A Generative Approach to Define Rich Domain-Specific Trace Metamodels

Erwan Bousse, Tanja Mayerhofer, Benoit Combemale, Benoit Baudry

A Generative Approach to Define Rich Domain-Specific Trace Metamodels

Erwan Bousse[1], Tanja Mayerhofer[2], Benoit Combemale[3], and Benoit Baudry[3]

[1] University of Rennes 1, France
erwan.bousse@irisa.fr
[2] Vienna University of Technology, Austria
mayerhofer@big.tuwien.ac.at
[3] Inria, France,
{benoit.combemale, benoit.baudry}@inria.fr

Abstract. Executable Domain-Specific Modeling Languages (xDSMLs) open many possibilities for performing early verification and validation (V&V) of systems. Dynamic V&V approaches rely on execution traces, which represent the evolution of models during their execution. In order to construct traces, *generic trace metamodels* can be used. Yet, regarding trace manipulations, they lack both *efficiency* because of their sequential structure, and *usability* because of their gap to the xDSML. Our contribution is a generative approach that defines a *rich* and *domain-specific* trace metamodel enabling the construction of execution traces for models conforming to a given xDSML. Efficiency is increased by providing a variety of navigation paths within traces, while usability is improved by narrowing the concepts of the trace metamodel to fit the considered xDSML. We evaluated our approach by generating a trace metamodel for fUML and using it for semantic differencing, which is an important V&V activity in the realm of model evolution. Results show a significant performance improvement and simplification of the semantic differencing rules as compared to the usage of a generic trace metamodel.

1 Introduction

In recent years, a lot of efforts have been made to provide facilities to design executable Domain-Specific Modeling Languages (xDSMLs) [5, 19, 22]. Executability of models opens many possibilities in terms of early dynamic verification and validation (V&V) of models, such as debugging [6], runtime verification [16], model checking [4], and semantic differencing [15].

A central concept in dynamic V&V approaches is the *execution trace*, which is the representation of the evolution of a model's state during an execution. While a trace can take numerous forms, we focus in this work on traces containing a sequence of *states* of the model being executed and *event occurrences* related to state changes. All previously mentioned V&V approaches rely on traces: model checking consists in verifying a property of a model by analyzing all its possible traces and providing traces as counter-examples; runtime verification consists in

checking whether or not a trace satisfies a property; debuggers require traces to replay faulty scenarios; semantic differencing consists in comparing traces of two models in order to understand the semantic variations between them.

Therefore, there are at least two significant prerequisites for the V&V of executable models: (1) the definition of a *trace metamodel* to represent traces, and (2) facilities to manipulate large traces efficiently, *i.e.* with good scalability in time. The first prerequisite can be fulfilled by using an existing *generic trace metamodel* (e.g. Compact Trace Format defined in [10]), which can be adopted for any executable language. However, such metamodels cannot take the domain-specific concepts of an xDSML explicitly into account, which makes the development of domain-specific analyses of traces more difficult. To cope with that, a *domain-specific* trace metamodel that is specific to an xDSML (e.g. fUML trace metamodel defined in [18]) can be used. Yet, designing such a metamodel is a time consuming and error-prone task. Also, regarding the second prerequisite, existing trace metamodels only offer to explore a trace by enumerating all states and event occurrence one by one, which can only scale linearly at best.

In this paper, we propose a new way to define domain-specific trace metamodels for xDSMLs through two contributions: (1) a generic approach to automatically derive a domain-specific trace metamodel for a given xDSML by analyzing its definitions of execution states and events; (2) facilities to navigate efficiently within a trace conforming to such a generated metamodel by providing a variety of navigation paths. We evaluated this work by generating a rich and domain-specific trace metamodel for a real world xDSML, namely fUML [21], and by using it for *semantic differencing* [15]. The results show a simplification of the semantic differencing rules and better execution times when using the new trace structure, compared to the usage of a generic trace metamodel.

The remaining sections are organized as follows. Section 2 motivates the problem domain and explains our ideas. Section 3 presents what is executable metamodeling. Section 4 presents our contribution. Section 5 discusses the evaluation of our approach in the domain of semantic differencing. Finally, Section 6 discusses related work and Section 7 concludes the paper.

2 Motivation and Problem Statement

In this section, we first introduce two requirements we identified for trace metamodels, and then present our ideas for complying with these requirements.

2.1 Requirements for a Trace Metamodel

We consider a *metamodel* to be an object-oriented model defining a particular domain. Therefore it is composed of classes, which consist of properties. A property is either an attribute (typed by a datatype) or a reference to another class. A *model* is a set of objects that conforms to a metamodel. Conformity means that each object in the model is an instance of one class defined in the metamodel. An *object* is composed of fields, each representing the object's values for one property of the corresponding class.

In our previous work [3], we highlighted a number of issues that must be considered when constructing and manipulating execution traces. In particular, the potentially large size of a trace compromises the capacity to query it in a reasonable time. For instance, if some element of an executable model only changed at the end of an execution, we might still have to iterate through all states stored in the corresponding trace before noticing that change. Another issue is to manage the manipulation complexity of trace models. Trace analyses can either be *generic* (e.g. comparing the number of different states or the amount of event occurrences), or *domain-specific* (e.g. determining how many tokens traversed a Petri net place). In the former case, manipulations are simple and the structure or content of the trace has little influence on the complexity of the analysis task. However, in the latter case, manipulations handle domain-specific data that can be arbitrarily complex depending on the considered xDSML. Hence, in such cases, defining the right analysis can be error-prone and difficult. A good illustration of these issues is *semantic differencing* [15]. First, it is a hard problem because traces tend to be large and therefore expensive to process. But more importantly, semantic differencing consists in doing *domain-specific* analyses of traces, since they are written according to the semantics of a specific xDSML, and may therefore rely on complex domain-specific data. To sum up, we consider the following requirements on a good trace metamodel:

Scalability in time. It should provide good scalability in time when manipulating large traces, *i.e.* traces with a lot of state changes.

Usability. It should provide good usability both for generic analyses and domain-specific analyses, e.g. by facilitating the manipulation of traces containing complex domain-specific execution data.

Note that scalability in space or handling distributed systems constitute other important issues which we presented in [3]. In this paper, we only focus on the two aforementioned requirements and other issues are out the scope of this work.

2.2 From Generic to Rich Domain-Specific Trace Metamodels

Considerable effort has been made to design generic trace formats to represent traces of programs or models conforming to any possible language [1, 8, 10, 7]. However, while they may have interesting characteristics (modeling of logical time, handling of distributed systems, etc.), and may be compatible with generic trace analysis tools, they do not deal with the requirements previously mentioned. First, they do not provide facilities to browse traces efficiently: the only way to navigate in the trace is by enumerating each captured execution state one by one. Second, genericity implies a gap between the trace concepts defined by a trace format and the domain concepts specific to a particular xDSML. This semantic gap has a significant impact on usability. Moreover, most of these formats only capture events that occurred during an execution, such as the start of an operation execution, and lack a representation of the execution state, such as the values of the variables of a program. This is due to the large size of traces,

which leads to limiting the amount of information stored in them. Yet, as stated previously, we focus in this paper on execution traces containing both states and events. Indeed, traces containing only events need to be replayed in order to reconstruct the states, whereas traces containing states allow direct analyses.

To better comply with the requirements (*i.e.* scalability in time and usability), the underlying intuition of the approach we propose is the following: considering that the benefits of narrowing the scope of a language to a domain are well known [12], defining a trace metamodel specific to a language should bring similar advantages. In particular, by providing concepts of the xDSML directy in the trace metamodel, the usability of the trace should be improved. In previous work [18], we followed this idea by defining manually a complete trace metamodel for fUML and recognized the many benefits such a domain-specific trace metamodel brings. Yet, defining this metamodel was tedious and error-prone, and we observed redundancies between the trace metamodel and the concepts defined in fUML. These redundancies are simply explained: the definition of an xDSML specifies what the state of a model is during its execution as part of the xDSML's semantics [5], and a trace metamodel directly requires such a notion of state. Hence, a first difficulty is the definition of a domain-specific trace metamodel, which can possibly be mitigated by analyzing how the execution state is defined in the xDSML. A second difficulty is that while generic trace metamodels can benefit from existing trace analysis and visualization tools, domain-specific ones require specific tooling. Therefore, our first idea is to go from *generic* trace metamodels to a *generic meta-approach* to define domain-specific trace metamodels. More precisely, we propose to automatically derive a complete domain-specific trace metamodel using the definitions of execution state and events of an xDSML. Such a generic generative approach would allow both to avoid the difficulty of defining domain-specific trace metamodels, and to automatically provide suitable tools for manipulating domain-specific traces.

The second intuition is that while a trace is generally only seen as a *sequence* of states and events, there are in fact many imaginable ways to browse a trace. Having more navigation paths at disposal could be a great way to browse traces more efficiently. An example is finding the next value change of a given model element regardless of any other state changes in the model. Such query can be done easily by traversing the complete trace, yet reifing it as a *navigation path* dedicated to the investigated model element would avoid browsing the whole trace. Henceforth, our second idea is to create *rich* trace metamodels, *i.e.* metamodels that provide many navigation paths to explore a trace.

In a nutshell, our proposal is an approach to automatically generate *rich* and *domain-specific* trace metamodels for an existing xDSML. We evaluate the relevance of our contribution with respect to the following research questions:

RQ#1: Can a rich domain-specific trace metamodel provide better execution times for trace manipulations as compared to a generic trace metamodel?

RQ#2: Can a rich domain-specific trace metamodel simplify the definition of domain-specific analyses of traces as compared to a generic trace metamodel?

3 From Executable Metamodeling to Execution Traces

In this section, we first present what constitutes an xDSML, then give an example of an xDSML, and finally provide our definition of execution trace.

3.1 Executable Metamodeling

While the purpose of metamodeling is to define languages, *executable* metamodeling also aims at including execution semantics in the language definition. This is done through executable Domain-Specific Modeling Languages (xDSMLs), which are languages that include the definitions of the *execution state* of a model conforming to the language, and *execution semantics* that change this state.

To define the execution state of a model, we consider that an abstract syntax metamodel can be extended into an *execution metamodel* with new properties and classes. To this end, a mechanism equivalent to the well-known *package merge* operation can be used. Note that in practice, existing tools and approaches use different but similar extension mechanisms—e.g. Kermeta [13] uses aspect weaving, xMOF [19] uses generalization, Hegedüs et al. [11] use separate classes.

There are two general approaches to define execution semantics: *translational* and *operational* semantics. Translational semantics consists in translating a model m into a model m' to be executed. This means that the execution state of m must be constantly synchronized with the execution state of m'. Operational semantics consist in a set of transformation rules that directly work with the execution state of m. In this paper, we only deal with operational semantics.

Furthermore, we consider two additional elements of an xDSML. First, in order to execute a model originally expressed with the abstract syntax metamodel, the *initialization function* translates such a model into a model conforming to the execution metamodel. Second, the *event metamodel* defines the events that may occur between two states during the execution. Each event corresponds to a specific transformation rule of the semantics. Such a metamodel can be directly inferred from the semantics (e.g. an event per transformation rule) or manually defined for a subset of the rules. Note that our approach does not require this metamodel, and that in such a case it won't provide event-based facilities to construct or manipulate event occurrences in a trace.

Definition 1. *An* xDSML *is defined by:*

- *An* abstract syntax*, which is a metamodel. We call* immutable *a property introduced in this metamodel. At the model level, we also call immutable an object's field based on an immutable property.*
- *An* execution metamodel*, which extends the abstract syntax by package merge. We call* mutable *a property introduced in this metamodel. At the model level, we also call mutable an object's field based on a mutable property.*
- Operational semantics*, which are a set of transformation rules that modify a model conforming to the execution metamodel by changing values of mutable fields and by creating/destroying instances of classes introduced in the execution metamodel.*

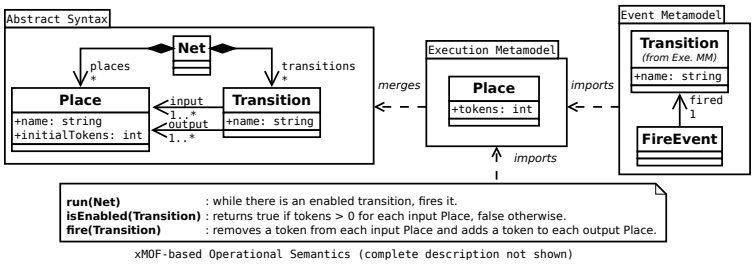

Fig. 1. Petri net xDSML defined using metamodels and xMOF.

- *An* event metamodel*, which is a metamodel containing events that may occur during an execution. Each event is related to a transformation rule. An instance of an event class is called an* event occurrence*.*
- *An* initialization function*, which given a model conforming to the abstract syntax, returns a model conforming to the execution metamodel.*

Figure 1 shows an example of a Petri net xDSML. On the top left corner, its abstract syntax is depicted with three classes Net, Place and Transition. Next to the abstract syntax, the execution metamodel is shown. It extends the class Place using *package merge* with a new mutable property `tokens`. The initialization function (not shown) transforms each original object (e.g. a Place object without a `tokens` field) into an executable object (e.g. a Place object with a `tokens` field) as defined in the execution metamodel. It also initializes each `tokens` field with the value of `initialTokens`. At the bottom, the rules defined in the operational semantics with xMOF [19] are depicted. On the right, the event metamodel is shown containing a single class FireEvent corresponding to the *fire* rule.

3.2 Execution Trace

While execution traces can take various forms, we consider in this work that an *execution trace* is a sequence of states and event occurrences. Thereby, an execution state contains all the values of all the mutable fields of a model, *i.e.* the values of the fields defined by properties introduced in the execution metamodel. At each application of a transformation rule defined by the operational semantics, the execution state of the model changes. As rules are responsible for state changes, events associated to these rules occur between states.

Definition 2. *An* execution trace *is a sequence of execution states and event occurrences. While the first state is given by the initialization function, each other state is reached through the application of a transformation rule and contains at least the values of the mutable fields of the executed model. If this transformation rule is associated to an event, there is a corresponding event occurrence preceding the state.*

4 Generating Rich Domain-Specific Trace Metamodels

To answer RQ #1 and RQ #2, we propose a generative approach to define rich and domain-specific trace metamodels that provide facilities for efficiently processing traces. In this section, we present this approach by first presenting the challenges we had to overcome, second explaining our generation procedure based on the introduced Petri net xDSML, third discussing the resulting benefits of the approach, and fourth providing details on our implementation.

4.1 Observations and Challenges

There are many possible ways to generate a domain-specific trace metamodel for an xDSML. Regarding the execution states, a simple yet working idea is to reuse the complete execution metamodel of the xDSML in the trace metamodel. As the executed model conforms to the execution metamodel, we can *clone* it at each execution step and store it as a state in the trace. However, this solution has multiple drawbacks. First, by duplicating the whole model to store each execution state, we create redundancies between the states for both immutable fields (as they never change) and mutable fields (as they may not change in each step). Scalable model cloning [2] would mitigate this issue at runtime by sharing immutable data among clones, but would not be of any help when serializing the trace. Second, the mutable fields we are interested in are scattered among the immutable fields, which may require complex queries to access them within a state. These issues compromise RQ #2. Lastly, such a trace metamodel does not provide any efficient way to browse a trace, since the only possibility is to enumerate each state one by one. Thus it would be, for instance, tedious and inefficient to look for the next value of a given mutable field, compromising both RQ #1 and RQ #2. From these observations, we identified three *challenges*:

(1) Narrowing the concepts introduced in a trace metamodel, e.g. by focusing on the mutable properties of the execution metamodel.
(2) Avoiding redundancy in traces, e.g. by not storing the same value twice consecutively for a given mutable field.
(3) Providing alternative navigation paths, e.g. among the sequence of values of a specific mutable field.

4.2 Trace Metamodel Generation

Algorithm 1 shows our trace metamodel generation procedure. Note that the algorithm is simplified for illustration purposes, meaning that some parts are reduced to functions, and that special cases, such as abstract classes, are not considered. The inputs of the procedure are the abstract syntax (mm_{as}), the execution metamodel (mm_{exe}) and the event metamodel (mm_{events}) of an xDSML. The procedure is independent from executable models, since the obtained metamodel is valid for any execution trace of any model of the considered xDSML. Note that the classes Trace and ExecutionState are always created (lines 2–3) and

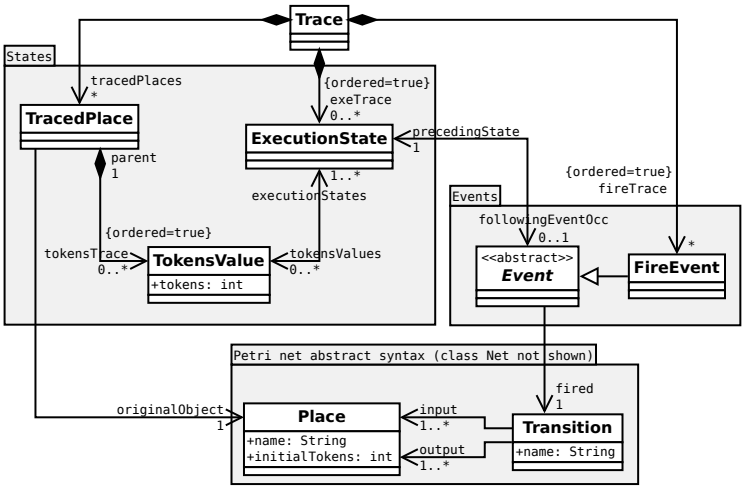

Fig. 2. Trace metamodel generated for the Petri net xDSML.

Algorithm 1: Trace metamodel generation (simplified)

Input: mm_{as}, mm_{exe}, mm_{events}
Result: mm_{trace}: the trace metamodel

begin
  $c_{\text{trace}}, c_{\text{exeState}} \leftarrow$ createBaseGenericClasses()
  $mm_{\text{trace}} \leftarrow \{c_{\text{trace}}, c_{\text{exeState}}\}$
  foreach $c_{\text{exe}} \in \{c \mid \text{containsMutableProperties}(c)\}$ **do**
    $c_{\text{traced}} \leftarrow$ createClass()
    $mm_{\text{trace}} \leftarrow mm_{\text{trace}} \cup \{c_{\text{traced}}\}$
    c_{trace}.createReferenceTo(c_{traced}, [0..*], *unordered*)
    if containsImmutableProperties(c_{exe}) **then**
      $c_{\text{orig}} \leftarrow$ getClassFromAbstractSyntax(c_{exe})
      c_{traced}.createReferenceTo(c_{orig}, [1..1])
    foreach $p \in$ getMutablePropertiesOf(c_{exe}) **do**
      $c_{\text{value}} \leftarrow$ createClass()
      $mm_{\text{trace}} \leftarrow mm_{\text{trace}} \cup \{c_{\text{value}}\}$
      c_{value}.`properties` $\leftarrow$ { copyProperty(p) }
      c_{traced}.createReferenceTo(c_{value}, [0..*], *ordered*)
      c_{value}.createReferenceTo(c_{traced}, [1..1])
      c_{exeState}.createReferenceTo(c_{value}, [0..*], *unordered*)
      c_{value}.createReferenceTo(c_{exeState}, [1..1])
  if $mm_{\text{events}} \neq \varnothing$ **then**
    $c_{\text{event}} \leftarrow$ createEventClass()
    $mm_{\text{trace}} \leftarrow mm_{\text{trace}} \cup \{c_{\text{event}}\}$
    c_{exeState}.createReferenceTo(c_{event}, [0..1])
    c_{event}.createReferenceTo(c_{exeState}, [1..1])
    foreach $c_{\text{exeevent}} \in mm_{\text{events}}$ **do**
      $c_{\text{eventcopy}} \leftarrow$ copyClass(c_{exeevent})
      $mm_{\text{trace}} \leftarrow mm_{\text{trace}} \cup \{c_{\text{eventcopy}}\}$
      $c_{\text{eventcopy}}$.`superTypes` $\leftarrow c_{\text{eventcopy}}$.`superTypes` $\cup \{c_{\text{event}}\}$
      c_{trace}.createReferenceTo($c_{\text{eventcopy}}$, [0..*], *ordered*)
  replaceReferencesToExecutionMM($mm_{\text{trace}}, mm_{\text{as}}, mm_{\text{exe}}$)

that the class Event is created only when the event metamodel is not empty (lines 20–21). In the following paragraphs, we explain the generation procedure based on the Petri net xDSML, starting with trace concepts for capturing the smallest unit of an execution state, *i.e.* an object's field values, up to the concepts for capturing the complete execution state of a model. The trace metamodel generated for the Petri net xDSML is shown in Figure 2.

Capturing the Values of Fields (lines 11–14). At any given point in time, all mutable fields of an object of the executed model have a *value*. To represent such a value in a trace, we create one class per mutable property of the execution metamodel, and we copy this mutable property into this new class (lines 12–14). This enables us to capture each value of a mutable field as an instance of this generated class. For Petri nets this means creating one class called TokensValue for the property `tokens`. Thereby, we precisely narrow the trace metamodel to the mutable part of the execution metamodel (challenge 1).

Capturing the States of Objects (lines 4–10, 15–16). The state of an object of the executed model at any point in time is defined by the values of all its mutable fields. To represent all states reached by an object, we create one class for each class of the execution metamodel containing at least one mutable property (lines 4–5). In addition, we make all instances of these generated classes accessible through a single instance of the class Trace. For Petri nets this means creating a class TracedPlace for the class Place, and a reference `tracedPlaces` from the class Trace. An instance of such a generated class shall contain all values reached by all mutable fields of an object of the considered type in chronological order. This is achieved by creating an ordered unbounded reference to each corresponding generated value class discussed previously (line 15). For Petri nets this means generating a reference `tokensTrace` for the class TracedPlace to the class TokensValue. When creating an execution trace, one TracedPlace object will be created per Place object, each storing a sequence `tokensTrace` of all the values reached by the `tokens` field of the respective Place object. A first benefit of this structure is that we avoid redundancy by creating a single object per value change of a mutable field (challenge 2). A second benefit is that such sequences provide additional navigation paths in the trace, making it possible to directly access all changes of one specific mutable field (challenge 3). The last concern for capturing the state of an object is that the object may also contain *immutable* fields, which remain an important piece of information. Since the corresponding immutable properties are all defined in a class introduced in the abstract syntax, our solution is to create a reference to this class (lines 8–10). For Petri nets this means adding a reference `originalObject` for the traced class TracedPlace to the class Place of the abstract syntax. A TracedPlace object is thus linked to the Place object whose states it captures.

Capturing the State of the Model (lines 17–18). An execution state can be seen as the n-tuple of the values of all mutable fields in an executed model at a given point in time. However, n is not xDSML-specific, but *model*-specific, as the number of mutable fields depends on the number of objects in the executed model. For instance, in our Petri net xDSML, n equals the

number of `tokens` fields of one given model, *i.e.* the number of Place objects. In addition, n can change during the execution, as new objects can be created for classes introduced in the execution metamodel. To represent this n-tuple, we create a bidirectional reference between each generated value class and the class ExecutionState, which represents one execution state of a model. By that means, an execution state references an unbounded set of values of mutable fields. For Petri nets this means introducing the references `tokensValue` and `executionState` between the classes ExecutionState and TokensValue.

Capturing Event Occurrences (lines 19–28). An event may occur between two execution states if its corresponding transformation rule was responsible for the respective state change. This is represented by the references `precedingState` and `followingEventOcc` between the classes ExecutionState and Event (lines 22–23). Since the abstract class Event represents any kind of event, we need to copy all classes from the event metamodel into the trace metamodel and add generalization links to the class Event (lines 25–27). For Petri nets this means copying the class FireEvent and making it a subclass of Event. In the same manner as for values, all event occurrences are stored chronologically within the unique Trace object (line 28). For Petri nets this means having an ordered reference `fireTrace` in the Trace class to the class FireEvent. This gives direct access to all event occurrences of a specific event in chronological order, which is an interesting additional navigation path for a trace (challenge 3).

Replacing References to the Execution Metamodel (line 29). When mutable properties and event classes were copied in the trace metamodel, this included copying references to classes of the execution metamodel. Yet, such classes may contain mutable properties that were already copied in the trace metamodel. To avoid having twice the same concept in the trace metamodel (challenge 1) or twice the same value stored in a trace (challenge 2), our solution is to replace all references to the execution metamodel by references either to the abstract syntax or to classes representing the states of objects (e.g. TracedPlace). This is indicated by the function *replaceReferencesToExeMM* (line 29).

Example Trace. Figure 3 shows a rich domain-specific trace of a Petri net model. Note that to construct such a trace, one must instrument the semantics of an xDSML, which is out the scope of this paper. In the upper part, we use the concrete syntax of Petri nets to show the execution. In the lower part, we use an object diagram to show the content of the executed model and of the trace at the end of the execution. In the example model, the transitions $t1$ and $t2$ are fired, leading to a trace with three states and two event occurrences. To represent the states, three ExecutionState objects are linked to a set of TokensValue objects, which represent the marking of the Petri net. Some are linked to FireEvent objects, which represent the firing of $t1$ and $t2$. There is one `tokensTrace` sequence per `tokens` field: $(1, 0)$ for $p1$ and $p2$, $(0, 1, 0)$ for $p3$ and $(0, 2)$ for $p4$ (not shown). These sequences constitute alternative navigation paths that facilitate queries, e.g. we can find the maximum number of tokens reached by $p1$ by reading only two values. Moreover, we can go from one such sequence back to the complete trace, e.g. to find all states in which $p4$ had at least two tokens.

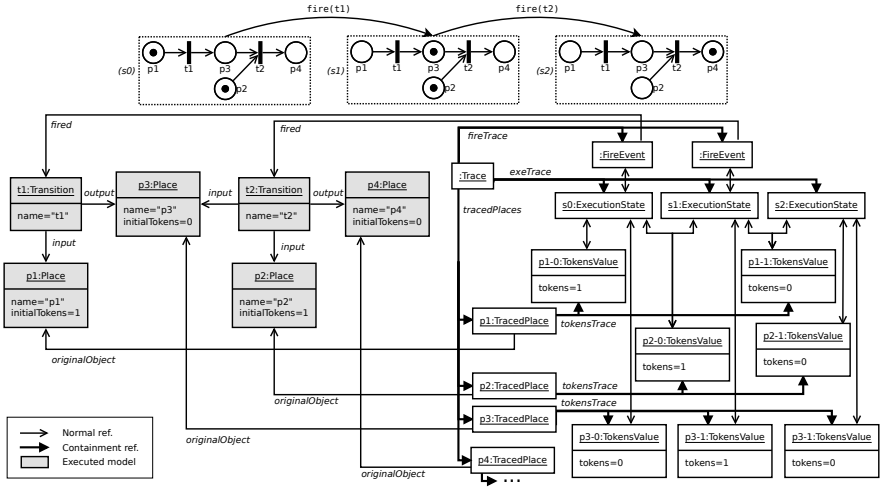

Fig. 3. Example of Petri net model and rich domain specific execution trace.

Regarding events, we have access to the list of the fired transitions by browsing the fireEvent trace, e.g. to find states following directly a firing of $t2$.

Note that this example does not illustrate the creation or deletion of objects within an execution. Such case is handled with the help of the variable number of references from a ExecutionState element to values. Hence, an object created just before a state means that this state and the following ones have references to the values of this object. Likewise, an object deleted just before a state means that this state and the following ones have no references to its values.

4.3 Resulting Benefits

Among all the concepts we create in a trace metamodel, some are generic (e.g. Trace), but the others are specific to the xDSML (e.g. TokensValue). Also, we make sure not to have any redundancy of concepts. In other words, we *precisely define* the structure of execution traces of models conforming to an xDSML. Thereby, domain-specific analyses of traces have direct access to these concepts, and do not have to rely on complex queries or introspection to use domain-specific data. We aim by that means to provide good usability (RQ #2).

In addition, we provide several *navigation paths* for browsing traces. Indeed, we create for each mutable property (e.g. `tokens`) and each event (e.g. FireEvent) of an xDSML a dedicated navigation path (e.g. `tokensTrace` and `fireTrace`). This allows to enumerate each value of a particular field, or each event occurrence of a particular event, without having to enumerate all the states of the trace. Moreover, all values and event occurrences are connected through execution states, allowing to go from one navigation path to another. These navigation facilities offer better usability and scalability in time (RQ #1 and RQ #2)

4.4 Implementation

We implemented our approach for the Eclipse Modeling Framework (EMF). Our completely generic trace metamodel generator is written using EMF and Xtend. Parts of our prototype are specific to the xMOF framework, including both a transformation that derives an event metamodel from xMOF semantics and a trace builder that can construct a trace from the execution of any xMOF-based model. For more information, the source code (EPL 1.0 licensed) is available at our project web page: `https://gforge.inria.fr/projects/lastragen/`.

5 Evaluation

In this section, we present the evaluation of our approach, which consists in a case study applying rich and domain-specific traces for semantic differencing. We first introduce our semantic differencing framework, then present our case study, and finally discuss the obtained results regarding RQ #1 and RQ #2.

5.1 Semantic Differencing

Semantic differencing of models is concerned with identifying differences among distinct versions of models. Thereby, not only syntactic differences among models are taken into account, but differences in their semantics are especially considered. In previous work [15], we have proposed a semantic differencing approach for xDSMLs, which is based on the analysis of execution traces. In this approach, execution traces obtained from the execution of two models to be compared are analyzed for identifying semantic differences among these models. This analysis is performed by applying semantic differencing rules on the traces, which are match rules [14] indicating which syntactic differences among the traces constitute semantic differences among the models. The match rules are specific to the used xDSML as well as the relevant semantic equivalence criterion.

Our semantic differencing approach utilizes a *generic trace metamodel* for capturing execution traces. More precisely, a trace conforming to this metamodel is a sequence of clones of the model after each event occurrence causing a state change. The usage of a generic trace metamodel has two key implications on the trace analysis: *(i)* As a state is simply a collection of objects of any type, type checks and type casting are required to analyze the captured execution data. This implies complex rules that are hard to read and comprehend. *(ii)* Analyzing state changes of an executed model requires the traversal of all states captured in a trace. This implies an execution time that scales at best linearly to the number of captured states. To mitigate these issues, we propose the application of rich and domain-specific traces as presented in this work.

5.2 Case Study

As proposed above, we have adapted our semantic differencing framework [15] so that it relies on execution traces conforming to generated rich and domain-specific trace metamodels instead of a generic trace metamodel. Thereby, we

conducted a case study with a real world xDSML, namely fUML [21]—a subset of UML comprising class and activity diagrams having well defined execution semantics. In the case study, we have defined the execution semantics of fUML using xMOF and used our proposed approach to generate a rich and domain-specific trace metamodel for fUML. The execution metamodel extends one metaclass and defines 57 new classes. The generated trace metamodel consists of 56 classes for values and 58 classes for object states. The implemented semantic differencing rules determine whether two fUML activity diagrams are trace equivalent, *i.e.* whether all sequences of action executions possible in one activity diagram are also possible in the other. We developed two variants of these rules: one for performing the analysis on trace models conforming to the generic trace metamodel, and one for performing the analysis on trace models conforming to the generated domain-specific trace metamodel.

For evaluating the performance improvement gained by relying on the proposed rich domain-specific trace metamodels, we applied the semantic differencing rules on example fUML models. The example models constitute real world models taken from the case study of Maoz et al. for evaluating their semantic differencing operator *ADDiff* [17]. These models may be found at `http://www.se-rwth.de/materials/semdiff/`.

5.3 Results

In the following, we present the results of the evaluation and discuss how they give answers to the research questions stated in Section 2.2.

Complexity Reduction of Semantic Differencing Rules (RQ #2). Table 1 compares the complexity of the semantic differencing rules defined for fUML based on the generic trace metamodel and the rich domain-specific trace metamodel. For all elements, we observe a significant reduction of the complexity of the rules reaching from 20% to 100%. This is mainly due to the rich structure of the generated domain-specific trace metamodel. In contrast to the generic trace metamodel, there is no need to traverse the complex data structure of the execution metamodel of fUML, but instead the actions and the evolution of their values can be directly accessed. Other improvements are due to the fact that the trace metamodel is domain-specific, such as type checks that become obsolete. These results allow us to answer RQ #2 as follows: rich domain-specific trace metamodels simplify the definition of domain-specific trace analyses.

Performance Improvement of Semantic Differencing Rules (RQ #1). Figure 4 shows the execution times measured for applying the semantic differencing rules on the traces of the considered example models. This experiment was performed on an Intel Core i7-4600U CPU, 2.10GHz, 2.69GHz, with 12GB RAM, running Windows 8.1 Pro. The X-axis of Figure 4 shows the number of states contained by the generic and domain-specific traces. The Y-axis shows the measured execution time on a logarithmic scale. Each execution time was measured ten times and the arithmetic mean values are shown in the figure. As can be seen from the measurements, the rules analyzing traces conforming to the domain-specific trace metamodel outperform the match rules analyzing generic

Elements	G	DS	Reduction
Lines of code	136	55	60%
Statements	58	21	64%
Operation calls	32	13	59%
Loops	5	4	20%
Type checks	4	0	100%

Table 1. Complexity of the semantic differencing rules of fUML defined for the generic (G) and rich domain-specific (DS) trace metamodel.

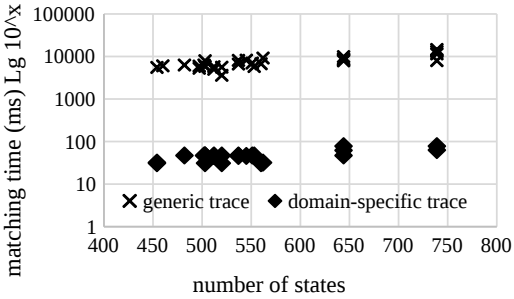

Fig. 4. Execution time of the semantic differencing rules of fUML for generic and rich domain-specific traces.

traces since they are between 170 and 400 times faster with an average of 250. The main reason for this result is the rich structure of the domain-specific trace metamodel allowing to efficiently explore the trace through dedicated navigation paths related to specific model elements. These results allow us to answer RQ #1 as follows: rich domain-specific trace metamodels enable better execution times for trace manipulations as compared to a generic trace metamodel.

6 Related Work

To our knowledge, little work has been done on the topic of domain-specific traces. Hegedus et al. [11] worked on many aspects of xDSMLs, such as trace replay and back-annotation. However they do not provide an approach to obtain trace metamodels for an xDSML. More recently, Meyers et al. introduced the ProMoBox framework [20], which generates a set of metamodels from an annotated xDSML, including a property metamodel and a trace metamodel. Their trace metamodel generation has multiple differences with our approach. Among others, they consider an abstract syntax whose properties are annotated either as *runtime* or *event* to identify mutable elements and event-related elements, while we consider the abstract syntax and the execution metamodel to be separated. Indeed, such separation makes possible a better separation of concerns and interchangeability of semantics. Also, they use generalization to extend a base trace metamodel, while we generate new classes to avoid having to rely on introspection and casting when manipulating traces. In addition, they do not provide alternative ways to explore a trace, while we provide various navigation paths. Finally, Gogolla et al. [9] generate *filmstrip models* from UML class diagrams. Such filmstrip models match what we call domain-specific trace metamodels, and also provide some navigation paths among objects states. However, they do not tackle redundancy since object states are always recreated at each model change, and they do not consider value states.

Regarding the richness of traces, we will in the future look more thoroughly at the mostly undocumented and transient traces manipulated by V&V tools.

7 Conclusion and Perspectives

Dynamic V&V of models requires the ability to model executions traces. We identified two important requirements regarding the definition of a *trace metamodel* for an xDSML: it must provide good *scalability in time* when manipulating traces, and good *usability* to analyze traces containing domain-specific data and events. Generic trace metamodels are not adequate because of their distance to the domain of an xDSML and because of their lack of alternative trace exploration means. The approach we presented consists in generating a *rich* and *domain-specific* trace metamodel of an xDSML, using its definition of what the execution state of a model is, and which events may occur during an execution. We reify the mutable properties of the execution metamodel into classes, allowing both to reduce redundancy and to narrow the trace metamodel. We also provide navigation paths both to follow the evolution of each mutable field of the model over time, and to follow the event occurrences of each event. This allows an efficient navigation of traces, *i.e.* an exploration without visiting each state of the trace. Our evaluation was done by the generation of a trace metamodel for fUML and its utilization for *semantic differencing* of several models. The results show a simplification of the semantic differencing rules and faster execution times of the rules, when compared to a naïve and generic trace metamodel.

The direct perspectives of this work include defining a common interface for all generated trace metamodels using model subtyping, enabling compression by detecting patterns in sequences of values, or handling *deltas* instead of states for certain types of value changes (e.g. collections, strings).

Acknowledgement. This work is partially supported by the ANR INS Project GEMOC (ANR-12-INSE-0011) and by the European Commission under the ICT Policy Support Programme grant no. 317859.

References

1. Alawneh, L., Hamou-Lhadj, A.: Execution Traces: A New Domain That Requires the Creation of a Standard Metamodel. In: Int. Conf. on Advanced Software Engineering and Its Application. CCIS, vol. 59, pp. 253–263. Springer (2009)
2. Bousse, E., Combemale, B., Baudry, B.: Scalable Armies of Model Clones through Data Sharing. In: 17th Int. Conf. on Model Driven Engineering Languages and Systems. LNCS, vol. 8767, pp. 86–301. Springer (2014)
3. Bousse, E., Combemale, B., Baudry, B.: Towards Scalable Multidimensional Execution Traces for xDSMLs. In: 11th Workshop on Model Design, Verification and Validation. CEUR-WS, vol. 1235, pp. 13–18. CEUR (2014)
4. Combemale, B., Crégut, X., Garoche, P.L., Thirioux, X.: Essay on Semantics Definition in MDE - An Instrumented Approach for Model Verification. Journal of Software 4(9), 943–958 (2009)
5. Combemale, B., Crégut, X., Pantel, M.: A Design Pattern to Build Executable DSMLs and Associated V&V Tools. In: 19th Asia-Pacific Software Engineering Conference. pp. 282–287. IEEE (2012)

6. Corley, J., Eddy, B.P., Gray, J.: Towards Efficient and Scalabale Omniscient Debugging for Model Transformations. In: 14th Workshop on Domain-Specific Modeling. pp. 13–18. ACM (2014)
7. DeAntoni, J., Mallet, F.: TimeSquare: Treat your Models with Logical Time. In: 50th Int. Conf. on Objects, Models, Components, Patterns. LNCS, vol. 7304, pp. 34–41. Springer (2012)
8. Eschweiler, D., Wagner, M., Geimer, M., Knüpfer, A., Nagel, W.E., Wolf, F.: Open Trace Format 2: The Next Generation of Scalable Trace Formats and Support Libraries. In: 14th Int. Conf. on Parallel Computing. Advances in Parallel Computing, vol. 22, pp. 481–490. IOS Press (2011)
9. Gogolla, M., Hamann, L., Hilken, F., Kuhlmann, M., France, R.B.: From Application Models to Filmstrip Models: An Approach to Automatic Validation of Model Dynamics. In: Modellierung 2014. LNI, vol. 225, pp. 273–288. GI (2014)
10. Hamou-Lhadj, A., Lethbridge, T.C.: A metamodel for the compact but lossless exchange of execution traces. Software & Systems Modeling 11(1), 77–98 (2010)
11. Hegedüs, A., Ráth, I., Varró, D.: Replaying Execution Trace Models for Dynamic Modeling Languages. Periodica Polytechnica - Electrical Engineering 56(3), 71–82 (2012)
12. Hutchinson, J., Whittle, J., Rouncefield, M., Kristoffersen, S.: Empirical Assessment of MDE in Industry. In: 33rd Int. Conf. on Software Engineering (ICSE). pp. 471–480. ACM (2011)
13. Jézéquel, J.M., Combemale, B., Barais, O., Monperrus, M., Fouquet, F.: Mashup of metalanguages and its implementation in the Kermeta language workbench. Software & Systems Modeling pp. 1–16 (2013)
14. Kolovos, D.S., Di Ruscio, D., Pierantonio, A., Paige, R.F.: Different Models for Model Matching: An analysis of approaches to support model differencing. In: 2009 ICSE Workshop on Comparison and Versioning of Software Models. pp. 1–6. IEEE (2009)
15. Langer, P., Mayerhofer, T., Kappel, G.: Semantic Model Differencing Utilizing Behavioral Semantics Specifications. In: 17th Int. Conf. on Model Driven Engineering Languages and Systems. LNCS, vol. 8767, pp. 116–132. Springer (2014)
16. Leucker, M., Schallhart, C.: A brief account of runtime verification. The Journal of Logic and Algebraic Programming 78(5), 293–303 (2009)
17. Maoz, S., Ringert, J.O., Rumpe, B.: ADDiff: Semantic Differencing for Activity Diagrams. In: 19th ACM SIGSOFT Symposium and 13th Europ. Conf. on Foundations of Software Engineering. pp. 179–189. ACM (2011)
18. Mayerhofer, T., Langer, P., Kappel, G.: A Runtime Model for fUML. In: 7th Workshop on Models@run.time. pp. 53–58. ACM (2012)
19. Mayerhofer, T., Langer, P., Wimmer, M., Kappel, G.: xMOF: Executable DSMLs based on fUML. In: 6th Int. Conf. on Software Language Engineering. LNCS, vol. 8225, pp. 56–75. Springer (2013)
20. Meyers, B., Deshayes, R., Lucio, L., Syriani, E., Vangheluwe, H., Wimmer, M.: ProMoBox: A Framework for Generating Domain-Specific Property Languages. In: 7th Int. Conf. on Software Language Engineering. LNCS, vol. 8706, pp. 1–20. Springer (2014)
21. Object Management Group: Semantics of a Foundational Subset for Executable UML Models (fUML), V 1.1 (August 2013), `http://www.omg.org/spec/FUML/1.1`
22. Tatibouët, J., Cuccuru, A., Gérard, S., Terrier, F.: Formalizing Execution Semantics of UML Profiles with fUML Models. In: 17th International Conference on Model Driven Engineering Languages and Systems. LNCS, vol. 8767, pp. 133–148. Springer (2014)