# Specifying, programming, and verifying in Maude

## Some applications to Model-Driven Engineering and Graph Rewriting (material based on a course by Narciso Martí-Oliet, Universidad Complutense de Madrid)

Artur Boronat

Dept. of Computer Science, Univ. of Leicester, UK
ab373@mcs.le.ac.uk

UoL, PhD Seminar, 23 October 2007

# Contents


1. Introduction
2. Functional modules
   - Many-sorted equational specifications
   - Order-sorted equational specifications
   - Equational attributes
   - Membership equational logic specifications
3. System modules
   - Rewriting logic

# Contents

5. Applications to Model-Driven Engineering
    - OO programming in Maude
    - Metamodel conformance through memberships
    - Graph rewriting
6. Reflection
    - Maude's metalevel: descent functions and internal strategies
    - Metaprogramming
        - Compilation of model transformation definition
        - Execution of graph-based model transformations

# Introduction

http://maude.cs.uiuc.edu

- Maude is a high-level language and high-performance system.
- It supports both equational and rewriting logic computation.
- Membership equational logic improves order-sorted algebra.
- Rewriting logic is a logic of concurrent change.
- It is a flexible and general semantic framework for giving semantics to a wide range of languages and models of concurrency.
- It is also a good logical framework, i.e., a metalogic in which many other logics can be naturally represented and implemented.
- Moreover, rewriting logic is reflective.
- This makes possible many advanced metaprogramming and metalanguage applications.

# Many-sorted equational specifications

- Algebraic specifications are used to declare different kinds of data together with the operations that act upon them.
- It is useful to distinguish two kinds of operations:
  - constructors, used to construct or generate the data, and
  - the remaining operations, which in turn can also be classified as modifiers or observers.
- The behavior of operations is described by means of (possibly conditional) equations.
- We start with the simplest many-sorted equational specifications and incrementally add more sophisticated features.

# Signatures

- The first thing a specification needs to declare are the types (or sorts) of the data being defined and the corresponding operations.
- A many-sorted signature $(S, \Sigma)$ consists of
  - a sort set $S$, and
  - an $S^* \times S$-sorted family

    $$\Sigma = \{\Sigma_{\bar{s},s} \mid \bar{s} \in S^*, s \in S\}$$

    of sets of operation symbols.
- When $f \in \Sigma_{\bar{s},s}$, we write $f : \bar{s} \to s$ and say that $f$ has rank $\langle \bar{s}, s \rangle$, arity (or argument sorts) $\bar{s}$, and coarity (or value sort, or range sort) $s$.
- The symbol $\varepsilon$ denotes the empty sequence in $S^*$.

# Terms

- With the declared operations we can construct terms to denote the data being specified.
- Terms are typed and can have variables.
- Given a many-sorted signature $(S, \Sigma)$ and an $S$-sorted family $X = \{X_s \mid s \in S\}$ of variables, the $S$-sorted set of terms

$$\mathcal{T}_\Sigma(X) = \{\mathcal{T}_{\Sigma,s}(X) \mid s \in S\}$$

is inductively defined by the following conditions:

1. $X_s \subseteq \mathcal{T}_{\Sigma,s}(X)$ for $s \in S$;
2. $\Sigma_{\varepsilon,s} \subseteq \mathcal{T}_{\Sigma,s}(X)$ for $s \in S$;
3. If $f \in \Sigma_{\bar{s},s}$ and $t_i \in \mathcal{T}_{\Sigma,s_i}(X)$ $(i = 1, \ldots, n)$, where $\bar{s} = s_1 \ldots s_n \neq \varepsilon$, then $f(t_1, \ldots, t_n) \in \mathcal{T}_{\Sigma,s}(X)$.

# Equations

- A Σ-equation is an expression

$$(\overline{x} : \overline{s})\, l = r$$

  where
  - $\overline{x} : \overline{s}$ is a (finite) set of variables, and
  - $l$ and $r$ are terms in $\mathcal{T}_{\Sigma,s}(\overline{x} : \overline{s})$ for some sort $s$.
- A conditional Σ-equation is an expression

$$(\overline{x} : \overline{s})\, l = r \ \text{ if } \ u_1 = v_1 \ \wedge \ldots \wedge \ u_n = v_n$$

  where $(\overline{x} : \overline{s})\, l = r$ and $(\overline{x} : \overline{s})\, u_i = v_i$ $(i = 1, \ldots, n)$ are Σ-equations.
- A many-sorted specification $(S, \Sigma, E)$ consists of:
  - a signature $(S, \Sigma)$, and
  - a set $E$ of (conditional) Σ-equations.

# Semantics

- A many-sorted $(S, \Sigma)$-algebra **A** consists of:
  - a carrier set $A_s$ for each sort $s \in S$, and
  - a function $A_f^{\overline{s},s} : A_{\overline{s}} \to A_s$ for each operation symbol $f \in \Sigma_{\overline{s},s}$.
- The meaning $[\![t]\!]_{\mathbf{A}}$ of a term $t$ in an algebra **A** is inductively defined.
- An algebra **A** satisfies an equation $(\overline{x} : \overline{s})\, l = r$ when both terms have the same meaning: $[\![l]\!]_{\mathbf{A}} = [\![r]\!]_{\mathbf{A}}$.
- An algebra **A** satisfies a conditional equation

$$(\overline{x} : \overline{s})\, l = r \ \text{ if } \ u_1 = v_1 \ \wedge \ldots \wedge \ u_n = v_n$$

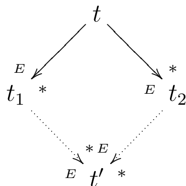
  when satisfaction of all the conditions $(\overline{x} : \overline{s})\, u_i = v_i$ $(i = 1, \ldots, n)$ implies satisfaction of $(\overline{x} : \overline{s})\, l = r$.

# Maude functional modules

```
fmod BOOLEAN is
  sort Bool .
  op true : -> Bool [ctor] .
  op false : -> Bool [ctor] .

  op not_ : Bool -> Bool .
  op _and_ : Bool Bool -> Bool .
  op _or_ : Bool Bool -> Bool .

  var A : Bool .

  eq not true = false .
  eq not false = true .
  eq true and A = A .
  eq false and A = false .
  eq true or A = true .
  eq false or A = A .
endfm
```

# Confluence and termination

- A set of equations $E$ is confluent (or Church-Rosser) when any two rewritings of a term can always be unified by further rewriting: if $t \rightarrow_E^* t_1$ and $t \rightarrow_E^* t_2$, then there exists a term $t'$ such that $t_1 \rightarrow_E^* t'$ and $t_2 \rightarrow_E^* t'$.




- A set of equations $E$ is terminating when there is no infinite sequence of rewriting steps $t_0 \rightarrow_E t_1 \rightarrow_E t_2 \rightarrow_E \ldots$

# Confluence and termination

- If $E$ is both confluent and terminating, a term $t$ can be reduced to a unique canonical form $t{\downarrow_E}$, that is, to a term that can no longer be rewritten.
- Therefore, in order to check semantic equality of two terms $t = t'$, it is enough to check that their respective canonical forms are equal, $t{\downarrow_E} = t'{\downarrow_E}$, but, since canonical forms cannot be rewritten anymore, the last equality is just syntactic coincidence: $t{\downarrow_E} \equiv t'{\downarrow_E}$.
- Functional modules in Maude are assumed to be confluent and terminating, and their operational semantics is equational simplification, that is, rewriting of terms until a canonical form is obtained.

# Natural numbers

```
fmod UNARY-NAT is
  sort Nat .

  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat .

  vars N M : Nat .

  eq 0 + N = N .
  eq s(M) + N = s(M + N) .
endfm
```

- Can we add the equation

  ```
  eq M + N = N + M .
  ```

  expressing commutativity of addition?

# Order-sorted equational specifications

- There are operations that are not defined for some values, like division on natural numbers.
- We can often avoid the possibility of considering partial functions by extending many-sorted equational logic to order-sorted equational logic.
- We can define subsorts corresponding to the domain of definition of a function, whenever such subsorts can be specified by means of constructors.
- An order-sorted signature adds a partial order relation to the set of sorts $S$, such that $s \leqslant s'$ is interpreted semantically by the subset inclusion $A_s \subseteq A_{s'}$ between the corresponding carrier sets in the algebras.
- Moreover, operations can be overloaded:
  - subsort overloading: addition both on natural numbers and on integers,
  - ad-hoc overloading: the same symbol can be used in unrelated sorts.

# Lists of natural numbers

```
fmod NAT-LIST-CONS is
  protecting NAT .

  sorts NeList List .
  subsort NeList < List .

  op [] : -> List [ctor] .                    *** empty list
  op _:_ : Nat List -> NeList [ctor] .        *** cons
  op tail : NeList -> List .
  op head : NeList -> Nat .
  op _++_ : List List -> List .               *** concatenation
  op length : List -> Nat .
  op reverse : List -> List .
  op take_from_ : Nat List -> List .
  op throw_from_ : Nat List -> List .

  vars N M : Nat .
  vars L L' : List .
```

# Lists of natural numbers

```
    eq tail(N : L) = L .
    eq head(N : L) = N .
    eq [] ++ L = L .
    eq (N : L) ++ L' = N : (L ++ L') .
    eq length([]) = 0 .
    eq length(N : L) = 1 + length(L) .
    eq reverse([]) = [] .
    eq reverse(N : L) = reverse(L) ++ (N : []) .

    eq take 0 from L = [] .
    eq take N from [] = [] .
    eq take s(N) from (M : L) = M : take N from L .
    eq throw 0 from L = L .
    eq throw N from [] = [] .
    eq throw s(N) from (M : L) = throw N from L .
endfm
```

# Equational attributes

- Equational attributes are a means of declaring certain kinds of equational axioms in a way that allows Maude to use these equations efficiently in a built-in way.
- Currently Maude supports the following equational attributes:
  - `assoc` (associativity),
  - `comm` (commutativity),
  - `idem` (idempotency),
  - `id:` ⟨*Term*⟩ (identity, with the corresponding term for the identity element),
  - `left id:` ⟨*Term*⟩ (left identity, with the corresponding term for the left identity element), and
  - `right id:` ⟨*Term*⟩ (right identity, with the corresponding term for the right identity element).
- These attributes are only allowed for binary operators satisfying some appropriate requirements that depend on the attributes.

# Matching and simplification modulo

- In the Maude implementation, rewriting modulo $A$ is accomplished by using a matching modulo $A$ algorithm.
- More precisely, given an equational theory $A$, a term $t$ (corresponding to the lefthand side of an equation) and a subject term $u$, we say that $t$ matches $u$ modulo $A$ if there is a substitution $\sigma$ such that $\sigma(t) =_A u$, that is, $\sigma(t)$ and $u$ are equal modulo the equational theory $A$.
- Given an equational theory $A = \cup_i A_{f_i}$ corresponding to all the attributes declared in different binary operators, Maude synthesizes a combined matching algorithm for the theory $A$, and does equational simplification modulo the axioms $A$.

# Basic natural numbers

```
fmod BASIC-NAT is
  sort Nat .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat .
  op max : Nat Nat -> Nat .

  vars N M : Nat .
  eq 0 + N = N .
  eq s(M) + N = s(M + N) .
  eq max(0, M) = M .
  eq max(N, 0) = N .
  eq max(s(N), s(M)) = s(max(N, M)) .
endfm
```

# Nonempty lists

```
fmod NAT-NE-LISTS is
  protecting BASIC-NAT .

  sort NeList .
  subsort Nat < NeList .
  op __ : NeList NeList -> NeList [ctor assoc] .
  op length : NeList -> Nat .
  op reverse : NeList -> NeList .

  var N : Nat .
  var L L' : NeList .

  eq length(N) = s(0) .
  eq length(L L') = length(L) + length(L') .
  eq reverse(N) = N .
  eq reverse(L L') = reverse(L') reverse(L) .
endfm
```

# Multisets

```
fmod NAT-MSETS is
  protecting BASIC-NAT .
  sort Mset .
  subsorts Nat < Mset .
  op empty-mset : -> Mset [ctor] .
  op __ : Mset Mset -> Mset [ctor assoc comm id: empty-mset] .
  op size : Mset -> Nat .

  vars N : Nat .
  var  S : Mset .

  eq size(empty-mset) = 0 .
  eq size(N S) = s(0) + size(S) .
endfm
```

# Membership equational logic specifications

- In order-sorted equational specifications, subsorts must be defined by means of constructors, but it is not possible to have a subsort of sorted lists, for example, defined by a property over lists.
- Membership equational logic solves problems of this kind by introducing sorts as predicates and allowing subsort definition by means of conditions involving equations and/or sort predicates.

# Membership equational logic

- A signature in membership equational logic is a triple $\Omega = (K, \Sigma, S)$ where $K$ is a set of kinds, $(K, \Sigma)$ is a many-kinded signature, and $S = \{S_k\}_{k \in K}$ is a $K$-kinded set of sorts.
- An $\Omega$-algebra is then a $(K, \Sigma)$-algebra **A** together with the assignment to each sort $s \in S_k$ of a subset $A_s \subseteq A_k$.
- Atomic formulas are either $\Sigma$-equations, or membership assertions of the form $t : s$, where the term $t$ has kind $k$ and $s \in S_k$.
- General sentences are Horn clauses on these atomic formulas, quantified by finite sets of $K$-kinded variables.

$$(\forall X)\ \ t = t' \ \text{ if } \ (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j)$$

$$(\forall X)\ \ t : s \ \text{ if } \ (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j).$$

# Membership equational logic in Maude

- Maude functional modules are membership equational specifications and their semantics is given by the corresponding initial membership algebra in the class of algebras satisfying the specification.
- Maude does automatic kind inference from the sorts declared by the user and their subsort relations.
- Kinds are not declared explicitly, and correspond to the connected components of the subsort relation.
- The kind corresponding to a sort `s` is denoted `[s]`.
- If `NzNat` $<$ `Nat`, then `[NzNat]` $=$ `[Nat]`.

# Membership equational logic in Maude

- An operator declaration like

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

  can be understood as a declaration at the kind level

  ```
  op _div_ : [Nat] [Nat] -> [Nat] .
  ```

  together with the conditional membership axiom

  ```
  cmb N div M : Nat if N : Nat and M : NzNat .
  ```

- A subsort declaration `NzNat < Nat` can be understood as the conditional membership axiom

  ```
  cmb N : Nat if N : NzNat .
  ```

# Palindromes

```
fmod PALINDROME is
  protecting QID .
  sorts Word Pal .
  subsort Qid < Pal < Word .

  op nil : -> Pal [ctor] .
  op __ : Word Word -> Word [ctor assoc id: nil] .

  var I : Qid .
  var P : Pal .

  mb I P I : Pal  .
endfm
```

# Rewriting logic

- We arrive at the main idea behind rewriting logic by dropping symmetry and the equational interpretation of rules.
- We interpret a rule $t \to t'$ computationally as a local concurrent transition of a system, and logically as an inference step from formulas of type $t$ to formulas of type $t'$.
- Rewriting logic is a logic of becoming or change, that allows us to specify the dynamic aspects of systems.
- Representation of systems in rewriting logic:
  - The static part is specified as an equational theory.
  - The dynamics is specified by means of possibly conditional rules that rewrite terms, representing parts of the system, into others.
  - The rules need only specify the part of the system that actually changes.

# Rewriting logic

- A rewriting logic signature is an equational specification $(\Omega, E)$ that makes explicit the set of equations in order to emphasize that rewriting will operate on congruence classes of terms modulo $E$.
- Sentences are rewrites of the form $[t]_E \longrightarrow [t']_E$.
- A rewriting logic specification $\mathcal{R} = (\Omega, E, L, R)$ consists of:
  - a signature $(\Omega, E)$,
  - a set $L$ of labels, and
  - a set $R$ of labelled rewrite rules $\quad r : [t]_E \longrightarrow [t']_E$ where $r$ is a label and $[t]_E, [t']_E$ are congruence classes of terms in $\mathcal{T}_{\Omega,E}(X)$.
- The most general form of a rewrite rule is conditional:

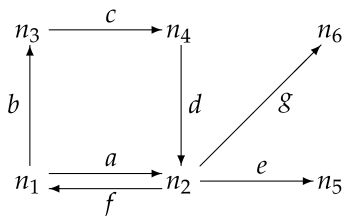
$$r : t \to t' \ \ \mathit{if} \ \ (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j) \wedge (\bigwedge_k p_k \to q_k)$$

# System modules

- System modules in Maude correspond to rewrite theories in rewriting logic.
- A rewrite theory has both rules and equations, so that rewriting is performed modulo such equations.
- The equations are divided into
  - a set $A$ of structural axioms, for which matching algorithms exist in Maude, and
  - a set $E$ of equations that are Church-Rosser and terminating modulo $A$;

  that is, the equational part must be equivalent to a functional module.

# Transition systems




```
mod A-TRANSITION-SYSTEM is
  sort State .
  ops n1 n2 n3 n4 n5 n6 : -> State [ctor] .
  rl [a] : n1 => n2 .     rl [b] : n1 => n3 .
  rl [c] : n3 => n4 .     rl [d] : n4 => n2 .
  rl [e] : n2 => n5 .     rl [f] : n2 => n1 .
  rl [g] : n2 => n6 .
endm
```

- not confluent: there are, for example, two transitions out of `n2` that are not joinable
- not terminating: there are cycles creating infinite computations

# Object-oriented systems

- An object in a given state is represented as a term

  `< O : C | p1 : v1,..., pn : vn >`

  where `O` is the object's name, belonging to a set `Oid` of object identifiers, `C` is its class, the `pi`'s are the names of the object's properties, and the `vi`'s are their corresponding values.
- Messages are defined by the user for each application.
- In a concurrent object-oriented system the concurrent state, which is called a configuration, has the structure of a multiset made up of objects and messages that evolves by concurrent rewriting (modulo the multiset structural axioms) using rules that describe the effects of communication events between some objects and messages.

# Object-oriented systems

```
fmod OO is
  ..
  sorts Object Oid Cid Property PropertySet Message
    ObjectCollection Configuration .

  subsort Property < PropertySet .
  op noneProperty : -> Property .
  op _,_ : PropertySet PropertySet -> PropertySet
    [assoc comm id: none] .

  op <_:_|_> : Oid Cid PropertySet -> Object .

  subsorts Object Message < ObjectCollection .
  op none : -> ObjectCollection .
  op __ : ObjectCollection ObjectCollection -> ObjectCollection
    [assoc comm id: none] .
  op <<_>> : ObjectCollection -> Configuration .
  ..
endfm
```

# Object-oriented rules - in Core Maude

- General form of rules in object-oriented systems:

$$
\begin{aligned}
& M_1 \ldots M_n \, \langle O_1 : F_1 \mid props_1, mps_1 \rangle \ldots \langle O_m : F_m \mid props_m, mps_m \rangle \\
& \longrightarrow \langle O_{i_1} : F'_{i_1} \mid props'_{i_1}, mps_1 \rangle \ldots \langle O_{i_k} : F'_{i_k} \mid props'_{i_k}, mps_m \rangle \\
& \qquad \langle Q_1 : D_1 \mid props''_1 \rangle \ldots \langle Q_p : D_p \mid props''_p \rangle \\
& \qquad M'_1 \ldots M'_q \\
& \mathit{if}\ C
\end{aligned}
$$

- By convention, the only object properties made explicit in a rule are those relevant for that rule:
  - the properties mentioned only in the lefthand side of the rule are deleted,
  - the original values of properties mentioned only in the righthand side of the rule do not matter, and
  - all properties not explicitly mentioned are left unchanged.

# Graph rewriting

- Object identifiers:

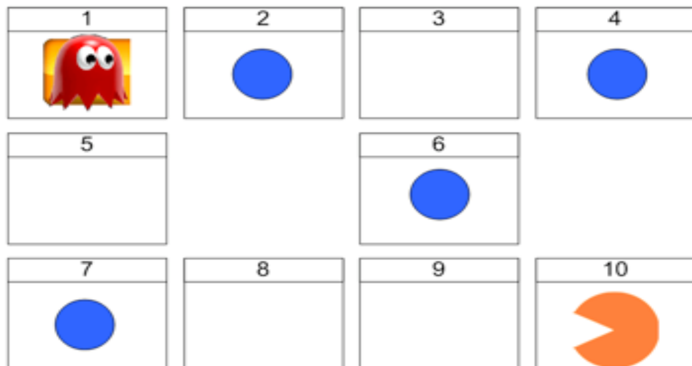
```
sorts Oid OidSet .
subsorts Qid < Oid < OidSet .
op none : -> OidSet .
op __ : OidSet OidSet -> OidSet [assoc comm id: none] .
```

- Two types of properties:
  - attributes: basic data types
  - references: object types (object identifier types)
- Graph rewriting given by term rewriting modulo AC (due to the constructor __ for the Configuration sort.)

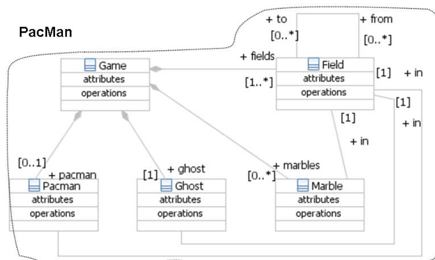
# Model-Driven Engineering

- From objects to models $\widetilde{M}$:
  - Models as collections of objects.
  - (Hierarchical) graphs.
- From object types to model types $\mathcal{M}$:
  - Model types $\mathcal{M}$ are defined as metadata: metamodels
  - Models as first-class citizens.
  - Type graphs.
  - Metamodel conformance relation $\widetilde{M} : \mathcal{M}$: a model $\widetilde{M}$ is syntactically correct wrt its model type $\mathcal{M}$
- From object manipulation to model manipulation:
  - Model transformations.
  - Production rules.

# PacMan Game

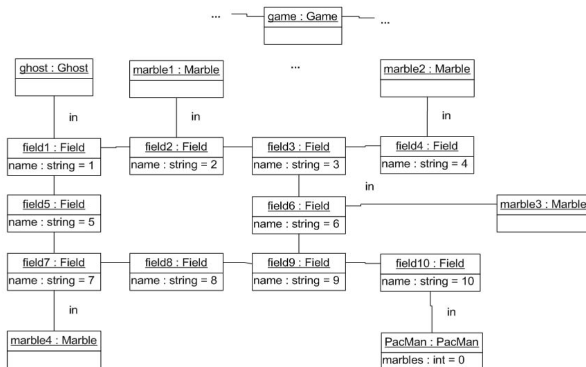


# PacMan Game: Metamodel/Type Graph




```
mod PACMAN is
  ..
  sort ModelType .

  sorts Game Pacman Ghost Marble Field .
  op Game : -> Game .
  ..
  op fields : OidSet -> Property .
  op pacman : [Oid] -> Property .
  op ghost : Oid -> Property .
  ..
endm
```

# PacMan Game: States/Typed Graphs




```
<<
  < '0 : Game | fields : 'f1 .. 'f10 , pacman : 'p, ghost : 'g, marbles : .. >
  < 'g : Ghost | in : 'f1 >
  < 'p : Pacman | in : 'f10 >
  < 'f1 : Field | name : "1", to : 'f2 'f5 >
  < 'f10 : Field | name : "10", to : 'f9 >
  ..
>>
```

# PacMan Game: Metamodel conformance

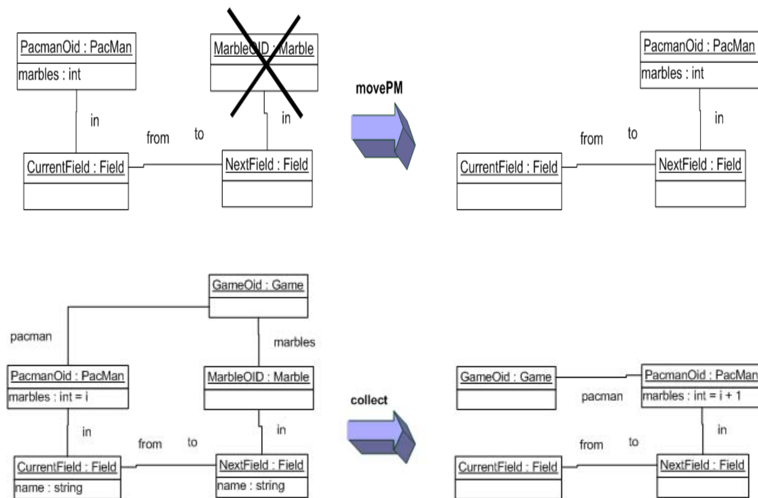
- The model type $\mathcal{M}$ is represented as a sort `ModelType`, whose semantics is provided by means of a membership.

```
mod PACMAN is
  ..
  var conf : Configuration .
  sort ModelType .

  cmb conf:Configuration : ModelType
  if well-defined(conf:Configuration) .

  op well-defined : Configuration -> Bool .
  ..
endm
```

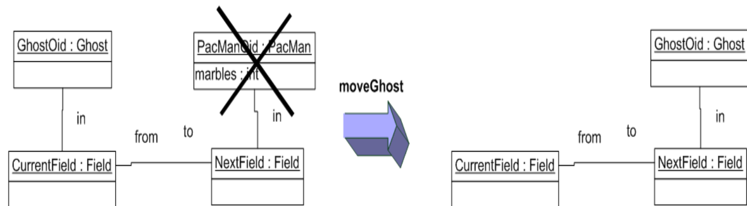
- Conformance checking:

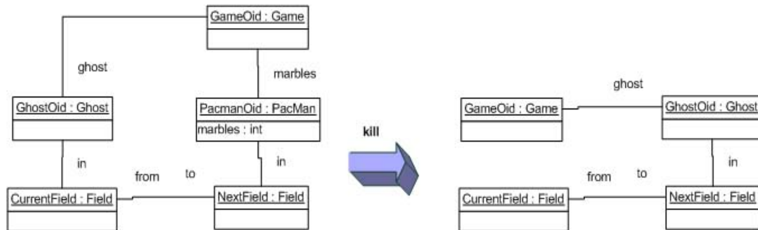
```
red <<
  < '0 : Game | fields : 'f0 .. 'f10 , pacman : 'p, ghost : 'g, marbles : .. >
  < 'p : Pacman | in : 'f1 >
  < 'g : Ghost | in : 'f10 >
  ..
>> :: ModelType .
```

# PacMan Game: Model Transformations/Production Rules

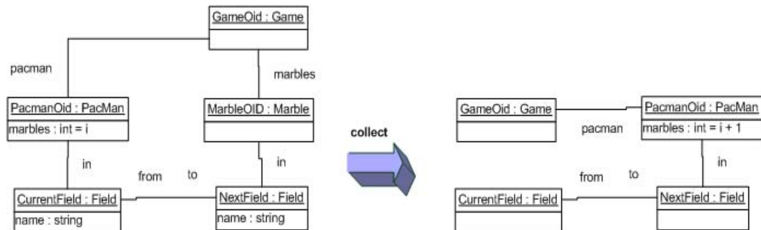


# PacMan Game: Model Transformations/Production Rules



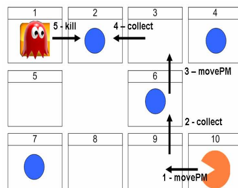
# PacMan Game: Model Transformations/Production Rules



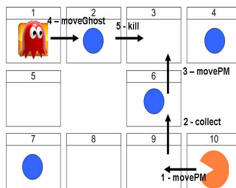

```
..
crl [collect] :
  < GameOid : Game | marbles : GameMarbles:OidSet, GamePS >
  < PacmanOid : Pacman | in = CurrentFieldOid, marbles" = PacmanMarbles:Int,
  PacmanPS >
  < CurrentFieldOid : Field | to = CurrentFieldTo, CurrentFieldPS >
  < NextFieldOid : Field | from = NextFieldFrom, MarbleMPS >
  < MarbleOid : Marble | in = NextFieldOid, MarbleMPS >
=>
  < GameOid : Game | marbles = (GameMarbles:OidSet -> excluding( MarbleOid )),
  GameMPS >
  < PacmanOid : Pacman | in = NextFieldOid, marbles = PacmanMarbles:Int + 1,
  PacmanPS >
  < CurrentFieldOid : Field | to = CurrentFieldTo, CurrentFieldPS >
  < NextFieldOid : Marble | in = NextFieldOid, MarbleMPS >
if (CurrentFieldTo -> includes ( NextFieldOid )) and
  (NextFieldFrom -> includes ( CurrentFieldOid )) .
```

# PacMan Game: simulation

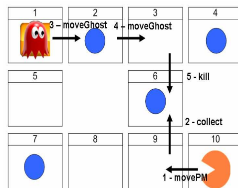
```
search [3] in PACMAN : model =>+
  <<
    < GameOid:Oid : Game |
        (pacman : O:[Oid]),
        GameMPS:MetaPropertySet >
    ObjCol:ObjectCollection
  >> .
```



Solution 1



Solution 2



Solution 3

# Applications to MDE: metamodeling

- Model type $\mathcal{M}$ is represented as the sort `ModelType` in the PACMAN theory.
- The semantics $[\![\mathcal{M}]\!]_{MOF}$ of the model type $\mathcal{M}$ is given as follows

$$[\![\mathcal{M}]\!]_{MOF} = \mathcal{T}_{PACMAN,ModelType}.$$

- Model definition $\widetilde{M}$ is a `ModelType` term, such that

$$\widetilde{M} \in \mathcal{T}_{PACMAN,ModelType}.$$

- Metamodel conformance

$$\widetilde{M} : \mathcal{M} \Leftrightarrow \widetilde{M} \in \mathcal{T}_{PACMAN,ModelType}.$$

- Model transformations and graph rewriting by means of term rewriting modulo AC.

# Reflection

- Rewriting logic is reflective, because there is a finitely presented rewrite theory $\mathcal{U}$ that is universal in the sense that:
  - we can represent any finitely presented rewrite theory $\mathcal{R}$ and any terms $t, t'$ in $\mathcal{R}$ as terms $\overline{\mathcal{R}}$ and $\bar{t}, \overline{t'}$ in $\mathcal{U}$,
  - then we have the following equivalence

$$\mathcal{R} \vdash t \longrightarrow t' \iff \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \bar{t} \rangle \longrightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle.$$

- Since $\mathcal{U}$ is representable in itself, we get a reflective tower

$$\begin{array}{c} \mathcal{R} \vdash t \rightarrow t' \\ \Updownarrow \\ \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \bar{t} \rangle \rightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle \\ \Updownarrow \\ \mathcal{U} \vdash \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \bar{t} \rangle} \rangle \rightarrow \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \overline{t'} \rangle} \rangle \\ \vdots \end{array}$$

# Maude's metalevel

In Maude, key functionality of the universal theory $\mathcal{U}$ has been efficiently implemented in the functional module `META-LEVEL`:

- Maude terms are reified as elements of a data type `Term` in the module `META-TERM`;
- Maude modules are reified as terms in a data type `Module` in the module `META-MODULE`;
- operations `upModule`, `upTerm`, `downTerm`, and others allow moving between reflection levels;
- the process of reducing a term to canonical form using Maude's `reduce` command is metarepresented by a built-in function `metaReduce`;
- the processes of rewriting a term in a system module using Maude's `rewrite` and `frewrite` commands are metarepresented by built-in functions `metaRewrite` and `metaFrewrite`;

# Maude's metalevel

- the process of applying a rule of a system module at the top of a term is metarepresented by a built-in function `metaApply`;
- the process of applying a rule of a system module at any position of a term is metarepresented by a built-in function `metaXapply`;
- the process of matching two terms is reified by built-in functions `metaMatch` and `metaXmatch`;
- the process of searching for a term satisfying some conditions starting in an initial term is reified by built-in functions `metaSearch` and `metaSearchPath`; and
- parsing and pretty-printing of a term in a module, as well as key sort operations such as comparing sorts in the subsort ordering of a signature, are also metarepresented by corresponding built-in functions.

# Representing terms

```
sorts Constant Variable Term .
subsorts Constant Variable < Qid Term .

op <Qids> : -> Constant [special (...)] .
op <Qids> : -> Variable [special (...)] .

sort TermList .
subsort Term < TermList .
op _,_ : TermList TermList -> TermList
    [ctor assoc gather (e E) prec 120] .

op _[_] : Qid TermList -> Term [ctor] .
```

# Representing terms: Example

- Usual term: `c (q M:State)`
- Metarepresentation

```
'__['c.Item, '__['q.Coin, 'M:State]]
```

- Meta-metarepresentation

```
'_`[_`]['`'__.Qid,
        '_`,_['`'c.Item.Constant,
              '_`[_`]['`'__.Qid,
                      '_`,_['`'q.Coin.Constant,
                            '`'M:State.Variable]]]]
```

# Representing modules

```
sorts FModule SModule FTheory STheory Module .
subsorts FModule < SModule < Module .
subsorts FTheory < STheory < Module .
sort Header .
subsort Qid < Header .
op _{_}  : Qid ParameterDeclList -> Header [ctor] .
op fmod_is_sorts_.____endfm : Header ImportList SortSet
     SubsortDeclSet OpDeclSet MembAxSet EquationSet -> FModule
     [ctor gather (& & & & & & &)] .
op mod_is_sorts_._____endm : Header ImportList SortSet
     SubsortDeclSet OpDeclSet MembAxSet EquationSet RuleSet
     -> SModule [ctor gather (& & & & & & & &)] .
op fth_is_sorts_.____endfth : Qid ImportList SortSet SubsortDeclSet
     OpDeclSet MembAxSet EquationSet -> FTheory
     [ctor gather (& & & & & & &)] .
op th_is_sorts_._____endth : Qid ImportList SortSet SubsortDeclSet
     OpDeclSet MembAxSet EquationSet RuleSet -> STheory
     [ctor gather (& & & & & & & &)] .
```

# Representing modules: Example at the object level

```
fmod VENDING-MACHINE-SIGNATURE is
  sorts Coin Item State .
  subsorts Coin Item < State .
  op __ : State State -> State [assoc comm] .
  op $ : -> Coin [format (r! o)] .
  op q : -> Coin [format (r! o)] .
  op a : -> Item [format (b! o)] .
  op c : -> Item [format (b! o)] .
endfm
```

# Representing modules: Example at the metalevel

```
fmod 'VENDING-MACHINE-SIGNATURE is
  nil
  sorts 'Coin ; 'Item ; 'State .
  subsort 'Coin < 'State .
  subsort 'Item < 'State .
  op '__ : 'State 'State -> 'State [assoc comm] .
  op 'a : nil -> 'Item [format('b! 'o)] .
  op 'c : nil -> 'Item [format('b! 'o)] .
  op '$ : nil -> 'Coin [format('r! 'o)] .
  op 'q : nil -> 'Coin [format('r! 'o)] .
  none
  none
endfm
```

# Representing modules: Example at the object level

```
mod VENDING-MACHINE is
  including VENDING-MACHINE-SIGNATURE .
  var M : State .
  rl [add-q] : M => M q .
  rl [add-$] : M => M $ .
  rl [buy-c] : $ => c .
  rl [buy-a] : $ => a q .
  rl [change] : q q q q => $ .
endm
```

# Representing modules: Example at the metalevel

```
mod 'VENDING-MACHINE is
  including 'VENDING-MACHINE-SIGNATURE .
  sorts none .
  none
  none
  none
  none
  rl 'M:State => '__['M:State, 'q.Coin] [label('add-q)] .
  rl 'M:State => '__['M:State, '$.Coin] [label('add-$)] .
  rl '$.Coin => 'c.Item [label('buy-c)] .
  rl '$.Coin => '__['a.Item, 'q.Coin] [label('buy-a)] .
  rl '__['q.Coin,'q.Coin,'q.Coin,'q.Coin]
    => '$.Coin [label('change)] .
endm
```

# Moving between levels

```
op upModule : Qid Bool ~> Module [special (...)] .
op upSorts : Qid Bool ~> SortSet [special (...)] .
op upSubsortDecls : Qid Bool ~> SubsortDeclSet [special (...)] .
op upOpDecls : Qid Bool ~> OpDeclSet [special (...)] .
op upMbs : Qid Bool ~> MembAxSet [special (...)] .
op upEqs : Qid Bool ~> EquationSet [special (...)] .
op upRls : Qid Bool ~> RuleSet [special (...)] .
```

In all these (partial) operations

- The first argument is expected to be a module name.
- The second argument is a Boolean, indicating whether we are interested also in the imported modules or not.

# Moving between levels: Example

```
Maude> reduce in META-LEVEL : upEqs('VENDING-MACHINE, true) .
result EquationSet:
  eq '_and_['true.Bool, 'A:Bool] = 'A:Bool [none] .
  eq '_and_['A:Bool, 'A:Bool] = 'A:Bool [none] .
  eq '_and_['A:Bool, '_xor_['B:Bool, 'C:Bool]]
    = '_xor_['_and_['A:Bool, 'B:Bool], '_and_['A:Bool, 'C:Bool]]
    [none] .
  eq '_and_['false.Bool, 'A:Bool] = 'false.Bool [none] .
  eq '_or_['A:Bool,'B:Bool]
    = '_xor_['_and_['A:Bool, 'B:Bool],'_xor_['A:Bool, 'B:Bool]]
    [none] .
  eq '_xor_['A:Bool, 'A:Bool] = 'false.Bool [none] .
  eq '_xor_['false.Bool, 'A:Bool] = 'A:Bool [none] .
  eq 'not_['A:Bool] = '_xor_['true.Bool, 'A:Bool] [none] .
  eq '_implies_['A:Bool, 'B:Bool]
    = 'not_['_xor_['A:Bool, '_and_['A:Bool, 'B:Bool]]] [none] .
```

# Moving between levels: Example

```
Maude> reduce in META-LEVEL : upEqs('VENDING-MACHINE, false) .
result EquationSet: (none).EquationSet
```

```
Maude> reduce in META-LEVEL : upRls('VENDING-MACHINE, true) .
result RuleSet:
  rl '$.Coin => 'c.Item [label('buy-c)] .
  rl '$.Coin => '__['q.Coin,'a.Item] [label('buy-a)] .
  rl 'M:State => '__['$.Coin,'M:State] [label('add-$)] .
  rl 'M:State => '__['q.Coin,'M:State] [label('add-q)] .
  rl '__['q.Coin,'q.Coin,'q.Coin,'q.Coin] => '$.Coin
    [label('change)] .
```

# Moving between levels: Terms

```
fmod UP-DOWN-TEST is protecting META-LEVEL .
  sort Foo .
  ops a b c d : -> Foo .
  op f : Foo Foo -> Foo .
  op error : -> [Foo] .
  eq c = d .
endfm

Maude> reduce in UP-DOWN-TEST : upTerm(f(a, f(b, c))) .
result GroundTerm: 'f['a.Foo,'f['b.Foo,'d.Foo]]

Maude> reduce downTerm('f['a.Foo,'f['b.Foo,'c.Foo]], error) .
result Foo: f(a, f(b, c))

Maude> reduce downTerm('f['a.Foo,'f['b.Foo,'e.Foo]], error) .
Advisory: could not find a constant e of
          sort Foo in meta-module UP-DOWN-TEST.
result [Foo]: error
```

# metaReduce

- Its first argument is the representation in `META-LEVEL` of a module $\mathcal{R}$ and its second argument is the representation in `META-LEVEL` of a term $t$.
- It returns the metarepresentation of the canonical form of $t$, using the equations in $\mathcal{R}$, together with the metarepresentation of its corresponding sort or kind.
- The reduction strategy used by `metaReduce` coincides with that of the `reduce` command.

```
Maude> reduce in META-LEVEL :
         metaReduce(upModule('SIEVE, false),
           'show_upto_['primes.NatList, 's_^10['0.Zero]]) .
result ResultPair:
  {'_._['s_^2['0.Zero], 's_^3['0.Zero], 's_^5['0.Zero],
        's_^7['0.Zero], 's_^11['0.Zero], 's_^13['0.Zero],
        's_^17['0.Zero], 's_^19['0.Zero], 's_^23['0.Zero],
        's_^29['0.Zero]],
   'IntList}
```

# metaRewrite

- Its first two arguments are the representations in `META-LEVEL` of a module $\mathcal{R}$ and of a term $t$, and its third argument is a natural $n$.
- Its result is the representation of the term obtained from $t$ after at most $n$ applications of the rules in $\mathcal{R}$ using the strategy of Maude's command `rewrite`, together with the metarepresentation of its corresponding sort or kind.

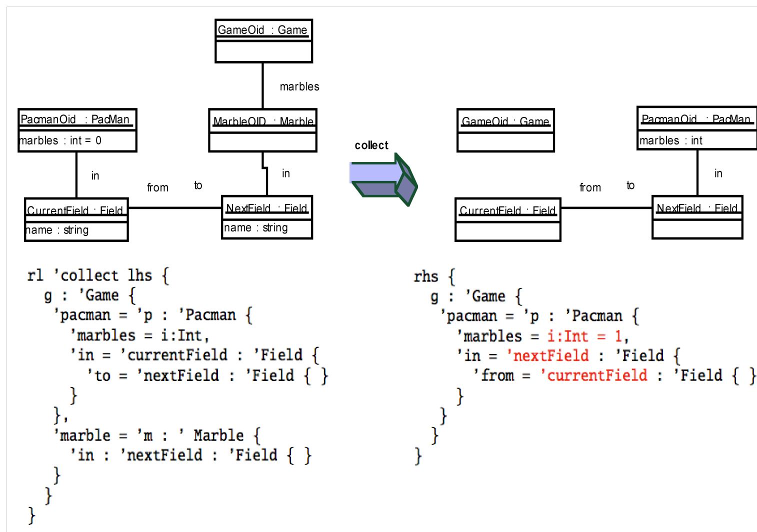
```
Maude> reduce in META-LEVEL :
        metaRewrite(upModule('VENDING-MACHINE, false),
          '__['$.Coin, '__['$.Coin, '__['q.Coin, 'q.Coin]]], 1) .
result ResultPair:
  {'__['$.Coin, '$.Coin, '$.Coin, 'q.Coin, 'q.Coin], 'State}

Maude> reduce in META-LEVEL :
        metaRewrite(upModule('VENDING-MACHINE, false),
          '__['$.Coin, '__['$.Coin, '__['q.Coin, 'q.Coin]]], 2) .
result ResultPair:
  {'__['$.Coin, '$.Coin, '$.Coin, 'q.Coin, 'q.Coin, 'q.Coin], 'State}
```

# Metaprogramming

- A metaprogram is a program that takes programs as inputs and performs some useful computation.
- It may, for example, transform one program into another.
- Or it may analyze such a program with respect to some properties, or perform other useful program-dependent computations.
- We can easily write Maude metaprograms by importing `META-LEVEL` into a module that defines such metaprograms as functions that have `Module` as one of their arguments.
- Examples:
  - compilation of grammar-based languages,
  - graph-based model transformations.

# Transformation Definition Language

- Example of grammar-based programming language:



# Semantics of Grammar-based Languages

- Context-free grammar of the transformation definition language:

```
< Program >          ::= .. < GTRuleExp > ..
< GTRuleExp >        ::= < GTRuleQualifier > < QvtIdentifier > < LHS > ';' < RHS> ';'
< GTRuleQualifier > ::= eq | rl
```

- A context-free grammar can be defined as an order-sorted signature:

```
sorts Program GTRuleExp GTRuleQualifier .
op .. GTRuleExp .. -> Program .
op ___;_;' : GTRuleQualifier QvtIdentifier LHS RHS -> GTRuleExp .
ops eq rl : -> GTRuleQualifier .
```

- The semantics of the language is given by the following reflective compiler function:

```
op compile :  Program -> Module .
```

# Execution of Model Transformations

- Model transformation definition $\widetilde{mt} : Program$
- Semantics of a model transformation definition is given by the rewrite theory $compile(\widetilde{mt})$, which
  - is metarepresented as $\overline{compile(\widetilde{mt})}$, and
  - subsumes the MEL theory that represents the metamodel (including the model type $\mathcal{M}$).
- Source model definition $\widetilde{M} : \mathcal{M}$ is metarepresented as a TERM term $\overline{M}$
- Model transformation is performed at the Maude meta-level by means of the metaRewrite command:

$$\langle \overline{compile(\widetilde{mt})}, \overline{M} \rangle \longrightarrow^{*} \langle \overline{compile(\widetilde{mt})}, \overline{M'} \rangle$$

where $\overline{M'}$ is the metarepresentation of the final model.

# Conclusions

- Maude is a high-level language and high-performance system.
- It supports both equational and rewriting logic computation.
  - Membership equational logic improves order-sorted algebra.
  - Rewriting logic is a logic of concurrent change.
- Applications to Model-Driven Engineering and Graph Rewriting:
  - Metamodeling: metamodel conformance
  - Model transformations: functional and concurrent behavior
  - Metaprogramming: code generation, semantics of grammar-based languages
  - Formal verification techniques: invariant checking through (bounded) search, LTL model checking

# Many thanks to

- Many thanks to:
  - Narciso Martí-Oliet
  - José Meseguer
  - The audience
- For further information about Maude:
  - http://maude.cs.uiuc.edu
  - Book: M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet and C. Talcott. "All About Maude". Springer LNCS Vol. 4350. 2007.