JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 6, No. 9, 2007

Formal and Tool Support for Model Driven Engineering with Maude

J. Raúl Romero[1], **José E. Rivera**[2], **Francisco Durán**[2], **Antonio Vallecillo**[2]
[1]Dept. Informática y Análisis Numérico, Universidad de Córdoba, Spain
[2]Dept. Lenguages y Ciencias de la Computación, Universidad de Málaga, Spain

Models and metamodels play a cornerstone role in Model-Driven Software Development. Although several notations have been proposed to specify them, the kind of formal and tool support they provide is quite limited. In this paper we explore the use of Maude as a formal notation for describing models and metamodels. Maude is an executable rewriting logic language specially well suited for the specification of object-oriented open and distributed systems. We show how Maude offers a simple, natural, and accurate way of specifying models and metamodels, and offers good tool support for reasoning about them. In particular, we show 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.

1 INTRODUCTION

Model-Driven Software Development (MDSD) is becoming a widely accepted approach for developing complex distributed applications. MDSD advocates the use of models as the key artifacts in all phases of development, from system specification and analysis, to design and implementation. Each model usually addresses one concern, independently from the rest of the issues involved in the construction of the system. Thus, the basic functionality of the system can be separated from its final implementation; the business logic can be separated from the underlying platform technology, etc. The transformations between models enable the automated implementation of a system from the different models defined for it.

We are currently witnessing how the Software Engineering community is embracing the MDSD initiative with increasing interest. So far, most of the efforts have been focused on the definition of models, metamodels and transformations between them. For instance, the Eclipse/GMT/AM3 project [19] is an example of an initiative that has built a repository containing hundreds of metamodels and transformations organized into sets of models of similar nature called *zoos*. There is for example a zoo of metamodels expressed in the Kernel MetaMetaModel (KM3, [4]), and some other auto-generated mirrors of it. The contents of these repositories are rapidly expanding, providing a publicly available source of experimental data to evaluate real life sets of model engineering artifacts [4].

Cite this document as follows: J. Raúl Romero, José E. Rivera, Francisco Durán, Antonio Vallecillo: *Formal and Tool Support for Model Driven Engineering with Maude*, in Journal of Object Technology, Special Issue TOOLS Europe 2007, 2007.

However, having such repositories of metamodels and transformations is not enough. We also need other kinds of additional model engineering artifacts, such as model editors, verification tools, model matchmakers and traders, quality measurement tools, code generators, etc. Without these we will not be able to achieve real software engineering.

It is our claim that these model driven engineering (MDE) components and tools need to be integrated into software development environments, and count with formal support. This does not mean that the average software engineer has to write complex formal specifications, but that the tools that he or she uses have formal underpinnings. Same as the hardware engineer working on a CAD workstation, who is able to achieve simulation and validation on the hardware designs he or she produces (which are just models!) without having to know all the formal groundwork supporting these processes.

In this paper we explore the use of Maude [10] as a formal notation and system for supporting many of the "model engineering" processes and operations, such as model typing and subtyping, matchmaking, and verification. 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.

The structure of this document is as follows. First, Sections 2 and 3 serve as a brief introduction to the Domain Specific Modeling and Maude, respectively. Then, Section 4 describes how models and metamodels can be represented in Maude, so they become amenable to formal analysis. Section 5 is dedicated to show some example of model operations that can be implemented with our proposal. Finally, Section 6 compares our work with other related proposals and Section 7 draws some conclusions and outlines some future research activities.

2 DOMAIN SPECIFIC MODELING

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 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 constraint 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.

DSM often also includes the idea of code generation: automating the creation of executable source code directly from the DSM models. Being free from the manual creation and maintenance of code implies significant improvements in developer productivity, reduction of defects and errors in programs, and a better resulting quality. Moreover, working with models of the problem domain instead of models of the code raises the level of abstraction, hiding unnecessary complexity and implementation-specific details, while putting the emphasis on already familiar terminology.

A DSM environment may be thought of as a metamodeling tool, i.e., a modeling tool used to define a modeling tool or CASE tool. The domain expert only needs to specify the domain specific constructs and rules, and the DSM environment provides a modeling tool tailored for the target domain. The resulting tool may either work within the DSM environment, or less commonly be produced as a separate stand-alone program. Using a DSM environment can significantly lower the cost of obtaining tool support for a DSM language, since a well-designed DSM environment will automate the creation of program parts that are costly to build from scratch, such as domain-specific editors, browsers and components. Examples of DSM environments include commercial ones such as MetaEdit+; open source environments, such as the Generic Eclipse Modeling System; or academic ones such as the Generic Modeling Environment (GME). The increasing popularity of DSM has led to DSM frameworks being added to existing integrated development environments, such as the Eclipse Modeling Project (EMP) and Microsoft's DSL Tools for Software Factories.

However, MDSD tools should provide more than just model editors and browsers. We also need to count on model analyzers, simulators, matchmakers, quality evaluators, etc., to perform real engineering tasks. In the following we will see how the use of some formal description techniques, with efficient tool support, can help achieve such MDSD components.

3 REWRITING LOGIC AND MAUDE

Maude [10, 11] 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 [15]. We informally describe in this section those Maude's features necessary for understanding the paper; the interested reader is referred to its manual [11] for more details.

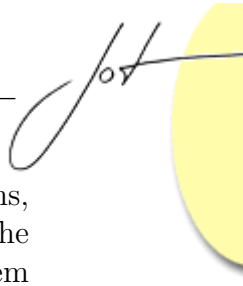
Rewriting logic is a logic of change that can naturally deal with state and with highly nondeterministic concurrent computations. A distributed system is axiomatized in rewriting logic by a *rewrite theory* $\mathcal{R} = (\Sigma, E, R)$, where (Σ, E) is an equational theory describing its set of *states* 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 declara-



tions 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.

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 (in some contexts, this also means that objects of class A can safely replace objects of class B). Multiple inheritance is also supported in Maude [10].

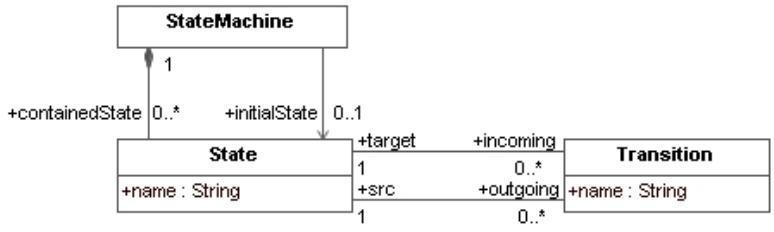


Figure 1: Simple State Machine Metamodel (from [18]).

4 FORMALIZING MODELS AND METAMODELS WITH MAUDE

Although the notion of model is not yet fully agreed, in this paper we will adopt the definition given by Jean Bézivin and Frederic Jouault in [4], where a model M is a triplet $M = (G, \omega, \mu)$ where G is a directed multigraph, ω is the *reference model* of M (i.e., its metamodel), and μ is a function associating elements (nodes and edges) of G to nodes of the multigraph that defines ω (μ defines the relation *conformsTo* between the model and its metamodel).

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 the KM3 specification of a metamodel for simple state machines, borrowed from [18], and depicted in Figure 1.

```
package SimpleStateMachine {
  class State {
      attribute name : String;
      reference stateMachine : StateMachine oppositeOf containedState;
      reference incoming [*] : Transition oppositeOf target;
      reference outgoing [*] : Transition oppositeOf src;
  }
  class StateMachine {
      reference initialState [0-1] : State;
      reference containedState [*] container : State oppositeOf stateMachine;
  }
  class Transition {
      attribute name : String;
      reference target [1] : State oppositeOf incoming;
      reference src [1] : State oppositeOf outgoing;
  }
}
```

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 [5].

In Maude, models will be represented by configurations of objects. Nodes will be represented by Maude objects. Nodes may have attributes, that will be represented by Maude objects' attributes. Edges will be 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 model 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 meta-metamodels, 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 StateMachine | initialState : Maybe{Oid}, containedState : Set{Oid} .
   class Transition | name : String, target : Oid, src : Oid .
   class State | name : String, stateMachine : Oid,
                 incoming : Set{Oid}, outgoing : Set{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 below 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.

```
< 'S : StateMachine | containedState : ('A, 'B), initialState : '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 >
```

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.

Given variables `MODEL`, `CONF`, `CONF1` and `CONF2` of sort `Configuration`, variables `O` and `SM` of sort `Oid`, variables `CS`, `IN` and `OUT` of sort `Set{Oid}`, and a variable `I` of sort `Maybe{Oid}`, this may be expressed in Maude as follows.

```
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, containedState : CS > CONF2)
  = isNullOrKindOf(I, 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 > CONF2)
  = isKindOf(T, State, MODEL)
    and-then isKindOf(S, State, MODEL)
    and-then validRefs(CONF1 < O : Transition | >, CONF2)
  if MODEL := CONF1 < O :  Transition | > CONF2 .
eq validRefs(CONF1, none) = true .
```

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 following configuration of Maude objects 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.

```
< 'SMP : KM3Package | name : "SimpleStateMachine", metamodel : 'MM,
   contents : ('STATE, 'STATEMACHINE, 'TRANSITION), 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 : 'STATEMACHINECONTAINEDSTATE, isContainer : false >
< 'STATEINCOMING : KM3Reference | name : "incoming", type : 'TRANSITION,
   package : 'SMP, owner : 'STATE, lower : 0, upper : many, isOrdered : false,
   isUnique : false, opposite : 'TRANSITIONTARGET, isContainer : false >
< 'STATEOUTGOING : KM3Reference | name : "outgoing", type : 'TRANSITION,
   package : 'SMP, owner : 'STATE, lower : 0, upper : many, isOrdered : false,
   isUnique : false, opposite : 'TRANSITIONSRC, isContainer : false >
< 'STATEMACHINE : KM3Class | name : "StateMachine", ... >
< 'TRANSITION : KM3Class | name : "Transition", ... >
```

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 Section 5). 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. There is a clear relationship between these two representations: we can easily obtain the first representation from the second one.

5 UTILITIES

Once we have described how models and metamodels are represented in Maude, this section shows how some of the basic operations on models can be then specified—namely, model subtyping, model-type inference, and metric evaluation.

Model Subtyping

Model typing is a critical issue for MDSD, specially for describing the input and output parameters of model operations and services. For instance, model transformations are defined in terms of the metamodels of their input and output models, and therefore we shall need to know whether a given model (conforming to a metamodel) can be a valid input for that transformation. This situation is even more justified in the case of initiatives such as the Model Bus [6], which allows modeling services to be connected. For connecting them, it is essential to check the type substitutability between the output of a service and the input of another, in such a way that type safety is guaranteed.

Another situation which requires type checking happens when looking for metamodels in a given repository (what is called metamodel *matchmaking*, a key mechanism for enabling reuse). For instance, suppose that you want to work with state machines, and want to know if there are already metamodels in a repository that deal with them. The easiest way to proceed would be to provide a very simple initial metamodel (such as the one depicted in Figure 1) and then look for subtypes of such a metamodel.

Despite being a core concept, a definition of the term *model type* is not widely agreed by the MDSD community yet. We will follow here the work by Steel and Jézéquel [18], for whom the type of a model is essentially its metamodel.

Then, these authors extended the notion of object subtyping to the realm of models, providing an algorithm for model subtyping (i.e., safe replaceability). This algorithm tries to be generic, and provides an elegant approach to address the problem of model type conformance, i.e., to check whether a *required model type* may be satisfied by a *provided model type*. This proposal considers a model type as the set of objects types for all the objects contained in a model and, thus, the subtyping algorithm operates on the set of features supported by the model elements (*structural conformance*) [17].

In the case of the KM3 environment, which is the one being considered here, a provided model type (i.e., a metamodel) satisfies (i.e., is subtype of) a required model type ($M_p \leq M_r$), if the following conditions hold:

- Metamodel M' is subtype of Metamodel M ($M' \leq M$) iff:

$$\forall K \in \{M.Package\} \bullet \exists K' \in \{M'.Package\} \bullet (K' \leq K)$$

- Package K' is subtype of Package K ($K' \leq K$) iff:

$$(K'.name = K.name) \wedge \forall C \in \{K.Class\} \bullet \exists C' \in \{K'.Class\} \bullet (C' \leq C)$$

- Class C' is a subtype of Class C ($C' \leq C$) iff:

$$(C'.name = C.name) \wedge (C.isAbstract \leq C'.isAbstract) \wedge$$

$$\forall P \in \{C.Attribute\} \bullet \exists P' \in \{C'.Attribute\} \bullet (P' \leq P) \wedge$$
$$\forall R \in \{C.Reference\} \bullet \exists R' \in \{C'.Reference\} \bullet (R' \leq R)$$

- Attribute P' is subtype of Attribute P ($P' \leq P$) iff:

$$(P'.name = P.name) \wedge (P'.type \leq P.type) \quad \wedge$$
$$(P'.multiplicity \leq P.multiplicity) \quad \wedge$$
$$(P'.isUnique \leq P.isUnique) \wedge (P'.isOrdered \leq P.isOrdered)$$

- Reference R' is a subtype of Reference R ($R' \leq R$) iff:

$$(R'.name = R.name) \wedge (R'.type \leq R.type) \quad \wedge$$
$$(R'.multiplicity \leq R.multiplicity) \wedge (R'.isUnique \leq R.isUnique) \quad \wedge$$
$$(R'.isOrdered \leq R.isOrdered) \wedge (R'.isContainer \leq R.isContainer) \wedge$$
$$(R.opposite \neq null) \Rightarrow (R'.opposite \neq null) \wedge (R'.opposite \leq R.opposite)$$

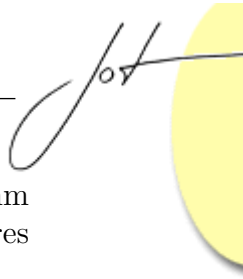
In these equations we assume that the subtyping relation $\leq$ is defined for boolean values as: $(a \leq b) \Leftrightarrow (b \Rightarrow a)$. Likewise, the $\leq$ operator can be defined for multiplicities ranges as follows: $[a'..b'] \leq [a..b] \Leftrightarrow (a' \leq a) \wedge (b' \geq b)$. Thus, $[0..*] \leq [0..1] \leq [1..1]$, and $[0..*] \leq [1..*] \leq [1..1]$, for example. Since the only primitive data types supported by KM3 are String, Boolean, Integer and Double, the subtyping relation $\leq$ for data types values will hold when both of them are the same, or in the Double $\leq$ Integer case.

One of the benefits of using Maude configurations for representing metamodels is that this subtyping algorithm can be easily implemented. In fact, previous clauses can be naturally specified in Maude using its equational logic capabilities. For instance, assuming M and M' are represented as configurations of objects, the following operation `_<=_` specifies the relation $M' \leq M$.

```
op _<=_ : Configuration Configuration -> Bool .
eq CONF' <= (< M : KM3Metamodel | contents : (P, PS) > CONF)
  = existsPackageSubtypeOf(P, CONF, CONF')
    and-then CONF' <= (< M : KM3Metamodel | contents : PS > CONF) .
eq CONF' <= (< M : KM3Metamodel | contents : empty > CONF) = true .
eq CONF' <= CONF = false [owise] .
```

As shown above, `_<=_` traverses all the packages in the provided metamodel. The auxiliary operation `existsPackageSubtypeOf(`P`,` M`,` M'`)` on a package P of a metamodel M checks whether there is a package P' in M' such that $P' \leq P$:

```
op existsPackageSubtypeOf : Oid Configuration Configuration -> Bool .
ceq existsPackageSubtypeOf(P, < P : KM3Package | > CONF,
                              < P' : KM3Package | > CONF')
  = isPackageSubtypeOf(P', P, < P : KM3Package | > CONF,
                              < P' : KM3Package | > CONF') .
```



In this way, successive Maude equations are used to specify the subtyping algorithm until reaching the most primitive metamodel elements, i.e., the structural features (attributes and references).

For space reasons we do not show here all the Maude equations of the subtyping algorithm. However, to illustrate how we deal with primitive structural features, the following operation `existsRefSubtypeOf(`SF`,` C`,` C'`,` M`,` M'`)` checks for a given reference SF defined in class C of metamodel M whether there exists a reference SF' in class C' of metamodel M' such that $SF' \leq SF$. The subtyping relationship between references is checked using the clauses defined above:

```
op existsRefSubtypeOf : Oid Oid Oid Configuration Configuration -> Bool .
eq existsRefSubtypeOf(SF', C', C,
     (< SF' : KM3Reference | name : N, owner : C', lower : L',
          upper : U', isOrdered : IO', isUnique : IU',
          isContainer : IC', opposite : RO' > CONF'),
     (< SF : KM3Reference | name : N, owner : C, lower : L,
          upper : U, isOrdered : IO, isUnique : IU,
          isContainer : IC, opposite : RO' > CONF))
  = conformsMultiplicity(L', L, U', U)
     /\ conformsOrder(IO', IO)
     /\ conformsUnique(IO', IO)
     /\ conformsContainer(IC', IC)
     /\ conformsOpposite(RO', RO, CONF, CONF') .
```

The operations `conformsMultiplicity`, `conformsOrder`, `conformsUnique`, and `conformsContainer` check for multiplicity, order, uniqueness and aggregation conformance, respectively, and the `conformsOpposite` predicate checks whether the opposite of `SF` is defined and if so, checks wether the opposite of `SF'` conforms to it. Note that the equality between the names of `SF'` and `SF` are implicitly constrained by using the same Maude variables `N`.

With these operations, checking that a metamodel Mr can be replaced by another metamodel Mp ($Mp \leq Mr$) is just a matter of reducing the term `Mp <= Mr`, where both `Mp` and `Mr` are expressed as configurations of Maude objects. Given a module with declarations

```
ops Mr Mp : -> Configuration .
eq Mr = < 'Mr : KM3Metamodel | contents : ('Pr1, 'Pr2, 'Prn ) >
        < 'Pr1 : KM3Package | name : "Package1", metamodel : 'Mr,
             contents : ('Cr1, 'Cr2, 'Cr3, 'Crn) > ... .
eq Mp = < 'Mp : KM3Metamodel | contents : ('Pp3) > ... .
```

we can execute the following reduction:

```
Maude> red Mp <= Mr .
```

From these equations we could extend very easily the capabilities of our structural type checking system. For instance, to determine whether two model types are *equivalent* we just need to define a new operation `_=_`, which makes use of the previous ones: $M_1 = M_2 \Leftrightarrow M_1 \leq M_2 \wedge M_2 \leq M_1$, which is specified as follows:

```
op _=_ : Configuration Configuration -> Bool .
eq (M1 = M2) = (M1 <= M2) and (M2 <= M1) .
```

Type inference

There are some situations in which inferring the type of a model can be interesting. Imagine, for instance, that we have the model shown in Section 4 (with one state machine, two states (`St1`, `St2`), and one transition `Tr`), but we do not know its type.

In general, there is no single type for a model. One thing we can do is to create an *initial* metamodel for it. By initial metamodel we consider the metamodel with the minimum set of elements that is a valid metamodel for the model. Once we have the initial metamodel for a model, we can traverse the metamodels in a given repository, looking for metamodels which can safely replace that metamodel, using the subtyping algorithm above. In that way, we can find some valid metamodels for the model, and choose the one that best fits our needs.

To create such a simple metamodel we use Maude the operation `inferModelType`, which transforms the configuration of objects representing the model into a configuration of objects that represents its initial metamodel (following the second way of modeling metamodels presented in Section 4).

```
op inferModelType : Configuration -> Configuration .
eq inferModelType(CONF)
  = combinePackages(combineMElts(inferMMElts(none, CONF, 0, 0))) .
```

This operation uses three auxiliary operations: `inferMMElts`, `combineMElts` and `combinePackages`. Firstly, the operation `inferMMElts` creates for each object in the model, one object representing its KM3 class, and for each attribute another object representing its KM3 structural feature (operation `inferStrFeat`).

```
op inferStrFeat : AttributeSet Qid Configuration Nat Nat -> Configuration .
ceq inferStrFeat((AT, ATS), CLASSID, MODEL, N1, N2)
  = < newId(N1, N2) : KM3Reference | name : getName(AT), owner : CLASSID, ... >
    inferStrFeat(ATS, CLASSID, MODEL, N1, (N2 + 1))
  if representsKM3Reference(AT) .
ceq inferStrFeat((AT, ATS), CLASSID, MODEL, N1, N2)
  = < newId(N1, N2) : KM3Attribute | name : getName(AT),
```

```
        owner : CLASSID, type : km3Type(getType(AT)), ... >
    inferStrFeat(ATS, CLASSID, MODEL, N1, (N2 + 1))
  if representsKM3Attribute(AT) .
eq inferStrFeat(none, CLASSID, MODEL, N1, N2) = none .

op inferMMElts : Configuration Configuration Nat Nat -> Configuration .
eq inferMMElts(CONF1, < O : C | ATS > CONF2, N1, N2)
  = < newId(N1) : KM3Class | name : string(C), isAbstract : false, ... >
    inferStrFeat(ATS, qid(C), CONF1 < O : C | ATS > CONF2, N1, N2)
    inferMMElts(CONF1 < O : C | ATS >, CONF2, (N1 + 1), N2) .
eq inferMMElts(CONF1, none) = none .
```

Auxiliary operations `representsKM3Reference` and `representsKM3Attribute` check, respectively, whether a Maude attribute represents a KM3 attribute (if its type is a KM3 primitive type `Integer`, `Bool`, etc.) or a KM3 reference (otherwise). Natural numbers `N1` and `N2` are used to create different identifiers for all the newly created objects, as is needed in Maude configurations.

The operation `combineMElts` removes duplicate entries and unifies any information related to the same KM3 concept. Since the source model may comprise more than one object of the same KM3 type, there could exist multiple Maude objects representing the same KM3 class. This may happen for structured features too. Thus, the operation `combineMElts` removes those spare objects, gathering as well the different information taken from the multiple objects that represent the same KM3 structural feature. Such an information, which may differ from one instance to the other, includes the KM3 cardinality, the type and the order indication.

```
op combineMElts : Configuration -> Configuration .
eq combineMElts(< O1 : KM3Class | name : NAME >
                < O2 : KM3Class | name : NAME > CONF )
  = combineMElts(< O1 : KM3Class | name : NAME > CONF) .
eq combineMElts(< O1 : KM3StructuralFeature | name : NAME, owner : OW,
                    lower : LO1, upper : UP1, isOrdered : B1, type : TYPE1 >
                < O2 : KM3StructuralFeature | name : NAME, owner : OW,
                    lower : LO2, upper : UP2, isOrdered : B2, type : TYPE2 >
                CONF)
  = combineMElts(< O1 : KM3StructuralFeature |
                     lower : minimun(LO1, LO2), upper : maximun(UP1, UP2),
                     isOrdered : (B1 or B2), type : max(TYPE1, TYPE2) > CONF) .
eq combineMElts(CONF) = CONF [owise] .
```

At this stage, what we have obtained a configuration of objects that represents all the KM3 classes, attributes and references of the metamodel. Then, the last step consists of bringing together the objects representing the main package, which contains all the inferred objects, the KM3 package for the primitive data types, and the metamodel itself.

```
op combinePackages : Configuration -> Configuration .
eq combinePackages(CONF)
  = < 'INITIALMETAMODEL : KM3Metamodel |
        contents : ('MAINPACKAGE, 'PTPACKAGE ), ... > .
    < 'MAINPACKAGE : KM3Package |
        name : "MainPackage", metamodel : 'INITIALMETAMODEL,
        contents : refs(CONF), ... >
    CONF
    < 'PTPACKAGE : KM3Package |
         name : "PrimitiveTypes", metamodel : 'INITIALMETAMODEL,
         contents : ('STRING, 'INTEGER, 'BOOLEAN, 'DOUBLE), ... >
    < 'STRING : KM3DataType | name : "String", package : 'PTPACKAGE >
    < 'INTEGER : ... >
    < 'BOOLEAN : ... >
    < 'DOUBLE : ... > .
```

Note that this allows a very flexible and powerful approach to typing models, similar to "duck-typing" (if it walks like a duck, and quacks like a duck, then it must be a duck).

Evaluating model metrics

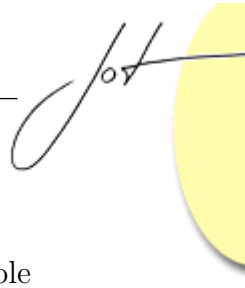
The fact of being able to manipulate models and metamodels as configuration of Maude objects also makes them amenable to formal reasoning, and to automate the specification of their properties.

For example, an interesting application of the proposed Maude formalization is the easy computation of metrics, both on the models and on the metamodels. Thus, most of the quality metrics for models defined in [8] can be easily formalized in Maude and therefore automatically computed. These metrics include, e.g., the number of classes, associations and inheritance relationships, or the maximum DIT (Depth of Inheritance Tree), the number of generalization hierarchies, or the average number of attributes, references or children per class.

For instance, the number of classes of a KM3 model coincides with the number of Maude objects of class `KM3Class` in its representation as an instance of the KM3 metamodel, which, given variables `O`, `C`, and `CONF` of sorts `Oid`, `Cid`, and `Configuration`, respectively, can be specified as follows.

```
op NoOfClasses : Configuration -> Nat .
eq NoOfClasses(< O : KM3Class | > CONF) = 1 + NoOfClasses(CONF) .
eq NoOfClasses(CONF) = 0 [owise] .
```

These specifications can then be executed, and form part of the Maude model tool-kit, which has been integrated in Eclipse as described in the next section.



Tool Support

One of the main advantages of using Maude is due to 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 has consisted in developing an Eclipse plug-in that allows to perform the previously described operations on KM3 models and metamodels. We use ATL (the ATLAS Transformation Language) to transform KM3 models to their Maude representations, and then execute the operations in the Maude environment. This tool has served as a proof of concept for our proposal, and the results are so far very encouraging. We are currently working on the definition of the reverse transformations, which will allow the KM3 user to invoke the model operations and to see the results in a transparent way, i.e., being unaware of all the Maude specifications and executions supporting the required operations.

6 RELATED WORK

There are several lines of research that are closely related to ours. First, we have the works that try to provide formal support for KM3. So far it is quite limited, there is only a very interesting proposal to formalize the semantics of KM3 using the Abstract State Machines notation [12], but there is no connection yet to any formal toolkit to reason about the formal specifications produced, nor all the potential benefits of having a formal representation of the models and metamodels have been fully exploited.

Secondly, there is quite a large number of papers about the importance of model management, the operations required to deal with models and metamodels, and about algorithms to implement such operations. For instance, there is the work by X. Blanc on the Model Bus [6], or the work by Bernstein on Model Management [2], and how model operations can help addressing classical meta-data management problems such as schema integration, schema evolution, and round-trip engineering. Many people have mentioned the importance of counting on model management operations such as model subtyping, match, merge, diff, compose, or apply, and some other people have developed algorithms for implementing some of these operations. Most of these implementations have been independently developed, and with no clear connection between them. It is true that some works have tried to deal with these operations on a unified and general way (e.g., [1, 14]) but mostly at a very high level, using category theory and institutions. What we have presented here is an integrated environment to both formally specify these operations, and to provide efficient implementations for them, which has been integrated into an Eclipse model engineering environment.

Finally, the work by Boronat *et al* [7] is very close to ours. They also use Maude to formalize models, although their representation is very different. They

have implemented a collection of generic operators to manipulate EMF models, providing some support for the QVT Relations language. Specifically, they use the Maude system to define directed declarative transformations. They have also integrated their toolset into the Eclipse environment.

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 quite limited. In this paper we have shown how Maude provides an accurate way of specifying models and metamodels, and offers good tool support for reasoning about them. In addition, Maude's possibilities of executing the specifications allows the specification and implementation of some key operations on models, such as model subtyping, type inference, and metric evaluation.

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 [16], 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 here.

Secondly, we are working on the specification on the reverse transformations from Maude to KM3, so all Maude computations can be made transparent to the user.

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. In this regard, these metamodels incorporate new elements (operations and more kinds of associations) that need to be taken into account in our algorithms.

Furthermore, some restrictions on models cannot be captured only by their metamodels, and thus OCL expressions need to be added to them, constraining their elements and their relationships. In this sense, we are working along the lines of Edwards et al. [13] on how to incorporate such kinds of constraints into our representation of models and metamodels.

Finally, DSLs are now defined in terms of their abstract and concrete syntax only. This metamodeling approach enables the rapid and inexpensive development of DSLs and their associated tools (e.g., editors). However, there is a growing interest in the MDSD community to be able to specify the behavioral semantics of DSLs too, something especially important for model operations like simulation and verification. Proposals such as the semantic anchoring method developed at Vanderbilt University [9] represent a (very important) first step in this direction. We think that Maude could also be very expressive for representing the dynamic behavior of models, and is something we plan to explore further.

Acknowledgements The authors would like to thank the anonymous referees for their insightful comments and very constructive suggestions, that have helped us significantly improve the contents and readability of the paper. This work has been supported by Spanish Research Project TIN2005-09405-C02-01.

REFERENCES

[1] S. Alagic and P. Bernstein. A model theory for generic schema management. In G. Ghelli and G. Grahne, editors, *Proc. of Database Programming Languages 2001 (DPL'01)*, volume 2397 of *Lecture Notes in Computer Science*, pages 228–246. Springer-Verlag, 2002.

[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 and F. Jouault. KM3: a DSL for metamodel specification. In *Procs. of the 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2006)*, volume 4037 of *Lecture Notes in Computer Science*, pages 171–185, Bologna, Italy, Apr. 2006. Springer-Verlag.

[5] 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.

[6] X. Blanc, M.-P. Gervais, and P. Sriplakich. Model bus: Towards the interoperability of modelig tools. 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 17–32. Springer-Verlag, 2005.

[7] A. Boronat, J. A. Carsí, and I. Ramos. Algebraic specification of a model transformation engine. In *Proc. of FASE 2006*, pages 262–277, Vienna, Austria, Mar. 2006.

[8] C. Calero, M. Genero, and M. Piattini. *Metrics for Software Conceptual Models*. Imperial College Press, London, 2005.

[9] 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.

[10] 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.

[11] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude 2.0 Manual*, June 2003. http://maude.cs.uiuc.edu.

[12] 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.

[13] J. Edwards, D. Jackson, and E. Torlak. A type system for object models. In *Proc. of the 12th International Symposium on Foundations of Software Engineering (SIGSOFT'04/FSE-12)*, pages 189–199. ACM Press, 2004.

[14] J. A. Goguen. Data, schema, ontology and logic integration. *Logic Journal of the IGPL*, 13(6):685–715, Nov. 2005.

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

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

[17] J. Steel and J. M. Jézéquel. Typing relationships in MDA. In *Proc. of the 2th Second European Workshop on Model Driven Architecture (MDA)*, pages 154–159. University of Kent, 2004.

[18] J. Steel and J.-M. Jézéquel. Model typing for improving reuse in model-driven engineering. In L. Briand and C. Williams, editors, *Procs. of MoDELS 2005*, volume 3713 of *Lecture Notes in Computer Science*, pages 84–96. Springer-Verlag, July 2005.

[19] The Eclipse/GMT/AM3 project. Generative Modeling Technologies/ATLAS MegaModel Management. http://www.eclipse.org/~gmt/am3/.