HAL
open science

A Generic Tool for Tracing Executions Back to a DSML's Operational Semantics

Benoit Combemale, Laure Gonnord, Vlad Rusu

To cite this version:

Benoit Combemale, Laure Gonnord, Vlad Rusu. A Generic Tool for Tracing Executions Back to a DSML's Operational Semantics. Seventh European Conference on Modelling Foundations and Applications, Jun 2011, Birmingham, United Kingdom. pp.35-51. hal-00593425

HAL Id: hal-00593425

https://hal.science/hal-00593425v1

Submitted on 15 May 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Generic Tool for Tracing Executions Back to a DSML's Operational Semantics

Benoît Combemale[1]*, Laure Gonnord[2], and Vlad Rusu[2]

[1] University of Rennes 1, IRISA, Campus de Beaulieu, Rennes, France
INRIA Rennes - Bretagne Atlantique (Triskell Project)
[2] LIFL - UMR CNRS/USTL 8022, INRIA Lille - Nord Europe
(DaRT Project) 40 avenue Halley, 59650 Villeneuve d'Ascq, France
First.Last@inria.fr

Abstract. The increasing complexity of software development requires rigorously defined *domain specific modeling languages* (DSML). Model-driven engineering (MDE) allows users to define a DSML's syntax in terms of *metamodels*. The behaviour of a language can also be described, either operationally, or via transformations to other languages (e.g., by code generation). If the first approach requires to redefine analysis tools for each DSML (simulator, model-checker...), the second approach allows to reuse existing tools in the targeted language. However, the second approach (also called *translational semantics*) imply that the results (e.g., a program crash log, or a counterexample returned by a model checker) may not be straightforward to interpret by the users of a DSML. We propose in this paper a generic tool for formally tracing such analysis/execution results back to the original DSML's syntax and operational semantics, and we illustrate it on xSPEM, a timed process modeling language.

1 Introduction

The design of a Domain-Specific Modeling Language (DSML) involves the definition of a metamodel, which identifies the domain-specific concepts and the relations between them. The metamodel formally defines the language's abstract syntax. Several works - [6,11,16], among others - have focused on how to help users define operational semantics for their languages in order to enable model execution and formal analyses such as model checking. Such analyses are especially important when the domain addressed by a language is safety critical.

However, grounding a formal analysis on a DSML's syntax and operational semantics would require building a specific verification tool for each DSML; for example, a model checker that "reads" the syntax of the DSML and "understands" the DSML's operational semantics. This is not realistic in practice.

Also, any realistic language will eventually have to be executed, and this usually involves code generation to some other language. Hence, model execution, resp. formal analyses, are performed via transformations of a *source* DSML to

* This work has been partially supported by the ITEA2 OPEES project.

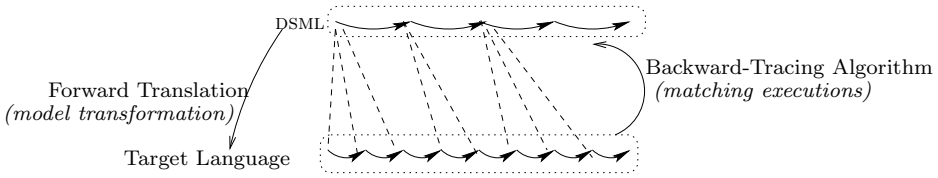

Fig. 1. Back-tracing executions.

some *target* language (the language chosen for code generation, resp. the input language of a model checker). The consequence is that execution and analysis results are typically not understandable by the source DSML practitioners. Hence, there should be a translation of the execution/analysis results back to the source.

In this paper we address the problem of formally tracing back results that are finite executions of a target language (that could have been obtained, e.g., as counterexamples to safety properties in a model checker, or as program crash logs) to executions of a source DSML, thereby allowing the DSML users to understand the results and to take action based on them. We propose a generic algorithm and its implementation in a DSML-independent tool to achieve this.

Our approach is illustrated in Figure 1. A *forward translation* typically implemented as a model transformation translates a DSML to a target language. Consider then an execution of that language, represented at the bottom of Figure 1. The *back-tracing algorithm* maps that execution to one that *matches* it in the source DSML, according to its syntax and operational semantics. We formally define this algorithm in the paper, and implement it in an MDE framework, using an aspect-oriented paradigm to make it reusable and generic for a broad class of DSML: those whose operational semantics are definable as finitely-branching transition systems (i.e., allowing for finite nondeterminism), and for executions of the target language presented as finite, totally ordered sequences of states.

The algorithm is parameterized by a relation R (depicted in Figure 1 using dashed lines) between states of the DSML and states of the target language; and by a natural number n that encodes an upper bound on the allowed "difference in granularity" between executions of the DSML and of the target language.

Our algorithm does not require the precondition that R is a bisimulation or a simulation relation between transition systems; indeed, detects situations where R is not so. However, in the case that R was proved to be such a relation (typically, in a theorem prover, as in, e.g., [5]) , our algorithm nicely complements the proof, by obtaining the parameters R and n that it needs, and by explicitly finding *which* DSML executions match a given target-language execution. Specifically,

- our algorithm requires the parameters R and n as inputs, and one can reasonably assume that these parameters characterise the bisimulation relation against which the model transformations was verified; hence, our algorithm benefits from that verification by obtaining two of its crucial inputs;
- our algorithm provides information that model transformation verification does not: DSML executions that correspond to given target-language ones.

Also, by combining model transformation verification with our back-tracing algorithm, we completely relieve DSML users of having to know *anything* about

the target language. This is important for such formal methods to be accepted in practice. A typical use of the combined approach by a DSML user would be:

- the user chooses a model conforming to the DSML, and a safety property;
- the model transformation automatically maps the model and property to the target language, here assumed to be the language of a model checker;
- the model checker runs automatically, producing the following output:
  - either *ok*, meaning that the property holds on the domain-specific model;
  - or a counterexample (in terms of the target language), that our tool automatically maps to an execution in terms of the source DSML.

This is an interesting (in our opinion) combination of theorem proving (for model transformation) and model checking (for model verification), set in the MDE and aspect-oriented paradigms to provide a DSML-independent implementation.

The rest of the paper is organized as follows. In Section 2 we illustrate our approach on an example (borrowed from [5]) of a process modeling language called xSPEM, transformed into Prioritized Time Petri Nets for verification by model checking. In Section 3 we present the generic implementation of our tool based on the advanced MDE capabilities and aspect-oriented features provided by the KERMETA environment [11], and we show the results of the implementation on the example discussed in Section 2. In Section 4 we detail the back-tracing algorithm implemented by the tool, and formally state its correctness. In Section 5 we present related work, and we conclude and suggest future work in Section 6.

2 Running Example

In this section we present a running example and briefly illustrate our approach on it. The example is a DSML called xSPEM [1], an executable version of the SPEM standard [14]. A transformation from xSPEM to Prioritized Time Petri Nets (PrTPN) was defined in [5] in order to benefit from the TINA verification toolsuite [2]. They have also proved (using the COQ proof assistant) that this model transformation induces a *weak bisimulation* between any xSPEM model's behavior and the behavior of the corresponding PrTPN. This implies in particular that for every PrTPN P and every execution of P returned by TINA - for instance, as a counterexample to a safety property - there *exists* a *matching* execution in the xSPEM model that transforms to P. However, their approach does not exhibit *which* xSPEM execution matches a given PrTPN execution.

This is the problem we address in this paper, with a generic approach that we instantiate on the particular case of the xSPEM-to-PrTPN transformation.

In the rest of this section we briefly describe the xSPEM language (Section 2.1): its abstract syntax, defined by the metamodel shown in Figure 2, and its operational semantics. After recalling a brief description of PrTPN (Section 2.2), we illustrate the model transformation on a model (Section 2.3). Finally, we show the expected result of our algorithm on this example (Section 2.4).

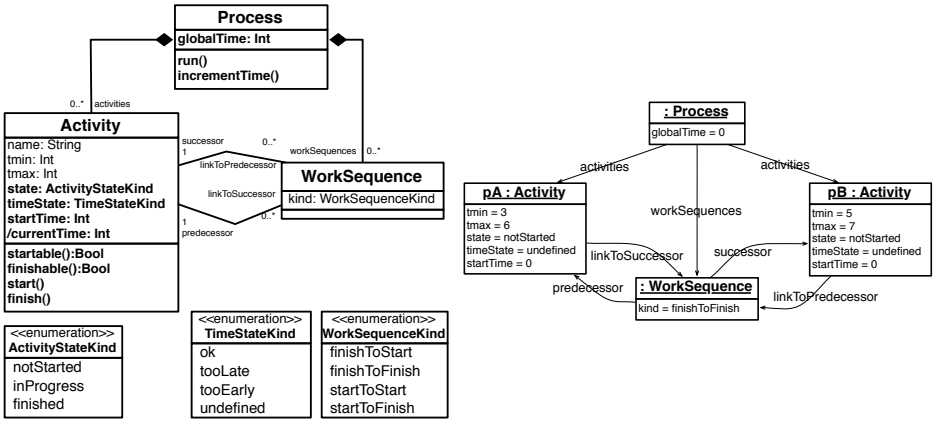

Fig. 2. xSPEM (simplified) metamodel and a model example

2.1 The xSPEM language and its operational semantics

In the metamodel shown in Figure 2 (left), an *Activity* represents a general unit of work assignable to specific performers. Activities are ordered thanks to the *WorkSequence* concept, whose attribute *kind* indicates when an activity can be started or finished. The values of *kind* are defined by the *WorkSequenceKind* enumeration. Values have the form *state*To*Action* where *state* indicates the state of the work sequence's source activity that allows to perform the *action* on the work sequence's target activity. For example, in the right-hand side of Figure 2, the two activities pA and pB are linked by a *WorkSequence* of kind *finishToFinish*, which expresses that pB will be allowed to complete its execution only when pA is finished. The *tmin* and *tmax* attributes of the *Activity* class denote the minimum and, respectively, the maximum duration of activities.

Operational Semantics The following attributes and methods (written in bold font in Figure 2) are used for defining the system's state and operational semantics[1]. An activity can be *notStarted*, *inProgress*, or *finished* (*state* attribute). When it is finished an activity can be *tooEarly*, *ok*, or *tooLate* (*timeState* attribute), depending on whether it has completed its execution in the intervals $[0, tmin[$, $[tmin, tmax[$, or $[tmax, \infty[$ respectively (all intervals are left-closed, right-open). The *timeState* value is *undefined* while an activity is not *finished*.

Time is measured by a global clock, encoded by the *globalTime* attribute of the *Process* class, which is incremented by the *incrementTime()* method of the class. The remaining attributes and methods are used to implement the state and time changes for each activity; *startTime* denotes the starting moment of a given activity, and the derived attribute *currentTime* records (for implementation reasons) the difference between *globalTime* and *startTime* (i.e., the current execution time of a given activity). Finally, the *startable()* (resp. *finishable()*)

[1] Defining state and operational semantics using attributes and methods is consistent with the Kermeta framework [11] in which we implement our tool.

$$
\begin{array}{l}
\forall ws \in a.linkToPredecessor, \\
\quad (ws.linkType = startToStart \ \&\& \ ws.predecessor.state \in \{inProgress, finished\}) \\
\ \ || \ (ws.linkType = finishToStart \ \&\& \ ws.predecessor.state = finished) \\
\qquad (notStarted, undefined, a.currentTime) \xrightarrow{start} (inProgress, tooEarly, 0)
\end{array}
$$

$$
\begin{array}{l}
\forall ws \in a.linkToPredecessor, \\
\quad (ws.linkType = startToFinish \ \&\& \ ws.predecessor.state \in \{inProgress, finished\}) \\
\ \ || \ (ws.linkType = finishToFinish \ \&\& \ ws.predecessor.state = finished) \\
\qquad \text{if } a.currentTime < a.tmin \text{ then} \\
\qquad (inProgress, tooEarly, a.currentTime) \xrightarrow{finish} (finished, tooEarly, a.currentTime) \\
\qquad \text{if } a.currentTime \in [a.tmin, a.tmax[\text{ then} \\
\qquad (inProgress, ok, a.currentTime) \xrightarrow{finish} (finished, ok, a.currentTime) \\
\qquad \text{if } a.currentTime \geq a.tmax \text{ then} \\
\qquad (inProgress, tooLate, a.currentTime) \xrightarrow{finish} (finished, tooLate, a.currentTime)
\end{array}
$$

Fig. 3. Event-based Transition Relation for Activities

methods check whether an activity can be started (resp. finished), and the *start()* and *finish()* methods change the activity's state accordingly.

Definition 1. *The state of an* xSPEM *model defining the set of activities A is the Cartesian product* $\{globalTime\} \times \Pi_{a \in A}(a.state, a.timeState, a.currentTime)$.

The initial state is $\{0\} \times \Pi_{a \in A}\{(notStarted,\ undefined, 0)\}$. The method *run* of the *Process* class implements this initialisation (Figure 2). The transition relation consists of the following transitions, implemented by the following methods:

- for each activity $a \in A$, the transitions shown in Figure 3. The first one starts the activity (implemented by the method *start* of the *Activity* class). An activity can be started when its associated constraints (written in the OCL language in Figure 3) are satisfied. These constraints are implemented in the *startable()* method of the metamodel. The three remaining transitions deal with finishing activities, depending on whether the activity ends too early, in time, or too late.
- the method *incrementTime* of *Process* increments the *globalTime*. It can be called at any moment. The values of *a.currentTime* are derived accordingly.

2.2 Prioritized Time Petri Nets

We translate xSPEM to *Prioritized Time Petri Nets* (PrTPN) for model checking.

A Petri Net (PN) example is shown in the left-hand side of Figure 4. Let us quickly and informally recall their vocabulary and semantics. A PN is composed of *places* and *transitions*, connected by oriented *Arcs*. A *marking* is a mapping from places to natural numbers, expressing the number of tokens in each place (represented by bullet in a place). A transition is *enabled* in a marking when all its predecessor (a.k.a. input) places contain at least the number of tokens specified by the arc connecting the place to the transition (1 by default when not represented). If this is the case then the transition can be *fired*, thereby removing the number of tokens specified by the input arc from each of its input places, and adding the number of tokens specified by the output arc to each of its successor (a.k.a. output) places. In the extended Petri net formalism that we

Fig. 4. A Petri Net and a Prioritized Time Petri Net

are using there is an exception to this transition-firing rule: if an input place is connected by a *read-arc* (denoted by a black circle) to a transition, then the number of tokens in this input place remains unchanged when the transition is fired. An *execution* of a Petri net is then a sequence of markings and transitions $m_0, t_1, m_1, \ldots, t_n, m_n$ (for $n \geq 0$) starting from a given initial marking m_0, such that each marking m_i is obtained by firing the transition t_i from the marking m_{i-1} (for $i = 0, \ldots, n$) according to the transition-firing rule.

Time Petri Nets (TPN) [9] are an extension of Petri Nets, dedicated to the specification of real-time systems. TPN are PN in which each transition t_i is associated to a *firing interval* that has non-negative rational end-points. Executions are now sequences of the form $(m_0, \tau_0), t_1, (m_1, \tau_1) \ldots, t_n, (m_n, \tau_n)$ (for $n \geq 0$), starting from a given initial marking m_0 at time $\tau_0 = 0$, and such that each marking m_i is obtained by firing the transition t_i at time τ_i from the marking m_{i-1} (for $i = 1, \ldots, n$) according to the transition-firing rule.

Finally, Prioritized Time Petri Nets (PrTPN) [3] allows for *priorities* between transitions. When two transitions can both be fired at the same time, the one that is actually fired is the one that has higher priority (the priorities are denoted by dotted arrows in the right-hand side of Figure 4 - the source of the arrow denotes the higher priority).

2.3 A Transformation from xSPEM to PrTPN

In [5] we have defined a model transformation from xSPEM to PrTPN. We illustrate this transformation by presenting its output when given as input the xSPEM model shown in the right-hand side of Figure 2. The result is shown in Figure 5. Each *Activity* is translated into seven places, connected by four transitions:

- Three places characterize the value of *state* attribute (*NotStarted*, *InProgress*, *Finished*). One additional place called *Started* is added to record the fact that the activity has been started, and may be either *inProgress* or *finished*.
- The three remaining places characterize the value of the *time* attribute: *tooEarly* when the activity ends before *tmin*, *tooLate* when the activity ends after *tmax*, and *ok* when the activity ends in due time.

We rely on *priorities* among transitions to encode temporal constraints. As an example, the *deadline* transition has a priority over the *finish* one (cf. Figure 5). This encodes the fact that the termination interval $[tmin, tmax[$ is right-open.

Finally, a *WorkSequence* instance becomes a *read-arc* from one place of the source activity, to a transition of the target activity according to the *kind* attribute of the *WorkSequence*. In our example, *kind* equals *finishToFinish*, meaning that *pA* has to complete its execution before *pB* finishes; hence the read-arc in Figure 5 from the *pA_finished* place to the *pB_finish* transition.

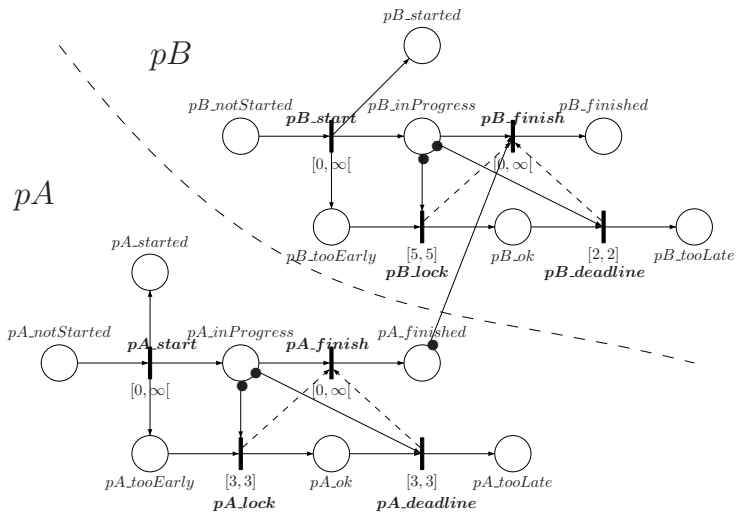

Fig. 5. PrTPN obtained from the xSPEM model in Figure 2 (right). The initial marking has one token in each of the *pA_notStarted* and *pB_notStarted* places.

2.4 An illustration of our Back-Tracing Algorithm

The PrTPN obtained from the transformation of a given xSPEM model can be analyzed by the TINA model checker. For example, to exhibit an execution where both activities end on time, we challenge TINA to prove that such an execution does not exist. This is expressed by the following temporal-logic formula:

$\Box\neg\,(pA_finished \wedge pA_ok \wedge pB_finished \wedge pB_ok)$

The tool returns *false*, and chooses one PrTPN execution that contradicts the temporal-logic property - that is, an execution where both activities end on time. For sake of simplicity, the markings m_i of the execution are not shown:

`(m_0,0),pA_start,(m_1,0),pA_lock,(m_2,3),pA_finish,(m_3,3),`
`pB_start,(m_4,3),pB_lock,(m_5,8),pB_finish,(m_6,8).`

That is, `pA_start` fires at time 0, then `pA_lock`, `pA_finish`, `pB_start` fire in sequence at time 3, and finally `pB_lock`, `pB_finish` fire in sequence at time 8.

Our back-tracing algorithm (cf. Section 4.2) takes this input together with a relation R between xSPEM and PrTPN states, and a natural-number bound n that captures the difference in granularity between xSPEM and PrTPN executions. Two states are in the relation R if for each activity, the value of its *state* attribute is encoded by a token in the corresponding place of the PrTPN, and when an activity is *finished*, the value of its *timeState* attribute is encoded by a token in the corresponding place of the PrTPN (cf. Section 2.3). For instance, $A.state = notStarted$ is encoded by a token in the *pA_nonStarted* place; and similarly for the *inProgress* and *finished* state values; and $A.timeState = ok$ is encoded by a token in the *pA_ok* place; and similarly for the *tooEarly* and *tooLate* time state values. Regarding the bound n, it is here set to 5 - because in xSPEM executions *globalTime* advances by at most one time unit, but in the given PrTPN execution, the maximum difference between two consecutive timestamps is $5 = 8 - 3$. Then, our algorithm returns the following xSPEM execution:

$globalTime$	xSPEM states : $(state_i, timeState_i, currentTime_i)$	
	$i = p_A$	$i = p_B$
0	$(notStarted, undefined, 0)$	$(notStarted, undefined, 0)$
0	$(inProgress, tooEarly, 0)$	$(notStarted, undefined, 0)$
3	$(inProgress, ok, 3)$	$(notStarted, undefined, 0)$
3	$(finished, ok, 3)$	$(notStarted, undefined, 0)$
3	$(finished, ok, 3)$	$(inProgress, tooEarly, 0)$
8	$(finished, ok, 3)$	$(inProgress, ok, 5)$
8	$(finished, ok, 3)$	$(finished, ok, 8)$

Note that indeed both processes finish in due time: pA starts at 0 and finishes at 3 (its $tmin = 3$); and pB starts at 3 and finishes at 8 (its $tmin = 5 = 8 - 3$).

3 A Generic Tool for Tracing Executions in Kermeta

Our implementation takes as input an execution of the target language and returns as output a corresponding execution of the source DSML. In our running example, the input execution trace comes from the TINA toolsuite and the tool returns an xSPEM model execution, as shown in the previous section.

3.1 Generic Implementation Using Executable Metamodeling

Kermeta is a language for specifying metamodels, models, and model transformations that are compliant to the Meta Object Facility (MOF) standard [12]. The abstract syntax of a DSML is specified by means of metamodels possibly enriched with constraints written in an OCL-like language [13]. Kermeta also proposes an imperative language for describing the operational semantics of DSML [11].

We implement in Kermeta the back-tracing algorithm given in detail in Section 4. Here, we focus more specifically on the genericity of the implementation. Accordingly, our implementation relies on a generic tree-based structure (cf. Figure 6, left). The algorithm is generically defined in the *treeLoading* method of the *SimulationTree* class. To use this method, the *SourceExecution* and its sequence of *SourceState* have to be specialized by an execution coming from an execution platform (*e.g.*, a verification tool), and the *TargetState* have to be specialized by the corresponding concept in the DSML[2].

The *treeLoading* method builds a simulation tree by calling the method *findStates* for each tree node. This method *findStates* computes the set of target states that are in relation with the next source state. The generic computation is based on calling the abstract method *execute* on the current *TargetState*. For a given DSML (xSPEM in our case), the method *execute* needs to be implemented to define one execution step according to the DSML operational semantics. This method depends on a given relation R (in our example, the one discussed in Section 2.4) defined in the method *simulationRelation* between *sourceState* and *targetState*. It also depends on a given maximal depth of search defined by the

[2] We assume in this work a naive definition of the domain-specific model state by specializing *TargetState* by *Process*. This work could be extended to well-distinguish the dynamic information in order to store only this one.

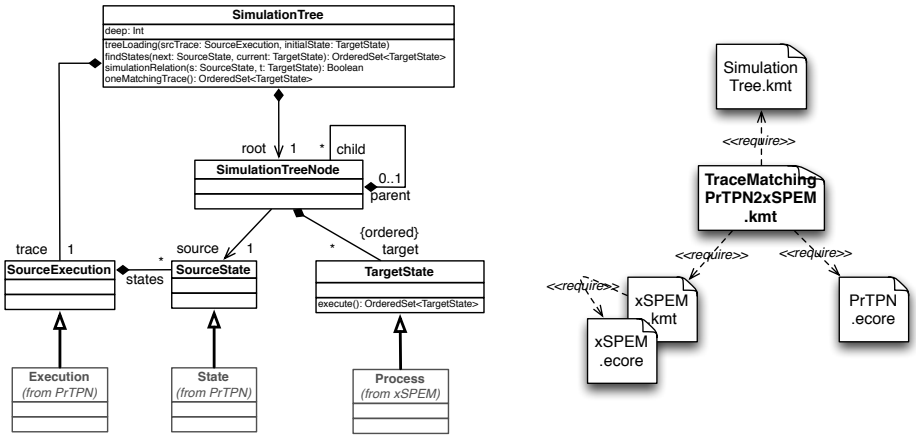

Fig. 6. The generic *SimulationTree* class (left) and how to use it (right).

attribute *deep*. Once the simulation tree is built, a DSML execution that matches any execution provided by the verification tool can be generated by the method *oneMatchingTrace*.

3.2 Tool Specialization for a Given Example Using Aspect-Oriented (Meta)Modeling

Among others, one key feature of Kermeta is its ability to extend an existing metamodel with constraints, new structural elements (meta-classes, classes, properties, and operations), and functionalities defined with other languages using the *require* keyword. This keyword permits the composition of corresponding code within the underlying metamodel as if it was a native element of it. This feature offers flexibility to developers by enabling them to easily manipulate and reuse existing metamodels. The static introduction operator *aspect* allows for defining these various aspects in separate units and integrating them automatically into the metamodel. The composition is performed statically and the composed metamodel is type-checked to ensure the safe integration of all units.

We use both model weaving and static introduction to specialize our generic implementation of the back-tracing algorithm to the particular context of computing xSPEM executions from a PrTPN execution. As described in Figure 6 (right) and in Listing 1.1, the *TraceMatchingPrTPN2xSPEM.kmt* program weaves the xSPEM metamodel (*xSPEM.ecore*) and its operational semantics (*xSPEM.kmt*), together with a metamodel for PrTPN (*PrTPN.ecore*) and our generic implementation for the back-tracing algorithm (*SimulationTree.kmt*) – cf. lines 4 to 7 in Listing 1.1. In addition to weaving the different artifacts, *TraceMatchingPrTPN2xSPEM.kmt* also defines the links between them. Thus we define the inheritance relations (cf. Figure 6, left) between *Execution* (from *PrTPN.ecore*) and *SourceExecution*, between *State* (from *PrTPN.ecore*) and *SourceState*, and between *Process* (from *xSPEM.ecore*) and *TargetState* – cf. lines 9 to 11 in Listing 1.1. We also define the relation R by specializing the

method *simulationRelation*) (cf. lines 13 to 18 in Listing 1.1) and the value of the attribute *deep* (cf. line 35 in Listing 1.1). Finally, we illustrate the content of the method *main* of *TraceMatchingPrTPN2xSPEM.kmt* in Listing 1.1, that loads a given PrTPN execution trace, a given process initial state, and computes a corresponding process execution trace.

Listing 1.1. *TraceMatchingPrTPN2xSPEM.kmt*

```
1  package prtpn2xspem;
2
3  require kermeta
4  require "./traceMatching.kmt"
5  require "./prtpn.ecore"
6  require "./xSPEM.ecore"
7  require "./xSPEM.kmt"
8
9  aspect class prtpn::Execution inherits traceMatching::SourceExecution { }
10 aspect class prtpn::State inherits traceMatching::SourceState { }
11 aspect class xspem::Process inherits traceMatching::TargetState { }
12
13 aspect class traceMatching::SimulationTree {
14   operation simulationRelation(next: SourceState, current: TargetState) :
        Boolean is do
15     // specification of the simulation relation between xSPEM and PrTPN
16     // ...
17   end
18 }
19
20 class TraceMatching
21 {
22   // Tracing a Petri net execution trace to an xSPEM process execution
23   operation main(inputTrace: String, initialState: String, deep: Integer)
        : Void is do
24     var rep : EMFRepository init EMFRepository.new
25     // loading of the Petri net trace  (using EMF API)
26     var resPN : Resource init rep.createResource(inputTrace, "./prtpn.
        ecore")
27     var pnTrace : prtpn::Execution
28     tracePN ?= resPN.load.one
29     // loading of the xSPEM process initial state (using EMF API)
30     var resProcess : Resource init rep.createResource(initialState, "./
        xSPEM.ecore")
31     var processInitState : xspem::Process
32     processInitState ?= resProcess.load.one
33     // trace matching
34     var prtpn2xspem : traceMatching::SimulationTree init traceMatching::
        SimulationTree.new
35     prtpn2xspem.deep := 5
36     prtpn2xspem.treeloading(pnTrace, processInitState)
37     // one possible result
38     prtpn2xspem.oneMatchingTrace()
39   end
40 }
41 endpackage
```

Thus, *TraceMatchingPrTPN2xSPEM.kmt* may be used for a given execution of PrTPN conforming to *PrTPN.ecore*, in our case, the Tina execution obtained in Section 2.4). As Tina only provides textual output, we had to parse and pretty-print it in the right format (XMI - *XML Metadata Interchange*). This was done in OCaml[3]. After running the method *SimulationTree*, we obtain the

[3] http://caml.inria.fr/ocaml/

following input (and corresponding model) using the method *oneMatchingTrace*:

```
Console
simulationTree.kmt_mtverification____Main_main [Kermeta Application] platform:/resource/fr.inria.mt.ver
One matching trace:
(0, {(pB, notStarted), (pA, notStarted)}) --> (0, {(pA, inProgress), (pB, notS
tarted)}) --> (3, {(pB, notStarted), (pA, inProgress)}) --> (3, {(pA, finish
ed@3=>ok), (pB, notStarted)}) --> (3, {(pB, inProgress), (pA, finished@3=>ok)}
) --> (8, {(pA, finished@3=>ok), (pB, inProgress)}) --> (8, {(pA, finished@3
=>ok), (pB, finished@5=>ok)})
```

4 Formalizing the Problem

In this section we formally define our back-tracing algorithm and prove its correctness. We start by recapping the definition of *transition systems* and give a notion of *matching* an execution of a transition system with a given (abstract) sequence of states, modulo a given relation between states.

4.1 Transition systems and execution matching

Definition 2 (transition system). *A transition system is a tuple* $\mathcal{A} = (A, a_{init}, \rightarrow_{\mathcal{A}})$ *where* A *is a possibly infinite set of states,* a_{init} *is the initial state, and* $\rightarrow_{\mathcal{A}} \subseteq A \times A$ *is the transition relation.*

Notations: $\mathbb{N}$ is the set of natural numbers. We write $a \rightarrow_{\mathcal{A}} a'$ for $(a, a') \in \rightarrow_{\mathcal{A}}$. An execution is a sequence of states $\rho = a_0, \ldots a_n \in A$, for some $n \in \mathbb{N}$, such that $a_i \rightarrow_{\mathcal{A}} a_{i+1}$ for $i = 0, \ldots, n-1$; $length(\rho) = n$ is the *length* of the execution ρ. Executions of length 0 are states. We denote by $exec(a)$ the subset of executions that start in the state a, i.e., the set of executions ρ of $\mathcal{A}$ such that $\rho(0) = a$. We restrict ourselves to *finitely branching* transition systems, meaning that for all states a there are at most finitely many states a' such that $a \rightarrow_{\mathcal{A}} a'$.

Definition 3 (R-matching). *Given a transition systems* $\mathcal{B} = (B, b_{init}, \rightarrow_{\mathcal{B}})$*, a set* A *with* $A \cap B = \emptyset$*, an element* $a_{init} \in A$*, a relation* $R \subseteq A \times B$*, and two sequences* $\rho \in a_{init}A^*$*,* $\pi \in exec(b)$*, we say that* ρ *is* R-matched *by* π *if there exists a (possibly, non strictly) increasing function* $\alpha : [0, \ldots, length(\rho)] \rightarrow \mathbb{N}$ *with* $\alpha(0) = 0$*, such that for all* $i \in [0, \ldots, length(\rho)]$*,* $(a_i, b_{\alpha(i)}) \in R$*.*

Example 1. In Figure 7 we represent two sequences ρ and π. A relation R is denoted using dashed lines. The function $\alpha : [0, \ldots, 5] \rightarrow \mathbb{N}$ defined by $\alpha(i) = 0$ for $i \in [0, 3]$, $\alpha(4) = 1$, and $\alpha(5) = 5$ ensures that ρ is R-matched by π.

In our framework, $\mathcal{B} = (B, b_{init}, \rightarrow_{\mathcal{B}})$ is the transition system denoting a DSML $\mathcal{L}$ (in our running example, xSPEM) and its operational semantics, with b_{init} being the initial state of a particular model $m \in \mathcal{L}$ (in our example, the model depicted in the right-hand side of Figure 2). The model m is transformed by some model transformation ϕ (in our example, the model transformation defined in [5]) to a target language (say, $\mathcal{L}'$; in our example, PrTPN, the input language of the TINA model checker). About $\mathcal{L}'$, we only assume that it has a notion of *state* and that its state-space is a given set A. Then, $\rho \in a_{init}A^*$ is the execution of the tool that we are trying to match, where a_{init} is the initial state

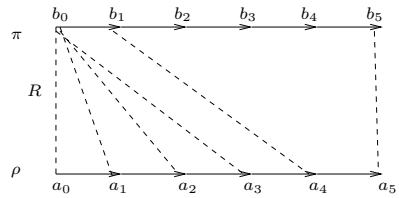

Fig. 7. R-matching of sequences. The relation R is represented by dashed lines. Note that R-matching does not require all π-states to be in relation to ρ-states.

of the model $\phi(m) \in \mathcal{L}'$ (here, the PrTPN illustrated in Figure 5 with the initial marking specified in the figure). The relation R can be thought of as a matching criterion between states of a DSML and those of the target language; it has to be specified by users of our back-tracing algorithm.

Remark 1. We do not assume that the operational semantics of $\mathcal{L}'$ is known. This is important, because is saves us the effort of giving operational semantics to the target language, which can be quite hard if the language is complex.

4.2 The back-tracing problem

Our back-tracing problem can now be formally stated as follows: given
- a transition system $\mathcal{B} = (B, b_{init}, \rightarrow_{\mathcal{B}})$
- a set A, an element $a_{init} \in A$, and a sequence $\rho \in a_{init}A^*$
- a relation $R \subseteq A \times B$,

does there exist an execution $\pi \in exec(b_{init})$ such that ρ is R-matched by π; and if this is the case, then, construct such an execution π.

Unfortunately, this problem is not decidable/solvable. This is because an execution π that R-matches a sequence ρ can be arbitrarily long; the function α in Definition 3 is responsible for this. One way to make the problem decidable is to impose that, in Definition 3, the function α satisfies a "bounded monotonicity property" : $\forall i \in [0, length(\rho) - 1]\ \alpha(i+1) - \alpha(i) \leq n$ for some given $n \in \mathbb{N}$. In this way, the candidate executions π that may match ρ become finitely many.

Definition 4 ((n, R)-matching). *With the notations of Definition 3, and given a natural number $n \in \mathbb{N}$ we say that the sequence ρ is* (n, R)-matched *by the execution π if the function α satisfies $\forall i \in [0, length(\rho) - 1]\ \alpha(i+1) - \alpha(i) \leq n$.*

In Example 1 (Figure 7), ρ is $(5, R)$-matched (but not $(4, R)$-matched) by π.

4.3 Back-tracing algorithm

For a set $S \subseteq A$ of states of a transition system $\mathcal{A}$, we denote by $\rightarrow^n_{\mathcal{A}}(S)$ $(n \in \mathbb{N})$ the set of states $\{a' \in A | \exists a \in S. \exists \rho \in exec(a).\, length(\rho) \leq n \wedge \rho(length(\rho)) = a'\}$; it is the set of successors of states in S by executions of length at most n. Also, for a relation $R \subseteq A \times B$ and $a \in A$ we denote by $R(a)$ the set $\{b \in B | (a, b) \in R\}$. We denote the empty sequence by ε, whose length is undefined; and, for a nonempty sequence ρ, we let $last(\rho) \triangleq \rho(length(\rho))$ denote its last element.

Algorithm 1 Return an execution $\pi \in exec(b_{init})$ of $\mathcal{B}$ that (n, R)-matches the longest prefix of a sequence $\rho \in a_{init}A^*$ that can be (n, R)-matched.

Require: $\mathcal{B} = (B, b_{init}, \rightarrow_{\mathcal{B}})$; A; $a_{init} \in A$; $\rho \in a_{init}A^*$; $n \in \mathbb{N}$; $R \subseteq A \times B$
  Local Variable: $\alpha : [0..length(\rho)] \rightarrow \mathbb{N}$; $\pi \in B^*$; $S, S' \subseteq B$; $\ell \in \mathbb{N}$
1: **if** $(a_{init}, b_{init}) \notin R$ **then return** ε
2: **else**
3: $\alpha(0) \leftarrow 0$, $k \leftarrow 0$, $\pi \leftarrow b_{init}$, $S \leftarrow \{b_{init}\}$
4: **while** $k < length(\rho)$ and $S \neq \emptyset$ **do**
5: $k \leftarrow k + 1$
6: $S' \leftarrow R(\rho(k)) \cap \rightarrow_{\mathcal{B}}^{n} (last(\pi))$
7: **if** $S' \neq \emptyset$ **then**
8: Choose $\hat{\pi} \in exec(last(\pi))$
  such that $\ell = length(\hat{\pi}) \leq n$ and $\hat{\pi}(\ell) \in S'$ ▷ ℓ can be 0
9: $\alpha(k) \leftarrow \alpha(k-1) + \ell$
10: $\pi_{\alpha(k-1)+1..\alpha(k)} \leftarrow \hat{\pi}_{1..l}$ ▷ effect of this assignment is null if $\ell = 0$
11: **end if**
12: $S \leftarrow S'$
13: **end while**;
14: **return** π
15: **end if**

Theorem 1 (Algorithm for matching executions). *Consider a transition system $\mathcal{B} = (B, b_{init}, \rightarrow_{\mathcal{B}})$, a set A with $A \cap B = \emptyset$, an element $a_{init} \in A$, a relation $R \subseteq A \times B$, and a natural number $n \in \mathbb{N}$. Consider also a sequence $\rho \in a_{init}A^*$. Then, Algorithm 1 returns an execution $\pi \in exec(b_{init})$ of $\mathcal{B}$ that (n, R) matches the longest prefix of ρ that can be (n, R)-matched.*

A proof can be found in the extended version of this paper [18]. In particular, if there exists an execution in $exec(b_{init})$ that (n, R)-matches the whole sequence ρ then our algorithm returns one; otherwise, the algorithm says there is none. Regarding the algorithm's complexity, it is worst-case exponential in the bound n, with the base of the exponent being the maximum branching of the transition system denoting the operational semantics of the source DSML. For deterministic DSML, the exponential disappears. In practice, n may be known if a proof of (bi)simulation between source and target semantics was performed; this is why our algorithm works best when combined with a theorem prover (as discussed in the introduction). If n is unknown, one can start with $n = 1$ and gradually increase n until a matching execution is found or until resources are exhausted.

Example 2. We illustrate several runs of our algorithm on the execution ρ depicted in the left-hand side of Figure 8, with the transition system $\mathcal{B}$ depicted in the right-hand side of the figure, and the relation R depicted using dashed lines. In the algorithm, let $n = 3$. The set S is initialized to $S = \{b_0\}$. For the first step of the algorithm - i.e., when $k = 1$ in the `while` loop - we choose $b = b_0$ and the execution $\hat{\pi} = b_0$; we obtain $\alpha(1) = 0$, $\pi(0) = b_0$ and $S' = R(a_1) \cap \{b_0, b_1, b_2, b_3, b_4\} = \{b_0, b_1, b_2\}$. At the second step, we choose $b = b_0$ and, say, $\hat{\pi} = b_0, b_2$; we obtain $\alpha(2) = 1$, $\pi(1) = b_2$ and

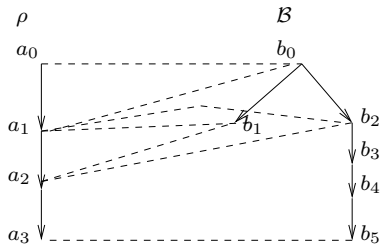

Fig. 8. Attempting to match execution ρ.

$S' = R(a_2) \cap \{b_2, b_3, b_4, b_5\} = \{b_2\}$. At the third step, we can only choose $b = b_2$ and $\hat{\pi} = b_2, b_3, b_4, b_5$; we obtain $\alpha(3) = 4$, $\pi_{2..4} = \hat{\pi}$, and $S' = \{b_5\}$, and now we are done: the matching execution π for ρ is $\pi = b_0, b_2 \ldots b_5$. Note that our non-deterministic algorithm is allowed to make the "most inspired choices" as above. A deterministic algorithm may make "less inspired choices" and backtrack from them; for example, had we chosen $\hat{\pi} = b_0, b_1$ at the second step, we would have ended up failing locally because of the impossibility of matching the last step a_2a_3 of ρ; backtracking to $\hat{\pi} = b_0, b_2$ solves this problem. Finally, note that with $n < 3$, the algorithm fails globally - it cannot match the last step of ρ.

Remark 2. The implementation of our algorithm amounts to implementing non-deterministic choice via state-space exploration. A natural question that arises is then: why not use state-space exploration to perform the model checking itself, instead of using an external model checker and trying to match its result (as we propose to do)? One reason is that it is typically more efficient to use the external model checker to come up with an execution, and to match that execution with our algorithm, than performing model checking using our (typically, less efficient) state-space exploration. Another reason is that the execution we are trying to match may be produced by something else than a model checker, e.g., a program crash log can also serve as an input to our algorithm.

5 Related work

The problem of tracing executions from a given target back to a domain-specific language has been addressed in several papers of the MDE community. Most of the proposed methods are either dedicated to only one pair metamodel/verification tool ([7], [10]) or they compute an "explanation" of the execution in a more abstract way. In [8], the authors propose a general method based on a *traceability* mechanism of model transformations. It relies on a relation between elements of the source and the target metamodel, implemented by means of annotations in the transformation's source codes.essentially By contrast, our approach does not require instrumenting the model transformation code, and is formally grounded on operational semantics, a feature that allows us to prove its correctness.

In the formal methods area, Translation Validation ([15]) has also the purpose of validating a compiler by performing a verification *after each run of the compiler* to ensure that the output code produced by the compilation is a correct translation of the source program. The method is fully automatic for the

developer, who has no additional information to provide in order to prove the translation: all the semantics of the two languages and the relation between states are embedded inside the "validator" (thus it cannot be generic). Contrary to our work, Translation Validation only focuses on proving correctness, and does not provide any useful information if the verification fails. Also, the Counterexample-Guided Abstraction Refinement (CEGAR) verification method ([4]) also consists in matching model-checking counterexamples to program executions. The difference between CEGAR and our approach is that CEGAR makes a specific assumption - that the target representation is an abstract interpretation of the source representation; whereas we do not make this assumption.

Finally, in the paper [17] we describe, among others, an approach for back-tracing executions, which differs from the one presented here in several aspects: the theoretical framework of [17] is based on observational transition systems and the relation between states is restricted to observational equality, whereas here we allow for more general relations between states; and in in [17], the relation is restricted such that the parameter n is always one, which means that one step in the source may match several steps in the target, but not the other way around. The are also practical differences: Maude is less likely to be familiar and acceptable to software engineers; and Maude allows for a direct implementation of the nondeterministic back-tracing algorithm, whereas in Kermeta a deterministic version of the algorithm had to be designed first.

6 Conclusion and Future Work

DSML are often translated to other languages for efficient execution and/or analysis. We address the problem of formally tracing executions of a given target language tool back into an execution of a DSML. Our solution is a generic tool implementing an algorithm that requires that the DSML's semantics be defined formally, and that a relation R be defined between states of the DSML and of the target language. The algorithm also takes as input a natural-number bound n, which estimates a "difference of granularity" between semantics of the DSML and of the target language. Then, given a finite execution ρ of the target language (e.g., a counterexample to a safety property, or program crash log), our algorithm returns an (n, R) *matching* execution π in terms of the DSML's operational semantics - if there is one - or it reports that no such execution exists, otherwise.

We implement our algorithm in Kermeta, a framework for defining operational semantics of DSML (among other features). Using Kermeta's abilities for aspect-oriented metamodeling, our implementation is generic: the user has to provide the appropriate metamodels, as well as the estimated bound n and relation R between the states of DSML and verification tool; the rest is automatic.

We illustrate our tool on an example where the DSML is xSPEM, a timed process modeling language, and the target language is Prioritized Time Petri Nets (PrTPN), the input language of the TINA model checker.

Regarding future work, the main direction is to exploit the combination of theorem proving (for model transformation) and model checking (for model verification) as described in the introduction. Another orthogonal research direction

is to optimise our currently naive implementation in Kermeta in order to avoid copying whole models when only parts of them (their "state") change.

References

1. Reda Bendraou, Benoit Combemale, Xavier Crégut, and Marie-Pierre Gervais. Definition of an eXecutable SPEM2.0. In *14th APSEC*. IEEE, 2007.
2. B. Berthomieu, P.-O. Ribet, and F. Vernadat. The tool TINA – construction of abstract state spaces for Petri nets and time Petri nets. *Int. Journal of Production Research*, 42(14):2741–2756, 2004.
3. Bernard Berthomieu, Florent Peres, and François Vernadat. Model checking bounded prioritized time petri nets. In *ATVA*, pages 523–532, 2007.
4. Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Proceedings of Computer Aided Verification, CAV*, volume 1855, pages 154–169. Springer, 2000.
5. Benoit 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, November 2009.
6. 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 *17th ASE*, pages 267–270. IEEE, 2002.
7. E. Guerra, J. de Lara, A. Malizia, and P. Daz. Supporting user-oriented analysis for multi-view domain-specific visual languages. *Information & Software Technology*, 51(4):769784, 2009.
8. A Hegedüs, G Bergmann, I Ráth, and D Varró. Back-annotation of simulation traces with change-driven model transformations. In *SEFM'10*, September 2010.
9. P. M. Merlin. *A Study of the Recoverability of Computing Systems*. Irvine: Univ. California, PhD Thesis, 1974.
10. J. Moe and D.A. Carr. Understanding distributed systems via execution trace data. In *Proceedings of the 9th International Workshop on Program Comprehension IWPC'01*. IEEE Computer Society, 2001.
11. Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving Executability into Object-Oriented Meta-Languages. In *MoDELS*, volume 3713 of *LNCS*, pages 264–278. Springer, October 2005.
12. Object Management Group. *Meta Object Facility 2.0*, 2006.
13. Object Management Group. *Object Constraint Language 2.0*, 2006.
14. Object Management Group. *Software Process Engineering Metamodel 2.0*, 2007.
15. A. Pnueli, O. Shtrichman, and M. Siegel. Translation validation: From SIGNAL to C. In *Proceedings of Conference on Correct System Design, LNCS 1710*, pages 231–255. Springer-Verlag, 1999.
16. José Eduardo Rivera, Cristina Vicente-Chicote, and Antonio Vallecillo. Extending visual modeling languages with timed behavior specifications. In *CIbSE*, pages 87–100, 2009.
17. V. Rusu. Embedding domain-specific modelling languages into Maude specifications. Available at `http://researchers.lille.inria.fr/~rusu/SoSym`.
18. Vlad Rusu, Laure Gonnord, and Benoît Combemale. Formally Tracing Executions From an Analysis Tool Back to a Domain Specific Modeling Language's Operational Semantics. Technical Report RR-7423, INRIA, october 2010.