HAL
open science

Formally Tracing Executions From an Analysis Tool Back to a Domain Specific Modeling Language's Operational Semantics

Vlad Rusu, Laure Gonnord, Benoit Combemale

To cite this version:

Vlad Rusu, Laure Gonnord, Benoit Combemale. Formally Tracing Executions From an Analysis Tool Back to a Domain Specific Modeling Language's Operational Semantics. [Research Report] RR-7423, INRIA. 2010. inria-00526561

HAL Id: inria-00526561

https://inria.hal.science/inria-00526561

Submitted on 15 Oct 2010

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.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Formally Tracing Executions From an Analysis Tool Back to a Domain Specific Modeling Language's Operational Semantics

Vlad Rusu — Laure Gonnord — Benoît Combemale

N° 7423

Octobre 2010

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--7423--FR+ENG

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE
INRIA
centre de recherche LILLE - NORD EUROPE

Formally Tracing Executions From an Analysis Tool Back to a Domain Specific Modeling Language's Operational Semantics

Vlad Rusu*, Laure Gonnord †, Benoît Combemale ‡

Thème : Architecture et compilation
Équipes-Projets DaRT (Inria Lille) et Triskell (Inria Rennes)

Rapport de recherche n° 7423 — Octobre 2010 — 21 pages

Abstract: The increasing complexity of software development requires rigorously defined *domain specific modelling languages* (DSML). Model-driven engineering (MDE) allows users to define their language's syntax in terms of *metamodels*. Several approaches for defining operational semantics of DSML have also been proposed. These approaches allow, in principle, for model execution and for formal analyses of the DSML. However, most of the time, the executions/analyses are performed via transformations to other languages: code generation, resp. translation to the input language of a model checker. The consequence is 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. In this research report, we propose a formal and operational framework for tracing such results back to the original DSML's syntax and operational semantics.

Key-words: Model-driven engineering (MDE), model transformation, operational semantics, verification, bisimulation, tracing-back executions

* INRIA Researcher at Lille Inria Research Center
† Research Assistant at Lille University
‡ Research Assistant at Rennes University

Centre de recherche INRIA Lille – Nord Europe
Parc Scientifique de la Haute Borne
40, avenue Halley, 59650 Villeneuve d'Ascq
Téléphone : +33 3 59 57 78 00 — Télécopie : +33 3 59 57 78 50

Remontée d'exécutions en terme de sémantique opérationnelle de langage métier

Résumé : Le besoin en définitions rigoureuses de langages de modélisation métiers (LMM) croît avec la complexitéé du développement logiciel. L'ingénierie dirigée par les modèles (IDM), permet à des utilisateurs de définir la syntaxe de nouveaux langages à l'aide de *metamodèles*. Quelques approches proposent également des façons de définir la sémantique opérationnelle de ces langages. Ces approches permettent, en principe, l'exécution des programmes (*modèles*) et leur analyse formelle. Cependant, la plupart du temps, les analyses sont réalisées à l'aide de transformations de ces modèles vers d'autres langages, pour être exécutés (langages généralistes, type C par exemple) ou vérifiés (langages d'entrées de model checkers, de simulateurs, ...). En conséquence, ces résultats (un log de crash de programme, une exécution contre exemple provenant de la vérification d'une propriété par un model-checker, par exemple) peuvent ne pas être directement compréhensibles par les utilisateurs d'un LMM. Dans ce rapport, nous proposons une méthode formelle et opérationnelle pour exprimer ces exécutions en terme de syntaxe et sémantique du LMM initial.

Mots-clés : Ingénierie dirigée par les modèles (IDM), transformation de modèles, sémantique opérationnelle, vérification, bisimulation, remontée d'exécutions

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 requires 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 and formal analyses are performed via transformations of the DSML to other *target* languages (the language chosen for code generation, resp. the input language of a model checker). The consequence is that execution/analysis results are typically not readily understandable by the original DSML practitioners. Hence, there should be a translation of the execution/analysis results into the original DSML.

In this paper we address the problem of formally tracing back results that are finite executions of a target language (e.g., counterexamples to safety properties in a model checker, or program crash logs) to an original DSML, thereby allowing the DSML users to understand the results and to take action based on them.

Our approach is illustrated in Figure 1. A *forward translation* typically implemented as a model transformation translates our 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 original DSML, according to its syntax and operational semantics. We formally define this algorithm in the paper, and implement in an MDE framework. The algorithm is parameterised 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 (informally speaking) encodes a certain "difference in granularity" between executions of the DSML and of the target language.

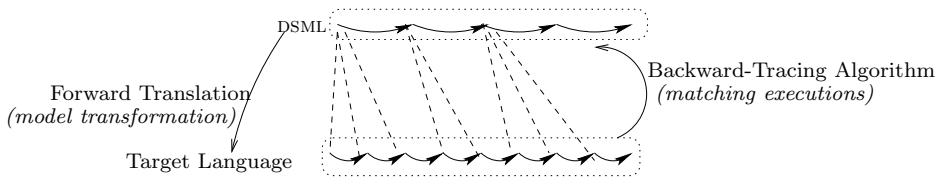

Figure 1:

Our back-tracing algorithm works best if the model transformation implementing the DSML-to-target-language translation has been formally verified in a theorem prover (e.g., [5] presents such a formal verification). The property that is verified is typically some kind of *bisimulation relation*, which implies that for every execution of a DSML instance there *exists* a matching execution in the translation of the DSML instance in the target language, and reciprocally. However, such theorem-proving based approaches typically do not exhibit *which* executions match. Our algorithm is complementary to such verifications because:

- 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 algorithms, 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 DSML instance and a safety property to be verified;
- the model transformation automatically maps the DSML instance and property to the target language, here assumed to be the language of a model checker; the checker runs automatically, producing the following output:
  - either *ok*, meaning that the property holds on the DSML instance;
  - or a counterexample, then automatically mapped to a DSML execution.

This is an interesting (in our opinion) combination of theorem proving (for model transformation)/model checking (for model verification) in an MDE framework.

The rest of the paper is organised as follows. In Section 2 we illustrate our approach on an example (borrowed from [5]) of a process description language called XSPEM, transformed into Prioritised Time Petri Nets for verification by model checking. In Section 3 we present our back-tracing algorithm and formally prove its correctness. In Section 4 we describe the implementation of the algorithm in KERMETA [11], and we show the results of the implementation on the example discussed in Section 2. 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 our 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 Prioritised 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 behaviour and the behaviour

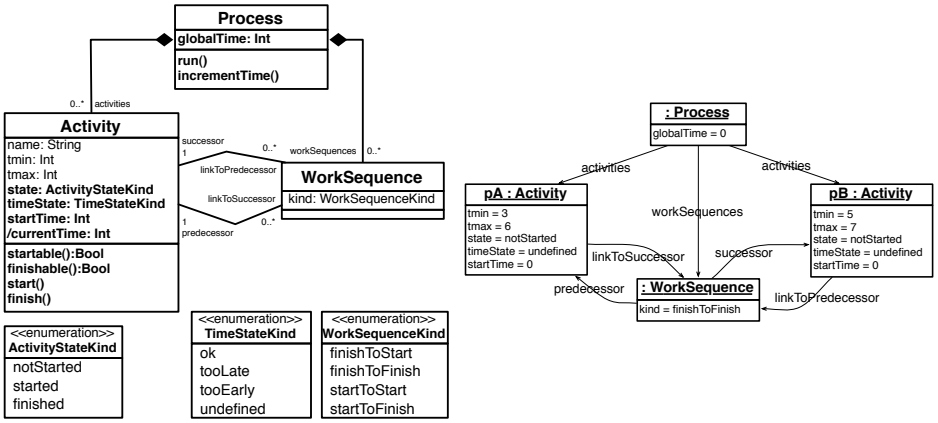

Figure 2: xSPEM (simplified) metamodel and a model example

of the corresponding PrTPN. This implies in particular that for every PrTPN P and every execution of it 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 general approach that we instantiate to 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 simple instance (section 2.3). Finally, we show the expected result of our algorithm on this example (section 2.4).

2.1 The xSPEM language and its operational semantics

In the metamodel of 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 processes 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 *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()*) 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 whose set of activities is A is the Cartesian product* $\{globalTime\} \times \Pi_{a \in A}(state_a, timeState_a, currentTime_a)$.

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 two transitions shown in Figure 3. The first one starts the activity (implemented by the method *start* of the *Activity* class) and the second one finishes it (implemented by the method *finish*). An activity can be started whenever it is not yet activated and when its associated constraints (written in the ocl language in Figure 3) are satisfied. These constraints are implemented in the *startable()* and *finishable()* methods of the metamodel. Note that the second transition's definition has three cases depending on the value of $currentTime_a$ and the time bounds of the activity.
- the method *incrementTime* of *Process* increments the *globalTime*. It can be called at any moment. The values of $currentTime_a$ are derived accordingly.

2.2 Prioritised Time Petri Nets

We translate xspem to *Prioritised 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

[1]Defining state and operational semantics using attributes and methods is consistent with the Kermeta framework [11], which we use for implementing our approach.

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

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

Figure 3: Event-based Transition Relation for Activities

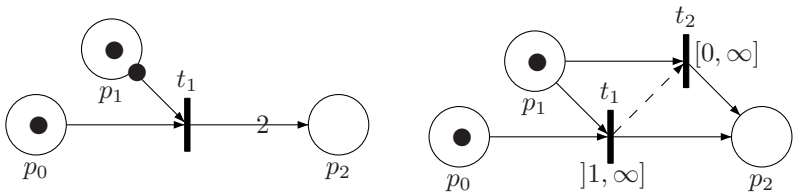

Figure 4: A Petri Net and a Prioritised Time Petri Net

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. However, 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 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. A *deadlock* is a marking where no transition is enabled.

Next, Time Petri Nets are Petri Nets in which each transition t_i is associated to a *firing interval*, with 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, Prioritised Time Petri Nets (PrTPN)

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 larger priority (the priorities denoted by dotted arrows in the right-hand side of Figure 4). The precise semantics of PrTPN is defined in [3].

2.3 A Transformation from xSPEM to PrTPN

In [5], Combemale et al. 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:

- Three places characterise the value of *state* attribute (*NotStarted*, *InProgress*, *Finished*). One additional place called *Started* is added to record that the activity has been started (and may either be *inProgress* or *finished*).
- The three remaining places characterise 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.

Four transitions govern the behaviour of the modeled activity. We rely on the use of priorities among transitions to soundly deal with temporal constraints. As an example, the *deadline* transitions are given priority over the *finish* ones 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 *linkKind* attribute of the *WorkSequence*. In our example, *linkKind* equals *finishToFinish*, meaning that *pA* has to complete its execution before *pB* finishes; hence, the read-arc from the *pA_finished* place to the *pB_finish* transition.

2.4 An illustration of our Back-Tracing Algorithm

The PrTPN obtained from the transformation of a given xspem model can be analysed by the Tina model checker. For example, to exhibit an execution where both activities end is due time, we give the following formula to Tina:

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

The tool responds *false* and returns a counter example as a PrTPN execution (for sake of simplicity, the markings m_i are not detailed here):

```
(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, a *state* of the PrTPN is a pair consisting of a marking and of a *time-stamp*. Transitions are fired between states at their respective time-stamps: `pA_start` fires at time 0, then `pA_lock`, `pA_finish` and `pB_start` fire in sequence at time 3, and finally `pB_lock` and `pB_finish` fire in sequence at time 8.

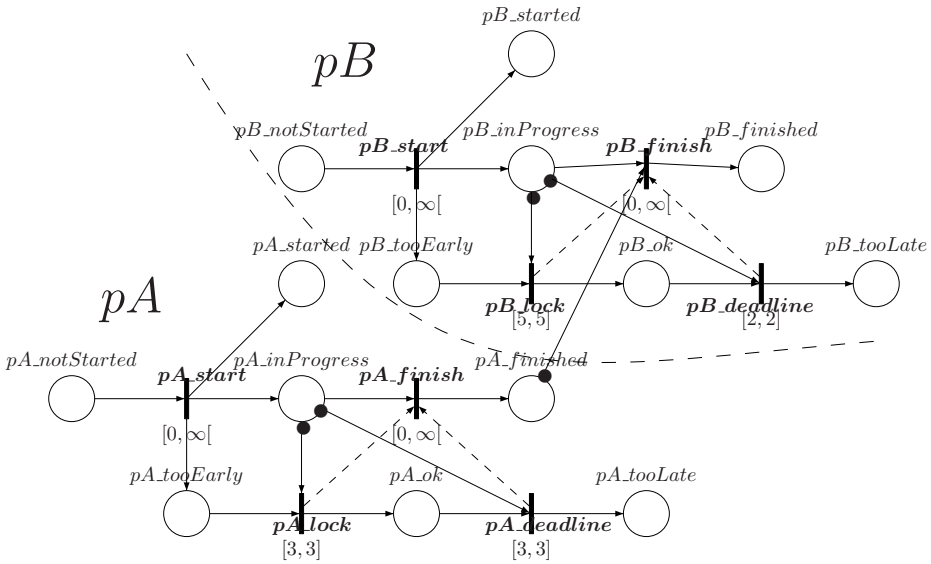

Figure 5: PrTPN obtained from the XSPEM model in Figure 2 (right). The initial marking has one token in each of the *pA_notStarted* and *pB_notStarted* places.

Our back-tracing algorithm (cf. Section 3.3) takes this input together with a *simulation 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

- their *globalTime* and time-stamp components are equal
- for each activity, the value of its *state* attribute is encoded by token in the corresponding place of the PrTPN,
- hen 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 exe-

cution, the maximum difference between two consecutive time-stamps 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 Formalising the Problem of Tracing Executions

In this section we formally define our 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. We then identify where the elements of our framework (i.e., the DSML, the target language, ...) stand in this picture, and formally state our backwards tracing problem. We identify a reasonably restricted solvable sub-problem, and propose and prove an algorithm for solving it.

3.1 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 $\rho(0) = a$.

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 strict) increasing function $\alpha : [0, \ldots, length(\rho)] \rightarrow \mathbb{N}$ with $\alpha(0) = 0$, such that for all $i \in [0, \ldots, length(\rho)]$, $(a_i, b_{\alpha(i)}) \in R$.*

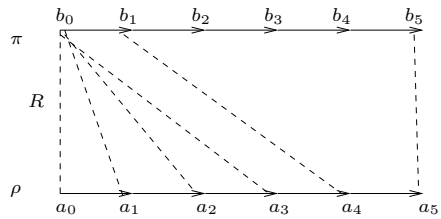

Figure 6: R-matching of sequences. The relation R is represented by dashed lines.

EXAMPLE 1 *In Figure 6 we represent two sequences ρ and π. A relation R is denoted using dashed lines. The function $\alpha : [0, \ldots, 5] \to \mathbb{N}$ 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.*

3.2 Formalising our framework

In our framework, $\mathcal{B} = (B, b_{init}, \to_{\mathcal{B}})$ 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 instance $m \in \mathcal{L}$ (in our example, the model depicted in the right-hand side of Figure 2). The instance 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 it 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 of the instance $\phi(m) \in \mathcal{L}'$ (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. Also, we do not need that the translation between $\mathcal{L}$ and $\mathcal{L}'$ be implemented using a model transformation, although this is convenient if the* DSML *$\mathcal{L}$ is defined using a metamodel (which it most often is). Finally, even if the translation is indeed implemented using a model transformation, we do not need to know that the transformation is a bisimulation to apply our back-tracing algorithm; indeed, we only need the transformation's output on a given input. However, if the transformation* was *verified - i,e, that R a* n-bisimulation *between* DSML *and target - meaning that to each execution in each* DSML *instance there exists a (n, R)-matching execution in the instance's traslation to the target language, and reciprocally - then we obtain the interesting (in our opinion) combination of theorem proving (for model transformation)/model checking (for model verification) described in Section 1.*

3.3 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 $R \subseteq A \times B$,

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**
2: **return** ε
3: **else**
4: $\alpha(0) \leftarrow 0, k \leftarrow 0, \pi \leftarrow b_{init}, S \leftarrow \{b_{init}\}$
5: **while** $k < length(\rho)$ and $S \neq \emptyset$ **do**
6: $k \leftarrow k + 1$
7: $S' \leftarrow R(\rho(k)) \cap \rightarrow^n_{\mathcal{B}} (last(\pi))$
8: **if** $S' \neq \emptyset$ **then**
9: 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
10: $\alpha(k) \leftarrow \alpha(k-1) + \ell$
11: $\pi_{\alpha(k-1)+1..\alpha(k)} \leftarrow \hat{\pi}_{1..l}$ ▷ effect of this assignment is null if $\ell = 0$
12: **end if**
13: $S \leftarrow S'$
14: **end while**;
15: **return** π
16: **end if**

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 6), ρ is $(5, R)$-matched (but not $(4, R)$-matched) by π.

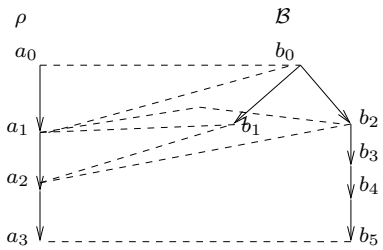

Figure 7: Attempting to match execution ρ.

3.4 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.

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.*

The proof of Theorem 1: is given in the Appendix. In particular, if there exists an execution in $exec(b_{init})$ that (n, R)-matches the whole sequence ρ then our back-tracing algorithm returns one; otherwise, the algorithm says there is none.

Example 2 *We illustrate several runs of our algorithm on the execution ρ depicted in the left-hand side of Figure 7, with the transition system $\mathcal{B}$ depicted in the right-hand side of the figure, and the relation R depicted using dashed lines. Let first the bound $n = 3$ in the algorithm. The set S is initialised 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 $S' = R(a_2) \cap \{b_2, b_3, b_4, b_5\} = \{b_2\}$. 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, instead 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.*

4 Implementation

Our implementation takes as input an execution given by the TINA toolsuite and returns as output an execution of the (initial) XSPEM model.

4.1 A generic implementation using Kermeta

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 meta-models 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].

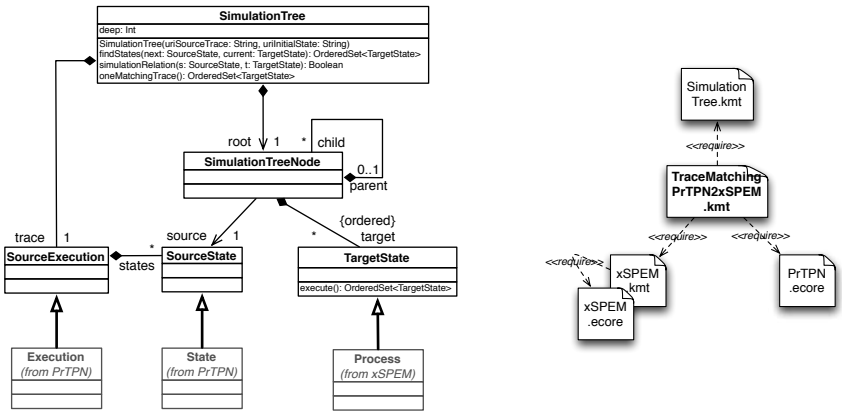

Figure 8: The generic *SimulationTree* class (left) and how to use it (right).

We implement in Kermeta a deterministic version of the non-deterministic Algorithm 1. Accordingly, our implementation relies on a generic tree-based structure (cf. Figure 8, left). The algorithm is generically defined in the *SimulationTree* method of the *SimulationTree* class. To use this method, the *SourceExecution* and its sequence of *SourceState* have to be specialised by an execution coming from the verification tool, and the *TargetState* have to be specialised by the corresponding concept in the DSML.

The *SimulationTree* method builds a tree, by calling at each tree node the method *findStates* to find the set of target states that simulate the next source state, according to the current target state and the operational semantics. This method depends on a given relation *R* defined in the method *simulationRelation* between *sourceState* and *targetState*. It also depends on a given maximal depth of search (n in Algorithm 1) defined by the attribute *deep*. Once the simulation tree is built, a DSML execution that simulates the execution provided by the verification tool can be generated by the method *oneMatchingTrace*.

4.2 On the example

One key feature of Kermeta is its ability to weave into a single metamodel, using the *require* keyword, existing metamodels, constraints, structural elements (metaclasses, classes, properties, and operations), and functionalities. This feature offers flexibility and modularity to developers by enabling them to easily manipulate and reuse existing metamodels.

Moreover, the static introduction operator *aspect* allows one to define these various aspects in separate units and to integrate them automatically into the metamodel. The composition is performed statically and the composed metamodel is typechecked to ensure the safe integration of all units.

We use both model weaving and static introduction to specialise our generic implementation of Algorithm 1 (as described previously) to the particular context of getting the XSPEM executions from a PrTPN execution. As described in Figure 8 (right), the *TraceMatchingPrTPN2xSPEM.kmt* program weaves the XSPEM metamodel (*xSPEM.ecore*) and its operation semantics (*xSPEM.kmt*), together with a metamodel for PrTPN (*PrTPN.ecore*) and our generic implementation for Algorithm 1 (*SimulationTree.kmt*). In addition to weaving the different artifacts, *TraceMatchingPrTPN2xSPEM.kmt* also defines the links between them. Thus we define the inheritance relations (cf. Figure 8, left) between *Execution* (from *PrTPN.ecore*) and *SourceExecution*, between *State* (from *PrTPN.ecore*) and *SourceState*, and between *Process* (from *xSPEM.ecore*) and *TargetState*. We also define the relation *R* by specialising the method *simulationRelation*) and the value of the attribute *deep*.

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. After running the method *SimulationTree*, we obtain using the method *oneMatchingTrace* the following output (and its corresponding model) :

5 Related work

The problem of tracking executions from a given target back to a domain-specific language has been addressed in several papers of the MDE community. Most of

```
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)})
```

Figure 9: Output on the given example

the proposed methods are either dedicated to only one pair metamodel/verification tool ([8], [10]) or they compute an "explanation" of the execution in a more abstract way (e.g., in [7], where the abstract formalism is Message Sequence Charts). In [9], 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. This relation is essentially structural (syntactical) - it is not *formally* related to the DSML's operational semantics. By contrast, our approach does not need any instrumentation of the model transformations code, and most importantly, it is formally grounded on operationally 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 focus on proving correctness, and does not provide any useful information if the verification fails. Finally, the Counterexample-Guided Abstraction Refinement (CEGAR) verification method ([4]) also consists in matching model-checking counterexamples to program executions. However, CEGAR attempts to match more "abstract" executions to more "concrete" ones, whereas we currently do the opposite.

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 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 the maximum "difference of granularity" between semantics of the DSML and of the target language. Then, given an finite execution ρ of the target language (e.g., a counterexample to a safety property, or

program crash log), our algorithm returns a (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 illustrate our algorithm on an example where the DSML is XSPEM, a timed language of activities and processes, and the target language is Prioritised Time Petri Nets (PrTPN), the input language of the TINA model checker. We implement our algorithm in Kermeta, a framework for defining operational semantics of DSML and model transformations between DSML.

Regarding future work, the main direction is to exploit the combination theorem proving (for model transformation)/model checking (for model verification) 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*, Japan, December 2007. IEEE Computer Society.

[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 *ASE*, pages 267–270. IEEE Computer Society, 2002.

[7] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Ws-engineer: A model-based approach to engineering web service com- positions and choreography. In *Proc. Int. of Test and Analysis of Web Services*, page 87Â119. Springer Berlin Heidelberg, 2007.

[8] E. Guerra, J. de Lara, A. Malizia, and P. DÂ´az. Supporting user-oriented analysis for multi-view domain-specific visual languages. *Information & Software Technology*, 51(4):769Â784, 2009.

[9] A Hegedüs, G Bergmann, I Ráth, and D Varró. Back-annotation of simulation traces with change-driven model transformations. In *Proc. Int. of the 8th International Conference on Software Engineering and Formal Methods (SEFM'10)*, Pisa, Italy, September 2010.

[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 L. Briand and C. Williams, editors, *Proceedings of the 8th IEEE/ACM International Conference on Model Driven Engineering Languages and Systems (MoDELS)*,

volume 3713 of *Lecture Notes in Computer Science*, pages 264–278, Montego Bay, Jamaica, October 2005. Springer.

[12] Object Management Group, Inc. *Meta Object Facility (MOF) 2.0 Core Specification*, January 2006. Final Adopted Specification.

[13] Object Management Group, Inc. *Object Constraint Language (OCL) 2.0 Specification*, May 2006.

[14] Object Management Group, Inc. *Software Process Engineering Metamodel (SPEM) 2.0*, March 2007.

[15] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. pages 151–166. Springer, 1998.

[16] José Eduardo Rivera, Cristina Vicente-Chicote, and Antonio Vallecillo. Extending visual modeling languages with timed behavior specifications. In Antonio Brogi, João Araújo, and Raquel Anaya, editors, *CIbSE*, pages 87–100, 2009.

Appendix: Proof of Theorem 1

The proof is based on the two following technical lemmas. The first one says that matching is in some sense compositional.

LEMMA 1 *With the notations of Definition 4, for all sequences $\rho, \rho' \in A^+$ and executions π, π' of $\mathcal{B}$ such that $\pi\pi'$ is also an execution of $\mathcal{B}$, if π (n, R)-matches ρ and π' (n, R)-matches ρ', then $\pi\pi'$ (n, R)-matches $\rho\rho'$.*

PROOF : Let $\alpha : [0..length(\rho)] \to \mathbb{N}$ and $\alpha' : [0..length(\rho')] \to \mathbb{N}$ be the functions in Definition 4, which ensure that π (n, R)-matches ρ and π' (n, R)-matches ρ'. To prove that $\pi\pi'$ (n, R)-matches $\rho\rho'$ we use the function $(\alpha\alpha') : [0..length(\rho) + length(\rho')] \to \mathbb{N}$ defined by $(\alpha\alpha')(i) = \alpha(i)$ if $0 \leq i \leq length(\rho)$, and $(\alpha\alpha')(i) = \alpha'(i - length(\rho) - 1) + 1 + \alpha(length(\rho))$ for $length(\rho) < i \leq length(\rho) + length(\rho')$. ■

The second lemma states a useful invariant of our back-tracing algorithm.

LEMMA 2 *In Algorithm 1, just before Label 5, the following invariant holds: $S \neq \emptyset \Longrightarrow \rho_{0..k}$* is $(n, R)-$matched by $\pi_{0..\alpha(k)}$.

PROOF : (sketch) For the base case: the first time the algorithm arrives at Label 5, $S = \{b_{init}\} \neq \emptyset$, $k = 0$, $\alpha(0) = 0$; and $\rho_{0..0} = a_{init}$ is matched by $\pi_{0..0} = b_{init}$.

For the inductive step: assume that the statement holds for $k = N$, we prove it for $k = N + 1$. Consider the set S_{N+1} in out statement, i.e., for the $k = N + 1$-th visit of the algorithm at Label 5. We have $S_{N+1} = R(\rho(N + 1)) \cap \to^n_{\mathcal{B}} (\pi_N(length(\pi_N)))$ thanks to the assignment at Label 7($last_N = \pi_N(length(\pi_N))$). We have to prove $S_{N+1} \neq \emptyset \Longrightarrow \rho_{0..N+1}$ is $(n, R)-$matched by $\pi_{0..\alpha(N+1)}$. Let then $S_{N+1} \neq \emptyset$. In the algorithm, at the $N + 1$-th iteration, the condition $S' \neq \emptyset$ of the `if` statement at Label 8 evaluates to *true*; this is because, during that iteration $S' = S_{N+1}$. Then, the `choose` statement chooses an execution $\hat{\pi}$ starting from the state $\pi(last)$ (where π ended at the last iteration) and ending in some state in S'; and by construction of that execution, $\hat{\pi}_{1..l}$ $(n$-$R)$-matches $\rho_{N..N+1}$. Hence, by compositionnality of matching (Lemma 1), $\rho_{0..N+1}$ is $(n$-$R)$-matched by $\pi_{0..\alpha(N)}\hat{\pi}_{1..l}$. Finally, we notice that at our point of interest - the $N + 1$-th visit to Label 5 - we have $\pi_{0..\alpha(N)}\hat{\pi}_{1..l} = \pi_{0..\alpha(N+1)}$ thanks to the assignments at Labels 10 and 11 during the N-th iteration. This concludes the proof. ■

We are now ready to Prove Theorem 1, reproduced below for convenience.

Theorem 1 (Algorithm for matching executions) *C*onsider a transition system $\mathcal{B} = (B, b_{init}, \to_{\mathcal{B}})$, 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.

PROOF : (Sketch) There are two cases, corresponding to the outermost `if-then-else` statement in the algorithm. If $(a_{init}, b_{init}) \notin R$ then even the first state a_{init} of the

sequence ρ cannot be matched. Hence, our procedure returns ε, i.e., the empty sequence, which in the current situation is indeed the longest prefix of the execution ρ that can be matched, and the proof is done in this case. Otherwise, $(a_{init}, b_{init}) \in R$. Now, assume that the algorithm is currently visiting Label 5 for the last time - i.e., it "stands just before" its last (ever) evaluation of the condition of the `while` loop. This means that the `while` loop's condition evaluates to *false*. Hence, $k = length(\rho)$ or $S = \emptyset$. If $k = length(\rho)$ and $S \neq \emptyset$ then using Lemma 2 we obtain $\rho_{0..length(\rho)}$ is simulated by $\pi_{0..\alpha(length(\rho))}$, meaning that the whole sequence ρ is simulated by the execution π returned by the algorithm, and the proof is done in this case.

The remaining case is $S = \emptyset$. In this case, notice that the current visit to Label 5 cannot be the *first* one, since at the first visit, $S = \{b_{init}\} \neq \emptyset$. Hence, there must have been a previous loop iteration, and at the corresponding previous visit to Label 5, we had $S \neq \emptyset$. The previous visit to Label 5 corresponds to the iteration $N - 1$, where N is the current value of k. Using Lemma 2 we obtain that $\rho_{0..N-1}$ is $(n, R)-$matched by $\pi_{0..\alpha(N-1)}$. Since the iteration $N - 1$ was the last loop iteration, the matching of ρ stopped at a prefix of length $N - 1 < length(\rho)$, and there is no further extension of the execution π to match "more" of the execution ρ, and the proof of this case is done as well. ■

Contents

1 Introduction **3**

2 Running Example **4**
2.1 The xSPEM language and its operational semantics 5
2.2 Prioritised Time Petri Nets 6
2.3 A Transformation from xSPEM to PRTPN 8
2.4 An illustration of our Back-Tracing Algorithm 8

3 Formalising the Problem of Tracing Executions **10**
3.1 Transition systems and execution matching 10
3.2 Formalising our framework 11
3.3 The back-tracing problem 12
3.4 Back-tracing algorithm 13

4 Implementation **14**
4.1 A generic implementation using Kermeta 14
4.2 On the example 15

5 Related work **15**

6 Conclusion and Future Work **16**

Centre de recherche INRIA Lille – Nord Europe
Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399