Software and Systems Modeling manuscript No.
(will be inserted by the editor)

Embedding Domain-Specific Modelling Languages in Maude Specifications

Vlad Rusu

Inria Lille Nord-Europe & Laboratoire d'Informatique Fondamentale de Lille, France

The date of receipt and acceptance will be inserted by the editor

Abstract We propose a formal approach for the definition and analysis of domain-specific modelling languages (DSML). The approach uses standard model-driven engineering artifacts for defining a language's syntax (using metamodels) and its operational semantics (using model transformations). We give formal meanings to these artifacts by translating them to the Maude language: metamodels and models are mapped to equational specifications, and model transformations are mapped to rewrite rules between such specifications, which are also expressible in Maude thanks to Maude's reflective capabilities. These mappings provide us, on the one hand, with abstract definitions of the MDE concepts used for defining DSML, which naturally capture their intended meanings; and, on the other hand, with equivalent executable definitions, which can be directly used by Maude for formal verification.

We also study a notion of operational semantics-preserving model transformations, which are model transformations between two DSML that ensure that each execution of a transformed instance is matched by an execution of the original instance. We propose a semidecision procedure, implemented in Maude, for checking the semantics-preservation property. We also show how the procedure can be adapted for tracing finite executions of the transformed instance back to matching executions of the original one. The approach is illustrated on xSPEM, a language describing the execution of activities constrained by time, precedence, and resource availability.

1 Introduction

Domain-Specific Modelling Languages (DSML) are languages dedicated to modelling specific application areas. Recently, the design of DSML has become widely accessible to engineers trained in Model-Driven Engineering (MDE). Designing a DSML amounts to defining a *metamodel* for the language's abstract syntax; then, the language's operational semantics is expressed using *model transformations* over the metamodel. The analogy with the Structured Operational Semantics (SOS) framework [1] is that models play the roles of abstract syntax trees, and model transformations play the role of SOS rules.

One can reasonably anticipate that this democratisation of language design will result in numerous languages. Formal approaches can benefit language designers by helping them to avoid or to detect errors. But, in order to have a chance of being accepted, formal approaches have to follow an accepted design process such as the MDE-based one mentioned above.

One domain where formal approaches can be beneficial is that of model-based, stepwise-refinement design processes. In each step of such a process there are two DSML $\mathcal{L}_1$ and $\mathcal{L}_2$, each endowed with an operational semantics, and a model transformation ϕ between $\mathcal{L}_1$ and $\mathcal{L}_2$. Here, $\mathcal{L}_1$ is a higher-level "specification" language, $\mathcal{L}_2$ is a lower-level "implementation" one, and ϕ is a refinement between these levels. A natural requirement for the refinement ϕ to be "correct" is that for each instance of $\mathcal{L}_1$, its image by ϕ can perform "no more" than the original - every execution of the copy must "correspond" to some execution of the original - because one does not want executions in the implementation that are not accounted for in the specification. When these conditions are met we say that the model transformation/refinement ϕ is *semantics preserving*. Also, one may wish to *compute*, for any given execution of the copy, the executions of the original that match it (resulting in so-called "execution traceability").

In this paper we propose a formal approach for defining the syntax and operational semantics of DSML, for defining and checking the semantical-preservation property of model transformations, and for defining and solving the execution-tracing problem stated above. Our approach is based, like others [2,3], on the Maude language and verification toolset [4].

The proposed approach extends our work [5], where we chose to represent metamodels (UML class diagrams possibly enriched with OCL constraints) and models as Maude equational specifications, such that model-to-metamodel conformance is automatically verifiable by equational reduction.

Contributions. The semantics of Maude specifications, based on algebras [6], provides models and metamodels with a formal semantics. We use it to propose an abstract definition of

model-to-metamodel conformance, as an “inclusion" of the semantics of the model into that of the metamodel. The advantage of this abstract definition is that it captures the intuition that a model conforms to a metamodel if the model "belongs to" the metamodel. The downside of the abstract definition is that it cannot be used for checking conformance: that requires an "executable" definition, like the one from [5]. We reconcile the two definitions by proving them equivalent.

In the same spirit, we propose abstract definitions for model transformations (which we use for defining a DSML's operational semantics, as well as translations between DSML) as computable *functions, or relations, between the semantics of their metamodels.* This captures the intuition that model transformations are functions/relations between metamodels. We prove that equivalent executable definitions for model transformations are *equationally defined functions, respectivelly, rewrite relations, over Maude specifications of models*, which are expressible in Maude thanks to its reflective nature.

Again, the abstract definition captures the intuition (that model transformations are functions/relation between metamodels), whereas the executable definition can be used by Maude for formal verification. That is, Maude can explore a language's executable operational semantics in order to model check temporal properties of instances of the languages.

We illustrate the approach by "defining" the language of finite automata, which is our running example; and we demonstrate its feasability by defining a more involved example adapted from [7]: xSPEM, a language for activities constrained by time, by resources, and by precedence relations. The examples suggest a natural and expressive language for expressing operational semantics and model transformations, mixing graphical rewrite rules and OCL [8] text for side-conditions.

We also show to optimise operational-semantics rules with respect to *given* initial DSML instances. The idea is that all reachable instances from a given initial instance can only differ from the initial instance with respect to a certain "dynamic" part, hence, rules can be "pre-instantiated" on the (invariant) "structural" part, yielding the same rewrite relation from the initial instance, but with fewer/simpler matchings.

Next, we turn to semantics-preserving model transformations. We formalise this notion by requiring that the transformation induces an *observational simulation* between the *observational transition systems* [9] generated by the operational semantics of the two DSML. The framework of observational transition systems and simulations is adequate for comparing executions of DSML because what actually changes during execution is typically a small part of a model - the "dynamic" part, which may consist of a few attributes and links. Observational transition system allow for "observations" of the dynamic part only, and observational simulations allow to compare executions only with respect to the dynamic part.

We then define a semidecision procedure and its implementation in Maude for automatically checking whether a model transformation between two instances of two DSML is semantics-preserving in the above sense. Semidecision here means that if the simulation does not hold, then our procedure will detect this; otherwise, the procedure may not terminate.

Hence, the procedure detects all semantics-preservation errors. Another interest of our procedure lies in the fact that it encodes semantical preservation as an invariance property, enabling (in principle) the use of theorem-proving techniques for invariants, also available in Maude [10, 11], for interactively proving that observational simulation does hold.

Finally, we give a version of the procedure that solves the "execution traceability" problem: given an execution ρ of an instance of the image by the transformation, it returns an encoding of all executions of the original that match ρ. We illustrate this on a transformation from xSPEM to *hierarchical extended state machines* (similar to UML state machines).

Organisation. The rest of the paper is organised as follows. In Section 2 we briefly present the Maude language. In Section 3 we present our Maude encoding of essential notions related to DSML: metamodel, model, conformance, operational semantics, and model transformations, and illustrate them on a simple example based on automata. In Section 4 we illustrate our approach on the xSPEM language. In Section 5 we deal with semantics-preserving model transformations and with the execution-tracing problem. Section 6 presents related work and future work, and concludes. The Appendix contains proofs of some technical lemmas. The Maude code for the examples in the paper is currently available online at `http://researchers.lille.inria.fr/~rusu/SoSym`.

2 Background

Maude specifications are written in Membership Equational Logic (MEL) or Rewriting Logic (RL), a superset of MEL. We briefly present them here, mostly by means of examples. The interested reader can consult [4] and the refereces therein.

2.1 Syntax

A MEL specification consists of a set of *sorts*; of a partial order on sorts called the *subsorting* relation; of a set of *operations*, which are functions between the sorts, each of which has an *arity*, where constants are 0-ary functions; and of a set of *axioms* defining the operations. Axioms are (possibly conditional) *equations* between terms, or *memberships* of terms into sorts. Among the equational axioms, some particularly important ones (associativity, commutativity, identity, ...) can be associated to some operators, saving to users the trouble of writing an explicit equation. A *term* is either a constant or a variable of a given sort, or the application of an operation to the appropriate number of terms of the appropriate sorts. A *ground term* is a term without variables. *Order-sorted* logic is a subset of MEL allowing only for equations as axioms (excluding memberships). Rewriting Logic is a superset of MEL, which also allows for (possibly conditional) *rewrite rules*.

Example 1 Two sample order-sorted specifications are shown in Figure 1, using (mostly, self-explanatory) Maude syntax.

```
fmod ELEMENT is
sort Element .
ops a b : ->  Element .
endfm

fmod ELEMENT-SET is
protecting ELEMENT .
sort Set .
subsort Element <  Set .
op empty : -> Set .
op _,_ : Set Set -> Set [assoc comm id: empty] .
eq X:Element, X:Element = X:Element .
endfm
```

Figure 1 Specifications ELEMENT and ELEMENT-SET.

They encode the standard way of defining finite sets in Maude. Sets are constructed using the empty constant, or by taking unions of sets, denoted by the _,_ operation in Figure 1, which is declared to be associative, commutative, and to have empty as its identity element. There is a sort Element for elements, which consists of the constants a and b. This sort is defined in another specification, called ELEMENT, which is *protectively extended* by the specification ELEMENT-SET. This means that the definitions in the protected specification become available in the protecting one, and that their semantics is not altered (more explanations on semantics follow).

Next, the subsorting relation Element<Set says that every element is a set. Note that, with this definition, a set would allow for multiple identical elements. To avoid this, the equation X:Element,X:Element=X:Element prevents elements to occur in a set more than once. However, if this equation is replaced by a *rewrite rule*, written in Maude syntax X:Element,X:Element=>X:Element, the intepretation is different: the equation is a part of the definition of sets; by contrast, the rule can be part of the definition of the *operational semantics* of a system whose states are multisets.

2.2 Semantics

The semantics of a MEL specification is defined in terms of *algebras*. Defining an algebra for a specification S consists in interpreting each sort of S as a set such that the subsorting relation is interpreted by the subset relation. The operations are then interpreted as functions between the corresponding sets (or by constants in the corresponding sets). It is required that the interpretation satisfies the specification's axioms. We shall denote by $\mathcal{A} \models \phi$ the satisfaction of a formula ϕ of a specification S by an algebra $\mathcal{A}$ of S, with the usual meaning - when interpreted in $\mathcal{A}$, the formula ϕ evaluates to *true*.

The *initial algebra* of a MEL specification is intuitively the "most natural interpretation" of the specification; for the specification depicted in Figure 1 it consists of sets of as and bs. Formally, the initial algebra interprets each sort s as the *set of equivalence classes of ground terms* that can be proved to be of sort s using MEL's deductive system [6] - where two terms are in the same equivalence class iff they can be proved equal using that same deductive system. The functions interpreting the non-constant operations are then implicitly defined by the specifications's axioms. Note that even though the initial algebra is the most natural interpretation of a MEL specification it is by no means the only one.

```
fmod 'ELEMENT-SET is
protecting 'ELEMENT .
sorts 'Set .
subsort 'Element <  'Set .
op 'empty : nil -> Set [none] .
op  '_,_ : Set Set ->  Set [assoc comm id('empty.Set) ] .
none .
eq '_,_['X:Element, 'X:Element] = 'X:Element [none] .
endfm
```

Figure 2 Metarepresentation of the specification ELEMENT-SET.

The *initial semantics* of a MEL specification consists of its initial algebra. We denote $(\!|S|\!)$ the initial semantics of a specification S. The *loose semantics* of a MEL specification S is the set of all its algebras. We use the initial semantics for MEL specifications denoting models, and a subset of the loose semantics for MEL specifications representing metamodels.

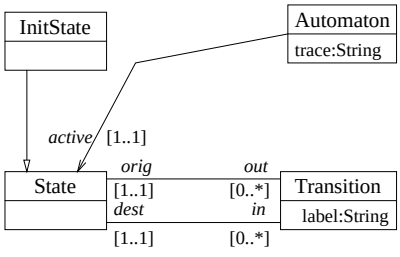
The initial semantics of a Maude RL specification is a *transition system* whose states are equivalence classes of ground terms, and whose transition relation interprets the *rewrite relation* of the RL specification (two classes $[t_1]$, $[t_2]$ are in relation if t_2 is obtained from t_1 by exactly one rewrite). We shall use this semantics to define the operational semantics of DSML, and, more generally, that of model transformations.

2.3 Reflectiveness

We shall use the fact that Maude is *reflective*: there exists a Maude specification that *metarepresents* all Maude specifications, including itself. For MEL specifications this is achieved by a function fmod_is_sorts_. ____endfm, defined at Maude's *meta-level*, which takes 7 arguments (corresponding to the number of underscores). For example, the specification ELEMENT-SET is obtained by applying the above function to metarepresentations of the following parameters: a name (here, ELEMENT-SET); a set of imported specifications (here, ELEMENT); a set of sorts (here, Set); a subsorting relation (here, Element < Set), a set of operation declarations (here, empty and _,_); a set of membership axioms (here, there are none); and a set of equations (here, the sole equation X:Element,X:Element=X:Element).

The resulting *metarepresentation* of the Maude specification ELEMENT-SET in Figure 1 is shown in Figure 2. Syntactical differences with the original are minor: for instance, all identifiers are *quoted*, and all operations are in prefix form.

More important differences lie in the fact that the metarepresentations $\overline{S}$ of Maude specifications S are *terms*, hence, they can be processed just as any term within other Maude specifications. We shall exploit this fact for defining operational semantics and model transformations, using equationally defined functions and/or rewrite rules over (metarepresentations of) Maude specifications denoting models conforming to a given metamodel. The one formal property of reflection that we shall use is that it is *injective*: for distinct specifications S_1, S_2, their metarepresentations $\overline{S_1}, \overline{S_2}$ are distinct.



$Transition.allInstances \rightarrow forAll(t : Transition | t.label \neq "")$

Figure 3 Metamodel for finite automata without silent transitions.

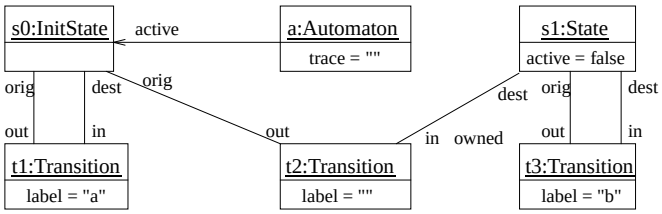


Figure 4 A non-conformant model of the metamodel in Fig 3.

3 Representing DSML into Maude

In this section we propose abstract definitions to the essential notions involved in DSML: metamodel, model, model-to-metamodel conformance, operational semantics, and model transformations. For conformance, operational semantics, and model transformations we show the equivalence of abstract definitions with executable ones, which can be used by Maude for verification, and which rely on Maude's reflectiveness.

We take the commonly shared view that meta-models are essentially UML class diagrams possibly with OCL constraints.

Example 2 The metamodel in Figure 3 represents finite automata. The unidirectional association from the class *Automaton* to the class *State* denotes the *active* state. The *InitState* subclass of *State* represents initial states of automata. The class *Automaton* has the *trace* attribute - a string of characters, obtained by concatenating *labels* of *transitions* fired by the automaton. Transitions are associated to *orig*in and *dest*ination states. The opposite roles, from the point of view of states, are those of *in*coming and *out*going transitions. The roles of associations are labelled with multplicities, e.g., transitions have one origin and one destination state. The OCL *invariant* below the diagram says that the automaton does not have "silent transitions": the labels of transitions are nonempty.

Figure 4 shows a model of an automaton as an object diagram of the class diagram in Figure 3. It is composed of: a self-loop labelled "a" on the (active and initial) state s_0; a transition from state s_0 to state s_1 labelled ""; and a self-loop labelled "b" on s_1. It does not conform to the metamodel in Figure 3 because it violates the metamodel's OCL invariant.

Model and metamodel representations in Maude. We give semantics to (meta)models by representing them in Maude.

We first discuss the already existing alternatives [2,3]. One the one hand, [2] base their representation on Maude's object oriented extension embodied in *Full Maude* (an extension of Maude, written in Maude itself). On the other hand, [3] represent metamodels as sorts, which are defined in Maude using membership axioms, and specify the constraints that the models conforming to a given metamodel must satisfy.

Our proposal is to represent both metamodels and models as Maude *specifications*, and to take advantage of the algebra-based semantics of Maude specifications to provide them with formal meanings. By doing so, we avoid the complexity of expressing conformance by means of memberships, or of having to rely on Maude's object-oriented extension. This relative simplicity allows us to avoid a theoretical problem encountered by [2,3]: their definitions for metamodels are quite complex, with the consequence that their encodings of model-to-metamodel conformance were not shown to be decidable.

Hence, we take a different approach - we represent metamodels and models as order-sorted specifications. Classes, inheritance, class attributes, and associations,, are mapped to existing constructions of order-sorted specifications: respectively, to sorts, to subsorting relations, and to functions between sorts. Constructions present in models are also mapped to corresponding constructions of order-sorted specifications, and OCL invariants are mapped to equations, such that, overall, the specifications representing object diagrams are *ground confluent and terminating* [12]. This ensures the decidability of model-to-metamodel conformance, and provides us with a correct and reasonably efficient procedure for checking it.

3.1 Metamodels

A metamodel is a class diagram possibly enriched with OCL invariants. We consider a minimal notion of class diagrams, consisting of a set of classes with attributes, of unidirectional associations between classes, whose roles have $[0..*]$ multiplicities, of a partial-order generalisation between classes, and of OCL invariants that are syntactically and semantically correct in the context of a given class diagram. We here assume that these concepts are known without further definitions. Other features of class diagrams (bidirectional associations, roles with multiplicities other than $[0..*]$, composition and aggregation associations...) are not considered since they do not any expressiveness - they can be equivalently encoded using the existing constructions and OCL constraints[1].

Definition 1 *For a metamodel $\mathcal{MM}$, we denote by* MEL$(\mathcal{MM})$ *the (order-sorted)* MEL *specification defined as follows:*

- *standard* MEL *specifications of basic types (Boolean,...) occuring in the metamodel are* protected *in* MEL$(\mathcal{MM})$*;*

[1] For example, a bidirectional association between the classes c_1 and c_2, which in the association play the roles r_1 and r_2, respectively, can be encoded using two unidirectional associations: one from c_1 to c_2, and the other one from c_2 to c_1, together with OCL invariants saying that the functions r_1, r_2 are inverse to each other.

– *each class c is translated into a sort c. A sort $Set\{c\}$ for multisets*[2] *of elements of sort c, with constructors $empty$, and $\_,\_$ is declared in* MEL($\mathcal{MM}$)*, together with two declarations: of the subsorting relation $c < Set\{c\}$, and of a constant, called $c.allInstances$, of sort $Set\{c\}$;*
– *the inheritance relation is represented by the subsorting relation: whenever c_1 directly inherits from c_2 in $\mathcal{MM}$ we have in* MEL($\mathcal{MM}$) *a subsort declaration $c_1 < c_2$;*
– *each attribute a of type t of a class c is translated to a function declaration $a : c \rightarrow t$;*
– *each association from c_1 to c_2, where c_2 plays the role r_2, is translated into a function $r_2 : c_1 \rightarrow Set\{c_2\}$;*
– *if the metamodel contains* OCL *invariants they are translated to equations, based on the translation defined in [12].*

We now describe the translation [12] of OCL to MEL and explain why it generates confluent and terminating equations.

– basic types (Booleans, integers, strings, ...) and the operations on them, as well as sets of such types, are already defined in Maude, so there is no need to redefine them;
– navigation is made available by the function declarations denoting attributes and links; for navigating from instances of a class c to attributes a of type t of c, the function $a : c \rightarrow t$ shall be used, and similarly for the navigation from instances to other instances via associations/roles;
– quantifiers (*forall*, *exists*) and iterators (*select*, *collect*) are expressed using equationally defined recursive functions. The only difficulty is that Maude functions do not allow for functions as arguments, whereas OCL does allow this. The solution is then to instantiate the iterators for the actual (finitely many) expressions over which they iterate. For example, an expression of the form $\rightarrow select(x : T|E_i)$, where E_i is a the i_{th} OCL expression occuring in the OCL invariants of metamodel (according to an arbitrary order) is defined by a function $select_i$, using the equations

$$select_E(empty) = empty$$

$$select_i(s, S) = (if\ E_i(s)\ then\ s\ else\ empty), select_i(S)$$

Note the presence of the index i in the *select* - it indicates the fact that it is meant for iterating over expression E_i. This is also useful for ensuring confluence (see below). Similarly, an expression of the form $\rightarrow forAll(x : T|E_j)$ is translated to a function $forall_j$, using the equations

$$forAll_j(empty) = true$$

$$forAll_j(s,S) = (if\ E_j(s)\ then\ true\ else\ false) \wedge forAll_j(S)$$

Similar equations define the other quantifer and iterators. Finally, a universally-quantified invariant of the form

$$c.allInstances \rightarrow forAll(x : T|E_i)$$

is encoded by an equation of the form

$$forAll_i(c.allInstances) = true$$

and similarly for existentially-quantified invariants.

[2] From here on we shall refer to multisets simply as sets.

```
fmod AUTOMATA-MM is
protecting Bool String .

--- sorts for classes, subsorting for inheritance
sorts Automaton Transition State InitialState .
subsort InitialState < State .

--- omitted: definitions for sorts Set{Automaton}, etc

--- constants for all instances of a given class
op Automaton.allInstances : -> Set{Automaton} .
op Transition.allInstances : -> Set{Transition} .
op State.allInstances : -> Set{State} .
op InitialState.allInstances : -> Set{InitialState} .

--- associations
op active : Automaton -> Set{State} .
op orig : Transition -> Set{State} .
op dest : Transition -> Set{State} .
op incoming : State -> Set{Transition} .
op outgoing : State -> Set{Transition} .

--- attributes
op trace : Automaton -> String .
op label : Transition -> String .

--- OCL invariant
op forAll-1 : Set{Transition} -> Bool .
eq forAll-1(empty) = true .
eq forAll-1(t:Transition, S:Set{Transition}) =
 (label(t) =/= "") and-then forAll-1(S:Set{Transition}  .

eq forAll-1(Transition.allInstances) = true .
endfm
```

Figure 5 MEL specification of the metamodel in Fig. 3.

Regarding confluence, it is ensured by the numbering of the *select, forall, ...* recursive functions, which avoids critical pairs (and also by the fact that, in the two equations per function, one takes the argument *empty*, and the other one, something nonempty). And termination is ensured by the fact that all recursive calls are made on structurally smaller arguments.

Example 3 For the metamodel shown in Figure 3, the result of the translation is for the most part shown in Figure 5. Other "implicit" OCL invariants (not shown in the figure) encode the 1..1 multiplicity constraints of some of the association roles, as well as the constraint that the unidirectional associatations encoding the bidirectional ones are inverse to each other.

For a metamodel $\mathcal{MM}$, we define a subset of the algebras of MEL($\mathcal{MM}$), which shall constitute by definition the metamodel's semantics. The idea is that models conforming to $\mathcal{MM}$ shall bijectively match algebras in the given set.

Definition 2 *For the* MEL *specification* MEL($\mathcal{MM}$) *of a metamodel $\mathcal{MM}$, we denote by $[\![$*MEL($\mathcal{MM}$)$]\!]$ *the smallest set of algebras of* MEL($\mathcal{MM}$) *containing all algebras A such that:*

1. *A interprets the specifications imported in* MEL($\mathcal{MM}$) *as their respective initial algebras;*
2. *A interprets each proper sort c of* MEL($\mathcal{MM}$) *(meaning that c is not imported in* MEL($\mathcal{MM}$)*) as a finite set $A(c)$;*
3. *for any pair c_1, c_2 of proper sorts of* MEL($\mathcal{MM}$) *that are in different connected components with respect to the subsorting relation of* MEL($\mathcal{MM}$)*, $A(c_1) \cap A(c_2) = \emptyset$;*
4. *A interprets all sorts of the form $Set\{c\}$ as $\mathcal{P}_f(A(c))$, that is, the set of finite parts of $A(c)$;*
5. *A interprets all the constants $c.allInstances$ as $A(c)$.*

Definition 3 (metamodel semantics) *The semantics of a metamodel $\mathcal{MM}$ is the set of algebras* $[\![\text{MEL}(\mathcal{MM})]\!]$.

3.2 Models

A model $\mathcal{M}$ is essentially an object diagram of some metamodel (i.e., class diagram) $\mathcal{MM}$. Remember that an object diagram is *of* a given class diagram if all objects have classes that belong to the class diagram; all atributes of an object are present in the object's class, and the value of the attributes have the same types as (or have subtypes of) the types declared in the class; and all links between objects instantiate an existing association between the two object's classes in the class diagram. We assume that these concepts are known.

Definition 4 *For a model $\mathcal{M}$ of a metamodel $\mathcal{MM}$, we denote by* $\text{MEL}_{\mathcal{MM}}(\mathcal{M})$ *the (order-sorted)* MEL *specification constructed as follows:*

- *the* MEL *translation of the metamodel $\mathcal{MM}$ is imported;*
- *each instance o of class c becomes a declaration $o :\rightarrow c$ of a constant o of sort c;*
- *each attribute a of an object o having value v is translated to an equation $a(o) = v$;*
- *for all classes c, the constant $c.allInstances$ is equated to the set of all constants of sort c declared in* $\text{MEL}_{\mathcal{MM}}(\mathcal{M})$.

A few explanations for this translation: (1) the *instances* of a class become *constants* of the sort denoting the class, whose declaration is imported from $\text{MEL}(\mathcal{MM})$; (2) *attributes values* become *equations*, which participate in the definition of the *function denoting the attribute*, whose declaration is also imported from $\text{MEL}(\mathcal{MM})$; (3) each set of links of an association translates to an equation that defines the *function denoting a role*, whose declaration is imported from $\text{MEL}(\mathcal{MM})$.

Moreover, for technical reasons we let all $\text{MEL}_{\mathcal{MM}}(\mathcal{M})$ share a unique identifier. This is to avoid two otherwise identical specifications be differentiated by their identifier only.

Example 4 For the model $\mathcal{M}$ from Figure 4 and the metamodel $\mathcal{MM}$ from Figure 3, $\text{MEL}_{\mathcal{MM}}(\mathcal{M})$ is depicted in Figure 6. Note that the specification of the model imports that of the metodel in *extending* mode (`extending` keyword). Like in the case of *protecting* imports, this makes all definitions from the imported specification become available in the importing one, but now, their semantics may be changed, e.g., by adding constants to sorts and equations between constants.

We now define the semantics of a model as the initial algebra of its corresponding MEL specification:

Definition 5 (model semantics) *The semantics of a model $\mathcal{M}$ of metamodel $\mathcal{MM}$ is the initial algebra* $(\!|\text{MEL}_{\mathcal{MM}}(\mathcal{M})|\!)$.

3.3 Conformance

Based on the abstract Definitions 3 and 5 for the semantics of metamodels and models, respectively, we obtain the following rather natural and abstract definition for conformance,

```
fmod AUTOMATON-MODEL is
extending Automata-MM .

--- constants denoting objects
op a : -> Automaton .
op s0  : ->  InitialState .
op s1 : -> State .
ops t1 t2 t3   : ->  Transition .

--- equations denoting attribute values
eq trace(a) = "" .
eq label(t1) = "a" .
eq label(t2) = "" .
eq label(t3) = "c" .

--- equations denoting links
eq active(a) = s0 .
eq orig(t1) = s0 .
eq dest(t1) = s0 .
eq orig(t2) = s0 .
eq dest(t2) = s1 .
eq orig(t3) = s1 .
eq dest(t3) = s1 .
eq in(s0) = t1 .
eq out(s0) = t1, t2 .
eq in(s1) = t1, t3 .
eq out(s1) = t3 .

--- equations characrerising all instances
--- for the declared classes
eq Automaton.allInstances = a .
eq State.allInstances = s1, s2 .
eq InitialState.allInstances = s1 .
eq Transition.allInstances = t1, t2, t3 .
endfm
```

Figure 6 MEL specification representing the model from Figure 4.

capturing the intuition that a model $\mathcal{M}$ conforms to a metamodel $\mathcal{MM}$ if the model $\mathcal{M}$ belongs to the metamodel $\mathcal{MM}$:

Definition 6 (conformance, abstract version) *A model $\mathcal{M}$ conforms to a metamodel $\mathcal{MM}$, denoted by $\mathcal{M} : \mathcal{MM}$, if* $(\!|\text{MEL}_{\mathcal{MM}}(\mathcal{M})|\!) \in [\![\text{MEL}(\mathcal{MM})]\!]$.

Note that the above definition also implies that $\mathcal{M}$ has metamodel $\mathcal{MM}$ (otherwise, $(\!|\text{MEL}_{\mathcal{MM}}(\mathcal{M})|\!)$ is not defined).

However, the abstract Definition 6 cannot be used for the automatic machine-checking of conformance, because it is a semantical definition, whereas computation requires syntax.

We now recall our executable definition of conformance from [5] and show that it is equivalent to the above abstract one. For a model $\mathcal{M}$ of metamodel $\mathcal{MM}$, the equational representation of the conjunction of all OCL invariants of $\mathcal{MM}$, which we shall denote by $\text{OCL}_{\text{MEL}}(\mathcal{MM})$, is automatically evaluated in $\text{MEL}_{\mathcal{MM}}(\mathcal{M})$. This is done by equational reduction, thanks to the ground confluence and termination of the equations denoting OCL invariants [12]. Then, conformance holds iff the canonical form of the conjunction $\text{OCL}_{\text{MEL}}(\mathcal{MM})$ in $\text{MEL}_{\mathcal{MM}}(\mathcal{M})$ is *true*. Since for ground confluent and terminating (order-sorted) MEL specifications, the initial algebra is the algebra of canonical forms of terms reduced by equations [4], we obtain that our executable definition for conformance from [5] amounts to the following one:

Definition 7 (conformance, executable version) *A model $\mathcal{M}$ conforms to a metamodel $\mathcal{MM}$, denoted by $\mathcal{M} :: \mathcal{MM}$, if* $(\!|\text{MEL}_{\mathcal{MM}}(\mathcal{M})|\!) \models (\text{OCL}_{\text{MEL}}(\mathcal{MM}) = true)$.

In order to show the equivalence of our abstract and operational definitions of conformance we need the following lemma, which says that the semantics of a metamodel is equal to the set of semantics of models that executably-conform to it:

Lemma 1 $[\![\text{MEL}(\mathcal{MM})]\!] = \{(\!|\text{MEL}_{\mathcal{MM}}(\mathcal{M})|\!) \,|\, \mathcal{M} :: \mathcal{MM}\}$.

The main result about conformance is that our abstract Definition 6 and the executable Definition 7 from [5] coincide:

Proposition 1 *$\mathcal{M} : \mathcal{MM}$ if and only if $\mathcal{M} :: \mathcal{MM}$.*

Proof by Definition 7, $\mathcal{M} : \mathcal{MM}$ if and only if

$$(\!|\text{MEL}_{\mathcal{MM}}(\mathcal{M})|\!) \in [\![\text{MEL}(\mathcal{MM})]\!]$$

By Lemma 1,

$$[\![\text{MEL}(\mathcal{MM})]\!] = \{(\!|\text{MEL}_{\mathcal{MM}}(\mathcal{M})|\!) \,|\, \mathcal{M} :: \mathcal{MM}\}$$

Hence, $\mathcal{M} : \mathcal{MM}$ if and only if

$$(\!|\text{MEL}_{\mathcal{MM}}(\mathcal{M})|\!) \in \{(\!|\text{MEL}_{\mathcal{MM}}(\mathcal{M})|\!) \,|\, \mathcal{M} :: \mathcal{MM}\}$$

which is equivalent to $\mathcal{M} :: \mathcal{MM}$. □

3.4 Operational Semantics: Three Equivalent Definitions

The operational semantics of a DSML is, intuitively, a function that maps models in the DSML to "next" models. Based on Definition 3 (semantics of metamodels) we propose the following abstract definition for operational semantics.

Definition 8 (operational semantics, abstract version) *The operational semantics of a* DSML *of metamodel $\mathcal{MM}$ is any recursive function $F : [\![\text{MEL}(\mathcal{MM})]\!] \to \mathcal{P}_f([\![\text{MEL}(\mathcal{MM})]\!])$.*

Here, $\mathcal{P}_f(S)$ denotes the set of finite subsets of S. Thanks to Definition 6 of conformance, we can identify a metamodel $\mathcal{MM}$ with the st of models $\mathcal{M}$ conforming to it. Hence, Definition 8 captures the intuition that, during execution, a model nondeterministically "chooses" a successor from a finite (possibly empty) set of models; and that this set is computable.

However, Definition 8 is not executable: one cannot compute with Maude over algebras (semantics) of Maude specifications; such computations require syntax to operate on.

The available syntax is that of *Maude specifications* denoting models conforming to a given meta-model. Hence, in order to obtain executable versions of operational semantics, we shall define Maude computations over Maude specifications. This is possible in Maude thanks to its reflective nature.

We shall need the two following lemmas. The first one makes the first step from semantics to syntax: it establishes a bijection between a metamodel's semantics and the set of Maude specifications of models conforming to the metamodel.

Lemma 2 *There is a bijection between the set $\{\text{MEL}_{\mathcal{MM}}(\mathcal{M}) \,|\, \mathcal{M} :: \mathcal{MM}\}$ and the metamodel's semantics $[\![\text{MEL}(\mathcal{MM})]\!]$.*

```
fmod Models_AUTOMATA-MM is
--- predefined Maude module for using reflection
extending META-LEVEL .

--- metamodel for automata
protecting AUTOMATA-MM .

--- definitions of sorts metarepresenting automata models
sort  Models_AUTOMATA-MM .

--- definition of the sort Models_AUTOMATA-MM
--- using a conditional membership
--- FModule is a predefined sort from Meta-Module
--- metarepresenting Maude MEL specifications

var X : FModule .
cmb X : Models_AUTOMATA-MM
 if  conformance-check(X,'AUTOMATA-MM) = true .
 --- conformance-check is the implementation of
 --- conformance by equational reduction from [5]
 endfm
```

Figure 7 MEL specification `Models_AUTOMATA-MM`.

Hence, in Definition 8, semantics ($[\![\text{MEL}(\mathcal{MM})]\!]$) can be replaced with syntax ($\{\text{MEL}_{\mathcal{MM}}(\mathcal{M}) \,|\, \mathcal{M} :: \mathcal{MM}\}$, i.e., with Maude specifications). This is not enough: computations in Maude are either functions or rewrite rules, and both require *sorts* to be defined upon. Hence, our second lemma constructs a Maude specification that defines a sort that, when intepreted properly, is inhabited by metarepresentations of Maude specifications of models that conform to a given metamodel[3].

Lemma 3 *For each metamodel $\mathcal{MM}$, there exists a* MEL *specification denoted by $\texttt{Models}_{\mathcal{MM}}$, where a sort $Models_{\mathcal{MM}}$ is defined, whose interpretation in the algebra $(\!|\texttt{Models}_{\mathcal{MM}}|\!)$ is in bijection with the set $\{\text{MEL}_{\mathcal{MM}}(\mathcal{M}) \,|\, \mathcal{M} :: \mathcal{MM}\}$.*

Example 5 A specification of the form $\texttt{Models}_{\mathcal{MM}}$ is shown in Fig. 7 (`AUTOMATA-MM` is the specification in Figure 5).

Our first executable definition considers the sorts defined in the MEL specification $\texttt{Models}_{\mathcal{MM}}$ of Lemma 3 and so-called "protective extensions" of this specification. Remember that a protective extension of a specification S_1 by a specification S_2 does not change the (initial) semantics of S_1: it uses the sorts and operations defined in S_1 without altering them.

Definition 9 (operational semantics, executable version 1) *The operational semantics of a* DSML *of metamodel $\mathcal{MM}$ is any function $F : Models_{\mathcal{MM}} \to Set\{Models_{\mathcal{MM}}\}$ equationally defined in some protective extension of $\texttt{Models}_{\mathcal{MM}}$, and interpreted in the initial semantics of the extension.*

Proposition 2 *Definitions 8 and 9 are equivalent.*

Proof The bijection between the semantics $[\![\text{MEL}(\mathcal{MM})]\!]$ of a DSML's metamodel and the set $\{\text{MEL}_{\mathcal{MM}}(\mathcal{M}) \mid \mathcal{M} :: \mathcal{MM}\}$ (cf. Lemma 2) ensures that we have the following

[3] Note the analogy with [3], where metamodels are encoded as sorts and models are encoded as terms of those sorts. The difference is that [3] perform their encoding directly in Maude's logic "by hand", whereas, in our case, Maude's reflection mechanism automatically reflects models as terms/metamodels-as-sorts for us, based on our encoding of models/metamodels as Maude specifications.

equivalent definition to Definition 8: the operational semantics of a DSML of metamodel $\mathcal{MM}$ is a recursive function from $\{\text{MEL}_{\mathcal{MM}}(\mathcal{M}) \mid \mathcal{M} :: \mathcal{MM}\}$ to $\mathcal{P}_f(\{\text{MEL}_{\mathcal{MM}}(\mathcal{M}) \mid \mathcal{M} :: \mathcal{MM}\})$. Then, using the bijection from Lemma 3, and a standard encoding of finite sets in Maude such as that shown in Figure 1, we obtain yet another equivalent definition to Definition 8, as recursive functions from $Models_{\mathcal{MM}}$ to $Set\{Models_{\mathcal{MM}}\}$. Next, a theorem by Bergstra and Tucker [13] says that recursive functions on a given domain/codomain are exactly those functions that can be equationally defined on algebraic specifications of the domain and codomain, by means of confluent, terminating equations. The equations are, in general, written in protective extensions of the specification $\texttt{Models}_{\mathcal{MM}}$ and interpreted in their initial algebras. □

This definition is already an executable one, in the sense that Maude can compute results of the equationally-defined semantics. However, in order to use Maude's automatic verification tools (namely, state-space exploration, an example is given below) it is better to equivalently represent such semantics using *rewrite rules* of Rewriting-Logic specifications.

Definition 10 (operational semantics, executable version 2) *The operational semantics of a* DSML *of metamodel $\mathcal{MM}$ is the rewrite relation over the sort $ModelsIn\mathcal{MM}$, of some* RL *protective extension of the* MEL *specification $\texttt{Models}_{\mathcal{MM}}$, and interpreted in the initial semantics of the extension.*

Proposition 3 *Definitions 9 and 10 are equivalent.*

Proof The equivalence holds thanks to the following observations. On the one hand, for any sort S and equationally defined function $F : S \to Set\{S\}$, and any two terms t_1, t_2 of sort S, $t_1 \in F(t_2)$ reduces to $true$ if and only $\{t_1\}$ rewrites to $\{t_2\}$ by using the rewrite rule $\{x\} \Rightarrow \{y\}$ *if* $y, z := F(x)$, where variables x and y have sort S, and z has sort $Set\{S\}$; that is, one can always encode the computation of such equationally defined functions by using rewrite relations[4]. On the other hand, the transition relation over the sort $Models_{\mathcal{MM}}$ of a rewriting-logic specification containing $\texttt{Models}_{\mathcal{MM}}$ is a computable function from $Models_{\mathcal{MM}}$ to $Set\{Models_{\mathcal{MM}}\}$. □

Example 6 We illustrate below the executable Definition 10 on our specifications on automata. Figure 8 depicts automata execution: if an automaton Y has a transition W with label L whose origin is X and destination is Z, and the currently active state is X, then the active state becomes Z, and the label L is concatenated to the automaton's trace T.

The corresponding Maude rewrite rule is shown in Figure 9. It closely matches the graphical rule: the links and attribute values are denoted by equations; the rule changes the set of equations in order to change the links and attribute values. Here, the link that changes is the *active* link, from

[4] More precisely, the rewrite rule $\{x\} \Rightarrow \{y\}$ *if* $y, z := F(x)$ will rewrite at the top any term $\{t_1\}$ to a term $\{t_2\}$, whenever $F(t_1)$ can be *matched* by some set containing t_2. Here, we exploit the so-called *matching condition* [4]: *if* $y, z := F(x)$ of the rewrite rule.

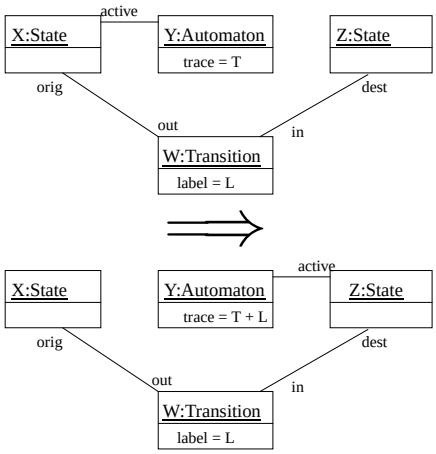


Figure 8 Executing automata: graphical rule.

```
mod AUTOMATA-EXECUTION  is
--- protective extension : the initial semantics
--- of Models_AUTOMATA-MM  is not modified
protecting Models_AUTOMATA-MM .

--- rule for executing automata
rl
(eq 'active[Y:Term]  = X:Term [none] .)
(eq 'orig[W:Term]  =  X:Term [none] .)
(eq 'dest[W:Term]  =  Z:Term [none] .)
(eq 'label[W:Term]  =  L:Term [none] .)
(eq 'trace[Y:Term]  =  T:Term [none] .)
=>
(eq 'active[Y:Term]  = Z:Term [none] .)
(eq 'orig[W:Term]  =  X:Term [none] .)
(eq 'dest[W:Term]  =  Z:Term [none] .)
(eq 'label[W:Term]  =  L:Term [none] .)
(eq 'trace[Y:Term]  =  '_+_[L:Term, T:Term] [none] )  .
endm
```

Figure 9 Executing automata: Maude rewrite rule.

`'active[Y:Term] = X:Term` to `'active[Y:Term] = Z:Term`. Operator names from the meta-model specification are quoted, and variables are of sort `Term`; this is due to the fact that we are using Maude's reflection (allowed by the importation of the `META-LEVEL` Maude specification). The attribute value that changes is `'trace[Y:Term]`, whose next-state value is the concatenation of the label L and trace T, expressed by reflection in Maude by means of the equation `'trace[Y:Term] = '_+_[L:Term, T:Term]`.

We can use now the Maude specification shown in Figure 9 to execute, e.g., the automaton whose model's specification in Maude is shown in Figure 6 and to verify some simple temporal properties for it. For example, the following command asks Maude whether an execution of the automaton exist such that the trace of the automaton is "aaabb":

```
search[1] upModule('AUTOMATON-MODEL =>* X
such that getTrace(X) = '"aaabb".String.
```

Maude instantly responds positively, and provides us upon request with the shortest path leading to the solution.

Extensions. The abstract Definition 8 of operational semantics may be extended to strictly more expressive *recursive relations* $R : [\![\text{MEL}(\mathcal{MM})]\!] \times [\![\text{MEL}(\mathcal{MM})]\!] \to Bool$. Such relations may lead to non-computable successor functions (the

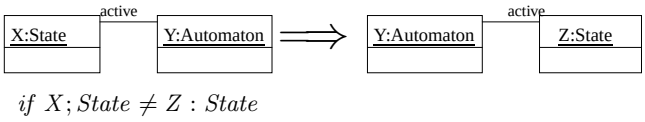


Figure 10 Graphical rule for arbitrary change of active state.

partial function F such that $R(x, F(x)) = true$ for all inputs x where F is defined, is in general only recursively enumerable). However, it is interesting to consider such semantics for theoretical reasons (can they also be represented in Maude?) and also for practical reasons: as we shall see, such relations naturally correspond to DSML for modelling *open* systems.

To represent such transition relations in Maude, we use the specification $\texttt{Models}_{\mathcal{MM}}$ from Lemma 3 and the sort $Models_{\mathcal{MM}}$ defined therein. Using the same reasoning as in the proof of Proposition 2 we obtain that the set of recursive relations $R : [\![\text{MEL}(\mathcal{MM})]\!] \times [\![\text{MEL}(\mathcal{MM})]\!] \to Bool$ are in bijection with the set of equationally defined relations $R : Models_{\mathcal{MM}}\ Models_{\mathcal{MM}} \to Bool$, written in some protective extension of the specification $\texttt{Models}_{\mathcal{MM}}$, and interpreted in the initial semantics of the extension. Now, these relations trivially coincide with the rewrite relations of rewrite rules of the form $(\dagger)\ \{x\} \Rightarrow \{y\}\ if\ R(x, y) = true$, with x and y of sort $Models_{\mathcal{MM}}$, interpreted in some rewriting-logic protective extension of the specification $\texttt{Models}_{\mathcal{MM}}$ and interpreted in the initial semantics of the extension. Moreover, the rewrite relations of rules are recursive (i.e., whether a rule can lead from one given term to another given one is decidable). Hence, we obtain an equivalent Maude characterisation of operational semantics that are recursive relations over $[\![\text{MEL}(\mathcal{MM})]\!]$, as rewrite relations over $Models_{\mathcal{MM}}$, definable in some RL protective extension of the specification $\texttt{Models}_{\mathcal{MM}}$, and interpreted in its initial semantics.

Finally, note that the rewrite rule $(\dagger)$ encoding the relation R has the free variable y in both its right-hand side and condition. The practical interest is that the free variable y may be interpreted as *input* from an *environment*. Hence, transition relations naturally correspond to DSML for modelling *open* systems, which receive inputs from an unknown environment.

However, to "execute" such a rule from a term matched by x, a rewrite engine must "choose" a term for y such that $R(x, y)$ evaluates to $true$. This is not possibly in general as it would require constraint-solving orver arbitrary domains. Such rules can be executed using *narrowing* in some cases [14].

Example 7 Assume that in a automata the user can arbitrarily change the active state to some other state. The rule in Figure 10 decribes this: the previous active state was X and the new active state is chosen to be some state Z, provided $X \neq Z$, which is a condition to the rule. In Maude this gives

```
crl (eq 'active[Y:Term] = X:Term [none] .)
=>  (eq 'active[Y:Term] = Z:Term [none] .)
if X:Term =/= Z:Term .
```

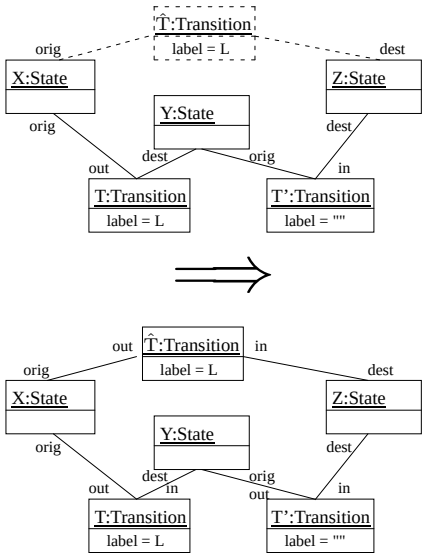


Figure 11 Bypassing internal actions: graphical rule.

3.5 Model Transformations

The operational semantics of DSML as defined in the previous section is just a particular case of an *endogenous* model transformation, i.e., a transformation where the source and target metamodels are the same. We naturally extend the abstract Definition 8 to model transformations between two different metamodels $\mathcal{MM}_1$ and $\mathcal{MM}_2$, as functions with domain $[\![\text{MEL}(\mathcal{MM}_1)]\!]$ and co-domain $\mathcal{P}_f([\![\text{MEL}(\mathcal{MM}_2]\!]))$. We also extend the executable Definitions 9, 10 to model transformations that rewrite terms of sort $Models_{\mathcal{MM}_1}$ to terms of sort $Set\{Models_{\mathcal{MM}_2}\}$ defined by reflection as in Section 3.4.

Example 8 We present a simple model transformation that implements the operation of *elimination of silent transitions* between the meta-models $\mathcal{MM}_1$ and $\mathcal{MM}_2$ of *automata*, resp. of *automata without silent transitions*, depicted in Figure 3 (without, respectively with, the OCL invariant). This also serves as illustration of conditional rules having *negative patterns* as conditions - patterns that must not match in order for the rule to apply - and of their encoding in Maude.

The transformation is expressed using two rules, one of which is shown in Figure 11. The solid-line pattern in the left-hand side consists of a transition T with a label L that may or may not be empty, followed by a silent transition T' whose label is the empty string. The origin and destination states of the transitions are also shown. The dotted-line pattern is a *negative pattern*: the rule *cannot* be applied if that pattern maches - essentially, if there is already a transition $\hat{T}$ labelled L from the origin of T to the destination of T'. (Without this negative pattern, the rule could always be applied, which leads to the undesired effect of nontermination). The effect of the rule consists in adding a transition such as $\hat{T}$.

A second rule, not shown here, erases the silent transitions from the model when the first rule cannot be applied.

The Maude rewrite rule shown in Figure 12 quite naturally corresponds to the graphical rule from Figure 11. Note

```
crl M => M' if
((eq 'out[X:Term] = TL:Term [none] .)
(eq 'orig[T:Term]  = X:Term [none] .)
(eq 'dest[T:Term]  =  Y:Term [none] .)
(eq 'orig[T':Term]  =  Y:Term [none] .)
(eq 'dest[T':Term]  =  Z:Term [none] .)
(eq 'in[Z:Term] = TL':Term [none] .)
(eq 'label[T:Term]  =  L:Term [none] .)
(eq 'label[T':Term]  =  '"".String [none] .)
 ES:EquationSet) := getEqs(M)
/\ negativePattern:EquationSet :=
((eq 'label['hatT:Variable] =  L:Term [none] .)
(eq 'orig['hatT:Variable] = X:Term [none] .)
(eq 'dest['hatT:Variable] = Z:Term [none] .))
/\ noMatch(negativePattern:EquationSet, ES:EquationSet)
/\ hatT:Term := newTransition('hatT:Variable) /\
M' := setEquations(addDecl(M, hatT:Term, 'Transition),
(eq 'out[X:Term] = '_',_[hatT:Term,TL:Term] [none] .)
(eq 'orig[T:Term]  = X:Term [none] .)
(eq 'dest[T:Term]  =  Y:Term [none] .)
(eq 'orig[T':Term]  =  Y:Term [none] .)
(eq 'dest[T':Term]  =  Z:Term [none] .)
(eq 'in[Z:Term] = '_',_[hatT:Term,TL':Term] [none] .)
(eq 'label[T:Term]  =  L:Term [none] .)
(eq 'label[T':Term]  =  '"".String [none] .)
(eq 'label[hatT:Term] =  L:Term [none] .)
(eq 'orig[hatT:Term] = X:Term [none] .)
(eq 'dest[hatT:Term] = Z:Term [none] .)
 ES:EquationSet) .

op noMatch : EquationSet EquationSet -> Bool .

eq noMatch(
  ((eq 'label['hatT:Variable] =  L:Term [none] .)
   (eq 'orig['hatT:Variable] = X:Term [none] .)
   (eq 'dest['hatT:Variable] = Z:Term [none] .)),
  ((eq 'label[hatT:Term] =  L:Term [none] .)
   (eq 'orig[hatT:Term] = X:Term [none] .)
   (eq 'dest[hatT:Term] = Z:Term [none] .)
    E:EquationSet) ) = false .
eq noMatch(E1:EquationSet,E2:EquationSet) = true [owise] .
```

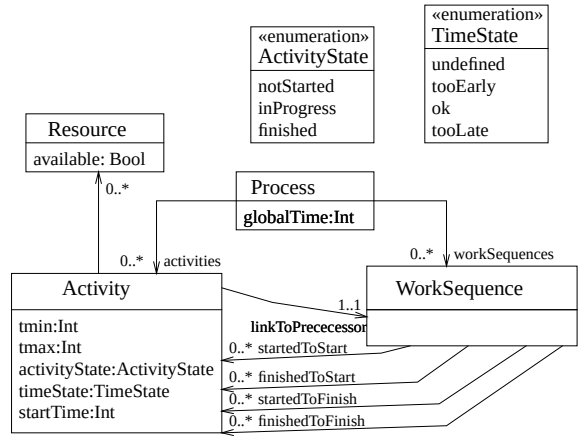
Figure 12 Maude Rewrite rule for bypassing transitions.

that it is a conditional rule; its condition is stated in the `if` clause, which does most of the work. A module `M` (denoting Maude specifications at Maude's metalevel) rewrites to another module `M'` if its set of equations does match the positive pattern and does not match the negative pattern in Figure 11. The latter condition is equationally specified by the function *noMatch*, which returns *false* if a certain match is found, and *true* otherwise (in the equation labelled `[owise]`). Then, `M'` is a copy of `M` whose declaration and equation sets are changed to fit the right-hand side of the rule in Figure 11. This is achieved by functions `newTransition`, `addDecl`, and `setEquations`, which we have omitted from the figure.

In Section 5 we shall study semantics-preserving model transformations and shall apply a procedure, defined there, for checking whether the silent-transition elemination transformation, discussed in this section, is semantics-preserving.

4 Defining the xSPEM language

In this section we study a DSML called xSPEM [15], which is an executable version of the SPEM language standard [16]. This further illustrates the approach presented in Section 3, and prepares an example of "execution tracing" for Section 5. We also propose an optimisation based on a partial evaluation of operational-semantics rules, which takes advantage of the



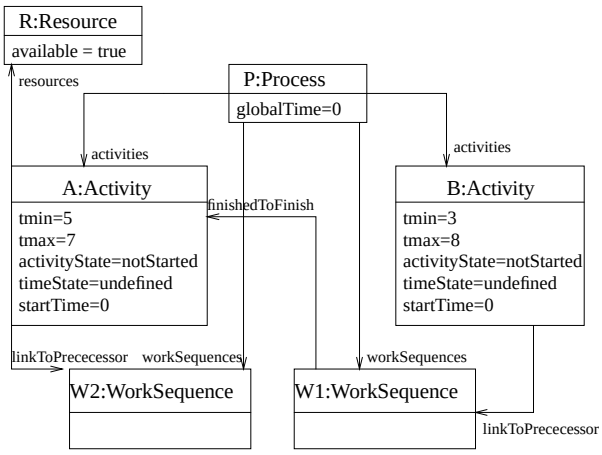


Figure 13 xSPEM metamodel (simplified), and one model.

inherent distinction between "structural" and "dynamic" parts of metamodels and of their models, hence, it is applicable in general for DSML. The optimisation concerns the execution of operational semantics starting from a given initial model.

4.1 The xSPEM *language and its operational semantics*

The language describes the execution of *activities* constrained by time, resources, and precedence relations. We show how to translate the xSPEM metamodel and models into Maude specifications, and how to encode the language's operational semantics as rewrite rules over such Maude specifications.

In the metamodel of Figure 13 *Activity* is the class of entities being executed. The *tmin* and *tmax* attributes of the *Activity* class denote the minimum and the maximum duration of activities, whose state with respect to execution is given by the value of the *activityState* attribute: the *notStarted*, *inProgress*, and *finished* values in the *ActivityState* enumeration.

Figure 14 Rule for Process incrementing the global time.

The execution of activities is also governed by explicit ordering constraints (*WorkSequence* class), and by the availability of resources (*Resource* class).

Each activity has a *WorkSequence* instance, which in turn may be linked to four (possiby empty) sets of activities:

- the activities that have to be started to allow for the current activity to be able to start (the *startedToStart* link);
- the activities that have to be finished to allow for the current activity to be able to start (the *finishedToStart* link);
- the activities that have to be started to allow for the current activity to be able to finish (the *startedToFinish* link);
- the activities that have to be finished to allow for the current activity to be able to finish (the *startedToFinish* link).

For example, in Figure 13, the activities B and A are linked by a *WorkSequence* via the link *finishedToFinish*, which expresses that B is allowed to finish only when A is finished. An activity may also have a number of *Resource* instances. Starting an activity requires that the resource be *available*, and makes the resources not available; when an activity finishes, it releases the resource by making it available again.

Time is measured by a clock, encoded by the *globalTime* attribute of the *Process* class. When an activity starts it records its starting time in the *startTime* attribute. Hence, its current execution time is the difference $globalTime - startTime$.

When it is finished, an activity can be *tooEarly*, *ok*, or *tooLate* (*timeState* attribute), depending on whether its current execution time is in $[0, tmin)$, $[tmin, tmax)$ or $[tmax, \infty)$ respectively (all intervals left-closed, right-open). The value of *timeState* equals *undefined* while an activity is not *finished*.

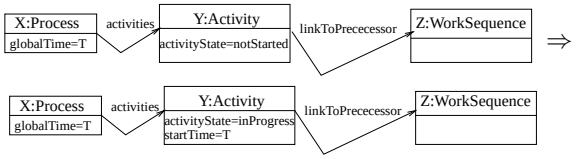
Operational Semantics. We express the operational semantics of xSPEM using graphical rewrite rules (Figures 14–16). In the first rule, the *process* instance increments *globalTime*. The next rule (Figure 15) deals with starting activities. If Y is an activity of process X whose *globalTime* attribute is T, and Y is linked to its predecessors by a work sequence Z, then starting the activity sets its *startTime* attribute to T and its *activityState* attribute to *inProgress*. However, the activity can only be started if certain other instances are in certain states. We add conditions (written in OCL) to the rules, for

- all activities in *Z.startedToStart* to be in progress;
- all activities in *Z.finishedTo Start* to be finished;
- all resources in *Y.resources* to be available;
- all resources in *Y.resources* become unavailable.

For the fourth constraint we use (cf. Figure 15) the construction *@Post* to indicate that the constraint is a postcondition.

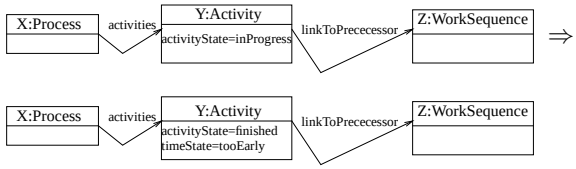
The last three rules (Figure 16) deal with finishing an activity and releasing the resources it held while it was executing. The rules differ on the value that they give to the *timeState* attribute, depending on how long has the activity been executing, i.e., on $X.globalTime - Y.startTime$:

- if the value in question is greater or equal than *Y.tmin*, but less than *Y.tmax*, then the attribute *timeState* is set to *ok*;
- if it is less than *Y.tmin* then *timeState* is set to *tooEarly*;
- otherwise, the attribute *timeState* is set to *tooLate*.

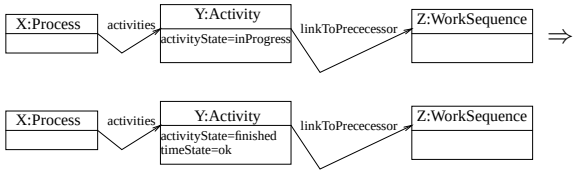


$$if\ Z.finishedToStart \rightarrow forAll(u : Activity | u.activityState = finished) \wedge$$
$$Z.startedToStart \rightarrow forAll(v : Activity | v.activityState = inProgress) \wedge$$
$$Y.resources \rightarrow forAll(r : Resource | r.available = true) \wedge$$
$$Y.resources \rightarrow forAll(r : Resource | r.available@Post = false)$$

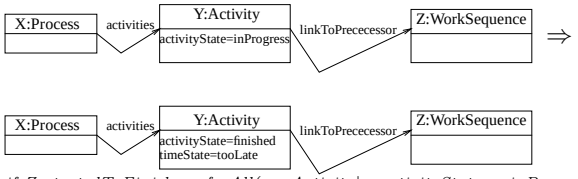
Figure 15 Rule for starting an Activity.



$$if\ Z.startedToFinish \rightarrow forAll(u : Activity | u.activityState = inProgress) \wedge$$
$$Z.finishedToFinish \rightarrow forAll(v : Activity | v.activityState = finished) \wedge$$
$$X.globalTime - Y.startTime < Y.tmin \wedge$$
$$Y.resources \rightarrow forAll\{r : resource | r.available@Post = true\}$$



$$if\ Z.startedToFinish \rightarrow forAll(u : Activity | u.activityState = inProgress) \wedge$$
$$Z.finishedToFinish \rightarrow forAll(v : Activity | v.activityState = finished) \wedge$$
$$\wedge X.globalTime - Y.startTime \geq Y.tmin \wedge$$
$$X.globalTime - Y.startTime < Y.tmax \wedge$$
$$Y.resources \rightarrow forAll\{r : resource | r.available@Post = true\}$$



$$if\ Z.startedToFinish \rightarrow forAll(u : Activity | u.activityState = inProgress) \wedge$$
$$Z.finishedToFinish \rightarrow forAll(v : Activity | v.activityState = finished) \wedge$$
$$X.globalTime - Y.startTime > Y.tmax \wedge$$
$$Y.resources \rightarrow forAll\{r : resource | r.available@Post = true\}$$

Figure 16 Rules for finishing an Activity.

4.2 Embedding xSPEM *in Maude*

We follow the guidelines of Section 3 for representing the xSPEM metamodel and sample model in Figure 13, as well as the operational semantics rules from Figures 14–16.

Metamodel and model. The Maude encoding of the xSPEM metamodel and model from Figure 13 is shown in Figure 17.

```
fmod ACTIVITYSTATE is
sort ActivityState .
ops notStarted inProgress finished : -> ActivityState .
endfm

fmod TIMESTATE is
sort TimeState .
ops tooEarly ok tooLate undefined : -> TimeState .
endfm

fmod xSPEM-METAMODEL is
protecting ACTIVITYSTATE .
protecting TIMESTATE .
protecting INT .
protecting BOOL .
sorts Process Activity WorkSequence Resource .
--- declarations for sets of Process, Activity,
--- WorkSequence, and Resource (omitted)
op globalTime : Process -> Int .
op activities : Process -> Set{Activity} .
op workSequences : Process -> Set{WorkSequence} .
ops tmin tmax startTime : Activity -> Int .
op activityState : Activity -> ActivityState .
op timeState : Activity -> TimeState .
op linkToPredecessor : Activity -> Set{WorkSequence} .
op resources : Activity  -> Set{Resource} .
--- equations for OCL invariant for cardinality of
--- role linkToPredecessor (omitted)
ops startedToStart finishedToFinish
      startedToFinish finishedToStart :
         WorkSequence  -> Set{Activity} .
op available : Resource -> Bool .
endfm

fmod xSPEM-MODEL is
extending xSPEM-METAMODEL .
op P : -> Process .
ops A B : -> Activity .
ops W1 W2 : -> WorkSequence .
op R : -> Resource .
eq globalTime(P) = 0 .
eq activities(P) = A, B .
eq workSequences(P) = W1, W2 .
eq tmin(A) = 5 .
eq tmax(A) = 7 .
eq startTime(A) = 0 .
eq activityState(A) = notStarted .
eq timeState(A) = undefined .
eq linkToPredecessor(A) = W2 .
eq resources(A) = R .
eq available(R) = true .
eq tmin(B) = 3 .
eq tmax(B) = 8 .
eq startTime(B) = 0 .
eq activityState(B) = notStarted .
eq timeState(B) = undefined .
eq linkToPredecessor(B) = W1 .
eq resources(B) = empty .
eq finishedToFinish(W1) = A .
eq startedToFinish(W1) = empty .
eq finishedToStart(W1) = empty .
eq startedToStart(W1) = empty .
eq finishedToFinish(W2) = empty .
eq startedToFinish(W2) = empty .
eq finishedToStart(W2) = empty .
eq startedToStart(W2) = empty .
endfm
```

Figure 17 Maude encoding of metamodel and model of Fig. 13.

There are two modules for the enumeration classes. They are imported (in protecting mode, to preserve their semantics) in the module denoting the metamodel. The module denoting the metamodel is imported (in extending mode, allowing to modify its semantics) by the module denoting the model.

Operational semantics. We show the Maude encoding of the rule in Figure 14 and of the conditional rule in Figure 15; the encoding of the remaining graphical rules is similar.

```
rl
(eq 'globalTime[X:Term] = T:Term [none] .) =>
(eq 'globalTime[X:Term] =
      upTerm(downTerm('_+_[T:Term,'s_['0.Zero]], errorNat)
        [none] .) .
```

Figure 18 Maude encoding of the rule in Figure 14.

We first write a module `Models_xSPEM-METAMODEL`, where a sort `Models_xSPEM-METAMODEL` is defined (by analogy to Fig. 7), which metarepresents all Maude specifications denoting xSPEM models. The xSPEM operational semantics rules are Maude rewrite rules operating over this sort.

The Maude encoding of the rule for time-passing in Figure 14 is shown in Figure 18. It says that whenever an equation stating that *the global time of some process (metarepresented by the term variable X) equals some value (metarepresented by the term variable T)* is found, then that equation is replaced by another one, which states that *the global time of the process X equals T plus the metarepresentation of 1, which is* `'s['0.Zero]`. The resulting meta-level term `'_+_[T:Term,'s_['0.Zero]]` is not directly evaluated because there is no equation to reduce it at the metalevel. In order to be evaluated, the term is casted from the metalevel down to the object level, where addition is performed by equational reduction. The casting is done by the built-in function `downTerm`, whose second argument is a constant returned in case the casting fails. Finally, the result of the addition is re-raised to the metalevel by the operation `upTerm`.

We now focus on the rule in Figure 15 for starting an activity, whose encoding in Maude is shown in Figure 19. The rule is conditional, and most of its computation is encoded in the condition. It says that a model `M` rewrites to a model `M'` if

- the equations of `M` encode the attribute values and the links corresponding to those in the left-hand side of the rule in Figure 15. In the conjunct `downTerm(Y:Term, ErrAct)` in `downTerm(L:Term,ErrAct)`, the predefined function `_in_` evaluates whether an activity metarepresented by `Y:Term` is in an activity list metarepresented by `L:Term`. For this evaluation to be performed, the metarepresentations are casted down to the object level;
- the OCL precondition evaluates to *true*. This condition is encoded in accordance to the Maude encoding of OCL invariants, presented earlier in Section 3. Note the definitions of the functions `forAll1`, `forAll2`, `forAll3`, which encode the first three $\rightarrow forAll()$ iterators in the graphical rule; to be evaluated on the adequate model `M`, the model is passed as an argument to those functions;
- finally, the model `M'` is also computed in the condition, by setting its equation set such as to encode the right-hand side of the graphical rule (the second "line" of Figure 15), and by applying the `forAll4` function to the result. The role of the latter is to encode the postcondition of the graphical rule (last conjunct in the condition in Figure 15). The function takes a model `M` and a set of resources and "assigns" all the resources' *available* attributes to *false*. For this, it replaces in `M` the equations that gave whatever "previous" values of the attribute, with

```
crl  M => M' if
((eq 'globalTime[X:Term] = T:Term [none] .)
(eq 'activities[X:Term] = L:Term [none] .)
(eq 'activityState[Y:Term] =
                  'notStarted.ActivityState [none] .)
(eq 'startTime[Y:Term] = '0.Zero [none] .)
(eq 'linkToPredecessor[Y:Term] = Z:Term [none] .)
 ES:EquationSet) := getEqs(M)  /\
downTerm(Y:Term,ErrAct) in downTerm(L:Term,ErrAct) /\
forAll1(M,startedToStart(downTerm(Z:Term,ErrWorkSeq))) /\
forAll2(M,finishedToStart(downTerm(Z:Term,ErrWorkSeq))) /\
forAll3(M',resources(downTerm(Y:Term,ErrAct))) /\
M' := forAll4(setEquations(M,
(eq 'globalTime[X:Term] = T:Term [none] .)
(eq 'activities[X:Term] = L:Term[none] .)
(eq 'activityState[Y:Term] =
                  'inProgress.ActivityState [none] .)
(eq 'startTime[Y:Term] = T:Term [none] .)
(eq 'linkToPredecessor[Y:Term] = W:Term [none] .)
 ES:EquationSet), resources(downTerm(Y:Term,ErrAct))) .
  ---- encoding of the OCL condition
 op forAll1 : Module Set{Activity} -> Bool .
 eq forAll1(empty) = true .
 eq forAll1(M:Module, (A:Activity, AS:Set{Activity})) =
   (if activityState(M:Module, A:Activity) == inProgress
    then true else false fi)
    and-then forAll1(M:Module,AS:Set{Activity}) .

 op forAll2 : Module Set{Activity} -> Bool .
 eq forAll2(M:Module, empty) = true .
 eq forAll2(M:Module, (A:Activity, AS:Set{Activity})) =
   (if activityState(M:Module, A:Activity) == finished
    then true else false fi)
    and-then forAll2(M:Module,AS:Set{Activity}) .

 op forAll3 : Module Set{Resource} -> Bool .
 eq forAll3(M:Module, empty) = true .
 eq forAll3(M:Module, (R:Resource, RS:Set{Resource})) =
   (if available(M:Module, R:Resource) == true
    then true else false fi)
    and-then forAll3(M:Module,RS:Set{Resource}) .

op forAll4 : Module Set{Resource} -> Module .
eq forAll4(M:Module, empty) = M:Module .
ceq forAll4(M:Module, (P:Resource, PS:Set{Resource})) =
replaceEq(E:Equation,forAll4(M:Module,PS:Set{Resource}))
if P:Term := upTerm(P:Resource)
/\ PS:Term := upTerm(false)
/\ E:Equation := (eq 'available[P:Term] = PS:Term[none].) .
```

Figure 19 Maude encoding of the rule in Figure 15.

equations stating that the new values are *false*. The equation replacement is done by the function `replaceEq`, which is omitted from Figure 15 for better readability.

The Maude rules shown in Figures 18 and 19, together with similar Maude encodings of the other graphical rules, are executable and be used for verification purposes. For example, one can ask whether there exists a path starting from the xSPEM model depicted in Figure 13 and leading to a model where both activities `A` and `B` are finished and have completed their execution in time. Assuming functions `allFinished` and `allOk`, which check whether all the equations encoding the attributes `activityState` and `timeState` have the right-hand sides `'finished.ActivityState` and `'ok.TimeState`, respectively, the answer to our question is returned by the following Maude search command:

```
search[1] upModule('xSPEM-MODEL,false)=>*M
such that allFinished(M) and allOK(M).
```

Maude responds instantly and returns a path to a solution.

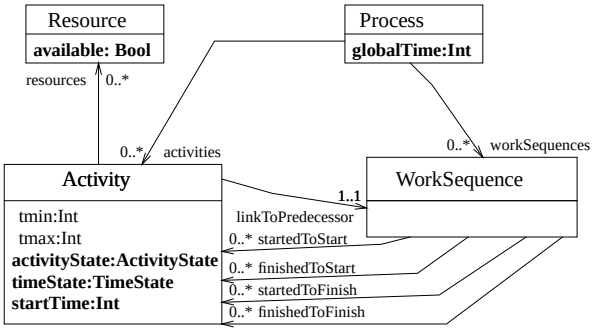


Figure 20 xSPEM metamodel; the dynamic part is in bold font.

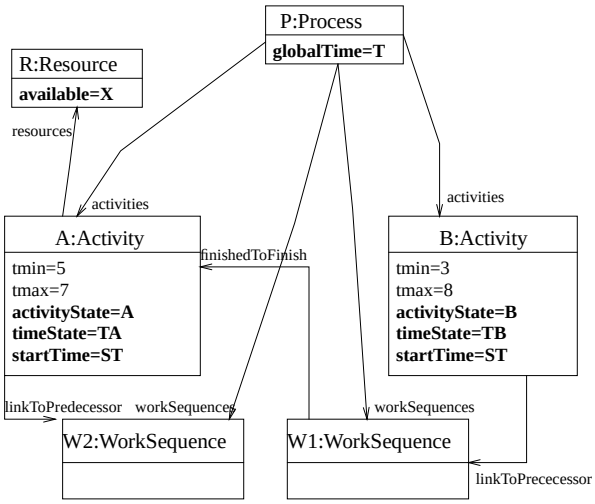


Figure 21 Pattern for xSPEM models reachable from that in Fig. 13.

Optimising the semantics by partial instantiation. We now describe how one can take advantage of the inherent distinction between the structural and dynamic parts of metamodels of DSML, in order to optimise their operational semantics.

To illustrate this we consider again the xSPEM metamodel in Figure 13. Its dynamic part consists of the *globalTime* attribute of the *Process* class, of the *activityState, timeState, startTime* attributes of the *Activity* class, and of the *available* attribute of the *Resources* class. These attributes are the only features of an xSPEM model that can change during model execution. All the rest is the structural part - including the instances and the links between them - and does not change.

This distinction between structural and dynamic parts is inherent to DSML defined using the MDE-based approach. The consequence is that once an "initial" model is chosen, all models reachable from it have the same structural part. In the case of xSPEM, if we start from the model shown in Figure 13, all the reachable models have the form shown in Figure 21.

In particular, this means that all the operational-semantics rules will perform their matching on the *same* structural part; we can take advantage of this observation by *partially instantiating* the operational-semantics rules on the structural part before executing them. This results in smaller and simpler rules, with less matching to do. Since matching (particularly set and multiset-matching, which we extensively use in our representation - e.g., we match over sets of equations) is the

most expensive part of rewriting we can expect substantial time gains when the simplified rules are executed.

The partial evaluation of rules consists in applying the following four operations to each operational-semantics rule:

1. match the left-hand side (lhs) of the rule with the pattern describing the form of reachable models (e.g., Figure 21); this results in a finite number of substitutions;
2. for each such substitution, generate a new rule by applying the substitution to the rule's lhs, rhs, and condition;
3. perform all reductions and simplifications possible on the new "partially instantiated" rule; for example, a conjunct reduced to *true* in a rule's condition is removed, and a conjunct reduced to *false* elimimates the rule;
4. make the rule *context-free*; for this, find a maximal context C such that the rule obtained after the last step has the form $C[lhs'] \rightarrow C[rhs']$ *if* $Cond$, and replace the rule with the *non-contextual* rule $lhs' \rightarrow rhs'$ *if* $Cond$, provided that the latter is indeed a rule and provided the fact that the contextual and the non-contextual rule induce the same matchings with the pattern of reachable models.

These operations are implemented by taking advantage of Maude's bultin metalevel functions for matching and reduction. They do not modify the rewrite relation starting from the chosen initial model, since Steps 1–3 are executed when the original operational-semantics rules are applied, and Step 4 explicitly checks that the matchings (hence, the rewritings) before and after the step are the same. What changes is the number of rules - one original operational-semantics rule possibly generates several simplified rules - but the number of rule applications for performing a given execution stays the same; it only involves simpler rules and less matching.

For the original rule shown in Figure 19, applying the four steps described above with the pattern shown on Figure 21 generates two rules: one for starting the activity A and the other one for starting the activity B. We show in Figure 22 the first rule (for A), which is indeed much simpler that the original one; in particular, the simplified rule is unconditional: all the conjuncts in its condition were reduced to *true*, and it is non-contextual: its lhs/rhs are sets of equations, not modules.

Just for the sake of the example, we have tried both versions of the operational semantics on a model obtained from the one shown in Figure 13 by multiplying the *tmin* and *tmax* constants by 10 and then by 20. On the `search` command (Page 13) the optimised rules terminated about twice faster than the original ones (18s and 5m3s, against 41s and 11m58s).

```
rl
(eq 'globalTime['P.Process] = T:Term [none]  .)
(eq 'activityState['A.Activity] =
  'notStarted.ActivityState [none] .)
(eq 'startTime['A.Activity] =  '0.Zero  [none] .)
(eq 'available['R.Resource] = 'true.Bool [none] .)
=>
(eq 'globalTime['P.Process] = T:Term [none] .)
(eq 'activityState['A.Activity] =
 'inProgress.ActivityState  [none] .)
(eq 'startTime['A.Activity] =  T:Term  [none] .)
(eq 'available['R.Resource] = 'false.Bool [none] .) .
```

Figure 22 Simplified rule for starting activity A.

5 Semantics-Preserving Model Transformations

Given two DSML $\mathcal{L}_1$ and $\mathcal{L}_2$, each endowed with an operational semantics, and given a model transformation ϕ between $\mathcal{L}_1$ and $\mathcal{L}_2$, how to define the fact that the transformation *preserves* the operational semantics when translating from $\mathcal{L}_1$ to $\mathcal{L}_2$? When $\mathcal{L}_1$ is a higher-level language, $\mathcal{L}_2$ is a lower-level one, and ϕ is a refinement between levels, semantical preservation means that the image in $\mathcal{L}_2$ of any model in $\mathcal{L}_1$ by the transformation does "at least as much" as the original, in the sense that to each execution of the copy there exists a "matching" execution of the original. This ensures that the refinement process does not add executions unaccounted for.

In this section we formalise the notion of semantics-preserving model transformation using a notion of *observational simulation* between *observational transition systems*. Observational transition systems are adequate for modelling operational semantics expressed in terms of model transformations because they allow for an emphasis on the "dynamic" part of models, that which changes during execution; and observational simulations compare executions with respect to the observations only. Another advantate of observational simulations is that they allow for one step of the higher-level semantics to match several steps of the lower-level one, which expresses a difference of "granularity" between the levels.

We propose a semidecision procedure to check semantical preservation. The procedure is complete: it detects all preservation violations, and may not terminate otherwise.

Thanks to its encoding of semantical preservation by an invariance poperty, the procedure also opens the possibility of using theorem-proving for invariance properties, also available in Maude [10,11], for proving that simulation does hold.

We also give a version of the procedure that computes an encoding of all the executions of an instance of $\mathcal{L}_1$ that "match" a given execution of the instance's image by ϕ, which provides us with "execution traceability". We give examples based on our encodings of automata and XSPEM in Maude.

5.1 Obervational Transition Systems

Observational transition systems OTS [9] are transition systems together with an observation domain and an observation function that maps states to observations in the domain.

Definition 11 (Observational Transition System) *An* OTS *is a tuple* $\langle A, a_0, \rightarrow, O, \omega \rangle$ *where* A *is a nonempty, possibly infinite set of* states*;* $a_0 \in A$ *is the initial state;* $\rightarrow \subseteq A \times A$ *is the* transition relation*;* O *is a nonempty, possibly infinite set of* observations*;* $\omega : A \rightarrow O$ *is the* observation function.

An *execution* is a finite sequence of states $\rho = a_0, ...a_i...a_n$ such that for $a_i \rightarrow a_{i+1}$ for $i = 0, 1, ..., n-1$ (note that we do not require that executions start in the initial state). We denote the *length* n of an execution $\rho = a_0, ...a_i...a_n$ by $len(\rho)$; hence, an execution of length 0 is a state. For a state

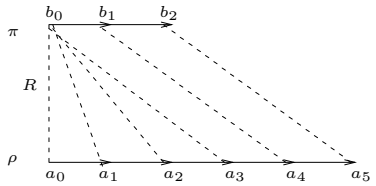


Figure 23 Matching executions.

a, we denote by $exec(a)$ the set of executions π such that $\pi(0) = a$, and by $exec(\mathcal{A})$ the set of executions $exec(a_{ini})$.

5.2 From DSML *to Observational Transition Systems*

We naturally identify a meta-model $\mathcal{MM}$ with the set of models that conform to it, and the operational semantics of a DSML of metamodel $\mathcal{MM}$ with a relation $\rightarrow \subseteq \mathcal{MM} \times \mathcal{MM}$. By choosing an "initial model" $\mathcal{M}^0 \in \mathcal{MM}$ we obtain a transition system $\langle \mathcal{MM}, \mathcal{M}^0, \rightarrow \rangle$, which expresses the execution of the model $\mathcal{M}^0$ according to our DSML's semantics. An observational transition system can obtained, e.g, by defining an OCL query on $\mathcal{MM}$ which expresses a "part" of models that "changes" during execution. For example, for the metamodel in Figure 3, we want to observe, say, the *trace* attribute of the *Automaton* class, which does change during execution. Assuming only one automaton per model of the metamodel, this can be written in OCL as $Automaton.allInstances.trace$.

5.3 Matching and Observational Simulation

Definition 12 (matching) *For* OTS $= (A, a_{ini}, \rightarrow_{\mathcal{A}}, O, \omega_{\mathcal{A}})$ *and* $\mathcal{B} = (B, b_{ini}, \rightarrow_{\mathcal{B}}, O, \omega_{\mathcal{B}})$ *and for two executions* $\rho \in exec(\mathcal{A})$ *and* $\pi \in exec(\mathcal{B})$*, we say that* ρ *is* matched *by* π *if there exists a function* $\alpha : [0, \ldots, len(\rho)] \rightarrow \mathbb{N}$ *with* $\alpha(0) = 0$ *and* $\forall i \in [0, \ldots, len(\rho) - 1]$, $\alpha(i+1) \in \{\alpha(i) + 1, \alpha(i)\}$, *such that for all* $i \in [0..len(\rho)]$, $\omega_{\mathcal{A}}(\rho(i)) = \omega_{\mathcal{B}}(\pi(\alpha(i)))$.

Example 9 We illustrate matching executions in Figure 23. States with identical observations are connected with dashed lines. The function $\alpha : [0, \ldots, 5] \rightarrow \mathbb{N}$ defined by $\alpha(0..3) = 0$ and $\alpha(4) = 1$, $\alpha(5) = 2$ ensures that the execution ρ (of length 5) is matched by the execution π (of length 2).

Our notion of matching in Def. 12 allows shorter executions to match longer ones. This is useful for relating executions of DSML whose semantics have different granularities; if $\mathcal{L}_2$ is a more "concrete" language than $\mathcal{L}_2$, it can be expected that one step of $\mathcal{L}_1$ is "implemented" by several steps of $\mathcal{L}_2$.

Definition 13 (observational simulation) *Given two observational transition systems* $\mathcal{A} = (A, a_{ini}, \rightarrow_{\mathcal{A}}, O, \omega_{\mathcal{A}})$ *and* $\mathcal{B} = (B, b_{ini}, \rightarrow_{\mathcal{B}}, O, \omega_{\mathcal{B}})$*, we say that* $\mathcal{A}$ *is* observationally simulated by $\mathcal{B}$ *if for all executions* $\rho \in exec(\mathcal{A})$ *there exists an execution* $\pi \in exec(\mathcal{B})$ *such that* ρ *is matched by* π.

```
crl ⟨M,S⟩ ⇒⟨M',S'⟩
if M',S'' := step2(M) ∧
S' := {M'' ∈ S ∪ step1(S) | ω1(M'') = ω2(M')}
```

Figure 24 Rewrite rule for checking semantical preservation.

5.4 Semantical preservation and a procedure to check it

We use observational simulations to define semantics-preserving model transformation connecting two instances of two DSML represented as OTS as suggested in Section 5.2.

Definition 14 (semantics-preserving model transformation) *Consider two* DSML $\mathcal{L}_1, \mathcal{L}_2$ *with metamodels* $\mathcal{MM}_i$ *and semantics* $\rightarrow_i$*, for* $i = 1, 2$*. Assume an observation set* O *and observation functions* ω_i*, for* $i = 1, 2$ *having codomain* O*. We say that a model transformation* ϕ *between* $\mathcal{MM}_1$ *and* $\mathcal{MM}_2$ *is* semantics-preserving for the instances $\mathcal{M}_1^0 \in \mathcal{MM}_1$ and $\mathcal{M}_2^0$*, if* $\mathcal{M}_2^0 \in \phi(\mathcal{M}_1^0)$ *and* $\langle \mathcal{MM}_2, \mathcal{M}_2^0, \rightarrow_2, O, \omega_2 \rangle$ *is observationally simulated by* $\langle \mathcal{MM}_1, \mathcal{M}_1^0, \rightarrow_1, O, \omega_1 \rangle$.

To check semantical preservation in Maude, we write two functions `step1` and `step2`, which take a set of models in $\mathcal{MM}_1$ and in $\mathcal{MM}_2$, respectively, and apply one step of the operational semantics of $\mathcal{MM}_1$ and of $\mathcal{MM}_2$, respectively.

We then write the conditional rewrite rule in Figure 24: Here, any pair $\langle \mathcal{M}, \mathcal{S} \rangle$ is rewritten to a pair $\langle \mathcal{M}', \mathcal{S}' \rangle$ where

- $\mathcal{M}'$ is *some* 1-step successor of $\mathcal{M}$ according to the operational semantics of $\mathcal{MM}$. This is done by the matching equation $\mathcal{M}', \mathcal{S}'' :=$ `step2`$(\mathcal{M})$ in the rule's condition;
- $\mathcal{S}'$ is the subset of the models in $\mathcal{S} \cup$ `step1`$(\mathcal{S})$ whose observation according to ω_1 equals the observation $\omega_2(\mathcal{M}')$.

Our procedure consists in performing the Maude command:

(‡) `search` $\langle \mathcal{M}_2^0, \mathcal{M}_1^0 \rangle$ `=>*` $\langle \mathcal{M}, \emptyset \rangle$.

Proposition 4 (semantical preservation) *A model transformation* φ *is semantics-preserving for* $\mathcal{M}_1^0$ *and* $\mathcal{M}_2^0 \in \phi(\mathcal{M}_1^0)$ *if and only if* $\omega_1(\mathcal{M}_1^0) = \omega_2(\mathcal{M}_2^0)$ *and the command* (‡) *fails.*

Proposition 4 states the correctness of our procedure. Completeness follows from the completeness of Maude's search command: if a term $\langle \mathcal{M}, \emptyset \rangle$ is reachable then it will be found.

Towards theorem proving. Our procedure also suggests an approach based on inductive theorem proving to show that a simulation does hold, i.e., that a model transformation preserves operational semantics: inductively prove that terms of the form $\langle \mathcal{M}, \emptyset \rangle$ cannot be reached from $\langle \mathcal{M}_2^0, \mathcal{M}_1^0 \rangle$ by the rule in Fig. 24, using, e.g., Maude's prover [10] and techniques for proving invariants [11]. This is left for future work.

Example 10 We illustrate the procedure on an example based on automata. Let $\mathcal{L}_1$ be the language of automata possibly with silent transitions, whose metamodel $\mathcal{MM}_1$ is shown in Fig. 3 (without the OCL invariant) and whose operational semantics $\rightarrow_1$ is given by the rule depicted in Fig. 8. Let $\mathcal{M}_1^0$ be the automaton model depicted in Fig. 4. The Maude representations of the metamodel, operational semantics, and model are shown in Figs. 5 (without the OCL invariant), 9, and 6.

Let $\mathcal{L}_2$ be the language of automata without silent transitions, having metamodel $\mathcal{MM}_2$ shown in Fig. 3. Its operational semantics $\rightarrow_2$ is also given by the rule in Figure 4. Let $\mathcal{M}_2^0$ be essentially the same automaton model as that depicted in Figure 4, except that the label of t_2 is "b" rather than "".

In order to turn the transition systems $\langle \mathcal{MM}_i, \mathcal{M}_i^0, \rightarrow_i \rangle$ (for $i = 1, 2$) into observational transition systems, we consider the observation domain O of Strings, and the observation functions that to each model associates the *trace* attribute of the $Automaton$ class, which changes during execution.

What is missing in order to illustrate our semantics-preservation checking procedure is a model transformation between $\mathcal{L}_1$ and $\mathcal{L}_2$. This shall be the operation of silent transition elimination, partially illustrated by the graphical rule shown in Fig. 11 and corresponding Maude rule in Fig. 12.

In order to check whether silent transition elimination ϕ is semantics-preserving for the instances $\mathcal{M}_1^0 \in \mathcal{MM}_1$ and $\mathcal{M}_2^0 = \phi(\mathcal{M}_1^0) \in \mathcal{MM}_2$ (cf. Definition 14) we use the Maude command `search` $\langle \mathcal{M}^0 2, \mathcal{M}_1^0 \rangle$ `=>*` $\langle \mathcal{M}, \emptyset \rangle$. The command does find a solution - meaning that ϕ fails to preserve operational semantics according to Definition 14.

The command also provides us with a path to the solution. By examining the path we realise the error in the model transformation: the automaton $\mathcal{M}_2^0$ can have the trace "ab" by firing two transitions, but $\mathcal{M}_1^0$ cannot: it needs three transitions starting from its initial state, including the one labeled by "". The origin of the error lies in the fact that silent transition elimination may generate several initial states - if the initial state of its input is the origin of a silent transition.

This example demonstrates that our procedure finds semantical-preservation errors in model transformations. The next section contains another example, and shows that the procedure can be adapted to solve the "execution tracing" problem.

5.5 Execution tracing

We consider a model transformation of xSPEM to *hierarchical extended state machines* (HESM, which are quite similar to the state machine diagrams of UML). Briefly, a transition can be fired if its origin state is active and if its guard (if present) evaluates to true; when a transition is fired, the assignments (if any) of the transition to the HESM's variables are performed, and the transition's destination state becomes active. A macro-state is a state containing a HESM. A transition originating in a macro-state is an abbreviation for a set of transitions with origins in all states of the macro-state. A parallel composition of HESM with shared variables consists in interleaving the firing of the transitions of the two HESM.

The effect of our transformation on a model consisting of a process P and a single activity A, whose worksequence has all its links empty, is shown in Figure 25. It consists of two HESM, among which the top one represents the process P incrementing the variable *globalTime* starting from zero.

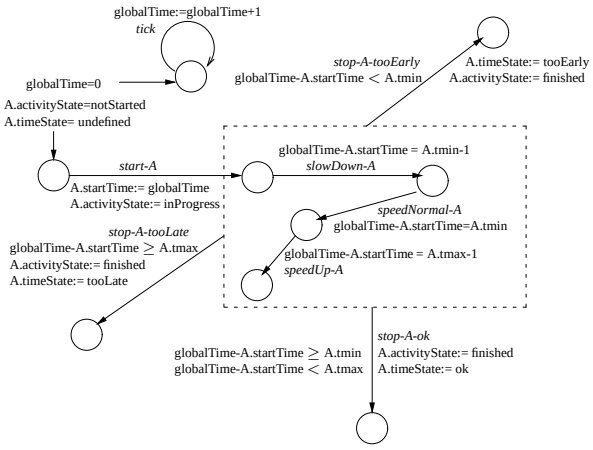


Figure 25 HESM encoding one Process P with one Activity A.

The bottom one encodes the activity's execution starting from the initial state (*A.activityState = notStarted, A.timeState = undefined*). The transition labelled *start-A* encodes the starting of the activity: the variable *startTime* records the current value of *globalTime*, and *A.activityState* is set to *inProgress*.

Up to this point our transformation does only the obvious. We now consider the following *refinement* of the xSPEM model: activities are now *tasks*, there is a notion of task *speed*, and our refinement "attempts" to avoid finishing a task too early or too late. Hence, one time unit before execution time reaches `A.tmin` the task is *slowed down*; after `A.tmin` is reached the speed of the task becomes *normal*; and one time unit before the time reaches `A.tmax` the task is *sped up*.

Eventually, the task completes its execution, and its variable *A.timeState* is set to one of among the values *tooEarly*, *ok*, or *tooLate*, depending on the time it took to complete. This is encoded by the three transitions originating in the macro-state (depicted as a rectangle with a dashed contour).

The general transformation from xSPEM to HESM encodes activities as the machine shown in Figure 25, possibly with more complex guards and assignments of transitions "starting" and "stopping" the activity, to take into account the states of the activities and resources that an activity is linked to.

Consider now the xSPEM model $\mathcal{M}$ that consists only of the process P, activity A, and worksequence W_2 in Figure 13. Its transformation to HESM is that represented in Figure 25 with *tmin =5* and *tmax=7*. We now consider the following execution of the HESM in Figure 25, which we describe only *via* the labels of the transitions that it fires in sequence:

*start-A (tick*4) slowDown-A tick speedNormal-A stop-A-ok*

We wish to trace back this execution to an xSPEM execution that matches in the sense of Definition 12. The answer depends on the observations functions on xSPEM and HESM.

For the case where the observation functions observe all the attributes in xSPEM, and all the (homonymous) variables in HESM, the answer is obviously the unique execution

*start (tick*5) stop-ok*

crl $\langle k, s\rangle \Rightarrow \langle k+1, s'\rangle$
if $s', \mathcal{S}$:=$\{\mathcal{M}'' \in \{\{s\} \cup$ step2$(\{s\}) \mid \omega_2(s') = \omega_1(\rho(k+1))\}$

Figure 26 Rewrite rule for tracing executions.

which we have described via the rules that are executed. Here, we have denoted by *tick* the rule in Figure 14, by *start* the rule in Figure 15, and by *stop-ok* the middle rule in Figure 16.

For other observation functions there may be also other matching executions. For example, if we choose not to observe in xSPEM and HESM the *globalTime* and *startTime* attributes and homonymous variables, then another matching execution inserts six *tick* actions between *start* and *stop-ok*.

For such simple examples it is easy to find the matching executions, but for more involved ones automation is necessary. This is achieved by a variant of our procedure for checking semantical preservation discussed earlier in this section.

For a sequence $\pi = \pi(0) \cdots \pi(i) \cdots \pi(n)$ we denote by $stuttering(\pi)$ the set $\{(\pi(0))^+ \cdots \pi(i)^+ \cdots (\pi(n))^+\}$, obtained by replicating each element of π finitely many times.

Proposition 5 *Consider two* OTS $\langle \mathcal{MM}_1, \mathcal{M}_1^0, \rightarrow_1, O, \omega_1\rangle$ *and* $\langle \mathcal{MM}_2, \mathcal{M}_2^0, \rightarrow_2, O, \omega_2\rangle$ *such that* $\omega_1(\mathcal{M}_1^0) = \omega_2(\mathcal{M}_2^0)$ *and an execution* $\rho \in exec(\mathcal{M}_2^0)$. *Consider also the rule in Figure 26 and the tree generated by the search command*

$(\sharp)$ search $\langle \rho(0), \mathcal{M}_1^0\rangle$ =>* $\langle \rho(len(\rho)), s\rangle$

Then, for every path in the tree, its projection on the second component belongs to the set $stuttering(\pi)$, *for some execution* $\pi \in exec(\mathcal{M}_1^0)$ *that matches the execution* ρ. *Reciprocally, for every execution* $\pi \in exec(\mathcal{M}_1^0)$ *that matches the execution* ρ, *there exists a path in the tree whose projection on the second component belongs to the set* $stuttering(\pi)$.

The reason why sequences in $stuttering(\pi)$ (not *the* matching execution π) occur in Proposition 5 is that π may be shorter that ρ - it may "stutter" when ρ takes a step. However, it is easy to reconstruct an execution π from a sequence in $stuttering(\pi)$, by trying to execute the sequence on the transition system for which π is supposed to be an execution.

6 Conclusion, Related, and Future Work

We propose a formal approach for defining and analysing Domain-Specific Modelling Languages. The approach is based on representing metamodels and models as Maude specifications, and on representing model transformations (which define the operational semantics of DSML as well as translations between DSML) as rewrite rules between Maude specifications, also expressible in Maude thanks to its reflectiveness.

This provides us, on the one hand, with abstract definitions of the MDE concepts used for defining DSML, which naturally capture their intended meaning; and, on the other hand, with equivalent executable definitions for those concepts, which can be used by Maude for formal verification.

Better execution and verification performances are obtained thanks to an optimisation that we propose, which is applied before execution/verification, and which consists of a form of partial evaluation of the rules that preserve the graph of reachable models starting from a given initial model. The optimisation takes advantage of the inherent distinction between structural and dynamic parts in DSML's metamodels.

We also propose a definition for the notion of semantics-preserving model transformations, which are translations between DSML that preserve operational semantics. We give a semidecision procedure, also implemented in Maude, for checking the semantics-preserving nature of a model transformation, and a version of the procedure for solving the "execution tracing" problem. We illustrate the approach on two examples: a simple one: finite automata, and a more elaborate one: xSPEM, a timed languages for executing activities constrained by time, precedence, and resource constraints.

Related Works. The closest related works are [2] and [3], who propose different encoding of metamodels, models, and model transformations in Maude. The main difference is that we encode metamodels as MEL specifications, while [2] base their representation on an object-oriented extension of Maude, and [3] use Maude sorts. This also induces differences in the way models and model transformations are represented.

We believe that our approach exploits better some of the simplest constructions of Maude: order-sorted specifications and their semantics based on algebras. We also study semantics preserving model transformations and execution tracing, which (to our best knowledge) are new for DSML in Maude. The optimisation of operational based on partial evaluation is also new. On the other hand, [2, 3] are more advanced in practical terms; their tools are integrated in the ECLIPSE environment, they propose user-friendly languages for users to define operational semantics, including real-time semantics [17, 18]; and they have performed significant case studies.

Among the many related works, graph transformations are formal modelling languages that have been used for defining semantics of DSML and of model transformations [19, 20, 21, 22, 23]. An advantage of Maude with respect to these approaches is that they abstract away from attribute values, whereas Maude is expressive enough to take into account attribute values as well as OCL constraints on them.

Another line of work based on theorem proving exploits type theory for formalising MDE artifacts, including a notion of correctness for model transformations [24, 25, 26].

Yet another, different approach is taken by the Kermeta framework[5], where methods written in an imperative language (also called Kermeta) language are *weaved* in a metamodel to make its underlying models executable [27]. This approach is not grounded in formal methods, but it is much more readily accessible to MDE practitioners who wish to define a DSML.

The present paper builds on our earlier work [5]. In addition to the representations of models, metamodels, and conformance from [5] we also study here operational semantics and model transformations, as well as semantical preservation and execution tracing for model transformations.

[5] http://www.kermeta.org

The paper [28] is a preliminary version of the present paper. The main additions with respect to [28] are the addition of the xSPEM example, the optimisation of operational semantics rules based on partial evaluation, more adequate definitions for semantical preservation, and the study of execution tracing, and more detailed proofs.

Execution tracing is also the object of [29]; the difference between [29] and the present paper is that [29] uses a different notion of execution matching - for example, it requires users to explicity define a relation beween states of transition systems, whereas in the present paper the relation is more conveniently induced by equality of observations on observational transition systems. From a more practical point of view, [29] is based on defining DSML in the Kermeta framework; Kermeta is an imperative language, which makes the implementation of execution tracing more difficult than in Maude (for instance, backtracking has to be performed explicitly, whereas in Maude it is done automatically). On the other hand, Kermeta is a well-accepted, user-friendly framework for DSML definition, whereas our Maude approach needs better interfaces in order to become acceptable to practitioners.

Regarding future work, we are planning to to connect to the ECLIPSE environment and to design a user-friendly, graphical-textual language (which was hinted at through several examples in this paper, but is not implemented yet) for expressing operational semantics and model transformations.

We also intend to explore the use of theorem-proving techniques for proving semantical preservation, based on its encoding into an invariance property shown in this paper.

References

1. G. D. Plotkin. A structural approach to operational semantics, 1981.
2. José Eduardo Rivera, Francisco Durán, and Antonio Vallecillo. Formal specification and analysis of domain specific languages using Maude. *Simulation: Transactions of the Society for Modeling and Simulation International*, 85(11 - 12):778–792, 2009.
3. Artur Boronat and José Meseguer. An algebraic semantics for MOF. *Formal Aspects of Computing*, 22(3-4):269–296, 2010.
4. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. *All About Maude, A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.
5. M. Egea and V. Rusu. Formal executable semantics for conformance in the MDE framework. *Innovations in Systems and Software Engineering*, 6:73–81, 2010.
6. José Meseguer. Membership algebra as a logical framework for equational specification. In Francesco Parisi-Presicce, editor, *WADT*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1997.
7. Benoît Combemale, Xavier Crégut, Pierre-Loïc Garoche, and Xavier Thirioux. Essay on semantics definition in MDE - an instrumented approach for model verification. *Journal of Software*, 4(9):943–958, 2009.
8. The object constraint language. `http://www.omg.org/spec/OCL`.
9. Kazuhiro Ogata and Kokichi Futatsugi. Simulation-based verification for invariant properties in the ots/cafeobj method. *Electr. Notes Theor. Comput. Sci.*, 201:127–154, 2008.
10. M. Clavel, M. Palomino, and A. Riesco. Introducing the ITP tool: a tutorial. *J. Universal Computer Science*, 12(11):1618–1650, 2006.
11. Vlad Rusu. Combining narrowing and theorem proving for rewriting-logic specifications. In *Tests and Proofs*, volume 6143 of *Lecture Notes in Computer Science*, pages 135–150, 2010.
12. Marina Egea. *An executable formal semantics for* OCL *with aopplications to model analysis and validation*. PhD thesis, Universidad Complutense de Madrid, 2008.
13. Jan Bergstra and John Tucker. Characterization of computable data types by means of a finite equational specification method. In *International Conference on Automata, Languages and Programming*, volume 81 of *Lecture Notes in Computer Science*, pages 76 – 90, 1980.
14. Manuel Clavel, Francisco Durán, Steven Eker, Santiago Escobar, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott. Unification and narrowing in maude 2.4. In Ralf Treinen, editor, *RTA*, volume 5595 of *Lecture Notes in Computer Science*, pages 380–390. Springer, 2009.
15. Reda Bendraou, Benoit Combemale, Xavier Crégut, and Marie-Pierre Gervais. Definition of an eXecutable SPEM2.0. In *14th APSEC*, Japan, December 2007. IEEE Computer Society.
16. Object Management Group, Inc. *Software Process Engineering Metamodel (SPEM) 2.0*, March 2007.
17. José Eduardo Rivera, Cristina Vicente-Chicote, and Antonio Vallecillo. Extending visual modeling languages with timed behavior specifications. In Antonio Brogi, João Araújo, and Raquel Anaya, editors, *CIbSE*, pages 87–100, 2009.
18. Artur Boronat and Peter Csaba Ölveczky. Formal real-time model transformations in MOMENT2. In David S. Rosenblum and Gabriele Taentzer, editors, *FASE*, volume 6013 of *Lecture Notes in Computer Science*, pages 29–43. Springer, 2010.
19. J. de Lara and H. Vangheluwe. Defining visual notations and their manipulation through meta-modelling and graph transformation. *Journal of Visual Languages and Computing*, 15(3-4):309–330, 2006.
20. Aditya Agrawal, Gabor Karsai, Sandeep Neema, Feng Shi, and Attila Vizhanyo. The design of a language for model transformations. *Software and System Modeling*, 5(3):261–288, 2006.
21. György Csertán, Gábor Huszerl, István Majzik, Zsigmond Pap, András Pataricza, and Dániel Varró. VIATRA - visual automated transformations for formal verification and validation of UML models. In *ASE*, pages 267–270. IEEE Computer Society, 2002.
22. Guilherme Rangel, Leen Lambers, Barbara König, Hartmut Ehrig, and Paolo Baldan. Behavior preservation in model refactoring using DPO transformations with borrowed contexts. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *ICGT*, volume 5214 of *Lecture Notes in Computer Science*, pages 242–256. Springer, 2008.
23. Dénes Bisztray, Reiko Heckel, and Hartmut Ehrig. Verification of architectural refactorings: Rule extraction and tool support. *ECEASST*, 16, 2008.
24. Iman Poernomo. The meta-object facility typed. In Hisham Haddad, editor, *SAC*, pages 1845–1849. ACM, 2006.
25. Iman Poernomo and Jeffrey Terrell. Correct-by-construction model transformations from partially ordered specifications in coq. In Jin Song Dong and Huibiao Zhu, editors, *ICFEM*, volume 6447 of *Lecture Notes in Computer Science*, pages 56–73. Springer, 2010.
26. Camillo Fiorentini, Alberto Momigliano, Mario Ornaghi, and Iman Poernomo. A constructive approach to testing model transformations. In Tratt and Gogolla [30], pages 77–92.

27. Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In *MoDELS*, volume 3713 of *Lecture Notes in Computer Science*, pages 264–278. Springer, 2005.
28. Vlad Rusu. Embedding domain-specific modelling languages into Maude specifications. *ACM Software Engineering Notes*. to appear in 2011.
29. Vlad Rusu, Laure Gonnord, and Benoît Combemale. Formally Tracing Executions From an Analysis Tool Back to a Domain Specific Modeling Language's Operational Semantics. Research Report RR-7423, INRIA, october 2010. submitted.
30. Laurence Tratt and Martin Gogolla, editors. *Theory and Practice of Model Transformations, Third International Conference, ICMT 2010, Malaga, Spain, June 28-July 2, 2010. Proceedings*, volume 6142 of *Lecture Notes in Computer Science*. Springer, 2010.

Appendix: additional proofs

Lemma 1 $[\![\text{MEL}(\mathcal{MM})]\!] = \{(\!|\text{MEL}_{\mathcal{MM}}(\mathcal{M})|\!) \mid \mathcal{M} :: \mathcal{MM}\}$.

Proof For the $\subseteq$ inclusion, from any $A \in [\![\text{MEL}(\mathcal{MM})]\!]$ we shall build a model $\mathcal{M}$ of $\mathcal{MM}$ such that $[\![\text{MEL}_{\mathcal{MM}}(\mathcal{M})]\!] = A$. Then, $A \in [\![\text{MEL}(\mathcal{MM})]\!]$ implies $A \models \text{OCL}_{\text{MEL}}(\mathcal{MM}) = true$ - because all algebras of a specification satisfy the equations of the specification. Finally, since $A = (\!|\text{MEL}_{\mathcal{MM}}(\mathcal{M})|\!)$ we obtain $(\!|\text{MEL}_{\mathcal{MM}}(\mathcal{M})|\!) \models \text{OCL}_{\text{MEL}}(\mathcal{MM}) = true$, which by Definition 7 is just the expected conclusion $\mathcal{M} :: \mathcal{MM}$.

To build $\mathcal{M}$ from A, for each proper sort c of $\text{MEL}(\mathcal{MM})$, we consider its interpretation $A(c)$, and let the elements of $A(c)$ be the objects of the class c in the model $\mathcal{M}$. We let the attribute values for those objects, as well as the links between the objects, to have values equal according to the algebra $\mathcal{A}$.

To conclude the $\subseteq$ inclusion we have to show that $\mathcal{M}$ is indeed a model of $\mathcal{MM}$ and that $[\![\text{MEL}_{\mathcal{MM}}(\mathcal{M})]\!] = A$.

- $\mathcal{M}$ is a model of $\mathcal{MM}$ because its objects, their attributes, and the links between them are valued according to an algebra A of $\text{MEL}(\mathcal{MM})$ that satisfies Definition 2. Note that the requirement that our model $\mathcal{M}$ is finite is ensured by the second item of Definition 2, and the requirement that its sets if objects of classes from different inheritance hierarchies must be disjoint is ensured by the third item;
- $[\![\text{MEL}_{\mathcal{MM}}(\mathcal{M})]\!] = A$ because, again, the constants denoting objects of $\mathcal{M}$, the functions denoting attributes of objects/links between objects are valued according to A.

$\supseteq$: consider a model $\mathcal{M}$ such that $\mathcal{M} :: \mathcal{MM}$. To show that $(\!|\text{MEL}_{\mathcal{MM}}(\mathcal{M})|\!) \in [\![\text{MEL}(\mathcal{MM})]\!]$ we show $(\!|\text{MEL}_{\mathcal{MM}}(\mathcal{M})|\!)$ satisfies Definition 2. First, by construction, the specification $\text{MEL}_{\mathcal{MM}}(\mathcal{M})$ imports $\text{MEL}(\mathcal{MM})$, hence, it also imports the specifications for the basic types, and as $(\!|\text{MEL}_{\mathcal{MM}}(\mathcal{M})|\!)$ is an initial algebra, its restriction to the basic types is also initial. Second, the interpretations of sorts of $(\!|\text{MEL}_{\mathcal{MM}}(\mathcal{M})|\!)$ denoting classes of $\mathcal{MM}$ are indeed finite, namely, they consist of the finitely many constants declared in $\text{MEL}_{\mathcal{MM}}(\mathcal{M})$. Third, sorts that are in distinct connected components according to the subsorting relation are indeed interpreted by disjoint sets, since they correspond to objects of classes in disjoint inheritance hierachies. Fourth, the sets of the form $Set\{c\}$ are indeed intepreted as finite sets of elements of sort c, because $\text{MEL}_{\mathcal{MM}}(\mathcal{M})$ includes the definitions of the sorts $Set\{c\}$, hence, $(\!|\text{MEL}_{\mathcal{MM}}(\mathcal{M})|\!)$ interprets them according to their initial algebra. Finally, $(\!|\text{MEL}_{\mathcal{MM}}(\mathcal{M})|\!)$ interprets the constants $c.allInstances$ as required by Definition 2, because those constants are equated in $\text{MEL}_{\mathcal{MM}}(\mathcal{M})$ to the respective sets of all constants (of sort c). This concludes the proof. □

Lemma 2 There is a bijection between the set $\{\text{MEL}_{\mathcal{MM}}(\mathcal{M}) \mid \mathcal{M} :: \mathcal{MM}\}$ and the metamodel's semantics $[\![\text{MEL}(\mathcal{MM})]\!]$.

Proof using Lemma 1 it is enough to show that there is bijection between the sets of specifications $\{\text{MEL}_{\mathcal{MM}}(\mathcal{M}) \mid \mathcal{M} :: \mathcal{MM}\}$ and the set of their semantics $\{(\!|\text{MEL}_{\mathcal{MM}}(\mathcal{M})|\!) \mid \mathcal{M} :: \mathcal{MM}\}$. For this, consider the mapping that to each specification associates its initial algebra. It is obviously a surjection between our two sets. To prove its injectiveness, we note that different models $\mathcal{M}_1$, $\mathcal{M}_2$ conforming to $\mathcal{MM}$ have at least two distinct objects, or different values for the same attribute of an object, or different links between objects. Hence, the specifications $\text{MEL}_{\mathcal{MM}}(\mathcal{M}_1)$, $\text{MEL}_{\mathcal{MM}}(\mathcal{M}_2)$ differ either in their constant declarations or in their equation sets (or both). Since by construction there are no equations in specifications of the form $\text{MEL}_{\mathcal{MM}}(\mathcal{M})$ between the constants denoting objects, the initial algebra of $\text{MEL}_{\mathcal{MM}}(\mathcal{M})$ interprets sorts as the constants defined of the respective sorts in $\text{MEL}_{\mathcal{MM}}(\mathcal{M})$, and interprets the functions between sort interpretations as defined by the equations of $\text{MEL}_{\mathcal{MM}}(\mathcal{M})$. Hence, for different models $\mathcal{M}_1$, $\mathcal{M}_2$ of $\mathcal{MM}$, either the sort intepretations or the functions intepretations (or both) differ, hence, we obtain $(\!|\text{MEL}_{\mathcal{MM}}(\mathcal{M}_1)|\!) \neq (\!|\text{MEL}_{\mathcal{MM}}(\mathcal{M}_2)|\!)$. □

Lemma 3 *For each metamodel $\mathcal{MM}$, there is a* MEL *specification denoted by* `Models`$_{\mathcal{MM}}$*, where a sort $Models_{\mathcal{MM}}$ is defined, whose interpretation in the algebra* $(\!|$`Models`$_{\mathcal{MM}}|\!)$ *is in bijection with the set* $\{\text{MEL}_{\mathcal{MM}}(\mathcal{M}) \mid \mathcal{M} :: \mathcal{MM}\}$.

Proof We use the fact that MEL is reflective: there exists a MEL specification called `Meta-Module`, where all MEL specifications (including itself) are reflected as *terms* of a certain sort called `Module`. We then write in Maude a specification `Models`$_{\mathcal{MM}}$ extending `Meta-Module`, where we define a subsort $Models_{\mathcal{MM}}$ of `Module`, which is interpreted as the set $\{\overline{\text{MEL}_{\mathcal{MM}}(\mathcal{M})} \mid \mathcal{M} :: \mathcal{MM}\}$ in the initial algebra of the specification `Models`$_{\mathcal{MM}}$ - here, $\overline{\text{MEL}_{\mathcal{MM}}(\mathcal{M})}$ is the *term* of `Models`$_{\mathcal{MM}}$ that reflects $\text{MEL}_{\mathcal{MM}}(\mathcal{M})$. The sort $Models_{\mathcal{MM}}$ is defined using a conditional *membership*, whose condition checks that our conformance checking procedure from [5] returns `true`[6]. Finally, the injectiveness of reflection ensures that the sets $\{\text{MEL}_{\mathcal{MM}}(\mathcal{M}) \mid \mathcal{M} :: \mathcal{MM}\}$ and $\{\overline{\text{MEL}_{\mathcal{MM}}(\mathcal{M})} \mid \mathcal{M} :: \mathcal{MM}\}$ are in bijection. □

[6] The condition implicitly checks that $\mathcal{M}$ is of metamodel $\mathcal{MM}$; these checks are performed by Maude's parser and typechecker.

Lemma 4 *Assume that $\omega_1(\mathcal{M}_1^0) = \omega_2(\mathcal{M}_2^0)$ in the command ($\ddagger$). Then, for each pair of the form $\langle \mathcal{M}, \mathcal{S}\rangle$ reachable in n rewriting steps from $\langle \mathcal{M}_2^0, \mathcal{M}_1^0\rangle$, $\mathcal{M}$ is last on some execution $\rho \in exec(\mathcal{M}_2^0)$ of length n, and $\mathcal{S}$ consists exactly of the all models that are last on some execution $\pi \in exec(\mathcal{M}_1^0)$ having length* at most *n, and such that ρ is matched by π.*

Proof By induction on n. The base case $n = 0$ is trivial: the model $\mathcal{M}$ is $\mathcal{M}_1^0$, which is last on the execution $\rho = \mathcal{M}_2^0$ of length 0; and the set $\mathcal{S}$ equals $\{\mathcal{M}_1^0\}$, which indeed is the set of all models that are last on executions π of length ≤ 0 that match ρ - here, there is only one such execution: $\pi = \mathcal{M}_1^0$.

For the induction step: by induction hypothesis, $\mathcal{M}$ is last on some execution of length ρ of length n, and $\mathcal{S}$ is the set of models that are last on some execution $\pi \in exec(\mathcal{M}_1^0)$ having length at most n that matches ρ. Assume that the rewrite rule in Figure 24 is applied from $\langle \mathcal{M}, \mathcal{S}\rangle$ and produces $\langle \mathcal{M}', \mathcal{S}'\rangle$.

- since $\mathcal{M}'$ is chosen to be a successor in one step of $\mathcal{M}$ (thanks to the matching equation $\mathcal{M}', \mathcal{S}''$ `:=step2`$(\mathcal{M})$ in the rule's condition) then, using the induction hypothesis, we obtain that $\mathcal{M}'$ is indeed the last state of some execution $\rho' \in exec(\mathcal{M}_2^0)$ that has the length $n + 1$;
- to prove the induction step regarding the set $\mathcal{S}'$, there are two subcases. Rememeber that $\mathcal{S}'$ the subset of $\mathcal{S} \cup$ `step1`$(\mathcal{S})$ whose observation according to ω_1 is $\omega_2(\mathcal{M}')$:
  - if $\mathcal{S} = \emptyset$, by induction hypothesis there are no executions π of length $\leq n$ that match ρ. Then, $\mathcal{S}' = \emptyset$ and there are no executions π' of length $\leq n+1$ that match ρ', which proves the inductive step in this subcase;
  - if $\mathcal{S} \neq \emptyset$ then consider the set of executions $\pi \in exec(\mathcal{M}_1^0)$ of length at most n that match ρ; by induction hypothesis, $\mathcal{S}$ is the set of all last states of the executions in this set. Then, the set of last states of executions of length $\leq n + 1$ that match ρ' is the subset of $\mathcal{S} \cup$ `step1`$(\mathcal{S})$ whose observation according to ω_1 equals $\omega_2(\mathcal{M}')$, i.e., the set $\mathcal{S}'$, which proves the inductive step in this case and concludes the proof. □

Proposition 4 (semantical preservation) *A model transformation φ is semantics-preserving for $\mathcal{M}_1^0$ and $\mathcal{M}_2^0 \in \phi(\mathcal{M}_1^0)$ if and only if $\omega_1(\mathcal{M}_1^0) = \omega_2(\mathcal{M}_2^0)$ and the command ($\ddagger$) fails.*

Proof ($\Rightarrow$) If φ is semantics-preserving for $\mathcal{M}_1^0$ and $\mathcal{M}_2^0 \in \phi(\mathcal{M}_1^0)$ then by Definition 14, $\langle \mathcal{M}\mathcal{M}_2, \mathcal{M}_2^0, \rightarrow_2, O, \omega_2\rangle$ is observationally simulated by $\langle \mathcal{M}\mathcal{M}_1, \mathcal{M}_1^0, \rightarrow_1, O, \omega_1\rangle$. By Definition 13, for the execution $\rho = \mathcal{M}_2^0$ there exists an execution $\pi \in exec(\mathcal{M}_1^0)$ that matches ρ. Using the Definition 12 we obtain in particular $\omega_2(\rho(0)) = \omega_1(\pi(\alpha(0))) = \omega_1(\pi(0))$, hence, $\omega_1(\mathcal{M}_1^0) = \omega_2(\mathcal{M}_2^0)$, which proves the first part of the ($\Rightarrow$) implication. For the second part of the implication, we reason by contradiction: assume the command ($\ddagger$) does not fail, hence, $\langle \mathcal{M}, \emptyset\rangle$ is reachable, thus, using Lemma 4[7] there exists an execution $\rho \in exec(\mathcal{M}_2^0)$ ending in $\mathcal{M}$ with no execution π of length at most $len(\rho)$ matching ρ. Since in our simulation framework longer executions can only be matched by shorter ones, there is no execution matching ρ at all, meaning that the observational simulation of $\langle \mathcal{M}\mathcal{M}_2, \mathcal{M}_2^0, \rightarrow_2, O, \omega_2\rangle$ by $\langle \mathcal{M}\mathcal{M}_1, \mathcal{M}_1^0, \rightarrow_1, O, \omega_1\rangle$ is violated, and by Definition 14 that φ is not semantics-preserving for $\mathcal{M}_1^0$ and $\mathcal{M}_2^0$. A contradiction has been reached: the command ($\ddagger$) fails, and the ($\Rightarrow$) implication is proved.

($\Leftarrow$) Assume that $\omega_1(\mathcal{M}_1^0) = \omega_2(\mathcal{M}_2^0)$ and that the command ($\ddagger$) fails. Consider an arbitrary execution $\rho \in exec(\mathcal{M}_2^0)$ and let $\mathcal{M}$ be the last state on ρ. Then, we have a reachable term of the form $\langle \mathcal{M}, \mathcal{S}\rangle$ with $\mathcal{S} \neq \emptyset$. We can apply Lemma 4 since we are assuming its hypothesis $\omega_1(\mathcal{M}_1^0) = \omega_2(\mathcal{M}_2^0)$, and obtain that the nonempty set $\mathcal{S}$ consists exactly of the all models that are last on some execution $\pi \in exec(\mathcal{M}_1^0)$ having length *at most* n, and such that ρ is matched by π. In particular, this means that there does exist an execution $\pi \in exec(\mathcal{M}_1^0)$ that does match ρ. By Definition 13 this means that there is an observational simulation of $\langle \mathcal{M}\mathcal{M}_2, \mathcal{M}_2^0, \rightarrow_2, O, \omega_2\rangle$ by $\langle \mathcal{M}\mathcal{M}_1, \mathcal{M}_1^0, \rightarrow_1, O, \omega_1\rangle$, and by Definition 14, this means that φ is semantics-preserving for $\mathcal{M}_1^0$ and $\mathcal{M}_2^0$, which concludes the ($\Leftarrow$) implication and the proof. □

Proposition 5 *Consider two* OTS *$\langle \mathcal{M}\mathcal{M}_1, \mathcal{M}_1^0, \rightarrow_1, O, \omega_1\rangle$ and $\langle \mathcal{M}\mathcal{M}_2, \mathcal{M}_2^0, \rightarrow_2, O, \omega_2\rangle$ such that $\omega_1(\mathcal{M}_1^0) = \omega_2(\mathcal{M}_2^0)$ and an execution $\rho \in exec(\mathcal{M}_2^0)$. Consider also the rule in Figure 26 and the tree generated by the search command*

($\sharp$) `search` $\langle \rho(0), \mathcal{M}_1^0\rangle$ `=>*` $\langle \rho(len(\rho)), s\rangle$

Then, for every path in the tree, its projection on the second component belongs to the set $stuttering(\pi)$, for some execution $\pi \in exec(\mathcal{M}_1^0)$ that matches the execution ρ. Reciprocally, for every execution $\pi \in exec(\mathcal{M}_1^0)$ that matches the execution ρ, there exists a path in the tree whose projection on the second component belongs to the set $stuttering(\pi)$.

Proof ($\Rightarrow$) By induction on the length of the path. If the length is 0 then $\mathcal{M}_1^0 \in stuttering(\mathcal{M}_1^0)$ is in $exec(\mathcal{M}_1^0)$ and matches $\rho = \mathcal{M}_2^0$, since we assumed $\omega_1(\mathcal{M}_1^0) = \omega_2(\mathcal{M}_2^0)$.

For the induction step, assume the statement holds for paths of length $n \leq len(\rho)-1$: $\langle \mathcal{M}_2^0, \mathcal{M}_1^0\rangle \cdots \langle \rho(n), s\rangle$. This means that the sequence $\hat{\pi} = \mathcal{M}_1^0 \cdots s$ is in $stuttering(\pi)$, for some $\pi \in exec(\mathcal{M}_1^0)$ that matches ρ. Now, a path of length $n + 1$ is obtained by applying the rule in Figure 26 to the term $\langle \rho(n), s\rangle$, resulting in a term $\langle \rho(n + 1), s'\rangle$ where s' is such that $\omega_1(s') = \omega_2(\rho(n + 1))$, and either $s' = s$ or $s \rightarrow_1 s'$. In both cases, $\mathcal{M}_1^0 \cdots ss'$ is in $stuttering(\pi)$ and matches $\rho[0..n + 1]$, which concludes the ($\Rightarrow$) implication.

($\Leftarrow$) By induction on the length of π. The base case where the length is 0 is trivial: the corresponding path is $\langle \rho(0), \mathcal{M}_1^0\rangle$.

For the induction step, its hypothesis says that there exists a path $\langle \mathcal{M}_2^0, \mathcal{M}_1^0\rangle \cdots \langle \rho(n), s\rangle$ such that the sequence $\hat{\pi} = \mathcal{M}_1^0 \cdots s$ is in $stuttering(\pi)$. Consider an execution π' that matches $\rho[0..n + 1]$. Then, using Definition 12, we have $\omega_1(\pi'(len(\pi'))) = \omega_2(\rho(n + 1))$, and either $\pi' = \pi$, or $\pi' = \pi s'$ such that $\pi(len(\pi)) \rightarrow_1 s'$. In both cases, the condition of the rule in Figure 26 is satisfied for $s = \pi(len(\pi))$, and s' can be chosen to be $\pi(len(\pi))$ in the rule's application, which generates the term $\langle \rho(n + 1), s'\rangle$. The path $\langle \mathcal{M}_2^0, \mathcal{M}_1^0\rangle \cdots \langle \rho(n), s\rangle\langle \rho(n+1), s'\rangle$ satisfies the conclusion for the induction step. This concludes the ($\Leftarrow$) implication.□

[7] Note that we can indeed apply Lemma 4 here, as we have just proved its hypothesis $\omega_1(\mathcal{M}_1^0) = \omega_2(\mathcal{M}_2^0)$.