Formal Specification and Analysis of Domain Specific Models Using Maude

José E. Rivera
Francisco Durán
Antonio Vallecillo
Departamento de Lenguajes y Ciencias de la Computación
Universidad de Málaga, Spain
{rivera, duran, av}@lcc.uma.es

Modeling languages play a cornerstone role in model-driven software development for representing models and metamodels. Modeling languages are usually defined in terms of their abstract and concrete syntax. This allows the rapid development of languages and some associated tools (e.g. editors), but does not allow the representation of their behavioral semantics, something especially important in certain industrial environments in which simulation and verification are critical issues. In this paper we explore the use of Maude as a formal notation for describing models, metamodels, and their dynamic behavior, making models amenable to formal analysis, reasoning, and simulation.

Keywords: Model Driven Engineering, Domain Specific Languages, Formal semantics, Model Simulation, Model Analysis, Maude

1. Introduction

Domain-specific modeling (DSM) is a way of designing and developing systems that involves the systematic use of domain-specific languages (DSLs) to represent the various facets of a system, in terms of models. Such languages tend to support higher-level abstractions than general-purpose modeling languages, and are closer to the problem domain than to the implementation domain. Thus, a DSL follows the domain abstractions and semantics, allowing modelers to perceive themselves as working directly with domain concepts. Furthermore, the rules of the domain can be included in the language as constraints, disallowing the specification of illegal or incorrect models.

DSLs play a cornerstone role in DSM. So far, defining a modeling language involved at least two aspects: the domain concepts and rules (abstract syntax), and the notation used to represent these concepts (concrete syntax), whether textual or graphical. The abstract syntax of a DSL is usually defined by a metamodel. A metamodel describes the concepts of the language, the relationships between them, and the structuring rules that constrain the model elements and their combinations in order to respect the domain rules. This approach enables the rapid and effective development of DSLs and some associated tools (e.g. editors, constraint checkers or metric evaluators).

Current DSM approaches have mainly focused on the syntactic (i.e. structural) aspects of DSLs. The explicit and precise specification of the behavioral semantics of models has not received much attention by the DSM community until recently, despite the fact that this creates a possibility for semantic mismatch between design models and the modeling languages of analysis tools. While this problem exists in virtually every domain where DSLs are used, it is more common in domains in which behavior needs to be represented explicitly. This issue is particularly important in safety-critical real-time and embedded system domains, where semantic ambiguities may produce conflicting results across different tools. Furthermore, the lack of explicit behavioral semantics strongly hampers the development of formal analysis and of simulation tools.

In order to remedy this situation, a growing number of proposals is trying to express the behavioral semantics of metamodels using diverse approaches (see Section 6). However, most of them present several limitations, especially when it comes to represent complex behaviors at a high level of abstraction. Moreover, many of them do not

SIMULATION, Vol. 85, Issue 11/12, Nov./Dec. 2009 778–792

DOI: 10.1177/0037549709341635
Figures 1, 2 appear in color online: http://sim.sagepub.com

count with proper tool support for the formal analysis of the specification produced.

In this paper we explore the use of Maude [12, 13] as a formal notation and system for supporting the specification of modeling languages. In particular, we show how models, metamodels, and their dynamic behavior can be specified in Maude using rewriting logic, and how we can make use of its analysis capabilities to reason about them: the fact that rewriting logic specifications are executable allows us to apply a flexible range of increasingly stronger formal analysis methods and tools, such as reachability analysis [13], model checking [22], or theorem proving [14]. Maude also offers a comprehensive toolkit for automating such kinds of formal analysis of specifications, efficient enough to be of practical use, and easy to integrate with software development environments such as Eclipse [38].

The structure of this document is as follows. First, Section 2 introduces the importance of adding semantics to a metamodel. Then, Section 3 serves as a brief introduction to Maude. Section 4 describes how models, metamodels and their dynamic behavior can be represented in Maude. Section 5 is dedicated to show how this Maude representation makes models amenable to formal analysis, providing some application examples. Finally, Section 6 compares our work with other related proposals and Section 7 draws some conclusions and outlines some future research activities.

2. Models, Metamodels and Their Semantics

A model is written in the language of its metamodel. A metamodel describes the concepts of the language, the relationships between them, and the structuring rules that constrain the model elements and their combinations in order to respect the domain rules. We normally say that a model *conforms to* its metamodel [3].

Metamodels are also models, and therefore they need to be written in another language, which is described by its meta-metamodel. This recursive definition normally ends at the meta-metalevel, since meta-metamodels conform to themselves.

As described by Chen et al. [11], the semantics of a DSL may be either structural or behavioral. The structural semantics describe the meaning of the models in terms of the structure of model instances: all of the possible sets of components and their relationships, which are consistent with the well-formedness rules, are defined by the abstract syntax. The behavioral semantics describe the evolution of the state of the modeled artifacts along some time model. Hence, the behavioral semantics need to be formally captured by a mathematical framework representing the appropriate form of dynamics.

Originally, the behavioral semantics of metamodels were not specified explicitly, or this was done using natural languages [27]. This way of describing semantics is not precise enough, and therefore it may lead to several (mis)interpretations of the same model. When a model is used to represent some ideas, illustrate a design or just as a roadmap to the corresponding implementation, semantics may be not needed. However this is not the case in DSM, especially for simulation purposes.

Some authors pointed out that it is normally possible to guess the meaning of most terms of a metamodel, since a good language designer probably chooses keywords and special symbols with a meaning similar to some accepted norm. However a computer cannot act on such assumptions [25]. To be useful in the computing arena, any language (either textual or visual, either used for programming, requirements, specification, or design) must come with rigid rules that clearly state allowable syntactic expressions and give a rigid description of their meaning. This issue is essential for realizing some model operations such as simulation and verification, and is of special relevance in the safety-critical real-time and embedded systems domain, where semantic ambiguities may produce conflicting results across different tools.

A lot of notation has already been proposed to specify models and metamodels. However, the kind of formal expressiveness and tool support they provide is quite limited (see Section 6). In this paper we propose Maude as a formal notation for describing models, metamodels, and their dynamic behavior, providing metamodels with semantics. In contrast to other proposals where abstract syntax and semantics live in different worlds, we can specify both abstract syntax and semantics using Maude, with no need for another notation to specify the mappings. Furthermore, these specifications will allow models to be simulated and analyzed by the Maude system and its formal analysis environment [14].

3. Rewriting Logic and Maude

Membership equational logic (MEL) [8] is a Horn logic whose atomic sentences are equalities $t = t'$ and *membership assertions* of the form $t : S$, stating that a term t has sort S. Such a logic extends order-sorted equational logic, and supports sorts, subsort relations, subsort polymorphic overloading of operators, and definition of partial functions with equationally defined domains. MEL has sound and complete rules of deduction, and initial and free algebras [8].

Rewriting logic (RL) [31] is a logic of change that can naturally deal with state and with highly non-deterministic concurrent computations. In RL, the state space of a distributed system is specified as an algebraic data type in terms of an equational specification (Σ, E), where Σ is a signature of sorts (types) and operations, and E is a set of equational axioms. The dynamics of a system in RL is then specified by rewrite *rules* of the form $t \rightarrow t'$, where t and t' are Σ terms. These rules describe the local, concurrent transitions possible in the system, i.e. when a part

of the system state fits the pattern t then it can change to a new local state fitting pattern t'. The model theory of RL has been developed in detail in [31], where initiality, soundness, and completeness theorems are proved.

In what follows, we use the syntax of Maude [13], a wide-spectrum programming language directly based on RL, with MEL as its underlying equational theory, to present rewrite theories. Thus, Maude integrates an equational style of functional programming with RL computation. Owing to its efficient rewriting engine and its metalanguage capabilities, Maude turns out to be an excellent tool to create executable environments for different logics, models of computation, theorem provers, or even programming languages. In addition, Maude has already been successfully used in software engineering tools and several applications [30]. We informally describe in this section those Maude's features necessary for understanding the paper; the interested reader is referred to [13] for details.

For example, the following Maude functional module `NATURAL` defines the natural numbers (with sorts `Nat` of natural numbers and `NzNat` of non-zero natural numbers), using the Peano notation, with the zero (`0`) and successor (`s_`) operators as constructors (note the `ctor` attribute). The addition operation (`_+_`) is also defined, being its behavior specified by two equational axioms. The operators `s_` and `_+_` are defined using *mixfix* syntax (underscores indicate placeholders for arguments):

```
fmod NATURAL is
 sorts NzNat Nat .
 subsort NzNat < Nat .
 op 0 : -> Nat [ctor] .
 op s_ : Nat -> NzNat [ctor] .
 op _+_ : Nat Nat -> Nat [assoc comm id: 0] .
 vars M N : Nat .
 eq s M + s N = s s (M + N) .
endfm
```

Computation in a functional module is accomplished by using equations as simplification rules from left to right until a canonical form is found. Some equations, such as those expressing the commutativity of binary operators, are not terminating but nonetheless they are supported by means of *operator attributes*, so that Maude performs simplification *modulo* the equational theories provided by such attributes, which can be associative (`assoc`), commutativity (`comm`), and identity (`id`).

While functional modules specify membership equational theories, rewrite theories are specified by *system modules*. A system module may have the same declarations of a functional module plus rules. Unconditional rules are written with syntax `rl [l] :` t `=>` t' `.`, and conditional rules as `crl [l] :` t `=>` t' `if` *Cond* `.`, with l the rule label and *Cond* its condition. The guards of conditional rules act as blocking pre-conditions, in the sense that a conditional rule can only be fired if its condition is satisfied.

Assuming a parameterized module `SET` specifying sets of elements, with a sort `Set{X}` of sets, `empty` as the empty set, and `_,_` the associative and commutative union operator, with identity element `empty`, let us consider the following `BOARD` system module:

```
mod BOARD is
 protecting SET{Nat} .
 vars N M : Nat .
 rl [replace] : N, M => (N + M) quo 2 .
endm
```

The `replace` rule takes any two numbers in a set of numbers and replaces them by their mean.

The `rewrite` command can be used to execute the system, using the Maude interpreter, which applies the rules and stops when no rule can be applied:

```
Maude> rewrite 6, 3, 2 .
result NzNat: 4
```

Note that the final result depends on the numbers selected in each application of the rule.

4. Models, Metamodels and Their Semantics in Maude

In [41], we presented a proposal for representing and manipulating models based on the use of Maude, which not only was expressive enough for these purposes, but also offered good tool support for operating with models. In particular, we showed how some basic operations on models, such as model subtyping, type inference, and metric evaluation, can be easily specified and implemented in Maude, and made available in development environments such as Eclipse. That work is extended here to cope with the specification of the behavioral semantics of metamodels, something we already outlined in our previous work [40]. In this paper, that representation has been improved for efficiency reasons (see Section 6). Moreover, here we show how to make use of Maude's formal analysis methods and tools to reason about the model specifications.

4.1 A Production System Example

For illustration purposes, let us introduce a modeling language for industrial production lines, which will serve as the motivating example to show the capabilities of our approach. It is inspired by a common example used in several works for describing DSLs (see, e.g., [10, 45]).

The production systems metamodel is shown in Figure 1. A production system is composed of different

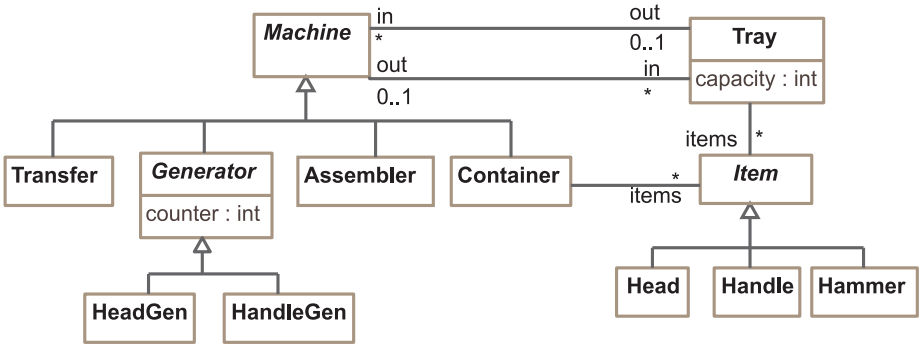

Figure 1. A production system metamodel

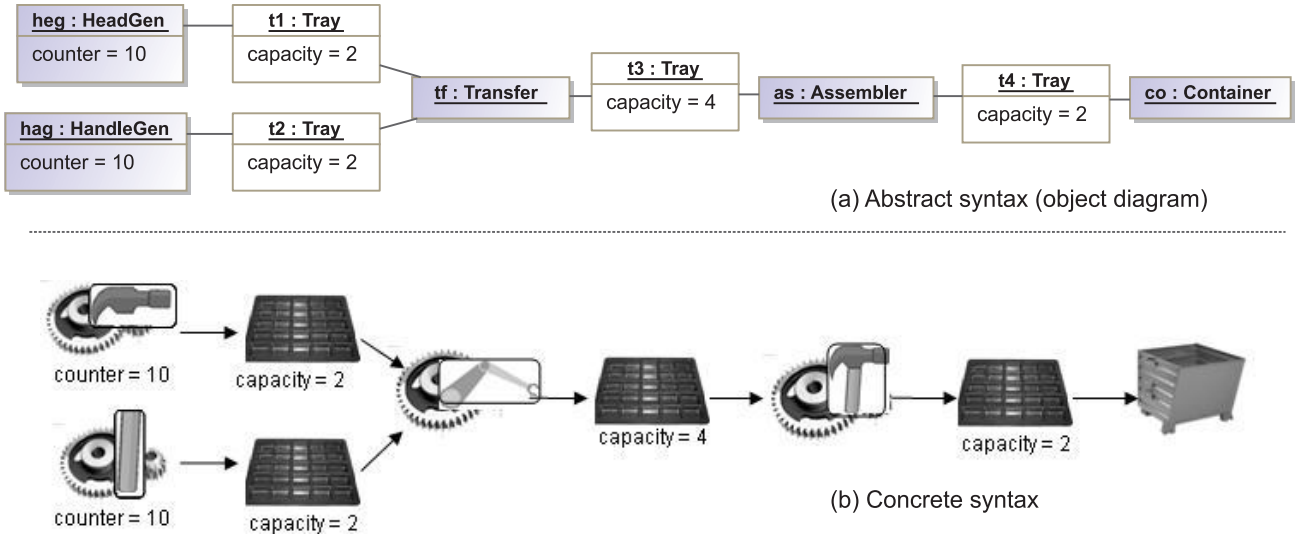

Figure 2. A production system model

kinds of machines: generators, transfers, assemblers and containers. Generators produce items, transfer machines move them through different trays, assemblers consume them to create new ones, and containers store the assembled items. Machines take their inputs from and put their results in trays which contain items up to their capacity.

Figure 2 shows a production model for hammers. It is composed of generators of heads and handles, a transfer machine that moves the generated heads and handles to the input tray of an assembler, an assembler that combines heads and handles to form hammers, and a container that stores (an unlimited number of) hammers. The figure also shows the number of handles and heads to be created (attribute `counter` of class `Generator`) as well as the maximum `capacity` of trays.

In the following, we refer to the model in Figure 2 as `initModel` (the *initial* model). Another model, resulting from modifying both generator counters to the value -1 (making generators create items indefinitely) will be considered as well. We refer to this second model as `infInitModel`.

4.2 Representing Models with Maude

Although the notion of model is not yet fully agreed, we prefer to adopt the definition given by Bézivin and Jouault in [4]: a model M is a triplet $M = (G, \omega, \mu)$ where G is a directed multigraph, ω is itself a model (called the reference model or metamodel of M) associated with a graph G_ω, and μ is a function associating elements (nodes and edges) of G to nodes of G_ω. The relation between a model and its reference model is called *conformance*.

Models are represented by sets of Maude objects. Nodes are represented as Maude objects, and node attributes by object attributes. Edges are also represented by object attributes, each one representing the reference (by means of object identifiers) to the target node of the edge. Thus, models are structures of the form

$mm\{obj_1 \ obj_2 \ \ldots \ obj_N\}$, where mm is the name of the metamodel, and the obj_i are the objects that represent the corresponding nodes. Maude objects are record-like structures of the form

$$< o : c \mid a_1 : v_1, \ \ldots, a_n : v_n >$$

where o is the object identifier, c is the class the object belongs to, a_i are attribute identifiers and v_i their corresponding current values.

Given the appropriate definitions for all classes, attributes, and references in its corresponding metamodel (as we shall see in next section), the following Maude term is a possible representation of the hammers production model depicted in Figure 2:

```
ProductionSystem {
 < 'heg : HeadGen | counter : 10,
   in : empty, out : 't1 >
 < 'hag : HandleGen | counter : 10,
   in : empty, out : 't2 >
 < 't1 : Tray | items : empty, in : 'heg,
   out : 'tf, capacity : 2 >
 < 't2 : Tray | items : empty, in : 'hag,
   out : 'tf, capacity : 2 >
 < 'tf : Transfer | in : ('t1, 't2),
   out : 't3 >
 < 't3 : Tray | items : empty, in : 'tf,
   out : 'as, capacity : 4 >
 < 'as : Assembler | in : 't3, out : 't4 >
 < 't4 : Tray | items : empty, in : 'as,
   out : 'co, capacity : 2 >
 < 'co : Container | in : 't4, out : null,
   items : empty > } .
```

Note that quoted identifiers (identifiers with a quote in front of them) are used as object identifiers. Relations with multiplicity `*` are represented as sets of references (of sort `Set{Oid}`). Relations with arity `0...1` are represented as values of sort `Maybe{Oid}`, which can be either an object identifier or the constant `null`.

4.3 Representing Metamodels with Maude

As part of the infrastructure we provide, there is a sort for every metamodel element: a sort `@Class` for classes, a sort `@Attribute` for attributes, a sort `@Reference` for references, etc. Thus, a metamodel is represented by declaring a constant of the corresponding sort, for each metamodel element. More precisely, each class is represented by a constant of a sort named after the class. This sort, which will be declared as subsort of sort `@Class`, is defined to support class inheritance through Maude's order-sorted type structure. Attributes and references are represented by constants of subsorts of `@Attribute` or `@Reference`, respectively, depending on their cardinality, and on whether their elements have to be ordered or can be duplicated. For instance, sort `@AttributeSimple` represents attributes with a [1–1] multiplicity; sort `@ReferenceMaybe` represents references with a [0–1] multiplicity; and sort `@ReferenceSet` represents non-ordered collections of non-duplicated references.

To illustrate this approach, the following fragment of Maude specification describes the production system metamodel depicted in Figure 1:

```
mod PRODUCTION-SYSTEM is
 extending MAUDELING-BASE .

 op ProductionSystem : -> @Metamodel .
 op PS : -> @Package .

 sort Machine .
 subsort Machine < @Class .
 op Machine : -> Machine .
  op in : -> @ReferenceSet .
  op out : -> @ReferenceMaybe .

 sorts Assembler Transfer .
 subsorts Assembler Transfer < Machine .
 op Assembler : -> Assembler .
 op Transfer : -> Transfer .

 sort Container .
 subsort Container < Machine .
 op Container : -> Container .
  op items : -> @ReferenceSet .

 sort Generator .
 subsort Generator < Machine .
 op Generator : -> Generator .
   op counter : -> @AttributeSimple .

 sort HeadGen HandleGen .
 subsorts HeadGen HandleGen < Generator .
 op HeadGen : -> HeadGen .
 op HandleGen : -> HandleGen .

 sort Item .
 subsort Item < @Class .
 op Item : -> Item .

 sort Head Handle Hammer .
 subsorts Head Handle Hammer < Item .
 op Head : -> Head .
 op Handle : -> Handle .
 op Hammer : -> Hammer .

 sort Tray .
 subsort Tray < @Class .
 op Tray : -> Tray .
  op capacity : -> @AttributeSimple .
  op items : -> @ReferenceSet .
```

```
  op in : -> @ReferenceSet .
  op out : -> @ReferenceMaybe .
endm
```

The other properties of metamodel elements, such as whether a class is abstract or not, the opposite of a reference (to represent bidirectional associations), or attributes and reference types, are expressed by means of Maude equations defined over the constant that represents the corresponding metamodel element. We can find, for example, the following definitions:

```
*** class Tray
eq isAbstract(Tray) = false .
...
*** attribute capacity (of class Tray)
eq type(capacity) = @Int .
eq lowerBound(capacity) = 1 .
eq upperBound(capacity) = 1 .
...
*** reference items (of class Tray)
eq type(items) = Item .
eq opposite(items) = null .
eq lowerBound(items) = 0 .
eq upperBound(items) = * .
...
```

Classes, attributes, and references are qualified with their containers' names, so that classes with the same name belonging to different packages, as well as attributes and references of different classes, are distinguished. For example, assuming that all of the classes in the model are included in a package `PS`, identifiers `Tray`, `capacity`, and `in` would really be `Tray@PS`, `capacity@Tray@PS`, and `in@Tray@PS` (or `in@Machine@PS`). These qualifications are omitted here to improve readability.

Note that this notation allows users to obtain all of the metamodel information from a model with no need to access the Maude metalevel. Thus, operations that need metamodel information are not only easier to specify, but also more efficient. A good example is the operation for validating models, which checks whether a model conforms to its metamodel (verifying, among other conditions, cardinalities and type constraints defined for a model).

Finally, please note that metamodels are models too, and thus they can also be represented by a set of objects. The classes of such objects will be those specified in the meta-metamodels, for example, the classes that define the MOF (or Ecore) metamodel. In this way, metamodels can be handled in the same way as models are, i.e. operations and analysis techniques defined over models can also be applied to metamodels.

4.4 Representing Behavior with Maude

As we have mentioned previously, explicit and formal specification of behavioral semantics has not received much attention by the DSM community so far, despite the fact that this lack of attention opens the door to the possibility for semantic mismatches.

For instance, it is not clear from the production system metamodel when the handles and heads are generated, or what is the precise behavior of the system when one of the trays is full. Can we execute the system for simulating different trays capacities, in order to find the most efficient production line? These are the sort of issues that need to be precisely clarified by a behavioral specification.

Dynamic behavior can be specified in Maude in terms of rewrite rules added to the corresponding metamodel specification. Thus, given a Maude module with the specification of a metamodel, we can 'extend' it with behavioral information (by adding the appropriate rewrite rules).

To illustrate this approach, the following Maude module `PRODUCTION-SYSTEM-WITH-BEHAVIOR` extends the above `PRODUCTION-SYSTEM` module with five rules:

```
mod PRODUCTION-SYSTEM-WITH-BEHAVIOR is
 inc PRODUCTION-SYSTEM .

 vars HEG T HE HA IT A HAM CO T' T" TR : Oid .
 vars ITEMS ITEMS' ITEMS" : Set{Oid} .
 vars SFS SFS' SFS" SFS"' SFS"" :
   Set{@StructuralFeatureInstance} .
 var OBJSET : Set{@Object} .
 vars CONT CAP CAP' CAP" : Int .

 crl [GenHead] :
  ProductionSystem {
    < HEG : HeadGen | counter : CONT,
        out : T, SFS >
    < T : Tray | capacity : CAP,
        items : ITEMS, SFS' >
    OBJSET }
  =>
  ProductionSystem{
    < HEG : HeadGen | counter : CONT - 1,
        out : T, SFS >
    < T : Tray | capacity : CAP,
        items : (ITEMS, HE), SFS' >
    < HE : Head | >
    OBJSET }
  if HE := headId(CONT)
     /\ CONT =/= 0
     /\ | ITEMS | < CAP .

 -- GenHandle rule similar to GenHead

 crl [Transfer] :
  ProductionSystem {
```

```
    < T : Tray | items : (IT, ITEMS),
        out : TR, SFS >
    < T' : Tray | items : ITEMS',
        out : TR, SFS' >
    < TR : Transfer | out : T", SFS" >
    < T" : Tray | capacity : CAP",
        items : ITEMS", SFS"' >
    OBJSET }
 =>
 ProductionSystem{
    < T : Tray | items : ITEMS,
        out : TR, SFS >
    < T' : Tray | items : ITEMS',
        out : TR, SFS' >
    < TR : Transfer | out : T", SFS" >
    < T" : Tray | capacity : CAP",
        items : (IT, ITEMS"), SFS"' >
    OBJSET }
 if | ITEMS" | < CAP" .

crl [Assemble] :
 ProductionSystem {
    < HA : Handle | SFS >
    < HE : Head | SFS' >
    < T : Tray | items : (HE, HA, ITEMS),
      out : A,  SFS" >
    < A : Assembler | out : T', SFS"' >
    < T' : Tray | capacity : CAP',
      items : ITEMS', SFS"" >
    OBJSET }
 =>
 ProductionSystem{
    < T : Tray | items : ITEMS,
      out : A, SFS" >
    < A : Assembler | out : T', SFS"' >
    < T' : Tray | capacity : CAP',
      items : (ITEMS', HAM), SFS"" >
    < HAM : Hammer | >
    OBJSET }
 if HAM := hammerId(HE, HA)
    /\ | ITEMS' | < CAP' .

rl [Store] :
 ProductionSystem {
    < T : Tray | items : (IT, ITEMS),
        out : CO, SFS >
    < CO : Container | items : ITEMS', SFS" >
    OBJSET }
 =>
 ProductionSystem{
    < T : Tray | items : ITEMS,
        out : CO, SFS >
    < CO : Container | items : (IT, ITEMS'),
        SFS" >
    OBJSET } .
endm
```

The `GenHead` and `GenHandle` rules specify the behavior of head and handle generators, respectively (the latter is not shown because it is analogous to the former). Since the value of the tray attribute `counter` is decreased every time an item is generated, a head (or handle) is created provided that: (a) there is room for it in the `out` tray (the number of items in it is smaller than its capacity), and (b) the number of items already created is smaller than its upper bound (i.e. its attribute `counter` is different from zero). The `headId` (respectively, `handleId`) function returns a new identifier `'he`*n* (respectively, `'ha`*n*) given a natural number *n*[1]. The $|\_|$ function returns the cardinality of a given set.

The `Transfer` rule specifies the behavior of transfer machines. If there is an item in one of its `in` trays, it will place it in its `out` tray provided that there is enough space in it.

The `Assemble` rule specifies the behavior of assemblers. If there is an assembler with a head and a handle in its `in` tray, and space on its `out` tray, then it will consume such a head and a handle and will put a hammer in its `out` tray. Given identifiers `'he`*n* and `'ha`*m*, the `hammerId` function returns a new identifier `'he`*n*`.ha`*m* .

The `Store` rule represents the hammer storage. Every time a hammer is placed on the container's `in` tray, the container stores it.

Note that the dynamic semantics of the model is precisely specified by these rewrite rules. For instance, if the `out` tray of a generator is full, no further items are created (until some item is taken from it); in case that a transfer can move several items from its `in` trays to its `out` tray, only one (at a time) will be moved in an non-deterministic way, etc.

Moreover, this specification can be used for simulation and execution purposes, as well as for formal analysis. We see below how this non-determinism can be removed by using Maude's *strategies* (Section 5.1) for controlling the execution of the system.

Finally, please note that several alternative behaviors can be specified by defining other modules extending the original `PRODUCTION-SYSTEM` module with different rules. This, together with the module inheritance Maude supports [13], provides a very powerful mechanism for refining and extending behavioral specifications.

5. Formal Analysis

Once the system specifications are written using the modeling approach presented in the previous sections, what we get is a rewriting logic specification of the system. Since the rewriting logic specifications produced are executable,

1. Front-end tools (see Section 6) will generate new object identifiers in some way. For example, our Maudeling tool generates them automatically from a `counter` object included in every simulation for this purpose.

this specification can be used as a prototype of the system, which allows us to simulate and analyze it.

Maude offers tool support for interesting possibilities such as model simulation, reachability analysis and model checking. We focus here on these three facilities, which will be described with the help of examples that illustrate Maude's possibilities in this context. Detailed descriptions of the commands and tools used here can be found in [13]. In addition, models can be further analyzed using other available tools in Maude's formal environment: the LTL satisfiability and tautology checker, the Church–Rosser checker, the coherence checker, the termination tool, etc. We refer the interested reader to [13, 14] for details on these tools.

5.1 Simulation

Given a Maude specification such as that described in the previous sections, it can be executed by just successively applying equations and rewrite rules on an initial term (model). Maude provides two different rewrite commands, namely `rewrite` and `frewrite`, which implement two different execution strategies, a top-down rule-fair strategy, and a depth-first position-fair strategy, respectively [13].

The `rewrite` and `frewrite` commands explore a possible execution of different initial models. However, a rewrite system do not need to be Church-Rosser and terminating[2] and there might be many different execution paths. Although these commands are sufficient in many practical situations where an execution path is sufficient for testing executability, the user might be interested in exploring all possible execution paths from the starting model, a subset of these, or a specific path.

There are currently three ways of controlling the execution process in Maude, which, ordered from the more complex to the easiest to use, are as follows.

- Using the reflective capabilities of Maude, with built-in metalevel functions such as `metaReduce`, `metaApply`, etc., which provide absolute control over the execution process [13].
- Using the Maude strategy language [21] to define strategy expressions that control the way terms are rewritten.
- Using invariant-driven strategies for executing specifications complying with given invariants [19].

We illustrate here the use of the strategy language as a mechanism to guide the execution process. In this language, a strategy is described as an operation that, when applied to a given term, produces a set of terms as a result (since, in general, this is a non-deterministic process). The basic strategies consist of the application of a rule (identified by the corresponding rule label) to a given term, and allowing variables in a rule to be instantiated before its application with a specific substitution. Basic strategies are combined by means of operators such as concatenation (`;`), union (`|`), iteration (`*` for zero or more times and `+` for one or more times), if–then–else, etc.

We can, for example, use the Maude `srewrite_using_` command to find all possible applications of the rules specified in Section 4.4 from `initModel` as follows:

```
srewrite initModel using all .
```

In this case the number of possible executions is huge, and although this may be helpful for some automatic checking, the result is not very useful here. More interesting is the result given by the following command:

```
srewrite initModel
     using (GenHead ; Transfer ; GenHandle ;
               Transfer ; Assemble ; Store) + .
```

In this case, the following sequence is enforced: first, a head is generated and moved to the `in` conveyor of the assembler; then, a handle is generated and moved as well; afterwards, both the head and the handle are assembled to generate a hammer; finally, this hammer is stored in the container. Ten solutions are found, one for each possible time that the sequence can be repeated before the generators reach their limit. For instance, the last solution found, where 10 hammers are created and stored, is the following:

```
ProductionSystem{
 < 'as : Assembler | in : 't3, out : 't4 >
 < 'co : Container | in : 't4, out : null,
       items : ('he1.ha1, 'he2.ha2, 'he2.ha2,
                 'he4.ha4, ... 'he10.ha10) >
 < 'he1.ha1 : Hammer | empty >
 < 'he2.ha2 : Hammer | empty >
 < 'he3.ha3 : Hammer | empty >
 < 'he4.ha4 : Hammer | empty >
 ...
 < 'he10.ha10 : Hammer | empty >
 < 'heg : HeadGen | counter : 0,
      in : empty, out : 't1 >
 < 'hag : HandleGen | counter : 0,
      in : empty, out : 't2 >
 < 't1 : Tray | capacity : 2, items : empty,
      in : 'heg, out : 'tf >
 < 't2 : Tray | capacity : 2, items : empty,
      in : 'hag, out : 'tf >
 < 't3 : Tray | capacity : 4, items : empty,
      in : 'tf, out : 'as >
```

2. For MEL specifications, being Church–Rosser and terminating means not only confluence (so that a unique normal form will be reached), but also a sort decreasingness property, namely that the normal form will have the least possible sort among those of all other equivalent terms.

```
  < 't4 : Tray | capacity : 2, items : empty,
       in : 'as, out : 'co >
  < 'tf : Transfer | in : ('t1,'t2),
       out : 't3 >}
```

The number of solutions to be shown can be limited using the corresponding (optional) argument of the `srewrite` command. Thus, for example, we can ask for the first 15 executions, using a less strict strategy (applying some of the rules with some freedom), as follows:

```
srewrite [15] initModel
     using ((GenHead ; Transfer) |
            (GenHandle ; Transfer) |
            Assemble | Store) + .
```

Among the solutions obtained we obtain that above (together with many others). The `rewrite` and `frewrite` commands also allow users to specify an upper bound for the number of rule applications. This upper bound can be very useful to simulate non-terminating system, or to execute a simulation step by step.

5.2 Reachability Analysis

Executing the system using the `rewrite` and `frewrite` commands means exploring just one possible behavior of the system. As in Section 5.1, this can be partially improved by using rewriting strategies. However, it does not allow us to analyze the solutions found. The Maude `search` command allows us to explore (following a breadthfirst strategy up to a specified bound) the reachable state space in different ways.

Let us start by looking for deadlock states. Although in this example we may easily guess how the deadlock states look, let us be a bit naive and look for final states (states on which no further rewrite may take place) using the `=>!` variant of the search command, with no hammers in the final container. For instance, starting from our initial model, and given variables `SFS` of sort `Set{@StructuralFeatureInstance}` and `OBJSET` of sort `Set{@Object}`, we can use the `search` command as follows:

```
search [10] initModel =>!
  ProductionSystem {
    < 'co : Container | items : empty, SFS >
    OBJSET } .
```

Please note that we have limited the number of solutions to 10, otherwise it gives hundreds of solutions. By observing them we realize that a possible source of deadlock is that the `'t3` tray may be full of items of the same type, not allowing the assembler machine to proceed. Let us look now for states satisfying this condition. For instance, starting from our initial model, and given variables `O1`, `O2`, `O3`, and `O4` of sort `Oid`, `I` of sort `Item`, `SFS` of sort `Set{@StructuralFeatureInstance}`, and `OBJSET` of sort `Set{@Object}`, we can search for states on which the tray `'t3` contains four items of the same type, using the `search` command as follows:

```
search [1] initModel =>*
  ProductionSystem {
    < 't3 : Tray | items : (O1,O2,O3,O4), SFS >
    < O1 : I | empty >
    < O2 : I | empty >
    < O3 : I | empty >
    < O4 : I | empty >
    OBJSET } .
```

Note the use of the `=>*` variant of the `search` command to look for any state, final or not. Note as well that we have requested only the first solution. The result given by Maude is the following:

```
Solution 1 (state 210)
OBJSET ->
  < 'as : Assembler | in : 't3, out : 't4 >
  < 'co : Container | in : 't4, out : null,
       items : empty >
  < 'heg : HeadGen | in : empty, out : 't1,
       counter : 6 >
  < 'hag : HandleGen | in : empty, out : 't2,
       counter : 10 >
  < 't1 : Tray | capacity : 2, items: empty,
       in : 'heg, out : 'tf >
  < 't2 : Tray | capacity : 2, items : empty,
       in : 'hag, out : 'tf >
  < 't4 : Tray | capacity : 2, items : empty,
       in : 'as, out : 'co >
  < 'tf : Transfer | in : ('t1,'t2), out : 't3 >
SFS -> capacity : 4, in : 'tf, out : 'as
O2 -> 'he7
O1 -> 'he8
O3 -> 'he9
O4 -> 'he10
I -> Head
```

As result, a collection of solutions (variable substitutions) that fulfil the given search pattern is obtained. In this case, a state with four heads placed in the tray `'t3` is found.

The shortest sequence of rewrites followed to reach this model can be displayed by typing `show path` *n*, where *n* is the number of a state. If we are only interested in the labels of the applied rules, the command `show path labels` *n* can be used instead. In this case, `show path labels 210` produces the sequence `GenHead GenHead Transfer GenHead Transfer GenHead Transfer Transfer`, which is one of the possible paths leading to this state.

Upper bounds for the number of rule applications and solutions can also be specified in the search. For instance, let us consider now the `infInitModel` initial model (with the counters of generators set to −1). Since `counter` attributes are decreased each time an item is generated, and generators stop when its value is equal to zero, it can result in a non-terminating execution if hammers are correctly produced (containers do not have an upper bound for contained items). Then, it could be interesting to know, for example, whether a model on which the container stores at least 10 hammers can be reached. Given variables `ITEMS` of sort `Set{Oid}`, `SFS` of sort `Set{@StructuralFeatureInstance}` and `OBJSET` of sort `Set{@Object}`, we can check it (by searching for only one solution) in the following way:

```
search [1] infInitModel =>*
  ProductionSystem {
    < 'co : Container | items : ITEMS, SFS >
    OBJSET }
  such that | ITEMS | >= 10 .
```

Note the use of the `such that` clause to look for states matching the specified pattern and satisfying the given condition. Although the number of states reachable from the initial state is infinite, a solution was found.

We can also use the `search` command to check safety properties. For instance, given variables `O` of sort `Oid`, `CAP` of sort `Int`, `ITEMS` of sort `Set{Oid}`, `SFS` of sort `Set{@StructuralFeatureInstance}` and `OBJSET` of sort `Set{@Object}`, we can check whether starting from `initModel` the capacity of any tray is exceeded:

```
search initModel =>*
  ProductionSystem {
    < O : Tray |
        capacity : CAP, items : ITEMS, SFS >
    OBJSET }
  such that | ITEMS | > CAP .

No solution.
```

Since no solutions are found, we can state that (starting from `initModel`) the capacities of the trays are never exceeded.

Whenever an invariant is violated, we are guaranteed to obtain a counterexample (up to time and memory limitations). However, when the number of states reachable from the initial state is infinite, and the invariant holds, we will search forever, never finding it. In these cases, bounded search (a search with a maximum number of rule applications) is a widely used procedure that: (a) together with a large depth can greatly increase the confidence that the invariants hold; and (b) can be quite effective in finding counterexamples, where applicable.

We can do the same check for `infInitModel` with a bound on the depth of the search of 1,000 as follows:

```
search [1,1000] infInitModel =>*
  ProductionSystem {
    < O : Tray |
        capacity : CAP, items : ITEMS, SFS >
    OBJSET }
  such that | ITEMS | > CAP .

No solution.
```

With this information, we could very easily modify the dynamic behavior of our model to avoid deadlocks. For example, we could add a condition to the `Transfer` rule so that an item is never put in the output tray if all of the items in it are of the same type (of the item).

5.3 LTL Model checking

Maude offers a linear temporal logic explicit-state model checker [22], which allows us to check whether every possible behavior starting from a given initial model satisfies a given temporal logic property. Maude's model checker can be used to prove safety and liveness properties of rewriting systems when the set of states reachable from an initial state is finite. Full verification of invariants in an infinite-state system can be accomplished by verifying the invariants on finite-state abstractions [32] of the original infinite-state system, that is, on an appropriate quotient of the original system whose set of reachable states is finite [13].

In order to specify safety and liveness properties, state predicates are needed. For instance, suppose that we want to check that a hammer that has been created is eventually stored in the container. In this case, we need two different state predicates: `exist` and `stored`, that will reflect whether an item exists or is stored, respectively.

State predicates are defined as operators of sort `Prop`, and their semantics are given by means of a set of equations that specify for what model pattern a given state predicate evaluates to true. In Maude, opposite to the standard LTL propositional logic, we can define parametric state predicates, that is, operators of sort `Prop` which need not be constants, but may have one or more sorts as parameter arguments. We include the following declarations in a module including the `MODEL-CHECKER` module:

```
ops exist stored : Oid -> Prop .

var O IT : Oid .
var ITEMS : Set{Oid} .
var SFS : Set{@StructuralFeatureInstance} .
var IC : Item .
var OBJSET : Set{@Object} .
var M : @Model .
var P : Prop .

eq ProductionSystem { < O : IC | SFS > OBJSET }
```

```
  |= exist(O) = true .
eq ProductionSystem {
     < O : Container | items : (IT, ITEMS),
     SFS > OBJSET }
   |= stored(IT) = true .
eq M |= P = false [otherwise] .
```

After defining these state predicates, we are ready to model-check different LTL properties. Making use of the *eventually* `< >` and *henceforth* `[ ]` LTL connectives, we can model check, for instance, that a created hammer (`'H10.S10`) is stored at some time, by typing

```
reduce modelCheck(initModel,
 [](exist('he10.ha10) -> <>
    stored('he10.ha10)) .

result Bool: true
```

Of course, this property is not very useful, but it becomes a bit more interesting if we check it for all possible hammers. Given the `exist` and `stored` propositions above, the following formula `allHammers` express the same property for all possible hammers, with identifiers created as in the above rules:

```
 vars N M K : Nat .

 op allHammers : Nat Nat -> Formula .
 op allHammers : Nat Nat Nat -> Formula .
 eq allHammers(N, M) = allHammers(N, M, M) .
 eq allHammers(s N, 0, K)
   = [] (exist(ID) -> <> stored(ID))
     /\ allHammers(N, K, K)
   if ID := hammerId(headId(s N), handleId(0)) .
 eq allHammers(N, s M, K)
   = [] (exist(ID) -> <> stored(ID))
     /\ allHammers(N, M, K)
   if ID := hammerId(headId(N), handleId(s M)) .
 eq allHammers(0, 0, K)
   = [] (exist(ID) -> <> stored(ID))
   if ID := hammerId(headId(0), handleId(0)) .
```

We can now check it as follows:

```
reduce modelCheck(initModel,
   allHammers(10,10)) .

result Bool: true
```

Of course, not all properties are true. For example, if we model-check that a created head, say `'he10`, is consumed at any time (to form a hammer) by typing:

```
reduce modelCheck(initModel,
 [](exist('he10) -> <> ~ exist('he10)) .
```

We obtain a counterexample showing why it fails (in this case due to deadlock states).

5.4 Tool Support

One of the main advantages of using Maude is due to its execution environment, which is able to provide efficient implementations of the specifications—comparable in resource consumption to most commercial programming languages' environments.

Our work so far has consisted in connecting Maude with the Eclipse Modeling Framework (EMF) and graph grammars. We want users to benefit from Maude simulation and analysis capabilities, but working with their preferred notation and tools, i.e. we think that our proposal provides a perfect target semantic domain for providing behavioral semantics to modeling languages.

Regarding graph grammars, we have integrated Maude in AToM3 [16] (a tool for multi-paradigm modeling based on graph grammars) to obtain the best of both platforms: allowing designers to work with domain-specific concepts and their concrete syntax in a graphical way (with AToM3), while providing them with the advanced analysis tools of Maude [39].

Regarding the integration with EMF, we have developed an Eclipse plug-in, called *Maudeling*, that enables the transformation of EMF models and metamodels to the corresponding Maude specifications. We are now working on a tool for representing rules that define the behavior of a metamodel in a visual fashion, called e-Motions [37], that will allow designers to simulate and analyze models and metamodels specified in the Eclipse platform.

Our Maudeling tool and all of the specifications in this paper are available from our website [37].

6. Related work

One way to specify the semantics of a language is to define a translation from expressions in that language into expressions in another language with well defined semantics [25]. These *semantic mappings* between semantic domains are very useful, not only to provide metamodels with semantics, but also to be able to simulate, analyze, or reason about them using the logical and semantical framework available in the target domain [15] (in general, each semantic domain is more appropriate to represent and reason about certain properties, and to conduct certain kinds of analyses).

For example, some works specify the behavioral semantics of models by using visual rules [17, 20, 25], which prescribe the preconditions of the actions to be triggered and the effects of such actions. These preconditions and postconditions are given visually as models that use the concrete syntax of the DSL. This kind of representation is quite intuitive, because it allows designers to work with domain-specific concepts and their concrete syntax

for describing the rules [18]. Graph transformation is one of these rule-based approaches. The most common formalization of graph transformation is the so-called algebraic approach, which uses category theory to express the rewriting [20]. This approach supports a number of interesting analysis techniques, such as detecting rule dependencies [20] or calculating critical pairs (minimal context of pairs of conflicting rules) [26].

However, graph transformation offers limited support for other kinds of analyses, such as reachability analysis, model checking, etc. This is why some authors define semantic mappings between graph transformation and other semantic domains, and then back-annotate the analysis results to the source notation [10, 17, 18, 26]. This possibility allows one to use the techniques specific to the target semantic domain for analyzing the source models. For example, in [2] rules are translated into Alloy in order to study the applicability of sequences of rules and the reachability of models; in [1] rules are translated into Petri nets to check safety properties; in [46] they are transformed into Promela for model-checking; and in [9, 10] rules are transformed into OCL preconditions and postconditions for rule analysis (e.g. conflict detection) using standard OCL tools.

One of the problems of these approaches is due to the fact that they require, from the DSL designer, deep knowledge of the target language in order to specify the transformations. However, this problem can be partially overcome if the transformations can be automated, using, e.g., model transformation techniques. For example, in [18] the authors are able to generate the transformations from rule-based domain-specific visual languages (DSVLs) into semantic domains with an explicit notion of transition, such as place-transition Petri nets. The generated transformation is expressed in the form of operational triple graph grammar rules that transform the static information (initial model) and the dynamics (source rules and their execution control structure). Similarly, in [10] the authors describe how graph transformation rules can be mapped into OCL preconditions and postconditions for rule analysis. However, as the authors of [10] describe, not all kinds of graph transformations can be automatically transformed into OCL, and the analyses that can be conducted are somewhat limited.

Other approaches use UML behavioral models to represent the system dynamics. For example, in [23] operational semantics are represented using UML collaboration diagrams, which are then formalized into graph transformation rules. In [24], *story diagrams* are presented as a new graph rewrite language based on UML and Java. More precisely, the authors propose the use of UML together with pieces of Java code to express a graph rewrite language mixed with object-oriented data concepts. The whole specification (including dynamic behavior) is then transformed into Java classes and methods, although not all kinds of story patterns can be translated automatically. Again, the kind of analysis is limited in these approaches.

Kermeta [33] is an extension of EMOF (part of the MOF 2.0 specification) for specifying operational semantics. It enriches the EMOF metamodel with an action specification metamodel, introducing another new language to express specifications of algorithms. Simulation and execution possibilities are available for this approach.

Other works propose model transformation languages to specify the semantics of DSLs. For example, in [28], QVT is proposed to specify the semantics of OCL. Then, QVT rules are specified between models conforming to the same metamodel, representing in this way the metamodel behavior, thus acquiring in-place transformations semantics (as graph transformations have). Analysis capabilities are not provided in this case. The MOMENT-QVT tool [6] is a model transformation engine that provides partial support for the QVT relations language. It is based on an algebraically defined operator, ModelGen, which permits the definition of directed declarative transformations.

Another interesting approach for defining semantic mappings in order to specify the semantics of a language is the semantic anchoring method developed at the Vanderbilt University [11]. *Semantic anchoring* relies on the use of well-defined 'semantic units' of simple, well-understood constructs and on the use of model transformations that map higher-level modeling constructs into configured semantic units. This approach uses abstract state machines as a semantic framework, and Microsoft's Abstract State Machine Language (AsmL) and associated tools for programming, simulating, and model checking ASM models [11]. One problem with this kind of approaches is that they normally force the introduction of yet another notation for specifying the mappings [34], or require the mappings to be proved to be correct. By using a single semantic domain (such as our proposal with Maude), we avoid the need to define the so-called semantic mappings.

Poernomo [35] also defines a formal metamodeling framework for MOF based on constructive type theory, where models are defined as terms (token models), that can also be represented as types (type models) by means of a reflection mechanism. Similarly, Boronat and Meseguer have recently proposed an algebraic semantics of MOF in MEL [7], now also part of the MOMENT2 project [43], which provides a reflective, algebraic, and executable framework for metamodeling, with support for MOF and OCL. These two approaches currently provide support for the formal specification of the structural aspects of (MOF-based) models and metamodels, and for different kinds of model management operations (as we also support in our proposal, cf. [41]).

Related to these proposals, other works also try formalize the concepts of concrete metamodeling languages, such as UML or OCL, in Maude. In fact, Maude offers several options to represent and manipulate object-oriented systems, depending on the way in which objects, attributes, and references are represented; whether

reflection is used or not, etc. For example, RIVIERA [42] is a framework for the verification and simulation of UML class diagrams and statecharts. It is based on the representation of class and state models as terms in Maude modules that specify the UML metamodel; it makes a heavy use of the reflective capabilities of Maude, defining and handling most model operations at the metalevel. MOVA [44] is another Maude-based modeling framework for UML, which provides support for OCL constraint validation, OCL query evaluation, and OCL-based metrication. In MOVA, both UML class and object diagrams are formalized as MEL theories. MOMENT [5, 43] is a generic model management framework which uses Maude modules to automatically serialize software artifacts. It supports OCL queries and is also integrated in Eclipse. MOMENT2 [6] is a new version of MOMENT in which the internal encoding of models and metamodels was changed, sharing many similarities with our previous representation [41].

The way in which we specify semantics in Maude differs from those proposals mainly in three aspects: (1) RIVIERA and MOVA define models in a fixed formalism (mainly UML) directly instead of explicitly discussing the metamodel definition too; (2) for those that use OCL, OCL allows the specification of structural semantics through invariants, and behavioral semantics through the definition of preconditions and postconditions on operations, but it does not allow (being side-effect free) the state of a model to be altered, i.e. OCL is not sufficient for expressing rich behavioral semantics; and (3) the proposals that represent models as Maude terms (namely MOMENT and RIVIERA) tend to make a heavy use of the reflective capabilities of Maude, defining and handling most model operations at the meta-level. This approach may have several drawbacks. For example, it increases the complexity of the specifications, makes them much more difficult to write and to maintain, and also has a significant impact on performance.

In fact, we changed the representation we used in our initial proposals [40, 41] to improve the efficiency of the operations and of the analysis tools. More precisely, we moved from Full Maude to Core Maude, and changed the usual representation of Full Maude classes and objects into Core Maude [13]. For instance, we have defined a sort for every metamodel element (metamodel, package, class, attribute, reference, etc.) instead of considering just classes and attributes. We have also defined several operations to gather all of the metamodel element properties, such as structural features cardinalities, reference types and subclasses relations. References types are no longer of the same type, `Oid`, but of the appropriate type so that model validation becomes quite efficient. Subclasses relation operation avoids the need to access the metalevel to check whether a class inherits from another (a common task in metamodel independent operations), something that degrades performance when dealing with large models. Finally, our attribute constructor (`@AttributeInstance`) clearly distinguishes attribute names (`@Attribute`) from attribute values (`@DatatypeInstance`), and therefore they can be accessed independently with a simple Maude match, instead of having to make use of metalevel operations if we used the common representation suggested in [13]. In addition, we have extended the scope of the analysis techniques used to simulate and reason about the models, based on the new notation.

7. Conclusions

According to the MDSD principles, models and metamodels become first-class citizens in the software engineering process. Several notations have been proposed to specify them, although the kind of formal and tool support they provide is always limited. In this paper we have shown how Maude provides an accurate way of specifying both the abstract syntax and the behavioral semantics of models and metamodels, and offers good tool support both for simulating and for reasoning about them.

Furthermore, Maude offers very good properties as a logical and semantic framework, in which many different logics and formalisms can be expressed [29]. Therefore, we expect that our proposal can be used to provide a rich semantic framework to which other proposals can be mapped, therefore allowing them to take advantage of the formal analysis methods and tools of Maude.

There are several lines of work in which we are currently engaged, or that we plan to address in the near future. For example, we are working on making Maude completely transparent to the user. With this aim in mind, we have already connected Maude with AToM3 (a tool for multi-paradigm modeling based on graph grammars) and with EMF. In particular, EMF models and metamodels can already be transformed into their corresponding Maude specifications. Our visual tool *e-Motions* that enables the representation of the behavioral rules is currently under development.

Furthermore, we also want to study the expressiveness of our approach to add semantic information to model transformations (at the end of the day they are also models), to be able to prove other kinds of properties, such as behavior preservation of model transformations [34, 36].

Finally, once we can model the basic behavior of models, we plan to explore the rich expressiveness of Maude to specify further behavioral aspects including, e.g., time, probabilities, quality of service and other non-functional properties, and combined hybrid behavioral semantics. Modeling and simulating continuous behavior of models with Maude is yet another open issue for research.

8. Acknowledgements

The authors would like to thank the anonymous referees for their insightful comments and very constructive suggestions, which have helped us to significantly improve

the contents and readability of the paper. This work has been partially supported by Spanish Research Projects TIN2008-03107, PET2006-0682-00, and P07-TIC-03184.

9. References

[1] Baldan, P., A. Corradini and B. König. 2001. A static analysis technique for graph transformation systems. In *Proceedings of CONCUR 2001* (*Lecture Notes in Computer Science*, Vol. 2154), Springer, Berlin, pp. 381–395.

[2] Baresi, L. and P. Spoletini. 2006. On the use of Alloy to analyze graph transformation systems. In *Proceedings of the Fifth International Conference on Graph Transformation (ICGT 2006)* (*Lecture Notes in Computer Science*, Vol. 4178), Springer, Berlin, pp. 306–320.

[3] Bézivin, J. 2005. On the unification power of models. *Software and Systems Modeling*, 4(2): 171–188.

[4] Bézivin, J. and F. Jouault. 2006. KM3: a DSL for metamodel specification. In *Proceedings of the 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2006)* (*Lecture Notes in Computer Science*, Vol. 4037), Springer, Berlin, pp. 171–185.

[5] Boronat, A., J. A. Carsí and I. Ramos. 2005. Automatic support for traceability in a generic model management framework. In *Proceedings of Model Driven Architecture: Foundations and Applications (ECMDA-FA 2005)* (*Lecture Notes in Computer Science*, Vol. 3748), Springer, Berlin, pp. 316–330.

[6] Boronat, A., J. A. Carsí and I. Ramos. 2006. Algebraic specification of a model transformation engine. In *Proceedings of FASE 2006* (*Lecture Notes in Computer Science*, Vol. 3922), Springer, Berlin, pp. 262–277.

[7] Boronat, A. and J. Meseguer. 2008. An algebraic semantics for MOF. In *Proceedings of FASE 2008*, Budapest, Hungary (*Lecture Notes in Computer Science*, Vol. 4961), Springer, Berlin, pp. 377–391.

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

[9] Büttner, F. and M. Gogolla. 2006. Realizing graph transformations by pre- and postconditions and command sequences. In *Proceedings of the 5th International Conference on Graph Transformation (ICGT 2006)* (*Lecture Notes in Computer Science*, Vol. 4178), Springer, Berlin, pp. 398–413.

[10] Cabot, J., R. Clarisó, E. Guerra and J. de Lara. 2008. Analysing graph transformation rules through OCL. In *Proceedings of the 1st International Conference on Model Transformations (ICMT 2008)* (*Lecture Notes in Computer Science*, Vol. 5063), Springer, Berlin, pp. 225–239.

[11] Chen, K., J. Sztipanovits, S. Abdelwalhed and E. Jackson. 2005. Semantic anchoring with model transformations. In *Proceedings of Model Driven Architecture: Foundations and Applications (ECMDA-FA 2005)* (*Lecture Notes in Computer Science*, Vol. 3748), Springer, Berlin, pp. 115–129.

[12] Clavel, M., F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and J. Quesada. 2002. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285: 187–243.

[13] Clavel, M., F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and C. Talcott. 2007. *All About Maude—A High-Performance Logical Framework*. (*Lecture Notes in Computer Science*, Vol. 4350), Springer, Berlin.

[14] Clavel, M., F. Durán, J. Hendrix, S. Lucas, J. Meseguer and P. Ölveczky. The Maude formal tool environment. In *Proceedings of Algebra and Coalgebra in Computer Science (CALCO'07)* (*Lecture Notes in Computer Science*, Vol. 4624), Springer, Berlin, pp. 173–178.

[15] Cuccuru, A., C. Mraidha, F. Terrier, and S. Grard. Enhancing UML extensions with operational semantics: Behaviored profiles with templates. In *Proceedings of MoDELS 2007* (*Lecture Notes in Computer Science*, Vol. 4735), Springer, Berlin, pp. 271–285.

[16] de Lara, J. and H. Vangheluwe. 2002. AToM3: a tool for multi-formalism and meta-modelling. In *Proceedings of Fundamental Approaches to Software Engineering (FASE 2002)* (*Lecture Notes in Computer Science*, Vol. 2306), Springer, Berlin, pp. 174–188.

[17] de Lara, J. and H. Vangheluwe. 2006. Defining visual notations and their manipulation through meta-modelling and graph transformation. *Journal of Visual Languages and Computing*, 15(3–4): 309–330.

[18] de Lara, J. and H. Vangheluwe. 2008. Translating model simulators to analysis models. In *Proceedings of Fundamental Approaches to Software Engineering (FASE 2008)*, Budapest, Hungary (*Lecture Notes in Computer Science*, Vol. 4961), Springer, Berlin, pp. 77–92.

[19] Durán, F., M. Roldán and A. Vallecillo. 2004. Invariant-driven strategies for Maude. In *Proceedings of the 4th International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2004)*, Aachen, Germany (*Electronic Notes in Theoretical Computer Science*, Vol. 124), Elsevier, Amsterdam, pp. 17–28.

[20] Ehrig, H., K. Ehrig, U. Prange and G. Taentzer. 2006. *Fundamentals of Algebraic Graph Transformation*, Springer, Berlin.

[21] Eker, S., N. Martí-Oliet, J. Meseguer and A. Verdejo. 2007. *Deduction, Strategies, and Rewriting* (*Electronic Notes in Theoretical Computer Science*, Vol. 174(11)), Elsevier, Amsterdam, pp. 3–25.

[22] Eker, S., J. Meseguer and A. Sridharanarayanan. 2002. The Maude LTL model checker. In *Proceedings of the 4th International Workshop on Rewriting Logic and its Applications (WRLA 2002)*, (*Electronic Notes in Theoretical Computer Science*, Vol. 71), Elsevier, Amsterdam, pp. 115–142.

[23] Engels, G., J. H. Hausmann, R. Heckel and S. Sauer. 2000. Dynamic meta modeling: a graphical approach to the operational semantics of behavioral diagrams in UML. In *Proceedings of UML 2000* (*Lecture Notes in Computer Science*, Vol. 1939), Springer, Berlin, pp. 323–337.

[24] Fischer, T., J. Niere, L. Torunski and A. Zündorf. 1998. Story diagrams: a new graph rewrite language based on the unified modeling language. In *Proceedings of the 6th International Workshop on Theory and Application of Graph Transformation*.

[25] Harel, D. and B. Rumpe. 2004. Meaningful modeling: What's the semantics of "semantics"? *Computer*, 37(10): 64–72.

[26] Heckel, R., J. M. Küster, and G. Taentzer. 2002. Confluence of typed attributed graph transformation systems. In *Proceedings of the 1st International Conference on Graph Transformation (ICGT 2002)* (*Lecture Notes in Computer Science*, Vol. 2505), Springer, Berlin, pp. 161–176.

[27] Kleppe, A. G. 2007 A language description is more than a metamodel. In *Proceedings of the 4th International Workshop on Software Language Engineering (ATEM 2007)*, Nashville, TN, 2007, http://megaplanet.org/atem2007/ATEM2007-18.pdf.

[28] Marković, S. and T. Baar. 2008. Semantics of OCL Specified with QVT. *Journal of Software and Systems Modeling*, 7(4): 399–422.

[29] Martí-Oliet, N. and J. Meseguer. 2002. Rewriting logic as a logical and semantic framework. In *Handbook of Philosophical Logic*, Vol. 9, 2nd edition, Kluwer, Dordrecht, pp. 1–87.

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

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

[32] Meseguer, J., M. Palomino and N. Martí-Oliet. 2003. Equational abstractions. In *Proceedings of CADE* (*Lecture Notes in Computer Science*, Vol. 2741), Springer, Berlin, pp. 2–16.

[33] Muller, P.-A., F. Fleurey and J.-M. Jézéquel. Weaving executability into object-oriented meta-languages. In *Proceedings of MoDELS/UML'2005* (*Lecture Notes in Computer Science*, Vol. 3713), Springer, Berlin, pp. 264–278.

[34] Narayanan, A. and G. Karsai. 2006. Using semantic anchoring to verify behavior preservation in graph transformations. In *Proceedings of the 2nd International Workshop on Graph and Model Transformation (GraMoT 2006)* (*Electronic Communications of the EASST*, Vol. 4), pp. 1–14.
[35] Poernomo, I. 2006. The meta-object facility typed. In *Proceedings of the SAC 2006 Track on Software Verification*, Dijon, France, 2006, ACM Press, New York, pp. 1845–1849.
[36] Poernomo, I. 2008. Proofs-as-model-transformations. In *Proceedings of the 1st International Conference on Model Transformations (ICMT 2008)* (*Lecture Notes in Computer Science*, Vol. 5063), Springer, Berlin, pp. 210–224.
[37] Atenea group. http://atenea.lcc.uma.es.
[38] Eclipse Official Site. http://www.eclipse.org/.
[39] Rivera, J. E., E. Guerra, J. de Lara and A. Vallecillo. Analyzing rule-based behavioral semantics of visual modeling languages with Maude. In *Proceedings of the 1st International Conference on Software Language Engineering (SLE'08)*, Toulouse, France, October 2008 (*Lecture Notes in Computer Science*, Vol. 5452), Springer, Berlin.
[40] Rivera, J. E. and A. Vallecillo. 2007. Adding behavioral semantics to models. In *Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007)*, IEEE Computer Society Press, Los Alamitos, CA, pp. 169–180.
[41] Romero, J. R., J. E. Rivera, F. Durán and A. Vallecillo. 2007. Formal and tool support for Model Driven Engineering with Maude. *Journal of Object Technology*, 6(9): 187–207.
[42] Sáez, J., A. Toval and J. L. Fernández Alemán. 2001. Tool support for transforming UML models to a formal language. In *Proceedings of the International Workshop on Transformations in UML (WTUML)*, pp. 111–115.
[43] The ISSI Research Group. MOMENT. http://moment.dsic.upv.es.
[44] The MOVA Group. 2006. The MOVA tool: a validation tool for UML. http://maude.sip.ucm.es/mova/.
[45] Vangheluwe, H. and J. de Lara. 2007. Automatic generation of model-to-model transformations from rule-based specifications of operational semantics. In *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM'07)*.
[46] Varró, D. 2004. Automated formal verification of visual modeling languages by model checking. *Journal of Software and Systems Modeling*, 3(2): 85–113.