, July 19-23, 2009*, pages 201–207, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[400] D. E. Rodríguez. Case studies in the specification and analysis of protocols in Maude. In Futatsugi [200], pages 257–273.

[401] D. E. Rodríguez. A secret-sharing protocol modelled in Maude. In Gadducci and Montanari [205], pages 223–239.

[402] J. R. Romero, A. Vallecillo, and F. Durán. Writing and executing ODP computational viewpoint specifications using Maude. *Computer Standards & Interfaces*, 29(4):481–498, 2007.

[403] G. Roşu, editor. *Proceedings of the Seventh 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*. Elsevier, 2009.

[404] G. Roşu and A. Ştefănescu. Matching logic: a new program verification approach (new ideas and emerging results track). In R. N. Taylor, H. Gall, and N. Medvidovic, editors, *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 868–871. ACM, 2011.

[405] G. Roşu, S. Eker, P. Lincoln, and J. Meseguer. Certifying and synthesizing membership equational proofs. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods, International Symposium of Formal Methods Europe, Pisa, Italy, September 8-14, 2003, Proceedings*, volume 2805 of *Lecture Notes in Computer Science*, pages 359–380. Springer, 2003.

[406] G. Roşu, C. Ellison, and W. Schulte. Matching logic: An alternative to Hoare/Floyd logic. In Johnson and Pavlovic [246], pages 142–162.

[407] G. Roşu, W. Schulte, and T. F. Şerbănuţă. Runtime verification of C memory safety. In S. Bensalem and D. Peled, editors, *Runtime Verification, 9th International Workshop, RV 2009, Grenoble, France, June 26-28, 2009. Selected Papers*, volume 5779 of *Lecture Notes in Computer Science*, pages 132–151. Springer, 2009.

[408] G. Roşu and T. F. Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[409] V. Rusu. Combining theorem proving and narrowing for rewriting-logic specifications. In G. Fraser and A. Gargantini, editors, *Tests and Proofs, 4th International Conference, TAP 2010, Málaga, Spain, July 1-2, 2010. Proceedings*, volume 6143 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 2010.

[410] V. Rusu and M. Clavel. Vérification d'invariants pour des systèmes spécifiés en logique de réécriture. In A. Schmitt, editor, *JFLA 2009, Vingtièmes Journées Francophones des Langages Applicatifs, Saint Quentin sur Isère, France, January 31-February 3, 2009, Proceedings*, volume 7.2 of *Studia Informatica Universalis*, pages 317–350, 2009.

[411] G. Santos-García, M. Palomino, and A. Verdejo. Rewriting logic using strategies for neural networks: An implementation in Maude. In J. M. Corchado, S. Rodríguez, J. Llinas, and J. M. Molina, editors, *Proceedings of the International Symposium on Distributed Computing and Artificial Intelligence, DCAI 2008, University of Salamanca, Spain, October 22-24, 2008*, volume 50 of *Advances in Soft Computing*, pages 424–433. Springer, 2009.

[412] R. Sasse, S. Escobar, C. Meadows, and J. Meseguer. Protocol analysis modulo combination of theories: A case study in Maude-NPA. In J. Cuéllar, J. Lopez, G. Barthe, and A. Pretschner, editors, *Security and Trust Management - 6th International Workshop, STM 2010, Athens, Greece, September 23-24, 2010, Revised Selected Papers*, volume 6710 of *Lecture Notes in Computer Science*, pages 163–178. Springer, 2011.

[413] R. Sasse and J. Meseguer. Java+ITP: A verification tool based on Hoare logic and algebraic semantics. In Denker and Talcott [139], pages 29–46.

[414] F. Schernhammer and B. Gramlich. Characterizing and proving operational termination of deterministic conditional term rewriting systems. *Journal of Logic and Algebraic Programming*, 79(7):659–688, 2010.

[415] F. Schernhammer and J. Meseguer. Incremental checking of well-founded recursive specifications modulo axioms. In Schneider-Kamp and Hanus [416], pages 5–16.

[416] P. Schneider-Kamp and M. Hanus, editors. *Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming, PPDP 2011, Odense, Denmark, July 20-22, 2011*. ACM, 2011.

[417] C. Schürmann and M.-O. Stehr. An executable formalization of the HOL/Nuprl connection in the metalogical framework Twelf. In M. Hermann and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 13th International Conference, LPAR 2006, Phnom Penh, Cambodia, November 13-17, 2006, Proceedings*, volume 4246 of *Lecture Notes in Computer Science*, pages 150–166. Springer, 2006.

[418] R. Segala. *Modelling and Verification of Randomized Distributed Real Time Systems*. PhD thesis, Massachusetts Institute of Technology, 1995.

[419] K. Sen, M. Viswanathan, and G. Agha. On statistical model checking of stochastic systems. In *17th conference on Computer Aided Verification (CAV'05)*, volume 3576 of *LNCS*, pages 266–280, Edinburgh, Scotland, 2005. Springer.

[420] K. Sen, M. Viswanathan, and G. A. Agha. VESTA: A statistical model-checker and analyzer for probabilistic systems. In *QEST 2005*, pages 251–252, 2005.

[421] T. F. Şerbănuţă. *A Rewriting Approach to Concurrent Programming Language Design and Semantics*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2010. `http://hdl.handle.net/2142/18252`.

[422] T. F. Şerbănuţă and G. Roşu. K-Maude: A rewriting based tool for semantics of programming languages. In Ölveczky [362], pages 104–122.

[423] T. F. Şerbănuţă, G. Roşu, and J. Meseguer. A rewriting logic approach to operational semantics. *Information and Computation*, 207(2):305–340, 2009.

[424] R. Shankesi, M. AlTurki, R. Sasse, C. A. Gunter, and J. Meseguer. Model-checking DoS amplification for VoIP session initiation. In Backes and Ning [36], pages 390–405.

[425] R. M. Smullyan. *Theory of Formal Systems*, volume 47 of *Annals of Mathematics Studies*. Princeton University Press, 1961.

[426] M. G. Sriram. Modelling protein functional domains in signal transduction using Maude. *Briefings in Bioinformatics*, 4(3):236–245, 2003.

[427] J. R. Stanton, Y. Amir, D. Hasse, G. Ateniese, Y. Kim, C. Nita-Rotaru, T. Schlossnagle, J. L. Schultz, and G. Tsudik. Secure group communication in asynchronous networks with failures: Integration and experiments. In *ICDCS*, pages 330–343, 2000.

[428] V. Stavridou, J. A. Goguen, A. Stevens, S. M. Eker, S. N. Aloneftis, and K. M. Hobley. Funnel and 2obj: Towards an integrated hardware design environment. In *Theorem Provers in Circuit Design (TPDC), Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, volume A-10 of *IFIP Transactions*, pages 197–223. North-Holland, 1992.

[429] L. J. Steggles and P. Kosiuczenko. A timed rewriting logic semantics for SDL: A case study of the alternating bit protocol. In Kirchner and Kirchner [262], pages 83–104.

[430] M.-O. Stehr. CINNI — A generic calculus of explicit substitutions and its application to λ-, ς- and π-calculi. In Futatsugi [200], pages 70–92.

[431] M.-O. Stehr. *Programming, Specification, and Interactive Theorem Proving — Towards a Unified Language based on Equational Logic, Rewriting Logic, and Type Theory.* PhD thesis, Fachbereich Informatik, Universität Hamburg, 2002.

[432] 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.

[433] 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.

[434] 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-Orientation to Formal Methods, Essays in Memory of Ole-Johan Dahl*, volume 2635 of *Lecture Notes in Computer Science*, pages 334–375. Springer, 2004.

[435] M.-O. Stehr, J. Meseguer, and P. C. Ölveczky. Rewriting logic as a unifying framework for Petri nets. In H. Ehrig, G. Juhás, J. Padberg, and G. Rozenberg, editors, *Unifying Petri Nets, Advances in Petri Nets*, volume 2128 of *Lecture Notes in Computer Science*, pages 250–303. Springer, 2001.

[436] J. G. Stell. Modelling term rewriting systems by sesqui-categories. Technical Report TR94-02, Keele University, 1994.

[437] W. J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton, 1994.

[438] M. Sun and J. Meseguer. Distributed real-time emulation of formally-defined patterns for safe medical device control. In Ölveczky [361], pages 158–177.

[439] M. Sun, J. Meseguer, and L. Sha. A formal pattern architecture for safe medical systems. In Ölveczky [362], pages 157–173.

[440] C. L. Talcott. Actor theories in rewriting logic. *Theoretical Computer Science*, 285(2):441–485, 2002.

[441] C. L. Talcott. Coordination models based on a formal model of distributed object reflection. In L. Brim and I. Linden, editors, *Proceedings of the First International Workshop on Methods and Tools for Coordinating Concurrent, Distributed and Mobile Systems, MTCoord 2005, Namur, Belgium, April 23, 2005*, volume 150(1) of *Electronic Notes in Theoretical Computer Science*, pages 143–157. Elsevier, 2006.

[442] C. L. Talcott. Formal executable models of cell signaling primitives. In T. Margaria and B. Steffen, editors, *Proceedings of the Leveraging Applications of Formal Methods Second International Symposium, ISoLA 2006, Paphos, Cyprus, November 15-19, 2006*, pages 298–302. IEEE, 2006.

[443] C. L. Talcott. Symbolic modeling of signal transduction in Pathway Logic. In L. F. Perrone, B. Lawson, J. Liu, and F. P. Wieland, editors, *Proceedings of the Winter Simulation Conference, WSC 2006, Monterey, California, USA, December 3-6, 2006*, pages 1656–1665. WSC, 2006.

[444] C. L. Talcott. Policy-based coordination in PAGODA: A case study. In G. Boella, M. Dastani, A. Omicini, L. van der Torre, I. Cerna, and I. Linden, editors, *Combined Proceedings of the Second International Workshop on Coordination and Organization, CoOrg 2006, and the Second International Workshop on Methods and Tools for Coordinating Concurrent, Distributed and Mobile Systems, MTCoord 2006, Bologna, Italy, June 13, 2006*, volume 181 of *Electronic Notes in Theoretical Computer Science*, pages 97–112. Elsevier, 2007.

[445] C. L. Talcott. Pathway logic. In M. Bernardo, P. Degano, and G. Zavattaro, editors, *Formal Methods for Computational Systems Biology, 8th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2008, Bertinoro, Italy, June 2-7, 2008, Advanced Lectures*, volume 5016 of *Lecture Notes in Computer Science*, pages 21–53. Springer, 2008.

[446] C. L. Talcott and D. L. Dill. The pathway logic assistant. In G. Plotkin, editor, *Proceedings of the Third International Workshop on Computational Methods in Systems Biology*, pages 228–239, 2005.

[447] C. L. Talcott and D. L. Dill. Multiple representations of biological processes. In C. Priami and G. D. Plotkin, editors, *Transactions on Computational Systems Biology VI*, volume 4220 of *Lecture Notes in Computer Science*, pages 221–245. Springer, 2006.

[448] 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, T. A. Jung, 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. World Scientific, January 2004.

[449] S. Tang. *Towards Secure Web Browsing*. PhD thesis, University of Illinois at Urbana-Champaign, 2011. 2011-05-25, `http://hdl.handle.net/2142/24307`.

[450] P. Thati and J. Meseguer. Complete symbolic reachability analysis using back-and-forth narrowing. *Theoretical Computer Science*, 366(1-2):163–179, 2006.

[451] 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 [205], pages 261–281.

[452] S. Thorvaldsen. Modeling and analysis of the OGDC wireless sensor network algorithm in Real-Time Maude. Master's thesis, Department of Informatics, University of Oslo, June 2005.

[453] C. Tinelli. A DPLL-based calculus for ground satisfiability modulo theories. In G. Ianni and S. Flesca, editors, *Proceedings of the 8th European Conference on Logics in Artificial Intelligence (Cosenza, Italy)*, volume 2424 of *Lecture Notes in Artificial Intelligence*, pages 308–319. Springer, 2002.

[454] A. Tiwari and C. L. Talcott. Analyzing a discrete model of aplysia central pattern generator. In M. Heiner and A. M. Uhrmacher, editors, *Computational Methods in Systems Biology, 6th International Conference, CMSB 2008, Rostock, Germany, October 12-15, 2008. Proceedings*, volume 5307 of *Lecture Notes in Computer Science*, pages 347–366. Springer, 2008.

[455] A. Tiwari, C. L. Talcott, M. Knapp, P. Lincoln, and K. Laderoute. Analyzing pathways using SAT-based approaches. In H. Anai, K. Horimoto, and T. Kutsia, editors, *Algebraic Biology, Second International Conference, AB 2007, Castle of Hagenberg, Austria, July 2-4, 2007, Proceedings*, volume 4545 of *Lecture Notes in Computer Science*, pages 155–169. Springer, 2007.

[456] X. Urbain. Modular & incremental automated termination proofs. *J. Autom. Reasoning*, 32(4):315–355, 2004.

[457] A. van Deursen, J. Heering, and P. Klint. *Language Prototyping: An Algebraic Specification Approach*. World Scientific, 1996.

[458] A. Verdejo. Building tools for LOTOS symbolic semantics in Maude. In D. Peled and M. Y. Vardi, editors, *Formal Techniques for Networked and Distributed Systems - FORTE 2002, 22nd IFIP WG 6.1 International Conference Houston, Texas, USA, November 11-14, 2002, Proceedings*, volume 2529 of *Lecture Notes in Computer Science*, pages 292–307. Springer, 2002.

[459] A. Verdejo. *Maude como Marco Semántico Ejecutable.* PhD thesis, Facultad de Informática, Universidad Complutense de Madrid, Spain, Mar. 2003.

[460] A. Verdejo and N. Martí-Oliet. Two case studies of semantics execution in Maude: CCS and LOTOS. *Formal Methods in System Design*, 27(1-2):113–172, 2005.

[461] A. Verdejo and N. Martí-Oliet. Executable structural operational semantics in Maude. *Journal of Logic and Algebraic Programming*, 67(1-2):226–293, 2006.

[462] A. Verdejo, I. Pita, and N. Martí-Oliet. Specification and verification of the tree identify protocol of IEEE 1394 in rewriting logic. *Formal Aspects of Computing*, 14(3):228–246, 2003.

[463] P. Viry. *La Réécriture Concurrente.* PhD thesis, Université de Nancy I, 1992.

[464] P. Viry. Rewriting: An effective model of concurrency. In C. Halatsis, D. G. 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.

[465] P. Viry. Input/output for ELAN. In Meseguer [317], pages 51–64.

[466] P. Viry. Adventures in sequent calculus modulo equations. In Kirchner and Kirchner [262], pages 21–32.

[467] P. Viry. Equational rules for rewriting logic. *Theoretical Computer Science*, 285(2):487–517, 2002.

[468] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in Stratego/XT 0.9. In C. Lengauer, D. S. Batory, C. Consel, and M. Odersky, editors, *Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Springer, 2004.

[469] L. Wang, Z. Kalbarczyk, and R. K. Iyer. Formalizing system behavior for evaluating a system hang detector. In *Proceedings of the 27th IEEE Symposium on Reliable Distributed Systems, SRDS 2008, Napoli, Italy, October 6-8, 2008*, pages 269–278. IEEE, 2008.

[470] I. Wehrman, D. Kitchin, W. R. Cook, and J. Misra. A timed semantics of Orc. *Theor. Comput. Sci.*, 402(2-3):234–248, 2008.

[471] 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.* Springer, 1999.

[472] M. Wirsing, G. Denker, C. L. Talcott, A. Poggio, and L. Briesemeister. A rewriting logic framework for soft constraints. In Denker and Talcott [139], pages 181–197.

[473] M. Wirsing and A. Knapp. A formal approach to object-oriented software engineering. In Meseguer [317], pages 322–360.

[474] M. Wirsing and A. Knapp. A formal approach to object-oriented software engineering. *Theoretical Computer Science*, 285(2):519–560, 2002.

[475] H. L. S. Younes and R. G. Simmons. Statistical probabilistic model checking with a focus on time-bounded properties. *Inf. Comput.*, 204(9):1368–1409, 2006.

[476] M. Zhang, K. Ogata, and M. Nakamura. Specification translation of state machines from equational theories into rewrite theories. In Dong and Zhu [143], pages 678–693.