A K-Based Formal Framework for Domain-Specific Modelling Languages

Vlad Rusu, Dorel Lucanu

A $\mathbb{K}$-Based Formal Framework for Domain-Specific Modelling Languages

Vlad Rusu[1] and Dorel Lucanu[2]

[1] Inria Lille, France, Vlad.Rusu@inria.fr
[2] University of Iasi, Romania, dlucanu@info.uauic.ro

Abstract. We propose a formal approach for the definition of domain-specific modelling languages (DSMLs). The approach uses basic Model-Driven Engineering artifacts for defining a DSML's syntax (using metamodels) and its operational semantics (using model transformations). We give formal meanings to these artifacts by mapping them to the $\mathbb{K}$ semantic framework. The mapping is implemented in the Rascal metaprogramming language. Since the resulting $\mathbb{K}$ definitions are executable, one obtains an execution engine for DSMLs and gains acces to $\mathbb{K}$'s formal analysis tools. We illustrate the approach on xSPEM, a language for describing the execution of tasks constrained by time, precedence, and resources.

1 Introduction

Domain-Specific Modelling Languages (DSMLs) are languages dedicated to modelling in specific application areas. Recently, the design of DSMLs has become widely accessible to engineers trained in the basics of Model-Driven Engineering (MDE): one designs a *metamodel* for the language's abstract syntax; then, the language's operational semantics is expressed using *model transformations* over the metamodel. The democratisation of DSML design catalysed by MDE is likely to give birth to numerous languages, and one can also reasonably expect that there shall be numerous errors in those languages. Indeed, getting a language right (especially its operational semantics) is hard, regardless of whether the language is defined in the modern MDE framework or in more traditional ones.

Formal methods can help detect or avoid errors in DSML definitions. However, the history of formal methods offers many examples of valorous methods that could not be transferred outside a circle of specialised users, because software engineers do not have the time or the background required for learning them. The lesson learned from these failures is that, in order to be accepted by software engineers, formal approaches have to operate with notions familiar to them.

We propose here such an approach, which formalises the basic MDE ingredients used in DSML definitions. From the point of vue a user, the approach is a black box (Figure 1): users can define their DSMLs using familiar MDE ingredients (metamodels for syntax, OCL [1] constraints for static semantics, model transformations for operational semantics). These inputs are parsed and processed by a Rascal [2] program (that we wrote) and are mapped to $\mathbb{K}$ [3] code, together

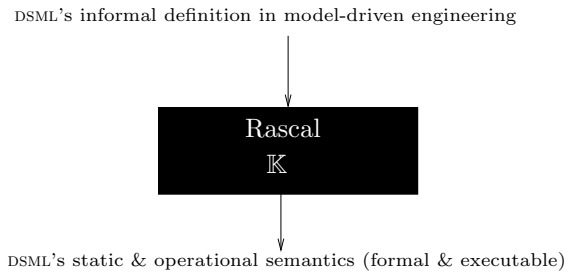


Fig. 1. Our approach, from the point of vue of a user defining a DSML.

with additional 𝕂 code (that we also wrote). In this way, users benefit from 𝕂's execution engine and formal analysis tools for free - without having to write code unfamiliar to them or unrelated to their task - which allows them to experiment with and to perform formal analyses on their languages in a transparent way.

We illustrate the approach on xSPEM [4], a DSML based on the OMG standard [5] for describing the execution of activities constrained by time, resources, and precedence relations. We show how users can automatically: check model-to-metamodel conformance (including OCL constraints); execute a DSML's semantics; and model check for reachability properties over the DSML's executions.

Contributions Our main contributions is providing a formal semantics to to the MDE notions employed in DSML definitions, using the 𝕂 semantical framework:

- metamodels, together with well-formedness OCL constraints, and models;
- model transformations for operational semantics. We have designed for this a language called KMRL (𝕂-based Model-Rewrite Language) by building on basic MDE notions. KMRL is composed of model-rewrite rules, where each rule consists of a model pattern similar to MDE-models, an optional condition written in OCL, and an optional piece of imperative code also based on OCL. KMRL should therefore look and feel familiar to our target users: software engineers familiar with such basic MDE notions as model and OCL constraints.

Note that our contribution is *not* an approach for defining DSMLs directly in 𝕂. We do not advocate this, since 𝕂 is unlikely to be accepted by software engineers, and we have stated that the main motivation for this work is to gain their acceptance (and, ultimately, to further the cause of formal methods in practice).

Organisation Section 2 provides preliminaries: it describes the 𝕂 semantical framework and the Rascal metaprogramming language, and illustrates the MDE-based definition of the xSPEM DSML. In Section 3 we present our approach and illustrate it on xSPEM. Section 4 concludes and presents related and future work.

2 Preliminaries

2.1 The $\mathbb{K}$ Framework and the Rascal Metaprogramming Language

$\mathbb{K}$ [3] is a framework mainly intended for defining and analysing semantics of programming languages[3]. The main features of $\mathbb{K}$ include:

- *executability*: the definitions are directly executable in order to be experimented with and analysed;
- *unique definition*: there is only one definition for a language, and several analysis tools that are sound with respect to this definition;
- *program logic*: the framework serve as a program logic with which the programs can be verified and analysed (see, e.g., [16]).

A $\mathbb{K}$ definition has three main ingredients: a *configuration*, which is a structure of nested cells abstracting the state structure of the machine on which the programs are executed; *computations*, which are sequences of tasks derived from the annotated syntax; and $\mathbb{K}$ *rules*, which describe the computation steps using minimal information (only what is needed for matching and rewriting). These concepts will be illustrated in Section 3, where we show how the DSML definitions can be formalized in $\mathbb{K}$. Then, the $\mathbb{K}$ execution engine can be used for executing the definition, and its model checker, for checking reachability properties.

Rascal [2] is a recent metaprogramming language for source code analysis and transformation. We use it to implement the mapping of the MDE concepts used in DSML definitions (metamodels, OCL, models, model transformations) to $\mathbb{K}$.

2.2 Defining a DSML using MDE: xSPEM

We illustrate our approach on a DSML called xSPEM [4], which is an executable version of the SPEM language standard [5]. The language describes the execution of *activities* constrained by time, resources, and precedence relations.

We first describe the syntax and static semantics of (a simplified version of) xSPEM by showing its metamodel as well as a sample model. Then we describe the language's operational semantics using a mixture of graphical and textual notations. These notations will become formal when we represent them in $\mathbb{K}$.

In the metamodel of Figure 2 (top), *activity* is the class of entities being executed. The *tmin* and *tmax* attributes of the *activity* class denote the minimum and maximum expected duration of activities. The *aS* attribute takes its values in the *activityState* enumeration: *notStarted*, *inProgress*, or *finished*; and the *tS* attribute takes its values in the *timeState* eumeration: *undefined*, *tooEarly*, *ok*, or *tooLate*. An activity may also have *resources*, which are reserved by the activity (and become unavailable to others) while the activity is running. In addition to the availability of resources (*resource* class) the execution of activities is also governed by explicit ordering constraints (*workSequence* class). Each activity also has exactly one *workSequence* instance, as indicated by the OCL invariant associated to the metamodel. The *workSequence* class has references to four possibly empty sets of activities, namely, the activities that must be

[3] e.g., a definition of C is available at `http://code.google.com/p/c-semantics/`.

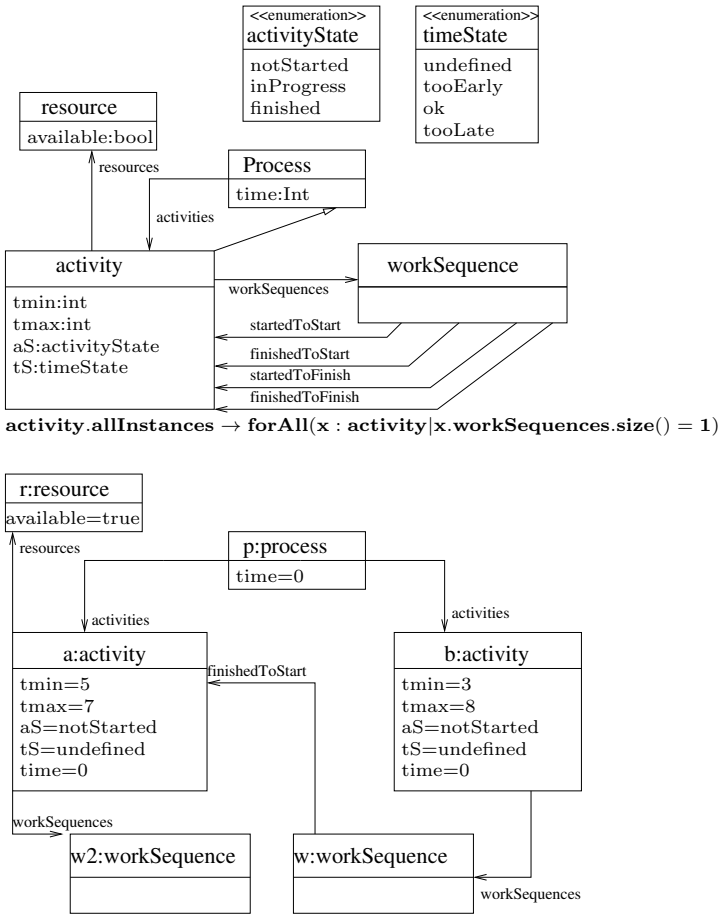


Fig. 2. (top): A simplified metamodel for xSPEM, together with an OCL constraint (bottom) one xSPEM model that conforms to the xSPEM metamodel.

- started for the current activity to start (*startedToStart* reference);
- finished for the current activity to start (*finishedToStart* reference);
- started for the current activity to finish (*startedToFinish* reference);
- finished for the current activity to finish (*finishedToFinish* reference).

Time is measured by a clock, encoded by the *time* attribute of the *process* class. Activities inherit from processes (inheritance is denoted in class diagram by an open arrrow). Hence, activities also have the *time* and *activities* features.

In the model depicted at the bottom Figure 2, the activities a and b are linked by a *workSequence* via the reference *finishedToStart*, meaning that b is allowed to start only when a is finished; and a has the (available) resource r.

We now give the operational semantics of xSPEM using a semi-formal notation mixing graphical and textual rules. The notation is isomorphic to the (textual) KMRL language that we shall formalise in Section 3.5. The first rule is shown in

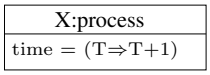


Fig. 3. Time-passing rule.

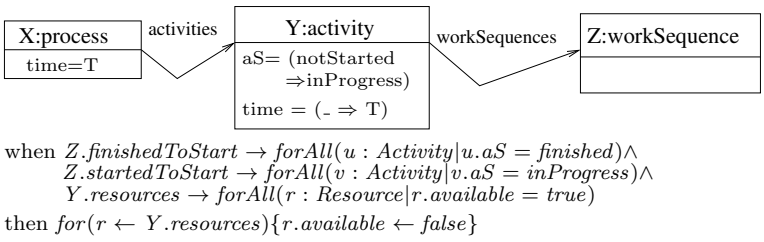


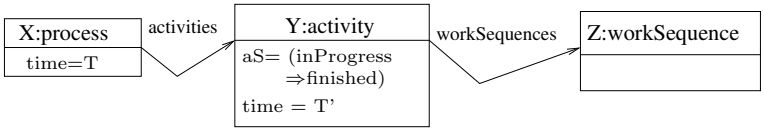
Fig. 4. Starting an activity.

Figure 3. It expresses the fact that processes "make time pass": the *time* attribute is increased by one. This is expressed by a local rewrite rule (a concept inspired from $\mathbb{K}$): $time = (T \Rightarrow T + 1)$, within a process X. Here, X and T are variables (of type process, respectively, integer) to be matched with correspondingly typed constants in a model; when this happens, the model is rewritten, just like in usual rewrite systems. For example, the execution of this rule on the model shown at the bottom of Figure 2 would produce the same model except that $p.time = 1$.

The next rule (Figure 4) expresses the starting of an activity. When an activity Y is started, the value of its aS attribute is rewritten from *notStarted* to *inProgress*, and the activity Y memorises in its *time* attribute the value of the homonymous attribute of the process, say, X, which owns the activity. This is expressed by the local rewrite rule $time = (\_ \Rightarrow T)$, which says that the *time* attribute of X is rewritten, from whatever value it had, to T; where "whatever" is denoted by an underscore, and T equals the value of $X.time$. Moreover, the activity Y may only be started if the (optional) *when* condition holds; and, finally, some additional imperative code, in the (optional) *then* clause, is executed.

Here, the *when* clause says that the activities that have to be started (resp. finished) for the current activity Y to start are indeed in the expected states, and that all the resources of Y are available. The additional code in the *then* clause is here an imperative *for* loop, which is in charge of assigning all the activity's resources *available* attributes to *false*. In general, the additional code can include assignments, loops, and conditionals, and can declare local variables for storing intermediate values. The practical utility of the imperative code can be illustrated on the current example: when an activity is started, it needs to make all its resources un-available to other activities; but this cannot be expressed in a graphical rewrite pattern, because an activity may have any number of resources; whereas a graphical pattern "draws" a fixed number of model elements.

The semantics of xSPEM includes one other rule in addition to the two ones shown above. The third rule is in charge of finishing activities. It is shown in Figure 5. The main difference with the rule for starting an activity lies in the more complex imperative code, which is here used for updating the tS attribute



when $Z.finishedToFinish \rightarrow forAll(u : Activity | u.aS = finished) \wedge$
$Z.startedToFinish \rightarrow forAll(v : Activity | v.aS = inProgress)$
then $var\ x;\ x \leftarrow T - T';$
$if(x < Y.tmin)\ Y.tS \leftarrow tooEarly$
$else\ if\ (x < Y.tmax)\ Y.tS \leftarrow ok$
$else\ Y.tS \leftarrow tooLate$
$endif$
$endif;$
$for(r \leftarrow Y.resources)\{r.available \leftarrow true\}$

Fig. 5. Finishing an activity.

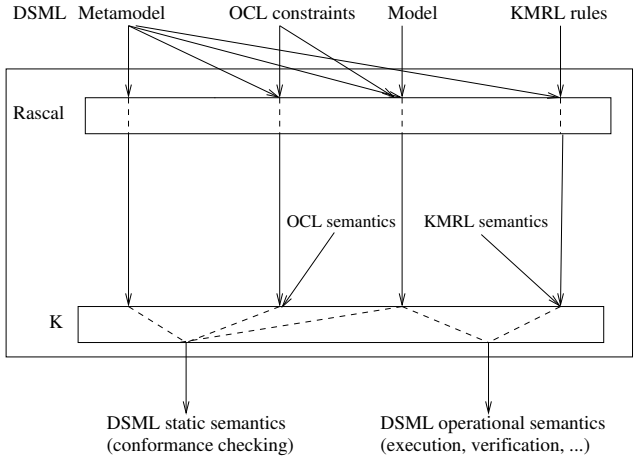


Fig. 6. Dataflow diagram of our framework.

of the activity Y being terminated, to *tooEarly*, *ok*, or *tooLate*, depending on whether its execution time is in [0,*tmin*), [*tmin*,*tmax*) or [*tmax*,∞), respectively. This is achieved using two nested *if-then-else-endif* conditionals. In order to avoid recomputing the execution time $T - T'$, we store it in the local variable x.

Hence, we have a flexible and expressive declarative/imperative language for describing operational semantics of DSMLs. In order to make it formal and executable we map it (together with languages for expressing metamodels and models for DSMLs) to the $\mathbb{K}$ framework, which we now briefly introduce.

3 A $\mathbb{K}$-Based Formal Framework for DSMLs

3.1 A General Overview

In this section we show how MDE concepts used in defining DSMLscan be mapped to $\mathbb{K}$. The mapping is implemented in the Rascal metaprogramming language [2].

A dataflow diagram of our approach is shown in Figure 6. It consists of:

```
metamodel xSPEM{
  enumeration activityState{notStarted; inProgress; finished}
  enumeration timeState{undef; tooEarly; ok; tooLate}
  class process{
    attribute time : int;
    attribute tS : timeState;
    reference activities : activity;
  }
  class activity extends {process}{
    attribute tmin : int;
    attribute tmax : int;
    attribute aS : activityState;
    reference workSequences : workSequence [1-1];
    reference resources : resource;
  }
  class workSequence{
    reference startedToStart : activity;
    reference startedToFinish : activity;
    reference finishedToStart : activity;
    reference finishedToFinish : activity;
  }
  class resource{
    attribute available : bool;
  }
}
```

Fig. 7. Textual representation of the xSPEM metamodel from Figure 2 (top).

- a Rascal program, which takes as input the ingredients of a DSML definition (metamodel, OCL constraints, models, KMRL rules) in a certain textual format, and produces intermediate representations suitable for $\mathbb{K}$;
- a $\mathbb{K}$ program, which takes the output produced by Rascal, together with $\mathbb{K}$ additional code defining the semantics of OCL and KMRL, and generates executable static semantics and operational semantics for the DSML.

Note that from the point of view of DSML designers, the outmost box is a black box: they do not need to know what is inside, but only need to provide the MDE artifacts for the definition of their DSML in a textual format; then, the static and operational semantics of the DSML is automatically generated for them.

We now describe the framework in detail by "scanning" the diagram in Figure 6 from left to right, and illustrate it on the xSPEM language from Section 2.2.

3.2 Metamodels and OCL constraints

We show in Figure 7 the textual syntax for the xSPEM metamodel from Figure 2. In order to generate input for $\mathbb{K}$, the metamodel in textual syntax is processed by a Rascal program. This includes parsing and some syntactical transformations, such as the replacement of multiplicity constraints and bidirectionality constraints by equivalent OCL invariants. For example, the `[1-1]` multiplicity of

the `workSequences` reference is replaced by the equivalent OCL invariant

```
allInstances(activity)→forAll(x:activity|x.workSequences.size() = 1).
```

The metamodel is thus stripped of its multiplicity and bidirectionality constraints and the result is translated to a set of $\mathbb{K}$ function declarations together with rules to evaluate them. The functions encode all the (stripped) metamodel's information: for each enumeration, its set of values, and for each class, its children classes, its attributes with their types, and its references with their types. They are used for checking the syntactical correctness of models with respect to the given metamodel. This is discussed in more detail in Section 3.3.

Well-formedness OCL constraints In addition to the implicit constraints induced by multiplicities and bidirectionality of references, a metamodel may include other OCL constraints, which enforce well-formedness requirements that models conforming to the metamodel must satisfy. For example, we shall require that in any well-formed xSPEM model there is exactly one "proper" process:

```
(allInstances(process) \ allInstances(activity)).size() = 1.
```

3.3 Models

In DSML terminology, models can be seen as "DSML programs", by analogy with the programs of usual programming languages. In this section we show the syntax of models required as input by our Rascal module in charge of processing models, and the representation of the models as $\mathbb{K}$ configurations generated by the module in question. In the next section we outline of the $\mathbb{K}$ semantics for OCL, and show how model-to-metamodel conformance checking is done in $\mathbb{K}$.

We also use textual language for the model description. The essential information about a model consists of the name of the metamodel it must conform to and a collection of objects (class instances). Each object is described by its name, the class it belongs to, and the values of its attributes and references.

In Figure 8 we give the equivalent textual syntax for a fragment of the xSPEM model example given in Figure 2. Using this input together with the information obtained from a metamodel and OCL constraints, a Rascal module generates a $\mathbb{K}$ configuration described as follows.

$\mathbb{K}$ Configuration for DSMLs $\mathbb{K}$ configurations are structures consisting of nested cells. The generic $\mathbb{K}$ configuration for DSMLs is graphically represented in Figure 9. The configuration includes: a cell $\langle \_ \rangle_{\mathsf{model}}$ for models (described later in this section); a cell $\langle \_ \rangle_{\mathsf{oclConstraint}}$ containing OCL constraints to be checked on the model; a cell $\langle \_ \rangle_{\mathsf{k}}$ containing computation tasks to be performed on the model (conformance checking, model execution, model checking, ...); and a cell $\langle \_ \rangle_{\mathsf{result}}$ for storing the results of the computation tasks. Initially, the cells are empty (as denoted by periods and dashes within them, depending on their type). They are filled by Rascal modules: the $\langle \_ \rangle_{\mathsf{model}}$ cell is filled by the module in charge of models, and cell $\langle \_ \rangle_{\mathsf{oclConstraint}}$ cell is filled by the module in charge of OCL constraints of the metamodel to which the model is supposed to conform.

```
onexSPEM = new xSPEM {
  p = new process {
    time = 0;
    activities = {a b};
  }
  a = new activity {
    tmin = 5;  tmax = 7;
    aS = notStarted; tS = undef;
    resources = {r}; time = 0;
    linkToPredecessor = {w2}; activities = {};
  }
  b = new activity {... // similar to activity a
  }
  r = new resource {
    available = true;
  }
  w1 = new workSequence {
    startedToStart = {}; startedToFinish = {};
    finishedToStart = {a};  finishedToFinish = {};
  }
  w2 = new workSequence { ... // similar to workSequence w1
  }
}
```

Fig. 8. Textual representation of the model from Figure 2.

The structure of the cell $\langle \_ \rangle_{\mathsf{model}}$ is similar to that of the textual language we use for model description. It consists of a set of $\langle \_ \rangle_{\mathsf{instance}}$ cells, each of which contains the instance's name, its class, and its attribute/reference values.

3.4 $\mathbb{K}$ Semantics of OCL

We have defined in $\mathbb{K}$ a substantial fragment of OCL based on the standard [1]. Due to limited space, a complete description will be given in a separate paper.

The elementary types in our definition of OCL are integer, string, and Boolean with the usual elementary operations on them. We also allows for collection types, built using navigation through attributes/references, iterators (`select`, `collect`), quantifiers (`forAll`, `exists`), as well as the usual set operations. The allInstances() query returns all the instances of a given class. This provides us with a rich language for constraints, which we may enrich in the future.

Here is, for instance, the $\mathbb{K}$ rule giving semantics to the query allInstances():

$$\frac{\langle\ \texttt{allInstances(Cls)}\ \cdots\rangle_{\mathsf{k}}}{\texttt{val(collectAllInstanceNames(Cls , children(Cls) , M))}}$$

$\langle \texttt{M} \rangle_{\mathsf{model}}$

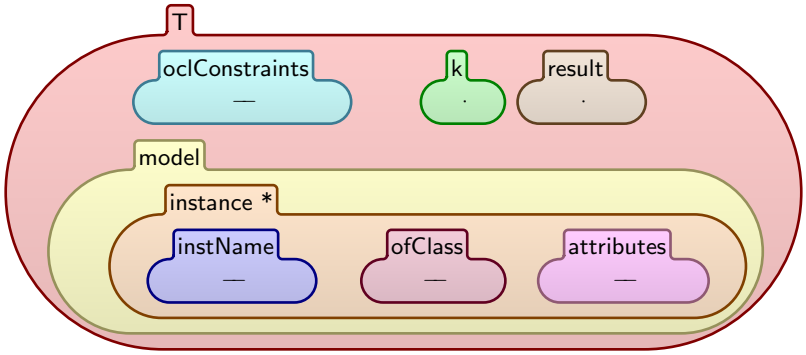


Fig. 9. $\mathbb{K}$ Configuration for DSMLs

The rule says that, in order to compute all the instances of a given class `Cls` in a model `M`, a helper function `collectAllInstanceNames()` must be called, with three parameters: the class `Cls`, its children `children(Cls)`, and the model `M`.

The fraction line denotes a local rewrite, here, at the the head of the computation cell $\langle \_ \rangle_{\mathsf{k}}$. The numerator is what is usually written in the left-hand side of rewrite rules, and the denominator is the equivalent of the right-hand side. A $\mathbb{K}$ rule may have several local rewrites in various configuration cells, and it can use other cells just for providing context of the rewrite. For example, in the above rule, the cell $\langle \_ \rangle_{\mathsf{model}}$ provides the model `M`. The operator `children()` is a part of the $\mathbb{K}$ description of the metamodel, and the function `collectAllInstanceNames()` traverses the $\langle \_ \rangle_{\mathsf{instance}}$ cells of `M` and collects their names. Finally, the `val` operator wraps the result in order for $\mathbb{K}$ to interpret it as a fully eveluated OCL value (in our case, a collection of instance names).

As another axample, we give the $\mathbb{K}$ semantical rules for the `forAll` operation in order to illustrate some interesting features of $\mathbb{K}$. Syntactically, the operation is written `Col->forAll(Id | Exp)`, and its meaning is that it is *true* if and only if the third argument `Exp` evaluates to *true* on each element (denoted by the second argument `Id`) of the first argument `Col`. Its $\mathbb{K}$ semantical rules are

$$\langle \underbrace{\texttt{val(}\cdot\texttt{)->forAll(Var | Exp)}}_{\texttt{true}} \cdots \rangle_{\mathsf{k}} \tag{1}$$

$$\langle \frac{\texttt{val(Hd, Tl)->forAll(Var | Exp)}}{\texttt{if Exp(Hd/Var) then val(Tl)->forAll(Var|Exp) else false}} \cdots \rangle_{\mathsf{k}} \tag{2}$$

The first rule describes the base case, when the first argument is an empty collection. The second rule describes the inductive step: when the first argument is a nonempty collection of the form `val(Hd, Tl)`, the result depends on the value of the expression `Exp` on the element `Hd`. This value is computed by applying the $\mathbb{K}$ visitor pattern [3] to perform substitution of `Var` by `Hd` in `Exp`. If the

value is *true*, then the overall result is that of the `forAll` operation, recursively evaluated on the smaller collection `val(Tl)`; otherwise, the overall result is *false*.

This relatively simple definition is possible due to a powerful mechanism of $\mathbb{K}$, which automatically generates rewrite rules in order to evaluate arguments of operators declared to be "strict". The actual $\mathbb{K}$ grammar for `forAll` is

```
Exp ::= Exp ->forAll( Id  | Exp ) [strict (1)]
```

This means that the `forAll` operator is "strict" in its first argument; that is, this argument will be evaluated before the `forAll` expression is evaluated.

In order to do this, $\mathbb{K}$ automatically generates "heating rewrites" of the form

$$E_1 \texttt{->forAll(}\ V \mid E_2\ \texttt{)} \longrightarrow E_1 \curvearrowright \square \texttt{->forAll(}\ V \mid E_2\ \texttt{)}$$

by which the first computation task becomes the evaluation of the first argument E_1; this can generate further computation tasks using the same mechanism, depending on the structure of E_1 (this is the case with the `if-then-else` expression in Rule 2). When E_1 is evaluated, "cooling rewrites" of the form:

$$\texttt{val(}\ L\ \texttt{)} \curvearrowright \square \texttt{->forAll(}\ V \mid E_2\ \texttt{)} \longrightarrow \texttt{val(}\ L\ \texttt{)->forAll(}\ V \mid E_2\ \texttt{)}$$

fill the "hole" left by E_1. Then, eventually, one of the rules (1–2) finishes the evaluation of the `forAll` operator.

Conformance Checking A model is well-formed if and only if it conforms to its metamodel, i.e., it is syntactically correct and satisfies the OCL constraints of the metamodel. Specifically, for a DSML, the class diagram of its metamodel defines its syntax, and the OCL constraints define its static semantics. The procedure for verifying that a model is well-formed is referred to as *conformance checking*.

In the rest of this section we briefly describe conformance checking in $\mathbb{K}$. Syntactical correctness is checked using the $\mathbb{K}$ functions specifying the metamodel. The OCL constraints are checked by first "loading" the content of the cell $\langle \_ \rangle_{\mathsf{oclConstr}}$ into the computation cell $\langle \_ \rangle_{\mathsf{k}}$. This is triggered by the following rule:

$$\underbrace{\langle \texttt{checkConformance} \rangle_{\mathsf{k}}}_{e}\ \langle e \rangle_{\mathsf{oclConstr}}$$

Then, the execution of the OCL semantics generates a computation of the form:

$$\begin{array}{l} \langle \langle e \rangle_{\mathsf{k}}\ \langle e \rangle_{\mathsf{oclConstr}}\ \langle \cdot \rangle_{\mathsf{mem}}\ \langle \cdot \rangle_{\mathsf{result}}\ \langle M \rangle_{\mathsf{model}}\ \cdots\ \rangle_{\mathsf{T}} \xrightarrow{*} \\ \langle \langle \cdot \rangle_{\mathsf{k}}\ \langle e \rangle_{\mathsf{oclConstr}}\ \langle \cdot \rangle_{\mathsf{mem}}\ \langle v \rangle_{\mathsf{result}}\ \langle M \rangle_{\mathsf{model}}\ \cdots\ \rangle_{\mathsf{T}} \end{array}$$

in which the result v of the OCL constraints e is placed in the $\langle \_ \rangle_{\mathsf{result}}$ cell.

Finally, a model M satisfies the OCL constraints e if and only if

$$\begin{array}{l} \langle \langle \texttt{checkConformance} \rangle_{\mathsf{k}}\ \langle e \rangle_{\mathsf{oclConstr}}\ \langle \cdot \rangle_{\mathsf{result}}\ \langle M \rangle_{\mathsf{model}}\ \cdots\ \rangle_{\mathsf{T}} \xrightarrow{*} \\ \langle \langle \cdot \rangle_{\mathsf{k}}\ \langle e \rangle_{\mathsf{oclConstr}}\ \langle \cdot \rangle_{\mathsf{mem}}\ \langle \mathit{true} \rangle_{\mathsf{result}}\ \langle M \rangle_{\mathsf{model}}\ \cdots\ \rangle_{\mathsf{T}} \end{array}$$

This provides us with a formal, executable definition for conformance checking.

```
rule start
forAll P Y Z timeVal
pattern {
  process P = {
    time = timeVal;
    activities ={Y _};
  }
  activity Y =  {
    aS = (notStarted => inProgress);
    time  = (0 => timeVal);
    linkToPredecessor  = {Z};
  }
}
when {
  Z.startedToStart->forAll(act| act.aS == inProgress);
  Z.finishedToStart->forAll(act| act.aS == finished);
  Y.resources->forAll(r| r.available == true)
}
then {
 var r;
 for (r <-Y.resources){(r.available) <- false} ;
 print("start"); print(Y)
}
```

Fig. 10. Textual version of the *start* from Figure 4.

3.5 Operational Semantics

The last ingredient in the definition of a DSML is the definition of its operational semantics. We propose for this the language KMRL ($\mathbb{K}$ Model-Rewrite Language), a mixed declarative/imperative language for model rewriting. A glimpse of KMRL has already been shown in Section 2.2, where we informally gave the semantics of xSPEM using (graphical) model-rewrite rules. To formalise the semantics of DSMLs, we need first to formalise the KMRL language itself. We provide it with a textual syntax, checked by a Rascal-generated parser, and with a formal semantics as a set of $\mathbb{K}$ rules. Those rules include the rules for the semantics of OCL since OCL is a sublanguage of KMRL.

We now illustrate the overall process with the rule *start* shown in Figure 4. The textual version of the rule is given in Figure 10. It is composed of four parts:

- global variable declaration (`forAll` keyword). Here, the notion of global variable[4] should be understood as variable in the rewrite-systems terminology: variables match terms, and are replaced by those terms when rewriting is performed; e.g., on our example, the variable `Y` matches activities. The scope of global variables is the whole rule; they can occur everywhere in the rule.

[4] We shall call global variables just "variables" when no confusion with local variables, to be introduced later, can occur. We shall do the same for local variables.

– rewrite pattern (`pattern` keyword). Patterns are very much like models, discussed in the previous section. The main difference is that rewrite patterns need not be completely specified models; informally speaking, they may match any model that is a "superset" of the pattern. Also, unlike models, rewrite patterns may refer to variables (those declared in the global variable declarations) in addition to constants; and their attributes and references can be local rewrite rules[5]. For example, the `time` attribute of the activity `Y` is rewritten from `0` to `timeVal`, where the latter is a variable, which was chosen to be the value of the `time` attribute of the process `P` of our pattern. This means that when the rule is applied, the `time` attribute of the activity `Y` gets assigned the value of the `time` attribute of the process `P`. Note also the value of the `activities` reference of `P`: the meaning of {`Y_`} is a set containing at least the value `Y` and possibly more. This is in contrast to, e.g., the set {`Z`}, which means the set containing exactly the value(s) as given.
– OCL condition (`when` keyword). This is a condition in the sense of conditional rewrite systems: a rule is applied only when its condition holds after its (global) variables (here, `Y` and `Z`) are substituted with matched terms.
– imperative code (`then` keyword). This is essentially a program composed of assignments, loops, and conditionals. The program starts with the declaration of a list of local variables, distinct from the global variables, which play the roles of usual variables in imperative programs: essentially, they serve for storage of intermediate computed results and as iterators of loops. In our example, the variable `r` serves as an iterator for the `for` loop. Regarding assignments, their left-hand side are OCL navigation expressions, i.e., expressions of the form $Variable.reference_1.\cdots.reference_n$, where *Variable* can be a global or a (previously evaluated) local variable (the latter case appears within the `for` loop of our example). Their right-hand sides can be arbitrary OCL expressions, including local variables. Finally, note the `print` statements: their arguments are arbitrary OCL expressions, and they are used to print output - useful for executing, debugging, or verifying KMRL code.

Parsing and processing KMRL with Rascal. The KMRL input defining the operational semantics of a DSML, together with the corresponding metamodel for the DSML's syntax, is processed by a Rascal program in three main steps.

The first step is to compute the types of the (global) variables and constants occurring in the rule: in our example, Rascal infers from the metamodel that `timeVal` is an integer, and that `inProgress` is an `activityState`. The second step is a form of "context transformation", also inspired from $\mathbb{K}$ but considerably simpler: since the rewriting pattern and its components are typically incomplete, we complete them with adequately typed variables; for example, the incomplete set {`Y,_`} is completed to {`Y, rest`}, where `rest` is a fresh variables that can match any number of set elements. And, finally, the third processing step consists in separating the rule's rewriting pattern into a left-hand and a right-hand side, which has indeed the effect of turning the pattern into a proper rewrite rule.

[5] This idea is borrowed from $\mathbb{K}$. The advantage is that produces simple/compact rules.

(Essentially, the right-hand side of a rewrite pattern is a copy of the pattern in which the left-hand sides of the local rewrite rules occurring in it are kept; and symetrically so for the right-hand sides. Those parts of the patttern that are not local rewrite rules appear in both sides, and serve as a rewriting context.)

Let r be a KMRL rule. We denote by `lhs(r)` and `rhs(r)` its left-hand and right-hand sides computed by Rascal (as described above), by `C(r)` its condition, and by `imp(r)` its imperative part. These are translated by Rascal to a $\mathbb{K}$ format which we do not present here since it is not essential for understanding. What is important is the the overall translation of the rule r, denoted by $\mathbb{K}(r)$:

$$\langle \underset{\texttt{when(C(r))}\curvearrowright\texttt{update(rhs(r))}\curvearrowright\texttt{apply(imp(r))}\curvearrowright\texttt{run}}{\texttt{run}} \cdots\rangle_{\mathsf{k}} \quad \langle \texttt{lhs(r)} \rangle_{\mathsf{model}}$$

This means that whenever the keyword *run* (which is by convention the instruction for model execution) is at the top of the $\langle \_ \rangle_{\mathsf{k}}$ cell, and the $\langle \_ \rangle_{\mathsf{model}}$ cell matches `lhs(r)`, the *run* keyword is rewritten into a sequence that evaluates the condition `C(r)`, and, if the condition holds, then it updates the $\langle \_ \rangle_{\mathsf{model}}$ cell by replacing its contents with `rhs(r)`, it applies the imperative program `imp(r)` to the result, and, finally, it recursively invokes model execution by reinserting `run` in the $\langle \_ \rangle_{\mathsf{k}}$ cell.

This effect is obtained by the $\mathbb{K}$ semantics of KMRL we now briefly describe.

$\mathbb{K}$ semantics of KMRL First, we give the rules for the `when` clause of $\mathbb{K}(r)$.

$$\langle \underset{\cdot}{\texttt{when(true)}} \cdots\rangle_{\mathsf{k}} \qquad\qquad \langle \underset{\texttt{stop}}{\texttt{when(false)}} \cdots\rangle_{\mathsf{k}}$$

That is, the `when` clause disappears (i.e., rewrites to $\cdot$) when its argument evaluates to `true`, and rewrites to the irreductible `stop` term otherwise. This simplicity is due to the fact that the `when` operation is strict: its argument (an OCL expression) is a Boolean when the `when` clause itself is evaluated. The evaluation of the argument consisted in applying the $\mathbb{K}$ rules for the semantics of OCL.

Then comes the rule for the `update` instruction. It simply consists in removing the `update` keyword from the top of the $\langle \_ \rangle_{\mathsf{k}}$ cell, and by replacing the content of the $\langle \_ \rangle_{\mathsf{model}}$ cell by the argument of the (just removed) `update` instruction:

$$\langle \underset{\cdot}{\texttt{update(M)}} \cdots\rangle_{\mathsf{k}} \ \langle \underset{\texttt{M}}{\_} \rangle_{\mathsf{model}}$$

Finally, there are rules for applying the imperative part `imp(r)` of $\mathbb{K}(r)$. We do not give these rules due to lack of space, but note that, except for assignment (which is quite specific to our present model-based framework) they are standard semantical rules for imperative programs constructs (loops and conditionals).

```
rule observer
  pattern  {
    process P;
  }
  when {
    P.activities->forAll(a | a.aS == finished and a.tS = ok)
  }
  then {
    print("reached")
 }
```

Fig. 11. Observer rule for model checking.

3.6 Execution and Model Checking

The operational semantics that we provide for DSML's semantics is executable, hence, it can directly be used for execution, and, with some user input, for model checking of reachability properties. Execution is here the nondeterministic execution of the ($\mathbb{K}$ semantics of) KMRL rules using the $\mathbb{K}$ execution engine.

We now illustrate model checking on xSPEM, whose metamodel and operational semantics rules were shown in Section 2.2. We consider the folllowing reachability problem: *is it possible, from the model shown at the bottom of Figure 2, to reach a model where all activities of the model's proper process*[6] *are finished within their expected time limits*? A model where all activities of a process P are finished within their expected time limits is characterised by the following OCL expression: $\texttt{P.activities} \rightarrow \texttt{forAll}(\texttt{a} \mid \texttt{a.aS} = \texttt{finished} \wedge \texttt{a.tS} = \texttt{ok})$. To search for such states, we add the KMRL rule in Figure 11 to the set of rules of the semantics of xSPEM. The rule is not part of the semantics of xSPEM, but acts like an observer, which runs together with the operational semantics rules, and, when it observes that the expected OCL query holds, it prints (by convention) the string "reached" - actually, it adds this string at the end of the $\langle \_ \rangle_{\mathsf{result}}$ cell.

The problem of finding states satisfying OCL Boolean queries has thus been reduced to searching for $\mathbb{K}$ configurations denoting models, reachable from a given initial model-configuration, such that the $\langle \_ \rangle_{\mathsf{result}}$ cell contains a sequence ending with the string "reached". We have automated this process using a script, that takes the compiled $\mathbb{K}$ semantics of our DSML enriched with the observer rule, launches it in the Maude rewriting engine to search for the shortest solution, and returns the solution, filtered for better readability. Recall that a compiled $\mathbb{K}$ definition is a Maude rewrite specification. Hence, all what users need to write is their observer rule in KMRL: the rest of the process is fully automatic.

Here is, for example, the result of model checking for the property defined by the above observer, as produced by our script (whe have taken advantage of the fact that rules print what they do: starting, clock ticking, and finishing):

```
["start"] [a : activity]
```

[6] Remember that we have imposed an OCL constraint stating that there is exactly one proper process - i.e., a process which is not an activity - in each xSPEM model.

```
["tick"] [1] ["tick"] [2] ["tick"] [3] ["tick"] [4] ["tick"] [5]
["finishOk"] [a : activity]
["start"] [b : activity]
["tick"] [6] ["tick"] [7] ["tick"] [8]
["finishOk"] [b : activity]
```

One thing that can be noticed is that the expected property is indeed satisfied: both activities are finished in time (cf. the `["finishOk"]` output printed by their "finishing" rules). Another thing that can be noticed is that `b` started only when `a` finished, satisfying the requirement illustrated in the xSPEM model in Figure 2 by the `finishedToStart` reference linking `b` to `a` via `w1`. If we replace this link by a e.g., `finishedToStart`, the shortest solution is a different one, in which the two activitie's starting and finishing events occur in a different order.

Model-checking reachability properties can straightforwardly be generalised to checking the feasability of *scenario executions*, which consists in checking whether certain partially specified sequences of actions are executable by a DSML.

We are currently working on a language for scenario definition, its mapping to KMRL rules, and on the scenario-feasability verification technique in $\mathbb{K}$.

4 Conclusion, Related, and Future Work

We have proposed a formal approach for the definition and analysis of DSMLs. The approach uses the $\mathbb{K}$ semantical framework, which has shown its efficiency at defining semantics of general progamming languages, and applies it to the MDE ingredients used in defining DSMLs: metamodels for syntax, OCL for static semantics, models for "programs", and a combined declarative/imperative model-transformation language for operational semantics of DSMLs, which we call KMRL.

The approach was illustrated on xSPEM, a DSML based on an OMG standard.

Metamodels, OCL, and models are standard MDE concepts, which can be assumed to be familiar to software engineers trained in the basics of MDE. We have designed KMRL to re-use as much as possibly MDE basics: OCL occurs in both conditions and imperative parts, and the declarative part of KMRL generalises the representation of MDE-models. Our hope is therefore that KMRL will be easy to learn by engineers familiar with the basics of MDE, which will enable them to formally define their DSMLs and to perform formal verifications on them.

We have much benefitted in this work from $\mathbb{K}$'s modularity. Each syntactical construct of OCL is defined semantically in terms of a few $\mathbb{K}$ rules, and adding new constructions does not alter the semantics of the existing ones. Once OCL was defined, it could be reused as such to define the semantics of conditions of KMRL rules, and, with a few more $\mathbb{K}$ rules, we defined the imperative part of KMRL. We have also much benefitted from the flexibility of the general Rascal context-free grammars for parsing, and also from Rascal's powerful primitives for navigating in and transforming abstract syntax trees on the fly.

Comparison with Related Work KERMETA [6] is a metamodelling language, which allows users to define the syntax of DSMLs using metamodels, and their operational semantics by means of imperative commands of the language (assignments, loops, ...). Compared to KERMETA, KMRL also has declarative features (model-rewrite rules), it is formally defined, and allows for formal verification.

The ATL language [7] is a mixed declarative/imperative model transformation language. A formal definition of ATL in Maude [8] has been given in [9]. We took inspiration from ATL in this work. Compared to ATL, the declarative features of KMRL are more developped: in ATL one can only match over one model element, whereas in KMRL we allow for matching over arbitrary model patterns. On the other hand, ATL's imperative features are more developped than KMRL's: in ATL rules can call each other, and can invoke methods of class diagrams. Another difference is that ATL can perform general model transformations (i.e., between different metamodels), whereas KMRL is currently limited to one metamodel.

Several other approaches [10–12] use the Maude algebraic and rewriting-based formal specification language [8]. In these approaches, model transformations (in particular, DSML operational semantics) can only be specified in a declarative manner, by mapping them to Maude equations/rewrite rules. Compared to these approaches, ours also includes imperative features, which are lower-level but allow for better control. The same comparison can be drawn with declarative model transformations based on graph rewriting [13, 14].

Finally, the so-called *translational* approach [15] consists in endowing in a DSML with a formal semantics by translating it to a target language that does have a formally defined semantics. For example, xSPEM has been defined by translation to timed Petri nets [15]. Our approach differs in that we define not individual DSMLs, but a DSML *definition framework* (here, the MDE-based one). Our approach is thus more general than the translational one, and is more likely to be accepted by nonexperts since it does not require from them specialised knowledge of a target language (for writing a translation from DSML to it). On the other hand, due to its generality, our approach is likely to be less efficient for execution/verification than specialised, "hard-coded" translational ones.

Future Work One can envisage a way to combine the benefits of the translational approach (efficiency) and of ours (generality). It would consist in first having the DSML specialists formally define their language as we propose; then, the definition can serve as reference for translation to specialised languages for, e.g., more efficient execution and verification. If the target language has a formally defined operational semantics, the translation between the formally defined DSML and the target language can even formally be proved correct if needed.

Regarding formal verification, we are now working on scenario verification, which constitutes a high-level validation technique, compatible with the high-level nature of our DSML definition framework. The framework itself is currently implemented as a loosely coupled set of tools, which requires some knowledge to operate with. We are working on an implementation under Eclipse that will present users a friendly interface for their DSML definitions and analyses.

References

1. The Objet Management Group. The object constraint language, version 2.2. Technical report, 2010. http://www.omg.org/spec/OCL/2.2/.
2. Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *SCAM*, pages 168–177. IEEE Computer Society, 2009.
3. G. Roşu and T.-F. Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.
4. Reda Bendraou, Benoît Combemale, Xavier Crégut, and Marie-Pierre Gervais. Definition of an executable spem 2.0. In *APSEC*, pages 390–397. IEEE Computer Society, 2007.
5. Software & systems process engineering metamodel specification (SPEM). http://www.omg.org/spec/SPEM/2.0/.
6. 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.
7. Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *Sci. Comput. Program.*, 72(1-2):31–39, 2008.
8. 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.
9. Javier Troya and Antonio Vallecillo. Towards a rewriting logic semantics for atl. In Laurence Tratt and Martin Gogolla, editors, *ICMT*, volume 6142 of *Lecture Notes in Computer Science*, pages 230–244. Springer, 2010.
10. Artur Boronat, Reiko Heckel, and José Meseguer. Rewriting logic semantics and verification of model transformations. In Marsha Chechik and Martin Wirsing, editors, *FASE*, volume 5503 of *Lecture Notes in Computer Science*, pages 18–33. Springer, 2009.
11. 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.
12. Vlad Rusu. Embedding domain-specific modelling languages into Maude specifications. *ACM Software Engineering Notes*, 2011. To appear. Extended version available at http://researchers.lille.inria.fr/~rusu/SoSym/.
13. Gabriele Taentzer. AGG: A graph transformation environment for modeling and validation of software. In John L. Pfaltz, Manfred Nagl, and Boris Böhlen, editors, *AGTIVE*, volume 3062 of *Lecture Notes in Computer Science*, pages 446–453. Springer, 2003.
14. 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.
15. B. Combemale, X. Crégut, P.-L. Garoche, and X. Thirioux. Essay on Semantics Definition in MDE. An Instrumented Approach for Model Verification. *Journal of Software*, 4(9):943–958, November 2009.
16. Grigore Roşu, Chucky Ellison, and Wolfram Schulte. Matching logic: An alternative to Hoare/Floyd logic. In Michael Johnson and Dusko Pavlovic, editors, *Proceedings of the 13th International Conference on Algebraic Methodology And Software Technology (AMAST '10)*, volume 6486, pages 142–162. LNCS, 2010.