Adding Behavioral Semantics to Models

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

Abstract

Domain Specific Languages (DSLs) play a cornerstone role in Model-Driven Software Development for representing models and metamodels. DSLs are usually defined in terms of their abstract and concrete syntax only. This allows the rapid and inexpensive development of DSLs and their associated tools (e.g., editors), but does not allow the representation of their behavioral semantics, something especially important for model operations like simulation and verification. In this paper we explore the use of Maude as a formal notation for describing models and metamodels, including the specification of their dynamic behavior.

1 Introduction

As software technology becomes a core part of business enterprises in all market sectors, customers demand more flexible enterprise systems. This demand coincides with the increasing use of personal computers and today's easy access to local and global communication networks, that together provide an excellent infrastructure for building open distributed systems. However, the specific problems and intrinsic complexity of these large systems are currently challenging the Software Engineering community, whose traditional methods and tools are finding difficulties for coping with the new requirements. One common way to deal with this complexity is by dividing the design activity into a number of areas of concern, each one dealing with a specific aspect of the system, and using specialized languages to specify them. These languages allow not only to raise the level of abstraction of the specifications produced, but also make them closer to the IT experts. Domain-Specific Modeling (DSM) can thus become a key mechanism for the effective and successful specification of large-scale distributed enterprise systems.

Domain-Specific Modeling 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 into the language as constraints, disallowing the specification of illegal or incorrect models.

DSLs play a cornerstone role in DSM. In general, defining a modeling language involves at least two aspects: the domain concepts and rules (abstract syntax), and the notation used to represent these concepts (concrete syntax)—let it be textual or graphical. Each model is written in the language of its metamodel. Thus, a metamodel will describe the concepts of the language, the relationships between them, and the structuring rules that constrain the model elements and 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 such meta-metalevel, since meta-metamodels conform to themselves.

This metamodeling approach enables the rapid and inexpensive development of DSLs and their associated tools (e.g., editors), and is becoming very popular in all DSM proposals—both academic and from major vendors and organizations (the OMG, Microsoft, IBM, etc).

So far, most of the efforts have been focused on the specification of the structure of models and metamodels, i.e., the concepts that comprise the language, and their structuring rules. However, there is something more in metamodels than these static semantics. In fact, there is also a growing interest in the Model-Driven Software Development (DSDM) community for the specification of the behavioral semantics of DSLs—something especially important for model operations like simulation and verification. This kind of semantics would also allow to define a reference semantics for a given DSL, and could also be used to

1541-7719/07 $25.00 © 2007 IEEE
DOI 10.1109/EDOC.2007.40

check the behavioral equivalence with different translations from that DSL to other technical spaces.

As described by Chen et al., the semantics of a DSL may be either structural or behavioral [6]. 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 needs to be formally captured by a mathematical framework representing the appropriate form of dynamics.

In this paper we explore the use of Maude [8] as a formal notation and system for supporting the specification of both the structural (i.e., the abstract syntax) and the behavioral semantics of models and metamodels, in a natural and integrated way. This paper builds on our previous work [21] on how to represent metamodels with Maude, extending it to cope with the specification of their behavioral semantics. The use of Maude provides additional advantages. 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 run-time verification [14], model checking [13], or theorem proving [9]. Maude 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. These facilities, together with the capabilities of Maude to serve as a logical and semantic framework in which many logics can be smoothly integrated [15], made us think of Maude as an appropriate notation and semantic framework for specifying models and metamodels.

The structure of this document is as follows. First, Section 2 serves as a brief introduction to Maude. Then, Section 3 describes how the abstract syntax of models and metamodels can be represented in Maude. Section 4 is dedicated to show how the behavioral semantics of metamodels can be naturally added to the syntactical specifications, and how they become amenable to formal analysis. Finally, Section 5 compares our work with other related proposals and Section 6 draws some conclusions and outlines some future research activities.

2 Rewriting Logic and Maude

2.1 Introduction to Maude

Maude [7, 8] is a high-level language and a high-performance interpreter and compiler in the OBJ algebraic specification family that supports membership equational logic and rewriting logic specification and programming of systems. Thus, Maude integrates an equational style of functional programming with rewriting logic computation. Because of its efficient rewriting engine, able to execute more than 3 million rewriting steps per second on standard PCs, and because of its metalanguage capabilities, Maude turns out to be an excellent tool to create executable environments for various logics, models of computation, theorem provers, or even programming languages. In addition, Maude has been successfully used in software engineering tools in applications [16]. We informally describe in this section those Maude's features necessary for understanding the paper; the interested reader is referred to the Maude book [8] for more details.

Rewriting logic is a logic of change that can naturally deal with state and with highly nondeterministic concurrent computations. A distributed system is axiomatized in rewriting logic by a *rewrite theory* $\mathcal{R} = (\Sigma, E, R)$, where (Σ, E) is an equational theory describing its set of *states* as the algebraic data type $T_{\Sigma/E}$ associated to the initial algebra (Σ, E), and R is a collection of rewrite rules. Maude's underlying equational logic is membership equational logic, 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.

For example, the following Maude functional module `NATURAL` defines the natural numbers (with sorts `Nat` of natural numbers and `NzNat` of nonzero 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
```

If a functional specification is terminating, confluent, and sort-decreasing, then it can be executed. Computation in a functional module is accomplished by using the equations as simplification rules from left to right until a canonical form is found. Some equations, like 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`), identity (`id`), and idempotence (`idem`). The above properties must

therefore be understood in the more general context of simplification *modulo* such equational theories.

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 of the form $t \rightarrow t'$, where t and t' are Σ-terms, which specify the dynamics of a system in rewriting logic. 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 guards of conditional rules act as blocking pre-conditions, in the sense that a conditional rule can only be fired if the condition is satisfied.

2.2 Object-Oriented Specifications: Full Maude

In Maude, concurrent object-oriented systems are specified by object-oriented modules in which classes and subclasses are declared. A class is declared with the syntax `class` C `|` $a_1 : S_1$, ..., $a_n : S_n$, where C is the name of the class, a_i are attribute identifiers, and S_i are the sorts of the corresponding attributes. Objects of a class C are then record-like structures of the form `<` O `:` C `|` $a_1 : v_1$, ..., $a_n : v_n$ `>`, where O is the name of the object, and v_i are the current values of its attributes. Objects can interact in a number of different ways, including message passing. Messages are declared in Maude in `msg` clauses, in which the syntax and arguments of the messages are defined.

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 using rules that describe the effects of the communication events of objects and messages. The predefined sort `Configuration` represents configurations of Maude objects and messages, with `none` as empty configuration and the empty syntax operator `_ _` as union of configurations.

```
sort Configuration .
subsorts Object Message < Configuration .
op none : -> Configuration [ctor] .
op _ _  : Configuration Configuration
    -> Configuration [ctor assoc comm id: none] .
```

Thus, rewrite rules define transitions between configurations, and their general form is:

```
crl [r] :
  < O1 : C1 | atts1 > ... < On : Cn | attsn >
  M1 ... Mm
  => < Oi1 : C'i1 | atts'i1 > ... < Oik : C'ik | atts'ik >
     < Q1 : C''1 | atts''1 > ... < Qp : C''p | atts''p >
     M'1 ... M'q
  if Cond .
```

where r is the rule label, $M_1 ... M_m$ and $M'_1 ... M'_q$ are messages, $O_1 ... O_n$ and $Q_1 ... Q_p$ are object identifiers, $C_1 ... C_n$, $C'_{i_1} ... C'_{i_k}$ and $C''_1 ... C''_p$ are classes, $i_1 ... i_k$ is a subset of $1 ... n$, and *Cond* is a Boolean condition (the rule's *guard*). The result of applying such a rule is that: (a) messages $M_1 ... M_m$ disappear, i.e., they are consumed; (b) the state, and possibly the classes of objects $O_{i_1} ... O_{i_k}$ may change; (c) all the other objects O_j vanish; (d) new objects $Q_1 ... Q_p$ are created; and (e) new messages $M'_1 ... M'_q$ are created, i.e., they are sent. Rule labels and guards are optional.

For instance, the following Maude module, `ACCOUNT`, specifies a class `Account` with an attribute `balance` of sort integer (`Int`), a message `withdraw` with an object identifier (of sort `Oid`) and an integer as arguments, and two rules describing the behavior of the objects belonging to this class. The rule `debit` specifies a local transition of the system when there is an object `A` of class `Account` that receives a `withdraw` message with an amount smaller or equal than the balance of `A`; as a result of the application of such a rule, the message is consumed, and the balance of the account is modified. The rule `transfer` models the effect of receiving a money transfer message.

```
(omod ACCOUNT is
   class Account | balance : Int .
   msg withdraw : Oid Int -> Msg .
   msg transfer : Oid Oid Int -> Msg .
   vars A B : Oid .
   vars M Bal Bal' : Int .
   crl [debit] :
     withdraw(A, M)
     < A : Account | balance : Bal >
     => < A : Account | balance : Bal - M >
     if M <= Bal .
   crl [transfer] :
     transfer(A, B, M)
     < A : Account | balance : Bal >
     < B : Account | balance : Bal' >
     => < A : Account | balance : Bal - M >
        < B : Account | balance : Bal' + M >
     if M <= Bal .
endom)
```

When several objects or messages appear in the left-hand side of a rule, they need to synchronize in order for such a rule to be fired. These rules are called *synchronous*, while rules involving just one object and one message in their left-hand sides are called *asynchronous* rules.

Class inheritance is directly supported by Maude's order-sorted type structure. A subclass declaration `C < C'`, indicating that `C` is a subclass of `C'`, is a particular case of a subsort declaration `C < C'`, by which all attributes, messages, and rules of the superclasses, as well as the newly defined attributes, messages and rules of the subclass characterize its structure and behavior. This corresponds to the traditional notion of subtyping: A is a subtype of B if every <X> that satisfies A also satisfies B. Multiple inheritance is also supported in Maude [7, 8].

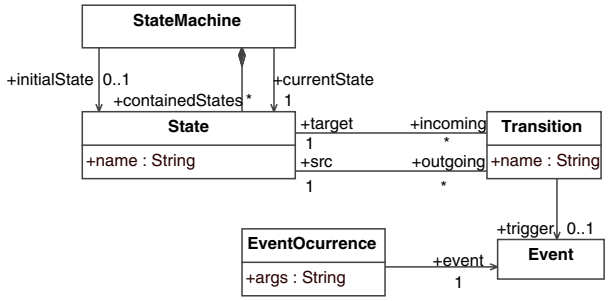

Figure 1. Simple State Machine Metamodel.

3 Formalizing models and metamodels with Maude

There are several notations to represent models and metamodels, from textual to graphical. One of particular interest to us is KM3, a specialized textual language for specifying metamodels, whose abstract syntax is based on Ecore and MOF 2.0. Thus, KM3 resembles the Ecore terminology and has the notions of package, class, attribute, reference and datatype. The following is a possible KM3 specification of a metamodel for simple state machines, which is also depicted in Figure 1.

```
package SimpleStateMachine {
 class State {
   attribute name : String;
   reference stateMachine : StateMachine
                    oppositeOf containedStates;
   reference incoming [*] : Transition
                    oppositeOf target;
   reference outgoing [*] : Transition
                    oppositeOf src;
 }
 class StateMachine {
   reference initialState [0-1] : State;
   reference currentState : State;
   reference containedStates [*] container : State
                    oppositeOf stateMachine;
 }
 class Transition {
   attribute name : String;
   reference target [1] : State
                    oppositeOf incoming;
   reference src [1] : State
                    oppositeOf outgoing;
   reference trigger [0-1] : Event;
 }
 class Event {
   attribute name : String;
 }
 class EventOccurrence {
   attribute args : String;
   reference event : Event;
 }
}
```

There are many interesting benefits of using KM3, such as: it is simple and easy to learn and to understand; it allows precise and easy definition and modification of the metamodels; it is possible to convert MOF, Ecore, and other metamodel languages to/from KM3 descriptions; and KM3 offers good tool support and is integrated with a proven and widely accepted MDSD environment, the ATLAS Model Management Architecture (AMMA). In addition, KM3 metamodels can be included into model repositories (zoos) and be ready to allow mega-modeling [4].

However, KM3 is not the only notation for specifying the syntax of models and metamodels. In [21] we presented a proposal based on the use of Maude, which not only was expressive enough for these purposes, but also offered good tool support for reasoning about 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. This section presents just a brief summary of that proposal.

In Maude, models are represented by configurations of objects. Nodes are represented by Maude objects. Nodes may have attributes, that are represented by Maude objects' attributes. Edges are represented by Maude objects' attributes, too, each one representing the reference to the target node of the edge.

Due to the way of representing models, we have two ways of representing metamodels. Firstly, we can represent a metamodel as a Maude object-oriented module, which contains the specification of the Maude classes to which the Maude objects that represent the corresponding models nodes belong. In this way, models conform to metamodels by construction.

Secondly, since metamodels are models too, they can be represented by configurations of objects. The classes of such objects will be the ones that specify the metametamodels, for example, the classes that define the KM3 metamodel.

To illustrate the first option of representing metamodels, the following piece of Maude specifications describe Simple State Machine metamodel as a Maude module.

```
(omod SimpleStateMachines is
   protecting STRING .
   class State |
       name : String,
       stateMachine : Oid,
       incoming : Set{Oid},
       outgoing : Set{Oid} .
   class StateMachine |
       containedStates : Set{Oid},
       currentState : Oid,
       initialState : Maybe{Oid} .
   class Transition |
       name : String, target : Oid,
```

```
        src : Oid, trigger : Maybe{Oid} .
    class Event |
        name : String .
    class EventOccurrence |
        args : String , event : Oid .
endom)
```

Then, KM3 classes correspond to Maude classes. Attributes are represented as Maude attributes. References are represented as attributes too, by means of sets of Maude object identifiers. Depending on the multiplicity, we can use: a single identifier (if the multiplicity is `1`); a `Maybe{Oid}` which is either an identifier or a `null` value, for representing a `[0-1]` multiplicity; a `Set{Oid}` for multiplicity `[*]`; or a `List{Oid}` in case the references are `ordered`. Notice that this representation abstracts away some KM3 notions, such as `oppositeOf`. This and other KM3 aspects will be considered in the alternative way of representing metamodels below.

The instances of such classes will represent models that conform to the example metamodel. For instance, the configuration of Maude objects shown in Figure 2 represents a possible state machine model that conforms to that metamodel. It represents a simple state machine with two states, named `St1` and `St2`, and one transition (`Tr`) between them. `St1` is the initial state of the state machine, and also its current state. `Tr` is triggered by the occurrences of event `Ev`.

The validity of the objects in a configuration is checked by the Maude type system. In addition, the valid types of the objects being referenced is expressed in Maude in terms of membership axioms that define the well-formedness rules that any valid model should conform to: a configuration is valid if it is made of valid objects, with valid references. In our example, the well-formedness rules of the simple state machines metamodel are shown in Figure 3.

Our second way of modeling metamodels is considering them as models, too. Therefore, they can also be represented as configurations of objects. The classes of such objects will be the ones that specify the meta-metamodels—for example, the classes defined in the KM3 metamodel.

To illustrate this approach, the configuration of Maude objects shown in Figure 4 represents the Simple State Machine metamodel. The Maude specification of the classes of these objects corresponds, of course, to the KM3 metamodel represented in Maude.

It is important to note that these two different representations of a metamodel are not completely equivalent. In fact, the second one contains all the information about the metamodel, while the former one describes just information about the models themselves—i.e., as a configuration of objects of certain classes with references to other objects. Thus, some information not relevant at this level is omitted, or checked with Maude equations, such as whether a class is abstract or not, or whether a reference is a container or the opposite of other. This is similar to the information captured by UML object diagrams, in which the relevant information are the object identifiers, their classifiers, and the links between them—but no information is shown about how the classifiers of such objects are organized into packages or structured in an inheritance hierarchy, or the kinds of associations of their links.

Then, in our proposal we use both approaches, because they are useful for different reasons. In all cases we represent metamodels as configurations of Maude objects (i.e., the second option above) to be able to capture all their relevant information, and to be able to reason about them using Maude (see [21]). But we also represent them as Maude specifications (i.e., using the first option that we have described) in order to be able to instantiate models from them in a natural way, and to add the behavioral semantics to them, as we shall see in next section. There is a clear relationship between these two representations: we can easily obtain the first representation from the second one.

In fact, using both representation in combination is a clear advantage of our proposal when compared to other approaches that use Maude for formalizing UML models (e.g., Moment [5] or Riviera [22]), as we discuss later in Section 5. The fact that each alternative representation is more appropriate for different usages allows us always work with the one that better suits each situation.

4 Semantics

As we have previously mentioned, there are many proposals that use metamodeling techniques (e.g., metamodeling languages, metamodels, or UML profiles) to describe the abstract syntax of DSLs. Such abstract syntax is expressed in terms of the concepts of the language, their relationships, and a set of structural (or well-formedness) rules.

However, explicit and formal specification of a model behavioral semantics has not received much attention by the MDSD community, despite the fact that this creates a possibility for semantic mismatch between design models and modeling languages of analysis tools. While this difficulty exists in virtually every domain where DSLs are used, it is more common in domains in which behavior needs to be explicitly represented. Furthermore, this issue is particularly problematic in safety-critical real-time and embedded system domains, where semantic ambiguities may produce conflicting results across different tools.

For instance, it is not clear from the `SimpleStateMachines` metamodel what happens if an event occurs and there is no transition that can be triggered. Is the event lost, or is it held until the state machine reaches a state with a transition that can be triggered by the event? What is the behavior of the system when it contains internal transitions (i.e., those that do not require the occurrence of any event to be triggered)? How

```
< 'S : StateMachine | containedStates : ('A, 'B), initialState : 'A, currentState : 'A >
< 'A : State | name : "St1", stateMachine : 'S, outgoing : 'T, incoming : empty >
< 'B : State | name : "St2", stateMachine : 'S, incoming : 'T, outgoing : empty >
< 'T : Transition | name : "Tr", src : 'A, target : 'B, trigger : 'E >
< 'E : Event | name : "Ev" >
```

Figure 2. Simple State Machine model.

```
vars MODEL CONF CONF1 CONF2 : Configuration .
vars O S T C E SM : Oid .
vars CS IN OUT : Set{Oid} .
vars i TR : Maybe{Oid} .

subsort ValidStateMachine < Configuration .

cmb CONF : ValidStateMachine if validRefs(CONF) .

op validRefs : Configuration -> Bool .
op validRefs : Configuration Configuration -> Bool .

eq validRefs(CONF) = validRefs(none, CONF) .

ceq validRefs(CONF1, < O : State | stateMachine : SM, incoming : IN, outgoing : OUT > CONF2)
  = isKindOf(SM, StateMachine, MODEL)
    and-then isSetOf(IN, Transition, MODEL)
    and-then isSetOf(OUT, Transition, MODEL)
    and-then validRefs(CONF1 < O : State | >, CONF2)
  if MODEL := CONF1 < O :  State | > CONF2 .

ceq validRefs(CONF1, < O : StateMachine | initialState : I, currentState : C,
                                          containedStates : CS > CONF2)
  = isNullOrKindOf(I, State, MODEL)
    and-then isKindOf(C, State, MODEL)
    and-then isSetOf(CS, State, MODEL)
    and-then validRefs(CONF1 < O : StateMachine | >, CONF2)
  if MODEL := CONF1 < O :  StateMachine | > CONF2 .

ceq validRefs(CONF1, < O : Transition | target : T, src : S, trigger : TR > CONF2)
  = isKindOf(T, State, MODEL)
    and-then isKindOf(S, State, MODEL)
    and-then isNullOrKindOf(TR, Event, MODEL)
    and-then validRefs(CONF1 < O : Transition | >, CONF2)
  if MODEL := CONF1 < O :  Transition | > CONF2 .

ceq validRefs(CONF1, < O : Event | > CONF2) = validRefs(CONF1 < O : Event | >, CONF2) .

ceq validRefs(CONF1, < O : EventOccurrence | event : E > CONF2)
  = isKindOf(E, Event, MODEL)
    and-then validRefs(CONF1 < O : EventOccurrence | >, CONF2)
  if MODEL := CONF1 < O :  Transition | > CONF2 .

eq validRefs(CONF1, none) = true .
eq validRefs(CONF1, CONF2) = false [owise] .
```

Figure 3. Well-formedness rules

```
< 'SMP : KM3Package | name : "SimpleStateMachine", metamodel : 'MM,
   contents : ('STATE, 'STATEMACHINE, 'TRANSITION, 'EVENT, 'EVENTOCCURRENCE), package : null >
< 'STATE : KM3Class | name : "State", isAbstract : false, package : 'SMP, superTypes : empty,
   structuralFeatures : ('STATENAME, 'STATESTATEMACHINE, 'STATEINCOMING, 'STATEOUTGOING) >
< 'STATENAME : KM3Attribute | name : "name", package : 'SMP, type : 'STRING,
   owner : 'STATE, lower : 1, upper : 1, isOrdered : false, isUnique : false >
< 'STATESTATEMACHINE : KM3Reference | name : "stateMachine", type : 'STATEMACHINE,
   package : 'SMP, owner : 'STATE, lower : 1, upper : 1, isOrdered : false,
   isUnique : false, opposite : 'STATEMACHINECONTAINEDSTATES, isContainer : false >
< 'STATEINCOMING : KM3Reference | name : "incoming", type : 'TRANSITION,
   package : 'SMP, owner : 'STATE, lower : 0, upper : *, isOrdered : false,
   isUnique : false, opposite : 'TRANSITIONTARGET, isContainer : false >
< 'STATEOUTGOING : KM3Reference | name : "outgoing", type : 'TRANSITION,
   package : 'SMP, owner : 'STATE, lower : 0, upper : *, isOrdered : false,
   isUnique : false, opposite : 'TRANSITIONSRC, isContainer : false >
< 'STATEMACHINE : KM3Class | name : "StateMachine", ... >
< 'TRANSITION : KM3Class | name : "Transition", ... >
```

Figure 4. Simple State Machine KM3 model expressed as Configuration of Maude objects

do they exactly behave? When are they triggered? These are the sort of issues that require to be precisely clarified by a behavioral specification.

The need for the specification of the behavioral semantics of models was also recently raised by Robin Milner in [17], where he mentioned that a (meta)model "consists of some concepts, *and a description of permissible activity in terms of these concepts*." (The emphasis is ours.) The question is then: how do we represent such activities and formalize them so they are amenable for reasoning about the properties of the system being specified?

4.1 Behavior as rewrite rules

Behavioral semantics are specified in Maude in terms of rewrite rules, which are added to the corresponding Maude specifications that contain the description of metamodels. Therefore, given a Maude module with the specification of the abstract syntax of a metamodel, we can "extend" it (by adding the appropriate rewrite rules) with the semantic information.

To illustrate this approach, the following Maude module `SSMWithBehavior1` extends the `SimpleStateMachines` specification with two rules.

```
(omod SSMWithBehavior1 is
  pr SimpleStateMachines .
  vars S T E A B X : Oid .

  rl [AnEventOccurs] :
   < S : StateMachine | currentState : A >
   < T : Transition | src : A, target : B,
                      trigger : E >
   < X : EventOccurrence | event : E >
   => < S : StateMachine | currentState : B >
      < T : Transition | > .

  rl [InternalTransition] :
   < S : StateMachine | currentState : A >
   < T : Transition | src : A, target : B,
                      trigger : null >
   => < S : StateMachine | currentState : B >
      < T : Transition | > .
endom)
```

First, rule `AnEventOccurs` specifies the behavior of the system when an event occurs, and the state machine is in a state with a transition that can be triggered by an event of that kind. In this case, the transition takes place (i.e., the `current` state of the state machine is changed) and the event occurrence is consumed: it disappears in the right hand side of the rule. (Please notice that in Maude, the attributes of an object which are unchanged by a rule do not need to be specified.)

Second, rule `InternalTransition` specifies the behavior of internal transitions. If the state machine is in a state that allows a transition to be triggered without the occurrence of any event, the transition takes place.

Note that if we do not add any further rule, the semantics of the rewriting rules in Maude precisely specify what happens if there is no transition that can be triggered when an event occurs: it is held until an state is reached in which a transition for that event occurrence can be triggered. Similarly, the rest of the behavioral aspects of the system are dictated by the semantics of Maude rewriting rules. Thus, in case that several rules can be fired under the occurrence of an event, only one will be fired in an indeterministic way. Of course, this behavior can be changed by the inclusion of the appropriate rules, or by the use of Maude's *strategies* for controlling the execution of the system [7].

Furthermore, several alternative behaviors can be specified by extending the original `SimpleStateMachines` specification in different ways. For instance, we could define another Maude modules `SSMWithBehavior2` and `SSMWithBehavior3` that include `SimpleStateMachines` and extend it with different

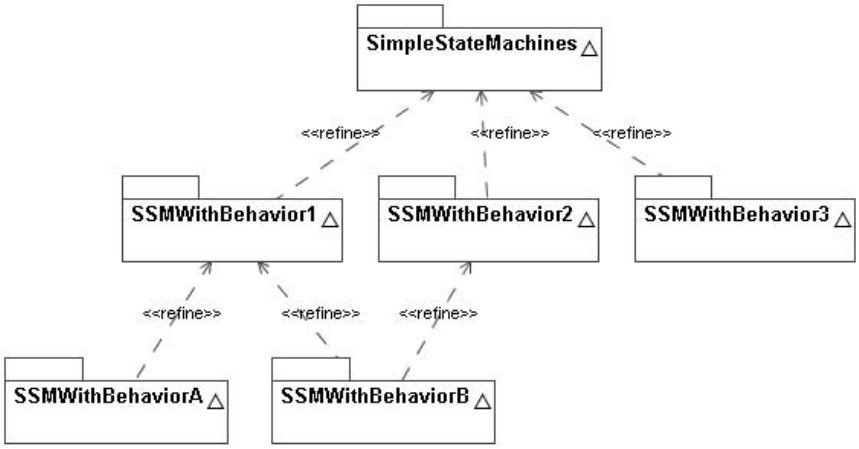

Figure 5. Different behavioral specifications for a metamodel.

rules (see Figure 5). Moreover, Maude defines a complete module algebra [12] which, among other operations, allows for module inheritance. This therefore becomes a very powerful mechanism in this context for refining behavioral specifications.

4.2 Formal Analysis

Once the system specifications are written using this modeling approach, what we get is a rewriting logic specification of the system. Since the rewriting logic specifications produced are executable, this specification can be used as a prototype of the system, which allows us to simulate it. Notice however that when executing the system we only get one of the possible executions, and this might not be enough in many situations. But then, Maude offers tool support for other interesting possibilities such as reachability analysis, model checking [13], or theorem proving [9].

For example, we could check whether a given state is reached in the state machine, starting from an initial state and a set of event occurrences. Then, suppose that we want to search for those executions in which the state of a state machine is `'B`, starting from the initial configuration `initialConf`. Given a variable `C` of sort `Configuration`, the `search` command can be used for such a search as follows.

```
(search initialConf =>*
  < 'S : StateMachine | currentState : 'B > C .)
```

The result will be a collection of solutions that fulfil the given search pattern. In this way we can search for executions leading to undesirable states that we want to avoid in the system, or for situations that violate any of the properties that we want to prove on the system.

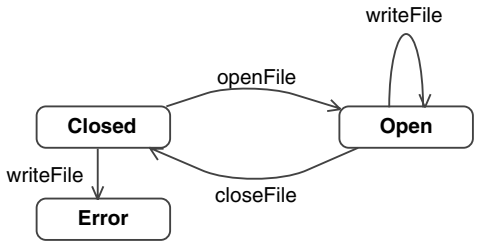

Figure 6. A File Simple State Machine.

For instance, suppose that we have the state machine model depicted in Figure 6. It is a simple state machine for Files with three states: `Open`, `Closed` and `Error`. Operations `openFile`, `closeFile` and `writeFile` can be issued. This state machine model can be represented in Maude as shown in Figure 7.

Thus, if `Error` is considered as an undesirable state, we would like to know if there exists a possible execution that reaches it starting, for example, from a couple of `writeFile` and `openFile` operation calls on a File in the `Closed` state. Firstly we define our `initialConf` as the union of the configuration shown in Figure 7 and these two event occurrences.

```
< 'OFOC : EventOccurrence | event : 'OF,
     args : "" >
< 'CFOC : EventOccurrence | event : 'WF,
     args : "Hello world ." > .
```

Then, given a variable `C` of sort `Configuration`, we can use the `search` command as follows:

```
(search initialConf =>*
```

```
< 'SM : StateMachine | containedStates : ('CS, 'OS, 'ES), initialState : 'CS, currentState : 'CS >
< 'CS : State | name : "Closed", stateMachine : 'SM, outgoing : ('C2O, 'C2E), incoming : 'O2C >
< 'OS : State | name : "Open", stateMachine : 'SM, outgoing : ('O2O, 'O2C), incoming : ('C2O, 'O2O) >
< 'ES : State | name : "Error", stateMachine : 'SM, outgoing : empty, incoming : 'C2E >
< 'C2O : Transition | name : "Closed2Open", src : 'CS, target : 'OS, trigger : 'OF >
< 'O2C : Transition | name : "Open2Closed", src : 'OS, target : 'CS, trigger : 'CF >
< 'O2O : Transition | name : "Open2Open", src : 'OS, target : 'OS, trigger : 'WF >
< 'C2E : Transition | name : "Closed2Error", src : 'CS, target : 'ES, trigger : 'WF >
< 'OF : Event | name : "openFile" >
< 'CF : Event | name : "closeFile" >
< 'WF : Event | name : "writeFile" >
```

Figure 7. The File Simple State Machine configuration of objects.

```
< 'SM : StateMachine | currentState : 'ES > C .)
```

Figure 8 shows the results of this command (the object `'SM` is not included in the resulting Maude term since it is part of the query). As expected, Maude obtains one possible execution that reaches such a state. It happens if the `writeFile` event occurrence `'CFOC` is consumed (by rule `AnEventOccurs`) before the `openFile` event occurrence `'OFOC` is consumed, i.e., if the write operation happens before the file is opened. (The opposite sequence would lead us to the `Open` state).

The search commands are very useful, and usually help to uncover many undesired situations. However, by searching we cannot reach any definitive conclusion, we can only gain certain level of confidence in the specification. Maude also offers a *linear temporal logic explicit-state model checker* [13], which also allows us to check whether every possible behavior starting from a given initial configuration satisfies a given (temporal) logic property. Furthermore, the *theorem prover* can be useful in case there is the need to thoroughly prove other properties (see [9] for useful examples and potential usages of the Maude model checker).

4.3 Tool Support

One of the main advantages of Maude is the possibility of using its execution environment, able to provide efficient implementations of the specifications—comparable in resource consumption to most commercial programming languages' environments. Our work so far consists of developing an Eclipse plug-in, called Maudeling [20], that allows to provide the formal specifications of KM3 models and metamodels. ATL (the ATLAS Transformation Language) is used by the tool to automatically transform the KM3 models into their corresponding Maude representations. Then the desired behavioral semantics can be added, and the corresponding operations can be executed in the Maude environment.

5 Related work

There are several lines of research that are closely related to ours. Firstly, there are some works that formalize some of the UML models using Maude. Among them, RIVIERA [22] is a framework for the verification and simulation of UML class diagram models and statecharts. It is based on the representation of class diagrams and statecharts as terms in Maude modules that specify the UML metamodel.

MOMENT [5] is a generic model management framework. It uses Maude modules to automatically serialize software artifacts. It supports OCL queries (but not OCL constraints over UML models). MOMENT is integrated in Eclipse, an open platform for tool integration. The specification of OCL queries is done manually, and it requires a deep understanding of both Maude and the MOMENT specific representation of UML diagrams.

Finally, ITP/OCL [10] is a rewriting- based tool that supports automatic validation of UML static class diagrams with respect to OCL invariants. From a conceptual point of view, the ITP/OCL tool is directly based on the equational specification of UML+OCL class diagrams in which class and object diagrams are specified as membership equational theories (and not as terms); invariants are represented as Boolean terms over extensions of those theories; and checking invariants over object diagrams is reduced to inspecting whether the corresponding Boolean terms rewrite to true or false.

We differ from those proposals mainly in three aspects: (1) other Maude proposals tend to define models in a fixed formalism (mainly UML) directly instead of explicitly discussing the metamodel definition too; (2) for those who use OCL, OCL allows to specify structural semantics through invariants, and behavioral semantics through the definition of pre- and post- conditions on operations, but does not allow (being side-effect free) to alter the state of a model, i.e., OCL is not sufficient for expressing rich behavioral semantics; and (3) we have dual representation of metamodels as both Maude specifications and Maude terms, being therefore able to handle them in the most appropriate way de-

```
rewrites: 1052 in 4344030914ms cpu (88ms real) (0 rewrites/second)
search in SSMExample :
    initialConf =>* C:Configuration < 'SM : StateMachine | currentState : 'ES > .

Solution 1 C:Configuration -->
    < 'C2E : Transition | name : "Closed2Error", src : 'CS, target : 'ES, trigger : 'WF >
    < 'C2O : Transition | name : "Closed2Open", src : 'CS, target : 'OS, trigger : 'OF >
    < 'O2C : Transition | name : "Open2Closed", src : 'OS, target : 'CS, trigger : 'CF >
    < 'O2O :Transition | name : "Open2Open", src : 'OS, target : 'OS, trigger : 'WF >
    < 'CS : State | incoming : 'O2C, name : "Closed", outgoing :('C2E, 'C2O), stateMachine : 'SM >
    < 'ES : State | incoming : 'C2E, name : "Error", outgoing : empty, stateMachine : 'SM >
    < 'OS : State | incoming :('C2O, 'O2O), name : "Open", outgoing :('O2C, 'O2O), stateMachine : 'SM >
    < 'CF : Event | name : "CloseFile" >
    < 'OF : Event | name : "OpenFile" >
    < 'WF : Event | name : "WriteFile" >
    < 'OFOC : EventOccurrence | event : 'OF,args : "" >

No more solutions.
```

Figure 8. Search command solution

pending on the operations we want to apply on them.

The proposals that represent models as Maude terms (namely MOMENT and RIVIERA) make a heavy use of the reflective capabilities of Maude, defining and handling most model operations at the meta-level. This approach has several drawbacks. For example, it heavily increases the complexity of the specifications, makes them much more difficult to write and to maintain, and also has a strong impact on performance. The fact that we define both the model operations and the model behavior as natural extensions of the Maude modules greatly simplifies the description of the specifications (and hence their understandability and maintenance), and is much more efficient when it comes to evaluating the operations or simulating the system behavior. Regarding the ITP/OCL tool, the approach it follows can be compared to our first way of representing metamodels. However, we have shown the benefits that representing models and metamodels as configuration of objects too (and not only as theories) can bring along. In particular, this dual representation allows the natural definition and evaluation of model operations on them, greatly simplifying their specification and also the way to reason about them.

Secondly, we have the works that try to provide formal support for KM3. So far it is quite limited. There is a very interesting proposal to formalize the semantics of KM3 using the Abstract State Machines notation [11], but there is no connection yet to any formal toolkit to reason about the formal specifications produced, nor have all the potential benefits of having a formal representation of the models and metamodels been fully exploited. And there is also a recent initiative by Thirioux et al. [23], who propose a framework to give a formal foundation of the Model-Driven Engineering (MDE) approach. They define the notions of model and reference model based on typed multigraphs, and formalize the two operations that are fundamental to the MDE approach, *conformsTo* that indicates whether a model is valid with respect to a reference model, and *promotion* which builds a reference model from a model. However, they just deal with the static semantics of metamodels, applied on EMOF. The way to deal with operational semantics is left open as future work, although they mention that it would allow to define a reference semantics for a given DSL, and could be used, for instance, to check the behavioral equivalence with different translations from this DSL to other technical spaces. This is precisely what we have shown that can be done with our proposal.

Thirdly, there are already several proposals for expressing the behavioral semantics of models and metamodels. Examples of such works specification include, among others, the transformational approaches that specify the operational semantics of DSLs using graph transformations (e.g., [1, 24]), or the semantic anchoring method developed at the Vanderbilt University [6], which represents an important step in this direction. *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 Microsofts Abstract State Machine Language (AsmL) and associated tools for programming, simulating and model checking ASM models [6]. Our work can be considered to be in this line too, but using Maude for both as a logical and semantical framework [15], and for specifying and reasoning about the models, avoiding in this way the definition of *mappings* between them.

6 Conclusions

According to the MDSD principles, models and metamodels become first-class citizens in the software engineer-

ing process. Several notations have been proposed to specify them, although the kind of formal and tool support they provide is quite 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 for reasoning about them.

There are several lines of work in which we are currently engaged, or that we plan to address in the near future. Firstly, we are working on the specification on more model operations, such as deep model copy [19], match, diff, merge, compose, or apply [2]. Our plan is to make them all available as part of the Maude model management tool-kit, in addition to the ones presented in [21].

Secondly, We are also working on improving the integration with other tools, being able to deal not only with KM3 models, but also with, e.g., MOF or Ecore metamodels. Although it is possible to convert MOF and Ecore to/from KM3 descriptions, these metamodels incorporate new elements (operations and more kinds of associations) that need to be taken into account in our algorithms. We are also working on the reverse transformations (from Maude to KM3, Ecore) in order for other tools to be able to use the results produced by Maude.

Finally, we also want to study the expressiveness of our approach when compared to other similar proposals, such as the one previously mentioned [6]. In particular, we want to use it to add semantic information to model transformations (at the end of the day they are models, too) and then be able to prove other kind of properties, such as behavior preserving of model transformations [18].

Acknowledgements The authors would like to thank the anonymous referees for their insightful comments and very constructive suggestions. This work has been supported by Spanish Research Project TIN2005-09405-C02-01.

References

[1] A. C. abd Reiko Heckel and U. Montanari. Graphical operational semantics. In *Proc. of the ICALP 2000 Satellite Workshops*, pages 411–418, 2000.

[2] P. Bernstein. Applying model management to classical metadata problems. In *Proc. of Innovative Database Research*, pages 209–220, 2003.

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

[4] J. Bézivin, F. Jouault, P. Rosenthal, and P. Valduriez. Modeling in the large and modeling in the small. In U. Aßmann, M. Aksit, and A. Rensink, editors, *Model Driven Architecture, European MDA Workshops: Foundations and Applications (MDA-FA 2003/2004)*, volume 3599 of *Lecture Notes in Computer Science*, pages 33–46. Springer-Verlag, 2005.

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

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

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

[8] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude – A High-Performance Logical Framework*. Number 4350 in Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, Germany, 2007.

[9] M. Clavel, F. Durán, S. Eker, and J. Meseguer. Building equational proving tools by reflection in rewriting logic. In K. Futatsugi, A. Nakagawa, and T. Tamai, editors, *CAFE: An Industrial-Strength Algebraic Formal Method*, pages 1–31. Elsevier, 2000.

[10] M. Clavel and M. Egea. ITP/OCL: A rewriting-based validation tool for UML+OCL static class diagrams. In M. Johnson and V. Vene, editors, *Proc. of AMAST 2006*, volume 4019 of *Lecture Notes in Computer Science*, pages 368–373. Springer-Verlag, 2006.

[11] D. di Ruscio, F. Jouault, I. Kurtev, J. Bézivin, and A. Pierantonio. Extending AMMA for supporting dynamic semantics specifications of DSLs. Technical Report 06.02, Laboratoire d'Informatique de Nantes-Atlantique (LINA), Nantes, France, Apr. 2006. Submitted for publication.

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

[13] S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker. In F. Gaducci and

U. Montanari, editors, *Proc. of the 4th International Workshop on Rewriting Logic and its Applications (WRLA 2002)*, volume 71 of *Electronic Notes in Theoretical Computer Science*, pages 115–142, Pisa, Italy, Sept. 2002. Elsevier.

[14] K. Havelund and G. Roşu. Monitoring programs using rewriting logic. In *Proc. of Automated Software Engineering 2001 (ASE'01)*, pages 135–143, California, Nov. 2001. IEEE CS Press.

[15] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume 9, pages 1–87. Kluwer Academic Publishers, 2 edition, 2002.

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

[17] R. Milner. Memories of Gilles Kahn, and the Informatic Future. Coloquim in memorial for Gilles Kahn. `http://www.inria.fr/gilleskahn/presentation/milner.pdf`, Jan. 2007.

[18] A. Narayanan and G. Karsai. Using semantic anchoring to verify behavior preservation in graph transformations. In *Proc. of the Second International Workshop on Graph and Model Transformation (GraMoT 2006)*, volume 4 of *Electronic Communications of the EASST*, pages 1–14, Brighton, United Kingdom, Sept. 2006. `http://eceasst.cs.tu-berlin.de/index.php/eceasst/article/viewFile/22/12`.

[19] I. Porres and M. Alanen. A generic deep copy algorithm for MOF-based models. In *Proc. of Model Driven Architecture: Foundations and Applications (MDA-FA 2003)*, pages 49–60, Enschede, The Netherlands, July 2003. `http://crest.abo.fi/publications/public/2002/TR486.pdf`.

[20] J. E. Rivera, F. Durán, A. Vallecillo, and J. R. Romero. Maudeling: Herramienta de gestión de modelos usando Maude. In *JISBD' 2007: Actas de XII Jornadas de Ingeniería del Software y Bases de Datos*, Sept. 2007.

[21] J. R. Romero, J. E. Rivera, F. Durán, and A. Vallecillo. Formal and tool support for model driven engineering with Maude. In B. Meyer and J. Bézivin, editors, *Proc. of TOOLS Europe 2007*, Zurich, Switzerland, Apr. 2007.

[22] J. Sáez, A. Toval, and J. L. Fernández Alemán. Tool support for transforming UML models to a formal language. In J. Whittle et al., editors, *Proc. of the International Workshop on Transformations in UML (WTUML)*, pages 111–115, Genoa, Italy, Apr. 2001.

[23] X. Thirioux, B. Combemale, X. Crégut, and P. loïc Garoche. A framework to formalise the MDE foundations. In R. Paige and J. Bézivin, editors, *Proc. of the TOWERS 2007 Workshop at TOOLS Europe 2007*, Zurich, Switzerland, May 2007.

[24] D. Varró. Automated formal verification of visual modeling languages by model checking. *Journal of Systems and Software*, 3(2):85–113, May 2004.