Advanced and efficient execution trace management for executable domain-specific modeling languages

Erwan Bousse, Tanja Mayerhofer, Benoit Combemale, Benoit Baudry

Softw Syst Model
DOI 10.1007/s10270-017-0598-5

SPECIAL SECTION PAPER

Advanced and efficient execution trace management for executable domain-specific modeling languages

Erwan Bousse[1] · **Tanja Mayerhofer**[1] · **Benoit Combemale**[2] · **Benoit Baudry**[3]

Received: 24 June 2016 / Revised: 30 December 2016 / Accepted: 18 April 2017

Abstract Executable Domain-Specific Modeling Languages (xDSMLs) enable the application of early dynamic verification and validation (V&V) techniques for behavioral models. At the core of such techniques, *execution traces* are used to represent the evolution of models during their execution. In order to construct execution traces for any xDSML, *generic trace metamodels* can be used. Yet, regarding trace manipulations, generic trace metamodels lack efficiency in time because of their sequential structure, efficiency in memory because they capture superfluous data, and usability because of their conceptual gap with the considered xDSML. Our contribution is a novel generative approach that defines a *multidimensional* and *domain-specific* trace metamodel enabling the construction and manipulation of execution traces for models conforming to a given xDSML. Efficiency in time is improved by providing a variety of navigation paths within traces, while usability and memory are improved by narrowing the scope of trace metamodels 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 technique 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.

Keywords Model execution · Domain-specific languages · Execution trace

Communicated by Prof. Alfonso Pierantonio, Jasmin Blanchette, Francis Bordeleau, Nikolai Kosmatov, Prof. Gabriele Taentzer, Prof. Manuel Wimmer.

✉ Erwan Bousse
bousse@big.tuwien.ac.at
https://big.tuwien.ac.at/people/ebousse/

Tanja Mayerhofer
mayerhofer@big.tuwien.ac.at
https://big.tuwien.ac.at/people/tmayerhofer/

Benoit Combemale
benoit.combemale@irisa.fr
http://people.irisa.fr/Benoit.Combemale/

Benoit Baudry
benoit.baudry@inria.fr
http://people.irisa.fr/Benoit.Baudry/

[1] TU Wien, Vienna, Austria

[2] IRISA - University of Rennes 1, Rennes, France

[3] Inria, Rennes, France

1 Introduction

A large amount of domain-specific modeling languages (DSMLs) has been proposed and used to model the behavior of systems [6,25,31,52,55]. Early *dynamic* verification and validation (V&V) techniques, such as omniscient debugging [9,16], semantic differencing [43], and runtime verification [44], are necessary to ensure that such models are correct. These techniques require models to be *executable*, which can be achieved by defining the execution semantics of the DSMLs used to define them. To that effect, a lot of efforts have been made to provide facilities to design so-called *executable DSMLs* (xDSMLs) [5,14,22,33,49,63,65].

While an executable model is an intentional representation[1] of some behavior, dynamic V&V techniques need more extensional representations[2] of behavior over time. A very common representation of a model's behavior is the *execution trace*, which records relevant information about an execution over time. While a trace can take numerous

[1] e.g., $\{t \in \mathbb{N} \mid f(t)\}$.

[2] e.g., $\{f(1), f(2), f(3)\}$.

Published online: 16 May 2017

forms, we focus in this work on traces containing the *execution states* reached by a model during the execution and the *execution steps* that were responsible for state changes. All previously mentioned V&V approaches rely on execution traces: omniscient debugging relies on an execution trace to revisit a previous execution state; semantic differencing consists in comparing execution traces of two models in order to identify the semantic variations between them; runtime verification consists in checking whether or not an execution trace satisfies a temporal property.

Consequently, providing execution trace management facilities is essential to support dynamic V&V for xDSMLs, which leads to two significant prerequisites: (1) the definition of an *execution trace metamodel* to represent traces of executable models of a particular domain in an accurate and usable way, and (2) the ability to manipulate large traces efficiently, *i.e.,* with good scalability in both memory and manipulation time.

The first prerequisite can be partly fulfilled with an existing *generic trace metamodel*, such as the ones defined and used in [32] and [43]. However, these metamodels cannot take the domain of an xDSML explicitly into account. This limits their usability, making the development of domain-specific analyses of traces more difficult. To cope with this, a *domain-specific* trace metamodel can be considered, like the trace metamodel defined for fUML in [48]. Yet, an existing domain-specific trace metamodel cannot be used for a new xDSML since it is bound to a particular domain. Furthermore, designing a new domain-specific trace metamodel is a time consuming and error-prone task [41]. These limitations lead to the development of generative approaches [27,51] that produce trace metamodels that are tailored to given xDSMLs.

However, existing generic and generative approaches for trace modeling do not fulfill the second prerequisite. In particular, they only offer to explore a trace by enumerating all execution states and steps one by one. This can only scale linearly in manipulation time at best. Moreover, traces generated with existing approaches have a substantial memory usage since these approaches rely on producing clones[3] of the complete model to capture execution states. As an example, part of the *ProMoBox* approach by Meyers et al. [51] consists in automatically generating a domain-specific trace metamodel for a considered xDSML, but such metamodel defines a trace as a sequence of clones of the complete executed model.

In this paper, we propose a novel generative approach for defining trace metamodels that are domain-specific and that enable efficient trace manipulations. This is achieved by the following contributions:

1. A generic approach to automatically generate a *multi-dimensional and domain-specific trace metamodel* for a given xDSML: The generated metamodel provides a data structure to capture domain-specific concepts in the execution traces. The generation process relies on the analysis of the xDSML's definitions of execution states and steps, to incorporate these concepts in the metamodel. Furthermore, the generated trace metamodel is multidimensional meaning that it provides alternative and combinable navigation paths to efficiently traverse and process traces.
2. A generic and generative approach to automatically derive *domain-specific trace constructors*: The generated constructors can be embedded in a model execution environment for the construction of execution traces conforming to the multidimensional and domain-specific trace metamodel generated for an xDSML.
3. An annotation mechanism for the *customization of trace metamodels and trace constructors* for particular application contexts of xDSMLs: *Tracing annotations* allow to restrict the level of granularity and the level of detail of execution traces to those execution states and steps that are required for a given trace analysis task. Such customized trace metamodels provide better usability and execution traces with smaller memory usage.

We evaluate our approach by generating a complete and a customized multidimensional and domain-specific trace metamodel for the real world xDSML *Foundational UML* (fUML) [55]. We compare the generated trace metamodels with a generic trace metamodel that relies on model cloning. We measure both their *usability* and *scalability* for performing semantic model differencing [43]. We also measure the runtime overhead induced by the construction of execution traces, and the memory consumption of these traces. Our results show that the proposed trace structure leads to a simplification of the semantic differencing rules, and to an improved performance of the semantic differencing rules. In addition, our approach induce a very small overhead when constructing traces during the execution of models, and reduce significantly the memory consumption of traces as compared to generic clone-based traces.

This paper is a significant extension of our previous work [10]. The extensions comprise (i) the inclusion of execution steps into domain-specific trace metamodels, (ii) the generation of trace constructors, (iii) the use of tracing annotations to customize trace metamodels, (iv) the evaluation of the runtime overhead caused by the construction of traces, (v) the evaluation of the memory consumption of traces, and (vi) a more thorough review of related work. Some of these ideas were already coarsely described or mentioned in our previous work on omniscient debugging [9].

[3] Also known as *snapshots* or *copies*.

The remainder of the paper is organized as follows:

- Section 2 presents background information on executable domain-specific modeling languages.
- Section 3 motivates the problem domain by looking at the state of the art and explaining our ideas for overcoming existing limitations.
- Section 4 gives an overview of our approach.
- Section 5 presents our approach for generating multidimensional and domain-specific trace metamodels.
- Section 6 explains the generation of trace constructors to make use of generated trace metamodels.
- Section 7 presents our customization mechanism for tailoring trace metamodels to the application context of an xDSML.
- Section 8 presents the implementation of our approach within the language and modeling workbench GEMOC Studio.
- Section 9 discusses the evaluation of our approach in the domain of semantic model differencing.
- Section 10 discusses related work.
- Section 11 summarizes the contributions of this paper.
- Section 12 provides an outlook on future work.

2 Background

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

2.1 Domain-specific modeling languages

A very common way to define the abstract syntax of a *domain-specific modeling language (DSML)* is by defining a *metamodel*. While there are many definitions of this term in the literature, we consider a metamodel to be an object-oriented model. Therefore, a metamodel is composed of *metaclasses*, each being composed of properties. A *property* is either an *attribute* (typed by a datatype, e.g., integer) or a *reference* to another metaclass. In addition, a metamodel possesses *static semantics*, which are additional structural constraints that must be satisfied by conforming models (e.g., multiplicities, containment references, and invariants on the structure of models).

Since a metamodel is a set of metaclasses, we consider a model as a set of objects that are instances of these metaclasses, and that satisfy the static semantics of the metamodel. This is commonly referred to as the *conformity* relationship between a model and its metamodel.

2.2 Executable domain-specific modeling languages

To execute a behavioral model and henceforth be able to use dynamic V&V techniques, the *execution semantics* of the considered DSML must be defined. We call *executable domain-specific modeling language* (xDSML) a DSML that aims at supporting the execution of models, and we call *executable model* a model that conforms to an xDSML.

There are two general approaches to define execution semantics, namely *translational* and *operational* semantics. Translational semantics consist in an exogenous model transformation that translates a model m into a model m' that conforms to a different executable language, in order to rely on the execution semantics of the latter. Operational semantics consist in an endogenous model transformation that changes the *execution state* of a model m. In this paper, we only deal with operational semantics, although our work can be directly adopted to translational semantics as long as a mechanism is provided to translate back the results of the execution (e.g., the execution states) into the source domain [12].

To define the execution state of a model, we consider that the abstract syntax metamodel of an xDSML can be extended into an *execution metamodel* with new properties and metaclasses. To this end, a mechanism equivalent to the well-known *package merge* operation of UML and MOF [53, 56] can be used. Note that in practice, existing tools and approaches use different but similar extension mechanisms—e.g., Kermeta [40] uses aspect weaving, xMOF [49] uses generalization, Hegedüs et al. [35] use separate metaclasses.

Next, we call *execution transformation* the model transformation that changes the execution state of a model. We consider this transformation to be endogenous (*i.e.,* both input and output models conform to the execution metamodel) and to be composed of a set of *transformation rules*, each defining a subset of the changes performed on the execution state. Depending on the model transformation language paradigm, rules can take different forms: In a declarative language, such as VIATRA [18] and ATL [39], they are composed of a source pattern and a target pattern and are executed using pattern matching; in imperative languages, such as Kermeta [40] and xMOF [49], they are operations that can call one another, one being the entry point starting the transformation. To avoid having to duplicate most of the model for the execution, we consider this transformation to be *in-place* (*i.e.,* the executed model is directly modified). This hypothesis takes into account that observing the in-place modifications made to a single model is a common pattern when defining tools for xDSMLs (e.g., graphical animation).

Because one of the purpose of xDSMLs is to analyze the behaviors of models, an important concern is to be able to observe the evolution of the execution state during the application of the execution transformation. In that respect, at least two problems must be taken into account:

- **Conformity:** By definition, it is only guaranteed that the executed model conforms to the execution metamodel

before and *after* the application of the complete model transformation responsible for the execution. Yet, the goal is to observe the model *during* the transformation, during which conformity is not guaranteed.
- **Atomicity:** The semantics of a language may specify what are the *atomic*[4] changes that can be made to the execution state, *i.e.,* changes during which the execution state is not consistent and should not be observed. A first example is a Java program, which reaches a new consistent state only after the execution of a complete statement, while the intermediate changes in the stack of the virtual machine are not observable. A second example is a Petri net, where an atomic change is the firing of a transition (*i.e.,* remove tokens from input places *and* add tokens to output places), while the intermediate removals or additions of tokens are not observable.

Therefore, to be observable, an execution transformation must: (1) be explicitly partitioned into distinct atomic changes and (2) ensure conformity of the model to the execution metamodel right before and after each atomic change. In this work, we consider that this is accomplished through *step rules*, which are designated rules of the model transformation that represent relevant changes in the model from the domain point of view. More precisely, considering the execution as a sequence of events *beginStepRule* (when the execution of a step rule starts) and *endStepRule* (when the execution of a step rule ends): Conformity must be guaranteed when such events occur, and there is one atomic change between each pair of successive events (e.g., inside a couple ⟨*beginStepRule*, *endStepRule*⟩ if a step rule does not call another step rule, or inside ⟨*beginStepRule*, *beginStepRule*⟩ if a step rule calls another step rule).

For example, a step rule may specify the complete execution of a Petri net, or the firing of a Petri net transition. As a comparison, a non-step rule may add tokens to the output places of a transition without removing tokens from the input places: It is an intermediate change that leads to a Petri net that indeed conforms to the execution metamodel, but whose marking should not be observed.[5]

Lastly, we consider one additional element of operational semantics: 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.

Definition 1 An *xDSML* is defined by:

- An *abstract syntax*, that is a metamodel.
- *Operational semantics*, composed of:
  - An *execution metamodel*, that defines the execution state of executed models by extending the abstract syntax with new properties and metaclasses using package merge, or any similar mechanism.
  - An *initialization transformation*, that is an exogenous model transformation that transforms a model conforming to the abstract syntax into a model conforming to the execution metamodel, while at least preserving the content of the input model.[6]
  - An *execution transformation*, that is an in-place model transformation that modifies a model conforming to the execution metamodel by changing values of dynamic fields and by creating/destroying instances of metaclasses introduced in the execution metamodel. The subset of transformation rules that are considered observable are called *step rules*.

For more precision, we call *static* a metaclass defined in the abstract syntax that is not extended by the execution metamodel with new properties. Likewise, we call *static* a property defined in the abstract syntax. At the model level, we also call *static* an object instance of a static metaclass, and an object's field that defines the object's value(s) for a static property. During the execution of the model, a static field cannot change, and a metaclass defined in the abstract syntax cannot be instantiated (e.g., it is not possible to add new transitions to a running Petri net).

In a similar fashion, we call *dynamic* a metaclass or a property introduced in the execution metamodel. A dynamic metaclass can either be an extension of a class of the abstract syntax, or a new metaclass only composed of dynamic properties. At the model level, we also call *dynamic* an instance of a dynamic metaclass, and an object's field that defines the object's value(s) for an dynamic property. During the execution of the model, a dynamic field can change, and a new dynamic metaclass (*i.e.,* only defined in the execution metamodel) can be instantiated.

Figure 1 shows an example of a simple Petri net xDSML. On the top left corner, its abstract syntax is depicted with three metaclasses Net, Place and Transition. Next to the abstract syntax, the execution metamodel is shown. It extends the metaclass Place using *package merge* with a new dynamic

[4] This term comes from the field of database systems where it refers to transactions comprising actions that are all executed indivisibly to ensure that the transaction preserves the consistency of the database [29].

[5] While this example shows that (conformity $\not\Rightarrow$ atomicity), note that an intermediate change may also break conformity, as is the case in many model transformations.

[6] For each object o of the input model conforming to a metaclass c, an instance o_{exe} of the corresponding execution metaclass c_{exe} is created and filled with the same values. This constraint implies that this transformation is not an arbitrary translation to a completely different target language—*i.e.,* we are not in the case of translational semantics—but in the preparation of an execution state for an upcoming execution.

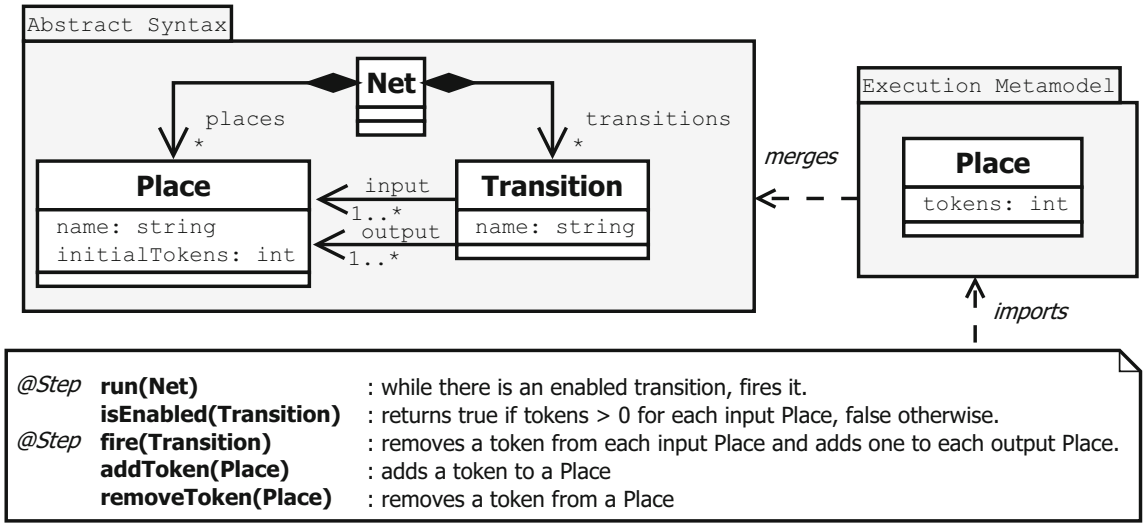

Fig. 1 Petri net xDSML

property `tokens`. The initialization function (not shown) transforms each original object (*i.e.,* a Place object without a `tokens` field) into an executable object (*i.e.,* 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 descriptions of the rules defined in the operational semantics are depicted. When called, these rules may change the `tokens` fields of the different Place objects, with *run* being the entry point of the transformation. The label @Step is used to show which transformation rules are step rules. By labeling both *run* and *fire* with @Step, we specify that the model is observable only before and after the application of these rules, *i.e.,* before and after the execution (since *run* is the entry point) and before and after firing transitions. The reason for this choice is that, according to Petri nets semantics, the firing of a transition is an atomic change. Hence, we should not observe a state after having only removed tokens from input places without adding tokens to output places, but only after the complete firing of a transition.

Algorithm 1: run

```
   Input:
   n : the Net object to run
1  begin
2  |  t_enabled :∈ {t ∈ n.transitions | isEnabled(t)}
3  |  while t_enabled ≠ null do
4  |  |  fire(t_enabled)
5  |  |  t_enabled :∈ {t ∈ n.transitions | isEnabled(t)}
```

Algorithm 1 shows the definition of the *run* transformation rule. First it finds an initially enabled transition using the *isEnabled* rule (line 2). Then it continuously fires enabled transitions using the *fire* rule, until no more transition is enabled (lines 3–5). Algorithm 2 shows the definition of the *fire* transformation rule. It first iterates over the input Place objects to remove one token from each input place of the transition using the *removeToken* rule (lines 2–3). Then it iterates over the output places to add one token to each output place using the rule *addToken* (lines 4–5).

Algorithm 2: fire

```
   Input:
   t : the Transition object to fire
1  begin
2  |  foreach p ∈ t.input do
3  |  |  removeToken(p)
4  |  foreach p ∈ t.output do
5  |  |  addToken(p)
```

2.3 Execution trace

As we have seen in Sect. 2.2, the operational semantics of an xDSML are based on a model transformation composed of transformation rules. More precisely, a subset of these rules are called *step rules* and produce consistent and observable changes of the execution state of a model.

We call *execution step* the application of a step rule. More precisely, we draw a distinction between a *small step*[7] and a *big step*,[8] the latter being composed of multiple execution steps. Big steps imply that the considered model transformation language gives the possibility to call or use

[7] Sometimes called *micro-step* [16,34].

[8] Sometimes called *macro-step* [16,34], *combo step* [24], or *compound step* [34].

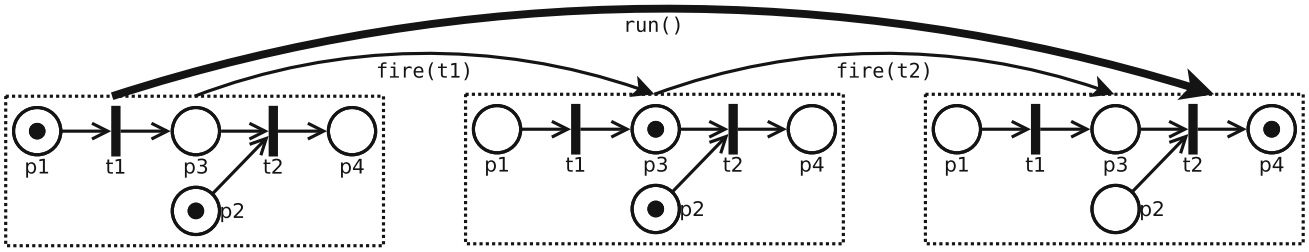

Fig. 2 Example of Petri net execution trace represented using concrete syntax

a transformation rule within another transformation rule. If a step rule calls another step rule, then the execution of the former constitutes a big step. This call can be indirect, e.g., within the call of a non-step rule during the big step. Lastly, if a step rule never calls another step rule, then the execution of the former constitutes a small step.

Definition 2 An *execution step* is the application of a step rule. An execution step that is not composed of other steps is called a *small step*, while an execution step composed of multiple steps is called a *big step*.

As an example, the Petri net xDSML depicted in Fig. 1 has two step rules: *run* and *fire*. The definition of *fire* in Algorithm 2 shows that it never relies on another step rule. Therefore, the application of *fire* results in a small step. However, the definition of *run* in Algorithm 1 shows that it relies on the step rule *fire*. Hence, the application of *run* results in a big step, composed of a number of *fire* small steps.

We call *execution state* the set of all values of all dynamic fields of an executed model, *i.e.,* the values of the fields defined by properties introduced in the execution metamodel, at a certain point in time of the execution. The execution state of a model changes each time a rule of the execution transformation modifies the value of some dynamic field, or creates a new dynamic object. Note that, at a given point in time, an object that was previously created during the execution is indirectly part of the execution state, since its fields are all dynamic.

Definition 3 An *execution state* is the set of the values of all dynamic fields of a model at a certain point in time of the execution. The execution state of a model is changed by the application of rules of the execution transformation.

To define the concept of execution trace, we leverage on existing trace theory from multiple domains, such as model checking [4], runtime verification [44], or more generally temporal logics [58], although we focus in this work on the finite and non-exhaustive exploration of a system's behavior. While in practice execution traces can take various forms (e.g., list of events, tree of method calls, list of memory dumps), we consider in this work that an execution trace is a sequence of execution states and execution steps. More precisely, we consider that an execution trace records all execution states of a model, as well as the execution steps causing changes on the model's execution state.

Definition 4 An *execution trace* is a sequence of execution states and execution steps (both small steps and big steps) responsible for the state changes.

Figure 2 shows an example of an execution trace obtained by executing a Petri net model using the operational semantics of the xDSML shown in Fig. 1. At the bottom, three execution states are depicted using the concrete syntax representation of the xDSML. At the top, two small steps are recorded: first the application of *fire* on the transition $t1$, then on $t2$. Both are part of the big step that is the application of *run*. This execution trace gives us all the required information to understand and analyze this execution: We know how the marking of the Petri net evolved, and we know which transitions were fired and in which order.

3 Motivation

In this section, we first introduce three requirements we identified for trace metamodels, then we present the limitations of existing solutions, and finally we present our ideas for complying with these requirements.

3.1 Requirements for an execution trace metamodel

In our previous work [8], 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, which might be an issue for large execution traces. Moreover, loading a large trace in memory can be a hindrance when resources are limited.

Another issue is to manage the manipulation complexity of trace models. Trace analyses can either be *generic* (e.g., computing the number of different execution states or

the amount of performed execution steps), or be *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* [43]. First, it is a hard problem because traces tend to be large and therefore expensive to process. But more importantly, semantic differencing consists in performing *domain-specific* analyses of traces, since they are written in accordance with the semantics of a specific xDSML, and may therefore rely on complex domain-specific data and properties.

To sum up, we consider the following requirements on a good trace metamodel:

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

Scalability in memory. Likewise, it should provide good scalability in memory when loading large traces.

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

Trace construction overhead. Using such a trace metamodel to construct an execution trace during the execution of a model should induce an acceptable overhead.

3.2 Limitations of existing trace formats and approaches

For a long time, considerable effort has been made to define both a variety of trace formats and approaches to define new trace formats. We call *trace format* a data structure describing the content of execution traces. With this term, we include metamodels (as defined in Sect. 2.1), but also ASCII or binary formats, which can all be used in practice for trace construction, persistence (*i.e.,* serialization into files or storage into a database system) or analysis.[9] In the following paragraphs, we highlight the limitations of different categories of formats and we present crosscutting problems they all share.

Existing generic trace formats. Few approaches [7,21,32,43,63] allow the capture and the manipulation of generic traces for any kind of execution from any kind of xDSML. However, *usability* is hindered regarding domain-specific trace manipulations, since relevant concepts related to the execution (*i.e.,* concepts defined in the execution metamodel) are not directly accessible. For instance, the concept of a Petri net *token* would not be present in a generic trace format.

Existing domain-specific trace formats. Most existing trace formats are domain-specific [2,19,23,26,30,48,54,64,64], hence specific to a selection of concerns, such as parallel software [23], operating systems [64], or to a specific xDSML [48]. Hence, while a domain-specific trace format may be relevant for specific xDSMLs, it is unlikely to be convenient to define the execution traces of a given arbitrary xDSML. For instance, a "system call" (from [64]) or an "fUML activity execution" (from [48]) are concepts that are not relevant when constructing an execution trace for a Petri net model. This semantic gap between the concepts defined in an existing domain-specific trace format and the domain concepts of a particular xDSML makes it impossible to reuse such formats for a given arbitrary xDSML.

Definition of new domain-specific trace formats. In order to take into account any possible xDSML while improving *usability* for domain-specific trace manipulations, a possible solution is the manual definition of an ad hoc execution trace format that is appropriate for the considered xDSML. A way to achieve this is to rely on *self-defined trace formats* [3,20,57,60] which are *meta-formats* allowing the definition of both the possible content of a trace and the trace itself in the same model. Another possibility is to consider approaches that provide a base trace metamodel along with some guidelines to define the complete domain-specific trace metamodel of an xDSML [13,34]. Yet, there are two main problems with these ideas. First, dedicated trace management tools must be manually developed along a trace format, which is expensive. Second, even with appropriate meta-formats or approaches, manually defining a trace format is likely to be a difficult task [41].

A last but interesting possibility relies on the *automatic* generation of a domain-specific trace metamodel. Such generative solutions [27,51] give the possibility to also automatically generate associated trace management tools. Defining such generation procedure is not trivial, since the possible contents of execution traces of any xDSMLs have to be captured precisely. Yet it avoids having to define a domain-specific trace metamodel by hand. Our contribution in this paper belongs to this latter category while overcoming the limitations discussed below of existing approaches.

Besides the issues specific to the different categories of existing tracing approaches mentioned above, the following limitations can be observed:

[9] However, note that we do not include purely theoretical definitions, such as the notion of execution trace from the realm of model checking [4], which is only used for laying formal foundations, while model checkers concretely rely on data structures of a different nature, such as binary decision diagrams (BDDs).

Linear structure. A limitation of almost all execution trace formats lies in the linear structure they propose for representing traces. In other words, the only way to navigate in a trace is by enumerating each captured execution state or execution step one by one. Few exceptions, such as [23] or [32], propose to browse an execution trace by focusing on the captured values of specific model elements. This appears as an interesting way to improve *scalability in time*.

Lack of a representation of execution states. The majority of trace management approaches [2,19,20,30,47,48,54,57] only capture events that occurred during an execution, such as execution steps, and lack a representation of the execution state, such as the values of the variables of a program. This is partly due to the large size of traces, which leads to the necessity of limiting the amount of information stored in them. Yet, traces containing only steps must be replayed in order to reconstruct the states, whereas traces containing states allow direct analyses. Therefore, representing execution states is important regarding the *usability* of a trace metamodel.

Use of model clones in generative solutions. Existing generative solutions to define domain-specific trace metamodels [27,51] define an execution state as a clone of the executed model. Hence, while this guarantees the capture of all the execution data, static parts of the model are copied despite the fact that they cannot change. This yields poor *scalability in memory*, and also poor *usability* due to the excess of data available in a trace. In addition, cloning a model is very costly, which means that it has a negative impact on the *overhead* caused by the construction of a trace during the execution of a model.

4 Approach overview

In this section, we first give intuitions and explain our reasoning to overcome the limitations of existing solutions, then we give a complete overview of our approach.

4.1 Proposal and research questions

To overcome the limitations observed in existing trace formats and approaches, and to better comply with the considered requirements, the underlying intuitions of the approach we propose are the following.

First, because generic trace formats provide little usability for the definition of domain-specific trace manipulations, the use of a domain-specific trace metamodel appears as a natural solution. This intuition is empirically supported by the very high number of domain-specific trace metamodels used nowadays, as we previously presented. Furthermore, the benefits of narrowing the scope of a language to a domain are well known [36,66], which means that defining a trace metamodel specific to a language can bring similar advantages. In particular, providing concepts of the xDSML directly in the trace metamodel can provide good usability for defining domain-specific manipulations. In [48], we followed this idea by defining manually a complete trace metamodel for fUML, which showed many benefits for analyzing executions of fUML models. Consequently, to improve *usability*, our first resolution is to rely on *domain-specific* trace metamodels for capturing traces of xDSMLs.

Second, manually defining a domain-specific trace metamodel for each xDSML has several disadvantages. In addition to being a tedious and error-prone task, it is necessarily followed by the task of defining domain-specific trace analysis and visualization tools. In contrast, generic trace formats can easily benefit from tools defined once for every possible xDSML, but they do not meet our usability requirement. Therefore, similarly to [27,51], our second resolution is to go from *generic* trace metamodels to a *generic generative 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 steps of an xDSML. This avoids the difficulty of defining domain-specific trace metamodels manually, and makes it possible to automatically provide suitable tools for manipulating domain-specific traces.

Third, as we already mentioned, most existing trace formats only provide support to browse an execution trace in a linear way, *i.e.,* by enumerating each captured execution state or execution step. Yet, there are in fact many imaginable ways to browse an execution trace. Having more navigation paths at disposal can enable a more efficient browsing of traces, and hence provide improved scalability in manipulation time. An example is finding the next value change of a given model element regardless of any other changes in the model. Such a query can be done easily by traversing the complete trace, yet reifing it as a *navigation path* dedicated to the investigated model element can avoid browsing the whole trace. Consequently, to improve *scalability in time*, we propose to create *multidimensional* trace metamodels similarly to [23] or [32], *i.e.,* metamodels that provide many navigation paths to explore a trace.

Fourth, since capturing the executions states reached by a model avoids having to replay an execution trace in order to analyze it, we propose to capture the execution states reached by a model in addition to the execution steps that led to them, thereby improving *usability* of our approach.

Lastly, due to the drawbacks of model cloning for capturing an execution trace, we propose to carefully and precisely capture in a trace metamodel only the concepts effectively required in a trace. By doing so, an execution state would only contain the data necessary for performing trace analyses,

which suggests better *scalability in memory*, better *usability*, and a smaller *trace construction overhead*.

In a nutshell, our proposal is an approach to *automatically* generate a *multidimensional* and *domain-specific* trace metamodel specific to an existing xDSML, and possibly also specific to a set of trace analyses.

To evaluate our approach, we consider a comparison to a linear generic trace metamodel that relies on the cloning of the complete executed model after each execution step, and that does not provide advanced navigation facilities. We evaluate the relevance of our contributions with respect to the following research questions:

RQ1 Can a multidimensional domain-specific trace metamodel provide a *smaller trace manipulation time* compared to a linear generic trace metamodel?

RQ2 Can a multidimensional domain-specific trace metamodel provide smaller memory consumption compared to a linear generic trace metamodel?

RQ3 Can a multidimensional domain-specific trace metamodel *simplify the definition of domain-specific trace manipulations* compared to a linear generic trace metamodel?

RQ4 Can a multidimensional domain-specific trace metamodel provide a *smaller trace construction overhead* compared to a linear generic trace metamodel?

4.2 Considered process

In the following, we present a complete overview of our approach, which is shown in Fig. 3. Elements in gray are part of our approach or are generated by our approach.

Definition and tooling of an xDSML. As we explained in Sect. 2.2, the first step toward the execution of models is the definition of an xDSML (a). To simplify Fig. 3, only the abstract syntax, execution metamodel, and execution transformation are depicted.

Next, the main part of our approach consists of a pair of generators (b) that take as input the definition of the xDSML. Two components are generated: a multidimensional domain-specific execution trace metamodel (c), and a domain-specific trace constructor (d), whose generation procedures are presented in Sects. 5 and 6, respectively.

To customize the trace metamodel for a given set of trace analyses (subsequently called *application context*), a set of tracing annotations (e) can be used to parametrize the pair of generators. These annotations point to elements of the execution metamodel (e.g., a subset of the dynamic fields) and elements of the execution transformation (*i.e.,* a subset of step rules) that must be traced, while the remaining elements can be ignored. The use of these annotations results in the production of a customized trace metamodel along with a corresponding trace constructor. This part of our approach is presented in Sect. 7.

Now that a domain-specific trace metamodel precisely scopes what are the possible execution traces of the considered xDSML, a domain-specific trace analysis (f) can be defined by using the concepts of this metamodel. Given such trace analysis, a complementary step of our approach is the use of a static footprint extractor (g) to compute the static metamodel footprint (h) of the considered trace analysis (f), *i.e.,* the elements of the trace metamodel that are statically referenced by the analysis program or model. Then, a tracing annotations generator (i) analyzes this footprint in order to annotate the set of elements of the execution metamodel that has to be traced for the analysis to work properly. This set of annotations is used to generate a new trace metamodel that can be seen as a *refinement* of the former one, since elements are only removed and not added. Note that this regeneration can be done as many times are required, especially if iterations are being made on the trace analysis. This part of our approach is presented in Sect. 7.5.

Execution of a model. After having defined an xDSML (a), an executable model (j) conforming to the execution metamodel can be executed. As explained in Sect. 2.2, such execution consists in applying the execution transformation to modify the execution state of the model.

From there, using the domain-specific trace constructor (c) generated by our approach, an execution trace (k) conforming to the generated trace metamodel (d) can be produced. More precisely, the trace constructor must be notified by the execution transformation of the execution steps that occur, so that the executable model can be observed at relevant instants to construct the trace.

Finally, once an execution trace is available, a domain-specific trace analysis (f) can be used to analyze the trace for dynamic V&V purposes.

5 Generation of multidimensional domain-specific trace metamodels

We propose a generative approach to define multidimensional and domain-specific trace metamodels that are appropriate for efficiently constructing and processing traces. In this section, we present this approach first by presenting the technical challenges we had to overcome, second by explaining our generation procedure based on the introduced Petri net xDSML, and third by discussing the resulting benefits of the approach.

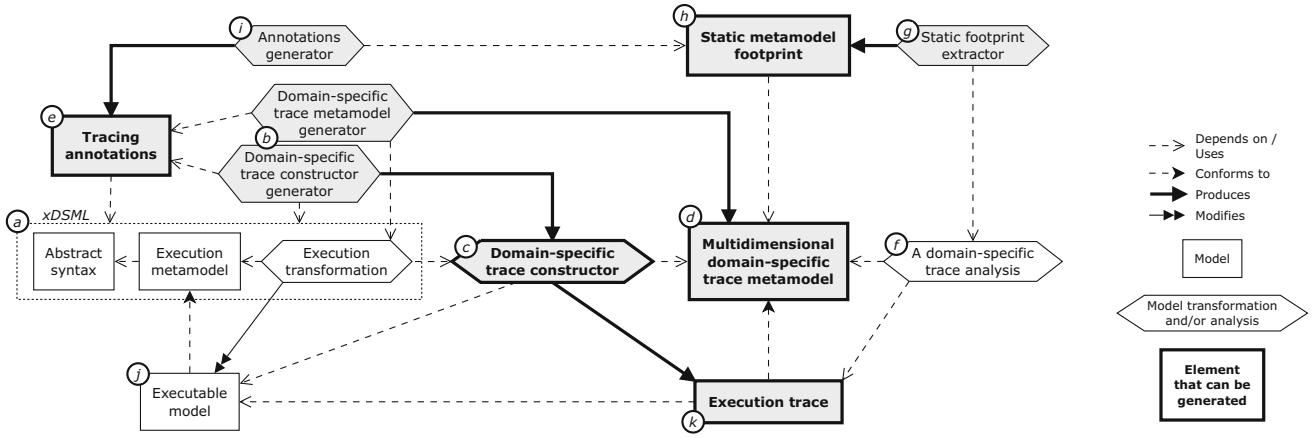

Fig. 3 Approach overview, with our contributions highlighted in gray

5.1 Observations and technical 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. While we presented such approaches in Sect. 3.2, we present below their limitations in more detail.

First, by cloning the whole model to store each execution state, redundancies appear between the states regarding all static fields (as they never change) and certain dynamic fields (as they may not change in each step). This impacts both usability (RQ3) and memory consumption (RQ2), although the scalable model cloning approach that we presented in [7] would mitigate this issue by sharing static data among clones at runtime.

Second, the dynamic fields we are interested in are scattered among the static fields, which may require complex queries to access them within a state. This issue compromise usability regarding the definition of domain-specific trace manipulation (RQ3).

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 dynamic field, compromising both scalability in time (RQ1) and usability (RQ3).

From these observations, we identified three technical challenges (TC):

(TC1) Narrowing the concepts introduced in a trace metamodel, e.g., by focusing on the dynamic properties of the execution metamodel.

(TC2) Avoiding redundancy in traces, e.g., by not storing the same value twice consecutively for a dynamic field.

(TC3) Providing alternative navigation paths, e.g., among the sequence of values of a specific dynamic field.

5.2 Execution trace metamodel generation

Algorithm 3: Trace metamodel generation

Input:
mm_{as} : the abstract syntax
mm_{exe} : the execution metamodel
exeTransf : the execution transformation
Result:
mm_{trace} : the trace metamodel

1 **begin**
2 $c_{\text{trace}}, c_{\text{exeState}}, c_{\text{step}}, c_{\text{smallStep}}, c_{\text{bigStep}} \leftarrow$ createBaseGenericClasses()
3 $mm_{\text{trace}} \leftarrow \{c_{\text{trace}}, c_{\text{exeState}}, c_{\text{step}}, c_{\text{smallStep}}, c_{\text{bigStep}}\}$
4 **foreach** $c_{\text{exe}} \in \{c \in mm_{\text{exe}} \mid \text{containsDynamicProperties}(c)\}$ **do**
5 $c_{\text{traced}} \leftarrow$ createClass()
6 $mm_{\text{trace}} \leftarrow mm_{\text{trace}} \cup \{c_{\text{traced}}\}$
7 c_{trace}.createReferenceTo(c_{traced}, [0..*], *unordered*)
8 **if** containsStaticProperties(c_{exe}) **then**
9 $c_{\text{orig}} \leftarrow$ getClassFromAbstractSyntax(c_{exe})
10 c_{traced}.createReferenceTo(c_{orig}, [1..1])
11 **foreach** $p \in$ getDynamicPropertiesOf(c_{exe}) **do**
12 $c_{\text{value}} \leftarrow$ createClass()
13 $mm_{\text{trace}} \leftarrow mm_{\text{trace}} \cup \{c_{\text{value}}\}$
14 c_{value}.`properties` $\leftarrow$ { copyProperty(p) }
15 c_{traced}.createReferenceTo(c_{value}, [0..*], *ordered*)
16 c_{value}.createReferenceTo(c_{traced}, [1..1])
17 c_{exeState}.createReferenceTo(c_{value}, [0..*], *unordered*)
18 c_{value}.createReferenceTo(c_{exeState}, [1..1])
19 **foreach** $r \in$ *exeTransf* **do**
20 $map_{\text{steps}} \leftarrow$ createMap()
21 createStepClass(r, map_{steps}, mm_{trace}, $c_{\text{smallStep}}$, c_{bigStep})
22 replaceReferencesToExecutionMM(mm_{trace}, mm_{as}, mm_{exe})

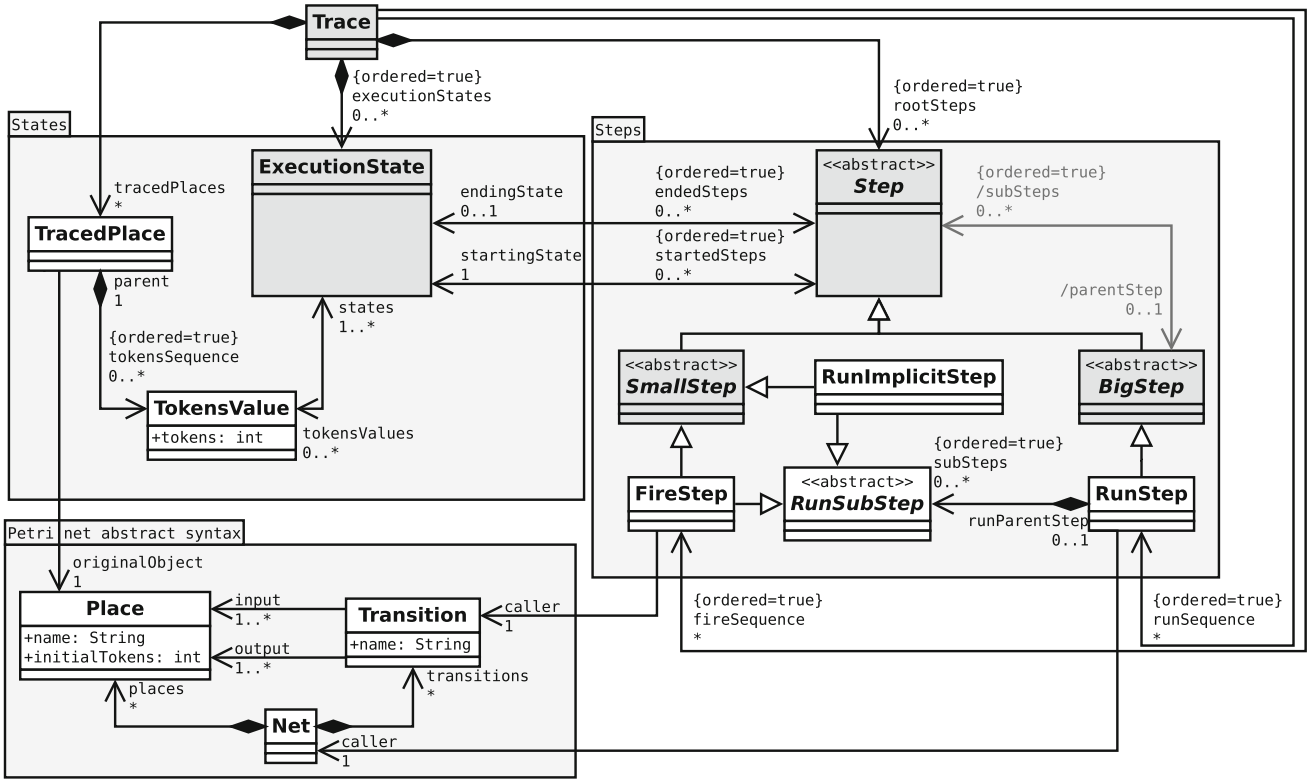

Fig. 4 Execution trace metamodel generated for the Petri net xDSML. Metaclasses in *dark gray* are always generated

Algorithm 3 shows our trace metamodel generation procedure. It relies on a recursive procedure *createStepClass* (called in line 21 of Algorithm 3), that is defined in Algorithm 4. Note that the algorithm is simplified for illustration purposes, meaning that some parts are reduced to functions, and that special cases, such as abstract metaclasses, are not considered. The inputs of the procedure are the abstract syntax (mm_{as}), the execution metamodel (mm_{exe}) and the execution transformation (*exeTransf*) 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. 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 Fig. 4. Note that the metaclasses Trace, ExecutionState, Step, SmallStep and BigStep (shown in dark gray) are always created (lines 2–3 of Algorithm 3).

Capturing the values of fields (lines 11–14 of Algorithm 3). At any given point in time, all dynamic fields of an object of the executed model have a *value*. To represent such a value in a trace, we create one metaclass per dynamic property of the execution metamodel, and we copy this dynamic property into this new metaclass (lines 12–14). This enables us to capture each value of a dynamic field as an instance of this generated metaclass. For Petri nets this means creating one metaclass called TokensValue for the property `tokens`. Thereby, we precisely narrow the trace metamodel to the dynamic part of the execution metamodel (TC1).

Capturing the states of objects (lines 4–10, 15–16 of Algorithm 3). The state of an object of the executed model at any point in time is defined by the values of all its dynamic fields. To represent all states reached by an object, we create one class for each metaclass of the execution metamodel containing at least one dynamic property (lines 4–6). In addition, we make all instances of these generated metaclasses accessible through a single instance of the metaclass Trace (line 7). For Petri nets this means creating a metaclass TracedPlace for the metaclass Place, and a reference `tracedPlaces` from the metaclass Trace.

An instance of such a generated metaclass shall contain all values reached by all dynamic 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 metaclass discussed previously (line 15). For Petri nets this means generating a reference `tokensSequence` for the metaclass TracedPlace to the

metaclass TokensValue. When creating an execution trace, one TracedPlace object will be created per Place object, each storing a sequence `tokensSequence` 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 dynamic field (TC2). 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 dynamic field (TC3).

The last concern for capturing the state of an object is that the object may also contain *static* fields, which remain an important piece of information. Since the corresponding static properties are all defined in a metaclass introduced in the abstract syntax, our solution is to create a reference to this metaclass (lines 8–10). For Petri nets this means adding a reference `originalObject` for the traced metaclass TracedPlace to the metaclass 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 of Algorithm 3). An execution state can be seen as the n-tuple of the values of all dynamic fields in an executed model at a given point in time. However, n is not xDSML-specific, but *model*-specific, as the number of dynamic 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 metaclasses introduced in the execution metamodel. To represent this n-tuple, we create a bidirectional reference between each generated value metaclass and the metaclass ExecutionState, which represents one execution state of a model. By that means, an execution state references an unbounded set of values of dynamic fields. For Petri nets this means introducing the references `tokensValues` and `states` between the metaclasses ExecutionState and TokensValue.

Capturing steps (lines 19–21 of Algorithm 3, and whole Algorithm 4). An execution step occurs when a step rule of the operational semantics is called. It has a starting state, which is the execution state of the model at the instant of the call, and an ending state, which is the execution state at the instant the rule has finished. This is represented by the references `startingState` and `endingState` between the metaclasses ExecutionState and Step. In addition, each step transformation of the operational semantics is reified into a metaclass of the same name (lines 3–5 of Algorithm 4), in which all parameters of the rule are copied (lines 6–7 of Algorithm 4). The resulting metaclass inherits either SmallStep if the rule does not call another step rule (lines 8–9 of Algorithm 4), or BigStep otherwise (10–11 of Algorithm 4). For Petri nets, this means creating the metaclasses FireStep inheriting from SmallStep, and RunStep inheriting from BigStep. By copying the parameters of the rules, each of these two metaclasses is given a reference `caller`, to be able to point to the Net or Transition object concerned by the rule.

Algorithm 4: createStepClass

Input:
- r : the step rule to transform into a step metaclass
- map_{steps} : a map with the step metaclass of each step rule
- mm_{trace} : the trace metamodel in construction
- c_{trace} : the trace root metaclass
- $c_{\text{smallStep}}$: the small step abstract metaclass
- c_{bigStep} : the big step abstract metaclass

```
1  begin
2    if r ∉ map_steps.keys then
3      c_step ← createClass()
4      mm_trace ← mm_trace ∪ c_step
5      map_steps ← map_steps ∪ (r ↦ c_step)
6      foreach p ∈ r.parameters do
7        c_step.properties ← paramToProperty(p)
8      if getStepRulesCalledBy(r) = ∅ then
9        c_step.superTypes ← c_smallStep
10     else
11       c_step.superTypes ← c_bigStep
12       c_sub ← createClass()
13       mm_trace ← mm_trace ∪ c_sub
14       c_step.createReferenceTo(c_sub, [0..*], ordered)
15       c_sub.createReferenceTo(c_step, [1..1])
16       foreach r_called ∈ getStepRulesCalledBy(r) do
17         createStepClass (r_called, map_steps, mm_trace,
                            c_trace, c_smallStep, c_bigStep)
18         c_called ← map_steps(r_called)
19         c_called.superTypes ← c_called.superTypes ∪ c_sub
20       if containsImplicitSteps(r) then
21         c_fill ← createClass()
22         mm_trace ← mm_trace ∪ c_fill
23         c_fill.superTypes ← {c_smallStep, c_sub}
24     c_trace.createReferenceTo(c_step, [0..*], ordered)
```

A step can be part of a big step, which is represented by the derived references `parentStep` and `subSteps`. This means that multiple steps may start (e.g., a big step and the first small step it contains) or end (e.g., a big step and the last small step is contains) in the same execution state. For instance, in the simple Petri net trace shown in Fig. 2, *fire* is called after *run* while the model is still in the initial state, and later both *run* and *fire* end in the final state. This is represented by the references `startedSteps` and `endedSteps` between the metaclasses ExecutionState and Step.

More precisely, a big step is the root of a tree whose internal nodes are big steps and whose leaves are small steps. To match the operational semantics as precisely as

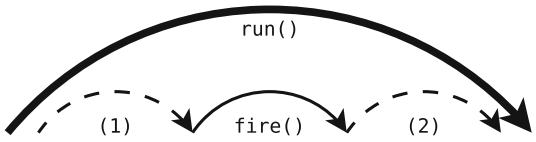

Fig. 5 Illustration of possible implicit steps within the *run* step rule

possible (TC1), we restrict the steps contained by a big step to the ones that may occur within its corresponding model transformation rule through the creation of a dedicated abstract metaclass (lines 12–13 of Algorithm 4). In addition, we rely on containment references to enforce the tree structure that is induced by big steps (lines 14–15 of Algorithm 4). For Petri nets, this means creating a metaclass RunSubStep representing all sorts of steps that may occur during a RunStep step, and two references `subSteps` and `runParentStep`.

Then, step metaclasses of all called step rules are created through a recursive call of the step metaclass creation procedure (Algorithm 4), and through the use of a map that associates each rule to its step metaclass (lines 16–19 of Algorithm 4). The first line of the algorithm is the stopping criterion to handle the recursion: We only create once the metaclass corresponding to a rule. For Petri nets, this means that the metaclass FireStep is defined as a subclass of RunSubStep, since this is the only operation called by *run*.

While most changes made during a big step are made by its substeps, some intermediate changes can be made directly by the big step itself, *i.e.,* outside of the substeps. For instance, a big step rule may start by making a change to the execution state before calling a first small step rule. Even though such change is not *explicitly* isolated within a dedicated transformation rule, it is nonetheless technically a small execution step. We call such anonymous small step an *implicit step*. These steps are taken into account by creating one dedicated metaclass per big step rule (lines 20–23 of Algorithm 4).

In the case of Petri nets, it is possible for the *run* operation to be responsible for other model changes in between the calls to *fire*. Figure 5 depicts such situation: Before and after calling *fire*, the code of *run* might be responsible for model changes, annotated (1) and (2). In the generated trace metamodel, they are represented by the metaclass RunImplicitStep inheriting both from SmallStep and RunSubStep. Note that such generation could be avoided provided an analysis of the *run* operation that would verify that no changes are made to the model apart from the calls to *fire*. We represent this analysis by a procedure called *containsImplicitSteps* (line 20 of Algorithm 4). Yet, for illustration purposes, we consider that we do not have such an analysis for Petri nets, and thus that this procedure returns *true*.

Finally, in the same manner as for values, all steps are stored chronologically within the unique Trace object (line 24 of Algorithm 4). For Petri nets this means having an ordered reference `fireSequence` in the Trace metaclass to the metaclass FireStep, and a similar reference `runSequence` to the metaclass `Run`. This gives direct access to all steps of a specific transformation rule in chronological order, which is an interesting additional navigation path for a trace (TC3).

Replacing references to the execution metamodel (line 22 of Algorithm 3). When dynamic properties and step metaclasses were copied in the trace metamodel, this included copying references to metaclasses of the execution metamodel. Yet, such metaclasses may contain dynamic properties that were already copied in the trace metamodel. To avoid having twice the same concept in the trace metamodel (TC1) or twice the same value stored in a trace (TC2), our solution is to replace all references to the execution metamodel by references either to the abstract syntax or to metaclasses representing the states of objects (e.g., TracedPlace). This is done by the function *replaceReferencesToExecutionMM* (line 22).

Example trace. Figure 6 shows a multidimensional domain-specific trace of a Petri net model. 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 shown execution, the transitions $t1$ and $t2$ are fired, leading to a trace with three states and two small steps contained in one big step.

To represent the states, three ExecutionState objects are linked to a set of TokensValue objects that represent the marking of the Petri net. The state objects are linked to FireStep objects, which represent the firing of $t1$ and $t2$. In addition, both the initial and the final states are linked to the RunStep object that represents the complete Petri net run. There is one `tokensSequence` list 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. Regarding steps, we have access to the list of the fired transitions by browsing the fireSequence list, e.g., to find states following directly a firing of $t2$. Likewise, we have access to the list of run nets with runSequence.

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 references from a ExecutionState object to value objects. 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

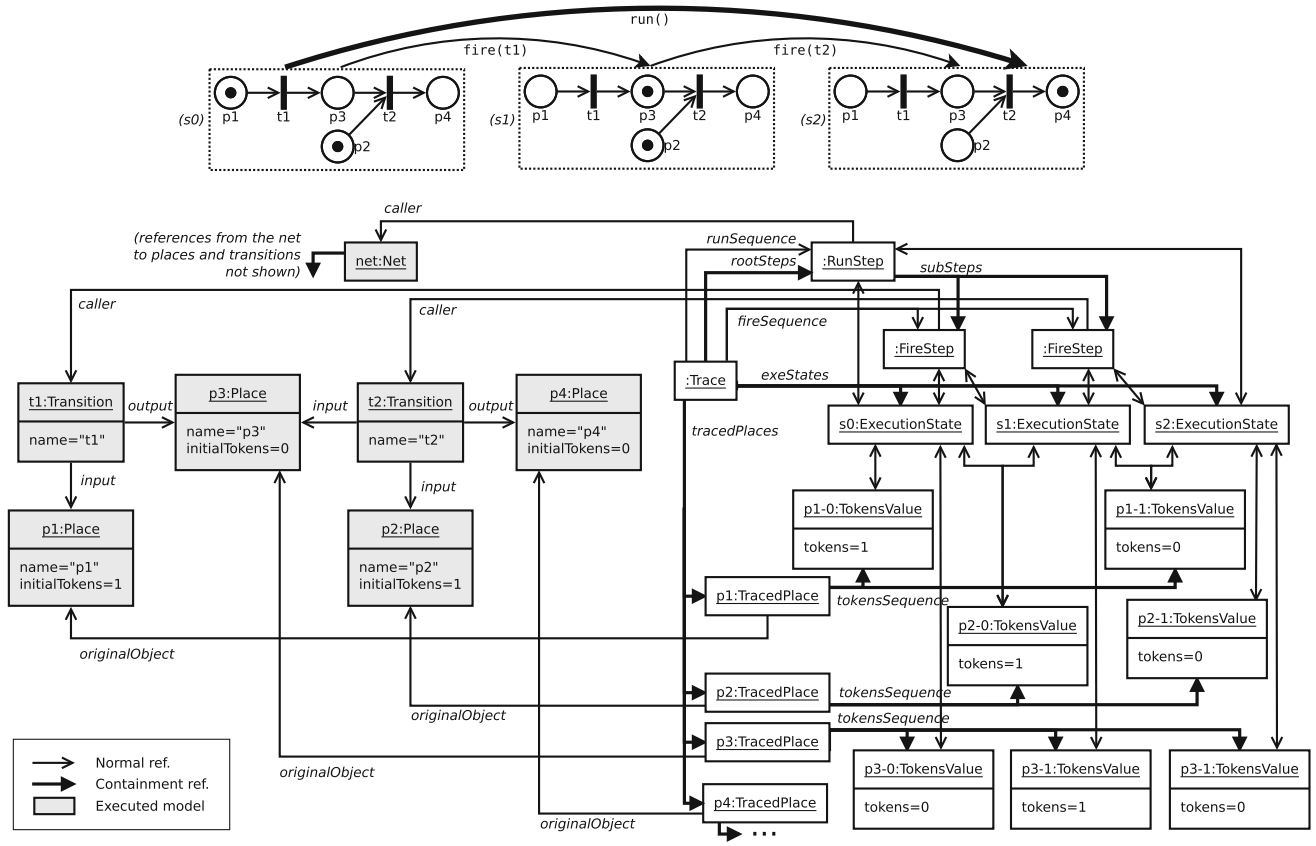

Fig. 6 Example of Petri net model and multidimensional domain-specific trace

before a state means that this state and the following ones have no references to its values.

5.3 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 (RQ3).

In addition, we provide several *navigation paths* for browsing traces. Indeed, we create for each dynamic property (e.g., `tokens`) and each step definition (e.g., FireStep) of an xDSML a dedicated navigation path (e.g., `tokensSequence` and `fireSequence`). This allows to enumerate each value of a particular field, or each step of a particular step rule, without having to enumerate all the states of the trace. Moreover, all values and steps are connected through execution states, allowing to go from one navigation path to another. These navigation facilities offer improved usability and scalability in time (RQ1 and RQ3).

5.4 Size of trace metamodels and models

To give a better understanding of the number of elements created for a generated trace metamodel, and created in a trace model conforming to such trace metamodel, we provide an analytical evaluation of our approach in the remainder of this section. We call *size* of a metamodel the number of metaclasses it contains, and *size* of a model the number of objects it contains. In order to provide short and intuitive formulas that still give relevant size estimates, we do not take into account the number of properties in metaclasses, and the number of fields in objects.

We use the notations NbC_X for a number of metaclasses, NbP_X for a number of properties, NbR_X for a number of transformation rules, and NbO_X for a number of objects.

Trace metamodel size. The size of a trace metamodel generated using our approach can be decomposed as:

$$NbC_{TMM} = NbC_{base} + NbC_{traced} + NbC_{value} + NbC_{step} + NbC_{implStep} + NbC_{subStep}$$

with:

- NbC_{TMM} the number of metaclasses in a trace metamodel generated using our approach.
- NbC_{base} the number of metaclasses that are always generated by the approach (e.g., the root metaclass Trace).
- NbC_{traced}, the number of metaclasses to trace dynamic objects (e.g., TracedPlace in Fig. 4).
- NbC_{step}, the number of metaclasses to trace execution steps (e.g., FireStep in Fig. 4).
- $NbC_{implStep}$, the number of metaclasses to trace implicit steps (e.g., RunImplicitStep in Fig. 4).
- $NbC_{subStep}$, the number of metaclasses to trace substeps (e.g., RunSubStep in Fig. 4).

We can then calculate NbC_{TMM} as follows for any xDSML:

- We have five base metaclasses (Trace, ExecutionState, Step, SmallStep, BigStep), hence $NbC_{base} = 5$.
- One traced metaclass is generated per dynamic metaclass of the considered xDSML, hence $NbC_{traced} = NbC_{dyn}$, with NbC_{dyn} the number of dynamic metaclass.
- One value metaclass is generated per dynamic property of the considered xDSML, hence $NbC_{value} = NbP_{dyn}$, with NbP_{dyn} the number of dynamic properties.
- One step metaclass is generated per step rule, hence $NbC_{step} = NbR_{step}$, with NbR_{step} the number of step rules.
- One implicit metaclass and one substep metaclass are generated for each big step metaclass, hence $NbC_{implicit} = NbC_{subStep} = NbR_{bigStep}$, with $NbR_{bigStep}$ the number of big step rules.

Finally, by replacing all the terms in the decomposition of NbC_{TMM}, we obtain the following sum to compute the size of a trace metamodel generated for a given xDSML:

$$NbC_{TMM} = 5 + NbC_{dyn} + NbP_{dyn} + NbR_{step} + 2NbR_{bigStep}$$

For instance, in the case of the Petri net xDSML shown in Fig. 1 we have $NbC_{dyn} = 1$, $NbP_{dyn} = 1$, $NbR_{step} = 2$, $NbR_{bigStep} = 1$, therefore $NbC_{TMM} = 11$. This corresponds to the number of metaclasses that can be found in the generated trace metamodel shown in Fig. 4.

Note that even if the size of a generated trace metamodel does linearly rise with the number of elements in the operational semantics, this has no significant negative impact on the approach. Regarding usability RQ3, the potentially large amount of metaclasses means that the generated metamodel precisely captures with many details the possible content of execution traces, which aims at facilitating the definition of trace analyses. Regarding scalability (RQ1 and RQ2), the size of a trace is only dependent on the executed model and on the inner workings of the execution transformation, and is entirely unrelated to the size of the trace metamodel. We explain this last aspect in more detail in the following.

Trace model size. The size of an execution trace conforming to a trace metamodel generated using our approach can be decomposed in the following way:

$$NbO_{trace} = NbO_{base} + NbO_{exeState} + NbO_{tracedObj} + NbO_{values} + NbO_{step}$$

with:

- NbO_{trace} the number of objects in a trace.
- NbO_{base} the number of base objects in a trace, which is always equal to 1 (one instance of the root metaclass Trace).
- $NbO_{exeState}$ the number of ExecutionState objects.
- $NbO_{tracedObj}$ the number of traced objects (e.g., instances of TracedPlace in Fig. 6).
- NbO_{values} the number of value objects (e.g., instances of TokensValue in Fig. 6).
- NbO_{step} the number of step objects (e.g., instances of FireStep in Fig. 6).

For example, in the case of the Petri net execution shown in Fig. 2, $NbO_{exeState} = 3$, $NbO_{tracedObj} = 4$, $NbO_{values} = 9$, $NbO_{step} = 3$, therefore $NbO_{trace} = 20$. This corresponds to the number of objects in the trace shown in Fig. 6, considering that the two value objects of $p4$ are not shown.

In summary, the size of a trace heavily depends not only on the initial state of an executed model (*i.e.,* the initial number of dynamic fields and objects), but mostly on how the execution transformation behaves for a given model (*i.e.,* the number of steps and the frequency of field changes). During an execution step, the changes in dynamic fields lead to new execution state objects, value objects and traced objects. Therefore, even a very large executed model (*i.e.,* with a large number of dynamic fields and objects) may lead to small execution traces if only very few dynamic fields change over time. Likewise, a very small executed model (*i.e.,* with a small number of dynamic fields and objects) may lead to a large execution trace if there is a large amount of steps along with a high frequency of changes in the dynamic fields.

6 Execution trace construction

In this section, we present the part of our approach dedicated to the construction of execution traces conforming to a generated multidimensional domain-specific trace metamodel. We first present how to generate a domain-specific trace constructor tailored for the trace metamodel of a spe-

cific xDSML. Then we explain how to make use of such constructors to capture execution traces of executed models.

6.1 Generating trace constructors

We call *trace constructor* a component that can be used during the execution of a model to construct an execution trace, *i.e.,* a model conforming to an execution trace metamodel. As shown in Fig. 3, this component must be called by the execution transformation and must access the model being executed to read its execution state.

Reflective versus generative trace construction. The construction of an execution trace conforming to a trace metamodel is necessarily dependent on the trace metamodel itself. However, in the previous section, we presented a procedure to automatically *generate* the domain-specific trace metamodel of a given xDSML, which means that the trace metamodel is not known in the general case. Fortunately, it is possible to analyze a trace metamodel generated using our procedure in order to discover the names of the required generated metaclasses and properties, as well as the dynamic subset of the execution metamodel. Such analysis is made possible thanks to the systematic structure of generated trace metamodels (e.g., the Trace class always references all traced classes, each of which references value classes). With this, there are two main solutions that can be considered for constructing traces conforming to a generated trace metamodel.

A first solution is to define a unique *reflective* generic trace constructor, which would realize such analysis of the considered generated trace metamodel during the construction of the trace (*i.e.,* "on-the-fly"), and rely on reflection to instantiate the discovered metaclasses.[10] However, this has several drawbacks: (1) analyzing the trace metamodel at each execution is a significant overhead; (2) using a reflective layer to create and modify objects is commonly regarded as being slow; (3) developing a correct reflective program is an error-prone task, since most manipulated objects are not statically typed, which strongly limits static checking; (4) if the trace metamodel generation approach evolves over time, the trace constructor would also evolve and hence become incompatible with previously generated trace metamodels.

A second solution is to *generate* a domain-specific trace constructor specific to the generated domain-specific trace metamodel of a given xDSML. The main advantage of such solution is that it only requires to analyze *once* the trace metamodel in order to generate the trace constructor, and avoids relying on reflection for constructing the trace. In addition, a generated trace constructor relies on typed objects and can therefore be statically checked for errors. Lastly, a trace constructor generated for a legacy trace metamodel can always be used regardless of changes made to the trace metamodel generation approach. For these multiple reasons, we chose in this paper to *generate* domain-specific trace constructors to construct execution traces, instead of relying on a unique reflective trace constructor.

Since there are multiple ways to generate valid trace constructors, and since any model transformation language or modeling environment can be used to implement them, we do not provide in this paper a complete generation algorithm. Instead, we first present a generic interface that each trace constructor generated by our approach must comply with, and we discuss the problem of efficiently capturing an execution state. We then present in Sect. 6.2 a simplified example of a generated trace constructor for the Petri net xDSML introduced in Fig. 1.

Generic interface for trace constructors. To limit the coupling between the execution transformation and the trace constructor, and to facilitate the evolution of the former, our approach relies on a *generic interface for trace constructors*[11] (shortened to *generic interface* in the remainder of the section) that is independent of the considered xDSML. As we explain thereafter in Sect. 6.3, this choice facilitates the automatic integration of a generated trace constructor with the execution transformation. We consider a generic interface composed of the following services:

- **createRoot**: create the root Trace object.
- **addInitialState**: create the first ExecutionState object of the model, with one traced object per initial dynamic object, and one value object per initial dynamic field.
- **addState**: add a new state in the trace if at least one dynamic field of the model changed, or if dynamic objects are created/deleted. This includes:
  - create a new ExecutionState object.
  - for each new dynamic object in the model, create a corresponding new traced object, create initial value objects for all its dynamic fields, and add these values both to the corresponding traced object and to the new execution state.
  - for each dynamic field that changed, create a value object, and add it to both the corresponding traced object and to the new execution state.
  - for each dynamic field that did not change, add the existing last value of this field to the execution state.
  - for each dynamic object removed from the model, the corresponding traced object will not be given new

[10] e.g., in EMF: `EObject o = EcoreUtil.create(eClass).`

[11] Please note that we do not mean a universal generic interface for any trace constructor for any trace construction approach, but simply a high-level specification of what is expected from the trace constructors generated by our approach. In other words, it is a way to describe how all our constructors behave and interact with other components.

values anymore, and new execution states will not refer to any value of the object anymore.

- **addStep**(*stepRuleID*, *stepRuleParams*): add a new Step in the trace. This includes:
  - create a new step object corresponding to the *stepRuleID* and containing the *stepRuleParams*,
  - if there was already a BigStep in progress, add the new step object as a substep of this BigStep,
  - set the current ExecutionState as the starting state of the new Step.
- **finishStep**: set the current ExecutionState as the ending state of the current Step.

Note that we intentionally do not specify the types of the parameters of these different services, since they may heavily depend on the considered modeling framework or model transformation language (e.g., a step rule may be identified by a name or by a pointer to the rule).

Efficient capture of execution state changes. An important requirement for efficient execution trace management is to limit the overhead induced by the construction of a trace during the execution of a model (RQ4). However, the execution state of a model (*i.e.,* the values of all dynamic fields, and the dynamic objects) can be arbitrarily large and complex and can therefore be costly to read and capture in a trace. To cope with this problem and to avoid reading the complete state of the model at each execution step, a solution is to only look at the changes that took place in the model since the last captured state. From there, a new state can be constructed in the trace by performing a shallow copy of this last captured state (*i.e.,* we do not copy the value objects, since they are may be used by different state objects), and by updating the copy based on the observed changes. Thereby, except for the initial one, each state can be constructed with little effort.

6.2 Example of a trace constructor

As we previously explained, we do not provide in this paper a complete code generation algorithm for trace constructors. Instead, we present a simplified example of a trace constructor generated for the Petri net xDSML introduced in Fig. 1, and we use it to explain the logic behind the expected generation procedure. Algorithms 5, 6, 7 and 8 show the trace construction operations generated for the Petri net xDSML introduced in Fig. 1. These operations comply with the generic interface we introduced above, and they construct an execution trace conforming to the execution trace metamodel generated for Petri net shown in Fig. 4.

Example of addInitialState. Algorithm 5 shows the *addInitialState* operation generated for Petri nets. Among other parameters, it requires a map map_{traced} relating each object of the executed model that contains at least one dynamic field (e.g., a Place) to the corresponding traced object (e.g., a TracedPlace). This map is initially empty, and is filled both by *addInitialState* during the creation of the initial state of the trace, or by *addState* when a new dynamic object is created in the executed model.

Algorithm 5: *addInitialState* generated for Petri nets

Input:
- $root$: root Trace object
- $model_{exe}$: the model being executed
- map_{traced} : map with the traced object of each object of the executed model

```
1  begin
2    state_initial ← createObject(ExecutionState)
3    root.executionStates.add(state_initial)
4    foreach o ∈ model_exe do
5      if o.is(Place) then
6        traced_o ← createObject(TracedPlace)
7        map_traced ← map_traced ∪ {o ↦ traced_o}
8        root.tracedPlaces ←
         root.tracedPlaces ∪ {traced_o}
9        v_new ← createObject(TokensValue)
10       v_new.tokens = o.tokens
11       state_initial.tokensValues.add(v_new)
12       traced_o.tokensSequence.add(v_new)
```

The creation of the initial state first requires the creation of the first ExecutionState object, which is added to the root of the trace (lines 2–3). Then a loop is generated to manage each possible sort of dynamic object using type checking (lines 4–5). For Petri nets, a TracedPlace object is created for each Place, and is added both to the aforementioned map and to the root (lines 6–8). Finally, the initial value of each `tokens` dynamic field is copied into a new TokensValue object, and is added both to the corresponding dimension in the traced object and to the execution state (lines 9–12).

Example of addState. Algorithm 6 shows the *addState* operation generated for the Petri nets xDSML trace constructor. As explained in Sect. 6.1, to construct a new state, we rely on the changes that occurred in the executed model since the last capture state. We consider that an operation *getFieldChanges* is available to obtain the list of changes in the mutable fields of the executed model (line 2). These changes can be obtained by defining an *observer* listening to the changes made to the executed model (e.g., using EMF notifications). Note that, in the general case, other operations similar to *getFieldChanges* must be provided for obtaining which objects were created or deleted from the executed model. Yet, in the case of Petri nets, such operations are not necessary since no objects can be created or deleted.

Algorithm 6: *addState* generated for Petri nets

```
Input:
    root          :  root Trace object
    model_exe     :  the model being executed
    map_traced    :  map with the traced object of each object of the
                     executed model
1   begin
2       changes ← getFieldChanges()
3       state_current ← root.executionStates.last()
4       if state_current = null then
5           addInitialState(root, model_exe, map_traced)
6       else if changes ≠ ∅ then
7           state_new ← copyState(state_current)
8           root.executionStates.add(newState)
9           foreach fieldChange ∈ changes do
10              o ← fieldChange.changedObject
11              traced_o ← map_traced(o)
12              if o.is(Place) then
13                  p ← fieldChange.changedProperty
14                  if p.is(Place.tokens) then
15                      v_new ← createObject(TokensValue)
16                      v_new.tokens = o.tokens
17                      v_old ← traced_o.tokensSequence.last()
18                      state_new.tokensValues.remove(v_old)
19                      state_new.tokensValues.add(v_new)
20                      traced_o.tokensSequence.add(v_new)
```

Algorithm 7: *addStep* generated for Petri nets

```
Input:
    root          :  root Trace object
    stepRuleID    :  ID of the step rule
    stepRuleParams:  parameters given to the rule
    stack_steps   :  stack of all ongoing current steps
1   begin
2       step_new ← null
3       state_current ← root.executionStates.last()
4       step_current ← stack_steps.peek()
5       if stepRuleID = getRuleID(RunStep) then
6           step_new ← createObject(RunStep)
7           step_new.caller ← stepRuleParams[0]
8           root.runSequence.add(step_new)
9           root.rootSteps.add(step_new)
10      else if stepRuleID = getRuleID(FireStep) then
11          step_new ← createObject(FireStep)
12          step_new.caller ← stepRuleParams[0]
13          root.fireSequence.add(step_new)
14          step_current.subSteps.add(step_new)
15      step_new.startingState ← state_current
16      stack_steps.push(step_new)
```

First, if the trace is still empty, the operation *addInitialState* is called to create the first ExecutionState object (line 3). Then, if the initial state was already created, the remainder of the operation consists in creating a shallow copy of the last state (lines 7–8), and updating it according to each change in the model (lines 9–20). A conditional expression is generated for each metaclass containing at least one dynamic field, and for each dynamic field. For Petri nets, the only possible change is within a Place object (line 12), more precisely the amount of tokens it holds (line 14). In that case, a new TokensValue object is created to replace the former one within the copied ExecutionState (lines 15–19). Finally, this new TokensValue is added to the `tokensSequence` dimension in the corresponding TracedPlace object (line 20).

Example of addStep. Algorithm 7 shows the *addStep* operation generated for the Petri net xDSML trace constructor. Among other parameters, it relies on a stack $stack_{steps}$ that contains all the ongoing steps, with the current one on the top (retrieved line 4). A step is pushed on the stack when it starts, and is popped from the stack when it ends.

For each possible step that can occur during the execution, a conditional expression is generated to obtain the step metaclass corresponding to the *stepRuleID* of the occurred step. For Petri nets, the two possibilities are the step rules *run* (line 5) and *fire* (line 10). In each case, we first create and initialize the corresponding step object (RunStep lines 6–7, FireStep lines 11–12), and we add the step to its corresponding dimension (lines 8 and 13). Next, the step object must be added either as a root step, or as a substep of an existing big step. For Petri nets, since *run* is only called once as a root step, the generated constructor can safely always add a RunStep as a root step in the trace (line 9). Likewise, since *fire* is only called from *run*, it can always safely be added as a substep of the current step (line 14). Finally, the starting state of the new step is set, and the new step is pushed onto the stack (lines 15–16).

Algorithm 8: *finishStep*

```
Input:
    root          :  root Trace object
    stack_steps   :  stack of all ongoing current steps
1   begin
2       state_current ← root.executionStates.last()
3       step_current ← stack_steps.pop()
4       step_current.endingState = state_current
```

Definition of finishStep. Algorithm 7 shows the definition of the *finishStep* operation. It only consists in retrieving the current state (line 2) and step (line 3) in order to set the ending state of the finishing step (line 4).

6.3 Integrating trace constructors with execution transformations

We described above how to generate a trace constructor that complies with a generic interface, with operations to add

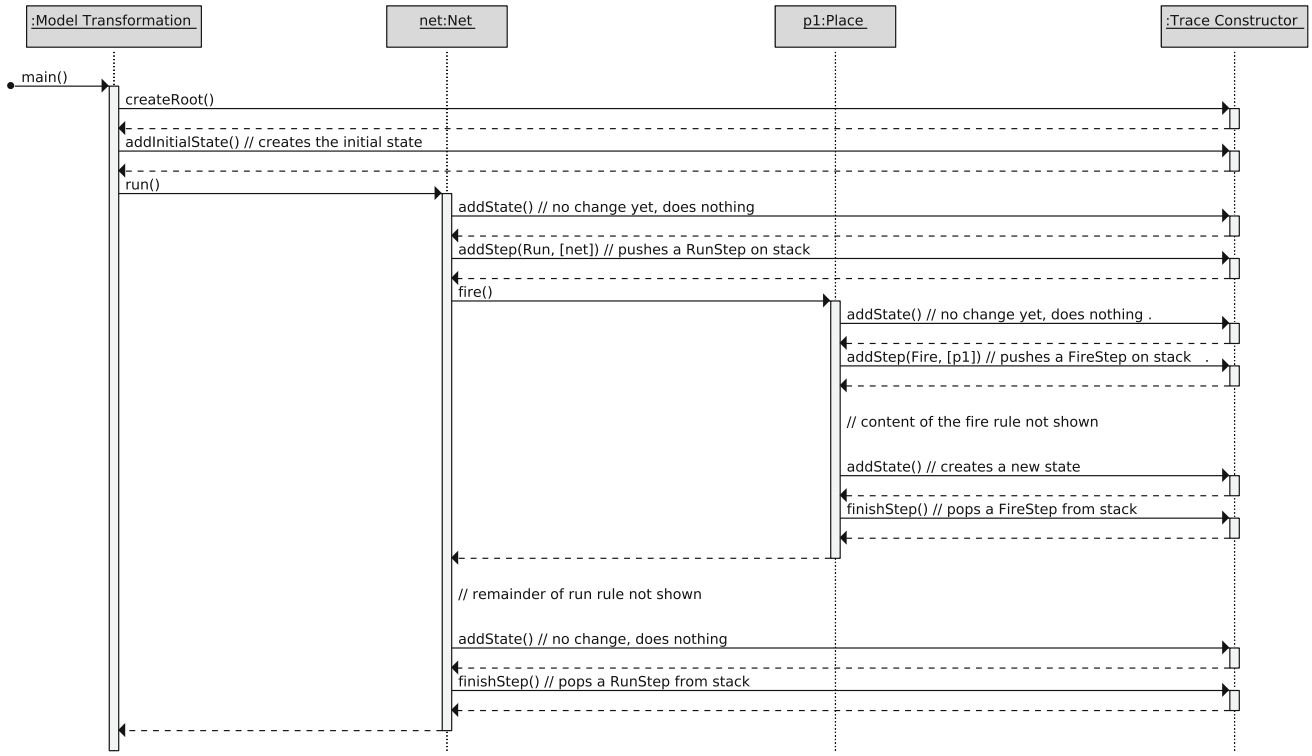

Fig. 7 Sequence diagram illustrating the use of a trace constructor during the execution of a Petri net model

states and steps to the execution trace. To actually construct a trace, these operations must be called by the execution transformation at relevant instants of the execution of the model. As we explained in Sect. 2.2, such instants are found at the beginning and at the end of execution steps responsible for changing the state of the executed model. Therefore, the following calls must be done during an execution:

- Just before the execution: *createRoot, addInitialState*
- Just before a step: *addState, addStep*
- Just after a step: *addState, finishStep*.

Figure 7 shows a sequence diagram illustrating the use of a trace constructor during the execution of a Petri net model. For the sake of clarity, the transformation rules *run* and *fire* are considered defined within the metaclasses Net and Place. The transformation is started through the use of an entry point called *main*. This triggers the call to *createRoot* to create the root Trace element of the trace, and *addInitialState* to create the initial ExecutionState object along with initial TracedPlaced objects for storing the tokens of all places. Then the first step transformation rule *run* is called on the object `net`. This triggers a call to *addState*, which does nothing since there was no change in the model yet. A call to *addStep* is also triggered, which adds a RunStep object in the trace and pushes this step on top of the stack of the constructor. Then, the step rule *fire* is called, which again triggers *addState* and *addStep*, adding a new FireStep on top of the stack. Note that *run* never changes the model before calling *fire*, which means that this second call to *addState* still does not create a new ExecutionState in the trace. However, after *fire* has modified the model, the third call to *addState* creates the second ExecutionState in the trace. A call to *finishStep* follows, which pops the FireStep from the stack and sets its ending state. The following calls to *fire* are not shown in the Figure. Finally, at the end of *run*, *addState* does not create any new ExecutionState since the state is not modified at the end of *run*, and *finishStep* pops the RunStep from the stack to set its ending state.

In practice, because these calls are made through a generic interface, they can be integrated in the execution transformation in many different ways. For instance:

- The execution transformation can be manually modified to integrate calls to the trace constructor. For instance, the *fire* step rule shown in Fig. 2 can be modified to call both *addState* and *addStep* at the very beginning, and both *addState* and *finishStep* and the very end.
- A model or program transformation can be defined to automatically make similar modifications to each step transformation rule of the transformation.
- Aspect oriented programming (e.g., AspectJ) can be used to specify *pointcuts* at the beginning and at the end of all step rules to call the trace constructor.

– If the considered model transformation language provides a notification mechanism to listen and react to applications of transformation rules (e.g., xMOF virtual machine, `@Step` annotation in Kermeta), a listener can be defined to call the operations of the trace constructor when step rules are applied.
– If the considered execution environment relies on a dedicated execution engine responsible for applying the transformation rules, this engine can produce calls to the trace constructor in between the applications of the rules.

For our implementation (see Sect. 8), we considered the GEMOC Studio as a target execution environment, which provides a complete and configurable execution engine. Therefore, we relied on the last solution.

7 Customization of domain-specific execution traces

In this section, we present the last part of our approach, which is the customization of trace metamodels to given application contexts of xDSMLs. We first explain and motivate this idea, and then introduce an illustrative example based on the xDSML fUML. Thereafter, we present how the customization of trace metamodels can be achieved by means of so-called tracing annotations.

7.1 Motivation and objective

Trace metamodels generated with the proposed approach provide domain-specific and optimized data structures for capturing exhaustive data about the behavior of executed models. Therewith, a trace metamodel generated with our approach provides the basis for realizing any kind of trace analysis for a given xDSML. In this paper, we call a specific set of trace analyses to be applied on traces obtained from the execution of any model conforming to a given xDSML *application context*. An application context is specific to the considered xDSML and concerns any model conforming to this xDSML.

Depending on the considered application context, only a subset of the exhaustive data captured by execution traces constructed with our approach may be required to perform the desired trace analyses. More precisely, an application context may require only a subset of the dynamic information captured about the execution states and execution steps of an executed model. By narrowing the scope of the trace metamodel down to the dynamic information required for the considered application context, we can further improve our approach with respect to all targeted research questions RQ1–RQ4. In particular, we can obtain a trace metamodel *customized* for a given application context, in order to trace only the dynamic information required in this context regarding the execution of any model conforming to the considered xDSML. Therewith, customized trace metamodels provide improved usability for defining trace analyses, lead to a reduction of the memory consumption and manipulation time of traces, as well as to a reduced runtime overhead induced by the construction of traces.

To further motivate and illustrate the customization potential of trace metamodels, we introduce in the following section a second xDSML, namely fUML [55], which is also used in Sect. 9 to evaluate our approach.

7.2 Illustrative example

Foundational UML (fUML) is an executable subset of UML standardized by the Object Management Group (OMG) [55]. It comprises subsets of UML's class modeling concepts and activity modeling concepts, and defines the execution semantics of these concepts. In this paper, we focus on the part of fUML that concern *activities*, which are elements representing the behavior of systems.

Excerpt of the fUML xDSML. Figure 8 depicts excerpts of both the abstract syntax and the operational semantics of fUML. At the top right, a subset of the abstract syntax shows that an activity is composed of activity nodes and activity edges. One specific type of activity nodes are actions defining the behavior of an activity. Activity nodes are connected via activity edges, in particular, via control flow edges and object flow edges denoting the flow of control and the flow of data among activity nodes, respectively.

fUML activities have token flow semantics similar to Petri nets, except that tokens in an fUML activity can express either control or data. The excerpt of the execution metamodel shown at the right of Fig. 8 defines the dynamic properties and metaclasses introduced for realizing these semantics. The dynamic metaclasses Token, ControlToken and ObjectToken are introduced to represent tokens that are created and held by activity nodes. In addition, the dynamic metaclass Offer is introduced to represent the tokens that are offered via activity edges. The excerpt shows also the dynamic property `firing` added to the metaclass Action to denote whether an action is currently being executed. At the bottom, a selection of rules of the execution transformation are shown: *sendOffers* and *receiveOffers* are responsible for handling token flows, while *fire* is responsible for executing activity nodes.

Excerpt of the generated fUML trace metamodel. We applied our generative approach from Sect. 5 to generate a multidimensional and domain-specific execution trace metamodel for fUML. The resulting metamodel is shown in Fig. 9. It defines four metaclasses for capturing the states

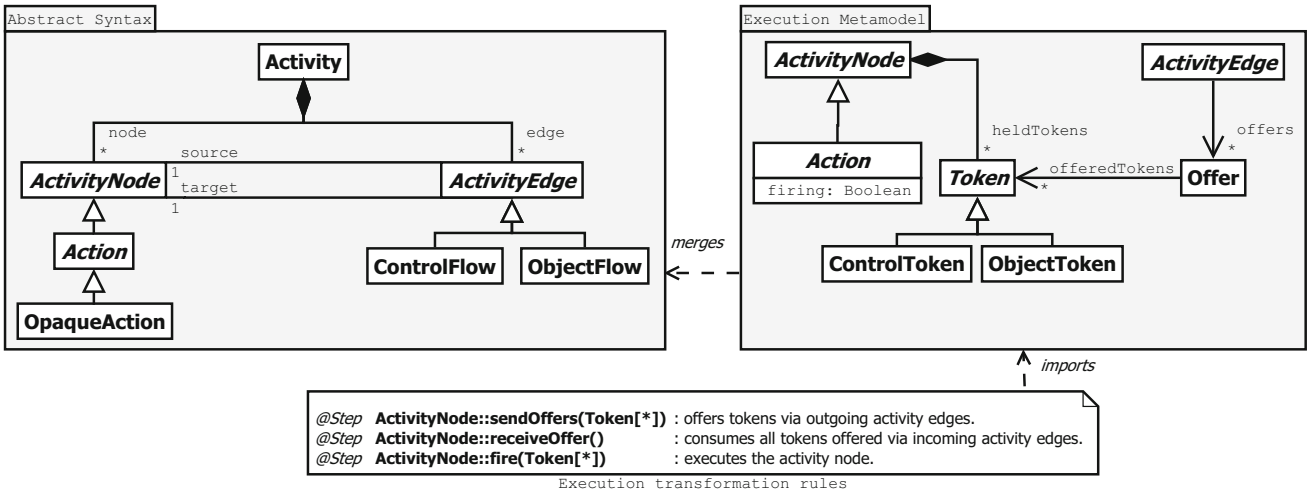

Fig. 8 Definition of the fUML xDSML (excerpt)

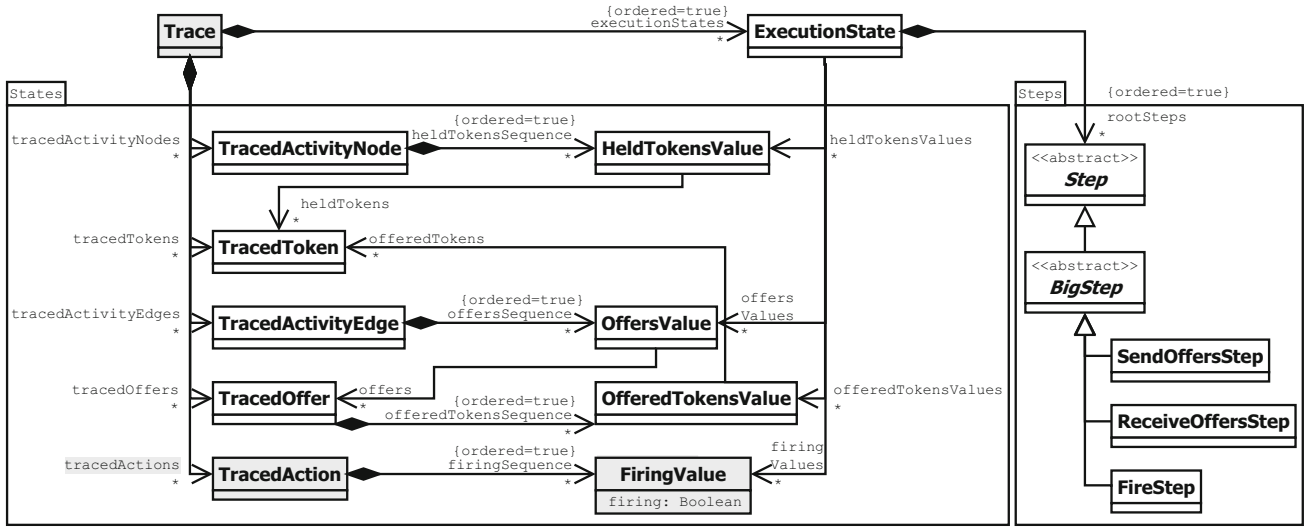

Fig. 9 Trace metamodel generated for the fUML xDSML (excerpt)

of the dynamic properties `heldTokens`, `offers`, `offeredTokens`, and `firing` introduced in the execution metamodel of the xDSML, as well as five metaclasses for capturing the states of objects of type ActivityNode, Token, ActivityEdge, Offer, and Action. Lastly, the trace metamodel contains three metaclasses for capturing the occurrences of the three big step rules *sendOffers*, *receiveOffers*, and *fire* defined as part of the execution transformation of fUML (the rules called by these big step rules are not shown).

fUML application contexts. The generated trace metamodel can be used in any application context of fUML. For instance, with the metaclasses HeldTokensValue and OfferedTokensValue, it is possible to trace the token flows between the nodes of a fUML activity. With this information, we can perform analyses on the token flows, such as analyzing the reachability of nodes and deadlocks in the activity. With the very same metaclasses, it is also possible to analyze the input values and output values of actions and activities. Furthermore, the metaclass FiringValue enables the tracing of executed actions and therewith allows the performance of analyses of executed actions, such as the analysis of the execution order of actions.

However, depending on the application context of fUML, different analyses are required. This means that the usage of traces may differ and, consequently, different levels of granularity and detail of traces may be required. For instance, if fUML models are used for modeling and analyzing high-level business processes, only the execution order of actions representing the steps of the modeled process might be of interest. In this case, the required granularity of traces is at the execution of one single action corresponding to the execution of the big step rule *fire*. This means, that only the state of the activity before and after the execution of a single

action are relevant in this context, but not intermediate states. Furthermore, the dynamic data that has to be provided in each captured state is reduced to the dynamic field `firing` of actions. Hence, the level of detail required for traces is substantially reduced in this application context. However, if fUML is used as an object-oriented programming languages as, for instance, advocated in [62], more detailed execution traces are required, since not only the execution order of actions are relevant for analyses of fUML activities, but also the inputs and outputs of executed action, as well as the object manipulations performed by them.

Example of fUML trace analysis. Let us have a more detailed look into a trace analysis that is part of the first application context of fUML mentioned above, namely the analysis of the executed actions of an fUML activity. Algorithm 9 shows this analysis, which processes a trace conforming to the trace metamodel generated for fUML in order to retrieve the set of executed actions of an fUML activity.

Algorithm 9: getExecutedActions

```
Input:
  trace : the analyzed Trace instance
Result:
  executedActions : the set of executed actions
1 begin
2   foreach action : TracedAction ∈ trace.tracedActions do
3     foreach actionFiringValue : FiringValue ∈
      action.firingSequence do
4       if actionFiringValue.firing then
5         executedActions ← executedActions ∪ action
```

The semantics of fUML specify that while an action is being executed, its dynamic field `firing` is set to *true*. This information is used by Algorithm 9 to compute the set of executed actions of an fUML activity. In particular, a traced action represented by a TracedAction object is added to the list of executed action, if one of its `firing` values represented by FiringValue objects holds the value `true`.

The trace analysis implemented by Algorithm 9 uses only a subset of the metaclasses and properties defined in the trace metamodel generated for fUML. The used metaclasses and properties are highlighted in gray color in the trace metamodel shown in Fig. 9. In particular, the trace metamodel elements relevant for the trace analysis are the metaclasses TracedAction and FiringValue, as well as their properties `firingSequence` and `firing`. For navigating to instances of TracedAction, the metaclass Trace and its property `tracedActions` are required as well. However, the trace metamodel elements dedicated to the tracing of token flows, such as the metaclasses TracedToken and TracedOffer, are not needed for the analysis. As a consequence, we could remove these concepts from the trace metamodel to obtain a trace metamodel customized to this application context of fUML.

7.3 Tracing annotations

In order to generate a domain-specific trace metamodel that is customized to a specific application context, we must first provide a way to select the subset of dynamic information that is relevant in this context for tracing the execution states and execution steps of models conforming to the xDSML. We propose to select this subset by annotating the operational semantics of an xDSML with so-called *tracing annotations*. There are two main cases to take into account: tracing annotations on the execution metamodel of an xDSML, and annotations on the execution transformation of an xDSML.

Tracing annotations applied on the execution metamodel of an xDSML select the dynamic properties and dynamic metaclasses that have to be considered for the construction of execution traces for the considered application context. In other words, each execution state stored within an execution trace should only capture dynamic field and dynamic objects corresponding to the annotated dynamic properties and dynamic metaclasses. This makes possible the reduction of the *level of detail* of execution traces, *i.e.,* the reduction of the dynamic information captured about the execution state of executed models. Please note that annotating a dynamic metaclass is the same as annotating all its dynamic properties. Furthermore, annotating a dynamic metaclass is also necessary in the case that no dynamic property of this metaclass has been annotated in order to trace the creation and destruction of instances of this dynamic metaclass.

Tracing annotations applied on the execution transformation of an xDSML select the step rules that have to be considered for the construction of execution traces for the considered application context. In other words, only execution steps corresponding to the execution of the annotated step rules are captured within an execution trace. This makes possible the reduction of the *level of granularity* of traces, *i.e.,* the amount of recorded execution steps and execution states.

Example of tracing annotations for fUML. Figure 10 shows the tracing annotations for fUML that are relevant for the application context of Algorithm 9. Tracing annotations are shown as labels with the prefix @ located next to the annotated element. This set of tracing annotations selects only those dynamic properties and step rules that are required for retrieving the set of executed actions in an activity, namely the dynamic property `firing` and the step rule *fire*.

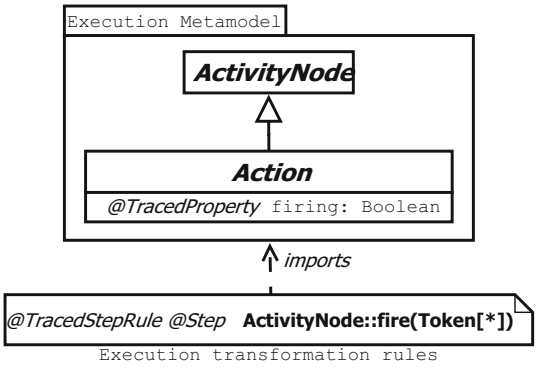

Fig. 10 Excerpt of the operational semantic of fUML, with tracing annotations relevant for the application context of Algorithm 9

7.4 Customization of trace metamodels and trace constructors based on tracing annotations

Using tracing annotations as introduced in the previous subsection, language engineers of an xDSML can explicitly define the dynamic properties and step rules of the xDSML that are relevant for a particular application context, and that should thus be traced during the execution of models. The goal when using tracing annotations is to eventually reduce the granularity of traces (*i.e.,* the number of execution states captured in a trace), as well as the level of detail of traces (*i.e.,* the data that is captured for each execution state).

To achieve this goal, the trace metamodel generation procedure introduced in Sect. 5 has to take defined tracing annotations into account, such that only the selected dynamic properties and step rules are considered in the generation of the trace metamodel. This is done by analyzing the tracing annotations and introducing only those elements into the trace metamodel that are relevant for tracing as defined by the tracing annotations. In particular, the tracing annotations are provided as additional input to the trace metamodel generation procedure defined in Algorithm 3. Based on this additional input it is checked before creating metaclasses in the trace metamodel whether the respective dynamic property or step rule is selected for tracing by a tracing annotation. Otherwise, no metaclass is added to the trace metamodel for the respective element.

Regarding the generation of trace constructors as described in Sect. 6, no adaptations are required since the trace constructors are directly generated from the trace metamodels. Because the generated trace metamodels only contain concepts that are needed for tracing as defined by the tracing annotations, corresponding trace constructors only record the selected dynamic fields, dynamic objects, and execution steps.

7.5 Automated generation of tracing annotations

Tracing annotations can be manually defined by an expert to select the dynamic properties and step rules of an xDSML that are relevant for a particular application context, *i.e.,* for performing a particular set of trace analyses. However, since xDSMLs and trace analyses can be arbitrarily complex, defining tracing annotations manually can be difficult and error-prone. For instance, an element of the xDSML may be annotated although it is never used in the application context, leading to unnecessary data being captured in any execution trace. Similarly, an element important for the application context might be missed by the expert, resulting in missing data in execution traces that may lead to undefinable trace analyses.

To cope with this problem, we propose an additional procedure to automatically derive tracing annotations on the operational semantics of an xDSML from the set of trace analyses used in a particular application context of the xDSML. This procedure takes as input (1) the set of trace analyses constituting the considered application context, (2) the trace metamodel that was used to define these trace analyses, and (3) the operational semantics of the xDSML. The steps are then as follows:

1. The static metamodel footprint of the trace analyses constituting the considered application context is computed. This static metamodel footprint captures the subset of the trace metamodel that is used in the definition of these trace analyses.
2. From the obtained static metamodel footprint, the tracing annotations for the considered application context are automatically derived and applied on the operational semantics of the considered xDSML.
3. The automatically derived tracing annotations can then be used to generate a new customized trace metamodel that focuses on the dynamic information needed for the considered application context.

In the remainder of the section, we explain the steps 1 and 2 of this procedure. We first explain the concept of static metamodel footprints and how to obtain such a footprint for trace analysis. Then we show how the static metamodel footprint of a trace analysis can be used to derive tracing annotations. Step 3, *i.e.,* generating a new customized trace metamodel and a corresponding trace constructor, is achieved by applying the proposed approach presented in Sect. 7.4.

Static trace metamodel footprints. As mentioned above, to automatically derive tracing annotations from a set of trace analyses, we propose to first calculate the *static metamodel footprint* of the considered trace analyses.

A *static metamodel footprint* is the set of metamodel elements used in the definition of a particular model oper-

ation [37]. Static metamodel footprints can be computed through a static analysis of the definition of the model operation. In our case, the investigated metamodel is a trace metamodel and the investigated model operation is a trace analysis. Hence, the static metamodel footprint is the set of elements in a trace metamodel that are used by a trace analysis. To calculate this *static trace metamodel footprint*, the abstract syntax tree of the definition of the trace analysis is traversed and for every statement it is checked whether it accesses a metaclass or property defined in the trace metamodel. If this is the case, the accessed metaclass or property is added to the footprint.

For trace analyses defined in imperative languages, such as Kermeta [40] and the Epsilon Object Language (EOL) [42], the static analysis is performed along the control flow graph of the trace analysis. Statements of the trace analysis that have to be investigated are, for instance, operation declarations, variable declarations, and property navigation expressions. For instance, for property navigation expressions, it is checked whether they access properties of metaclasses defined in the trace metamodel. Accessed properties are added to the footprint. In case the trace analysis is defined in a declarative language, such as triple graph grammars [61], the static analysis is performed on the left-hand side of all defined rules. For hybrid languages, those procedures are combined.

Computing static metamodel footprints has been extensively studied, and approaches exist for many languages, such as for Kermeta [37,38], ATL [11,59], and OCL [11]. As we discuss in Sect. 8, for our experimentations, we have implemented a static metamodel footprint calculator for the Epsilon Comparison Language (ECL) [42], which is a hybrid language allowing the definition of declarative rules and imperative rule bodies. Imperative rule bodies are defined with EOL.

It has to be noted that static metamodel footprints overapproximate the set of model elements accessed by the investigated model operation, *i.e.,* the set of collected elements includes more elements than actually used by the model operation. This is because static footprinting methods only consider types used in the model operation and do not investigate any conditions on the model's state. Furthermore, non-reachable statements in the model operation cannot be detected by static footprinting methods. To mitigate this problem of imprecision, dynamic analysis methods could be used. However, compared to dynamic methods, static metamodel footprints can be computed very efficiently and they are valid for any model processed by the investigated model operation. A detailed evaluation of static footprinting techniques is out of scope of this work, but for a detailed investigation of this topic we refer the interested reader to [37].

Example of a static trace metamodel footprint. Let us consider our illustrative example of a trace analysis determining the executed actions of a fUML activity (cf. Algorithm 9). This trace analysis comprises four statements, which access in total six elements of the trace metamodel of the fUML xDSML (the accesses to elements of the trace metamodel are underlined in Algorithm 9): the metaclasses Trace, TracedAction, and FiringValue, as well as the properties `tracedActions`, `firingSequence`, and `firing`. These elements comprise the static trace metamodel footprint of this trace analysis defined for fUML.

Generating tracing annotations from trace metamodel footprints. A trace metamodel footprint calculated from the static analysis of a trace analysis defines the elements of the trace metamodel that are accessed to perform the trace analysis. From this trace metamodel footprint, we can automatically derive the tracing annotations needed for generating a trace metamodel that is reduced to those elements relevant for the considered trace analysis. In particular, tracing annotations are created for those dynamic metaclasses and dynamic properties of the xDSML's execution metamodel that are traced by the metaclasses and properties of the trace metamodel that are accessed by the trace analysis as indicated by the trace metamodel footprint.

For identifying which dynamic metaclasses and dynamic properties of the xDSML's execution metamodel are traced by the trace metamodel elements captured in the static trace metamodel footprint, traceability information between the execution metamodel and the trace metamodel is needed. This traceability information can, for instance, be explicitly captured in an own model that maps elements of the execution metamodel to elements of the trace metamodel, or it can be automatically derived if the correspondences are known. In our case, we store traceability links between trace metamodel elements and execution metamodel elements explicitly, namely as annotations of the trace metamodel.

For our illustrative example of a trace analysis for fUML models defined in Algorithm 9, we derive two tracing annotations. One tracing annotation is produced for the dynamic property firing, because the metaclass FiringValue of the trace metamodel is used for tracing values of this dynamic property and is accessed in line 3 of the trace analysis. The second tracing annotation is generated for the dynamic metaclass Action, since the metaclass TracedAction of the trace metamodel is used for tracing instances of this metaclass Action and is accessed in line 2 of the trace analysis. The other elements accessed by the considered trace analysis are metaclasses and references that are introduced into the trace metamodel solely for navigation purposes but are not directly tracing dynamic objects or dynamic fields. Thus, no tracing annotations have to be produced for them.

In case that an application context comprises more than one trace analysis, the footprints of all these trace analyses are computed and merged before generating the tracing annotations. This way, tracing annotations are generated for all dynamic metaclasses and dynamic properties of the execution metamodel that need to be traced for performing the analyses. In case the set of trace analyses in the considered application context changes, the tracing annotations have to be recomputed.

Please note that the proposed procedure cannot be applied for automatically generating *tracing annotations for step rules* defined in the execution transformation of an xDSML. This is because the calculated trace metamodel footprint does not provide enough information to derive at which execution steps the state of an executed model should be recorded in the trace. For deriving this information, the execution transformation has to be investigated to determine which step rules have an impact on which traced dynamic objects and dynamic fields. This additional static analysis of an xDSML's execution transformation is left for future work. For now, we generate tracing annotations for each step rule defined in the execution transformation.

The procedure proposed in this section for automatically deriving tracing annotations from existing trace analyses is generally applicable for domain-specific trace metamodels. The only prerequisite is that traceability information between the execution metamodel and the trace metamodel is provided or can be derived, such that it can be determined which element of the execution metamodel is traced by which element of the trace metamodel.

8 Implementation

In this section, we present the implementation of our approach within the language and modeling workbench GEMOC Studio. We first introduce the GEMOC Studio and its execution framework. Then we give an overview of our implementation.

8.1 GEMOC Studio

The GEMOC Studio[12] is an Eclipse package built on top of the Eclipse Modeling Framework (EMF). It is composed of two main parts: a language workbench to define xDSMLs, and a modeling workbench to use defined xDSMLs.

Language workbench. The language workbench of the GEMOC Studio can be used to design and implement tool-supported xDSMLs. This includes defining the abstract syntax (using Ecore), the concrete syntax (using Sirius Animator[13]), and the operational semantics of an xDSML. The operational semantics can be defined using various transformation languages, e.g., [40] or xMOF [49]. Each considered model transformation language provides a way to define which transformation rules are step rules: Kermeta provides an annotation called `@Step`, and xMOF supports specific Ecore annotations for operations. As we explain afterward in Sect. 8.2, we implemented our approach as a component for the language workbench.

Modeling workbench. Once designed and implemented in the language workbench, the xDSMLs and associated domain-specific tools are automatically deployed in the modeling workbench of the GEMOC Studio. It allows system designers to edit, execute, simulate, and animate their models, while providing generic facilities for omniscient debugging on the basis of execution traces [9]. The modeling workbench includes advanced *execution engines* that can be used to execute any model conforming to an xDSML defined within the language workbench. In particular, an execution engine is responsible for applying the transformation rules of the operational semantics of an xDSML.

Engine add-ons are optional components that become notified by the engine about the progress of the execution (e.g., beginning of the execution, start of a step, end of a step). By reacting to engine notifications, an add-on may query the engine or the executed model for many purposes, such as providing a particular view on the execution, extracting information, or even controlling the execution of the model. As we will explain in Sect. 8.2, a trace constructor generated with our approach is implemented as an engine add-on that is deployed and used when executing a model in the modeling workbench.

8.2 Implementation in the GEMOC Studio

We implemented the three parts of our approach within the GEMOC Studio, namely the generation of trace metamodels, the generation of trace constructors, and the use of tracing annotations. All the code for this work is open-source (EPL 1.0 licensed), and can be found in publicly available source code repositories.[14]

Implementation of the generators. We implemented our trace metamodel and trace constructor generators for the GEMOC Studio using the Eclipse Modeling Framework

[12] http://gemoc.org/studio.

[13] https://www.eclipse.org/sirius.

[14] The Sirius Labs repository https://github.com/SiriusLab/ModelDebugging/tree/master/trace/generator contains the generators, and the Moliz repository https://github.com/moliz/moliz.gemoc contains the xMOF-specific libraries as well as the footprinting libraries.

(EMF) and the programming languages Xtend and Java. The input of our generators consists of an xDSML defined using Ecore for the abstract syntax, and using either Kermeta or xMOF for the operational semantics. The output is an Eclipse plugin containing the execution trace metamodel defined using Ecore, and an execution trace constructor written in Java and relying on EMF libraries. This plugin is generated from the specification of an xDSML within the language workbench, and is automatically deployed into the modeling workbench.

We have defined an intermediate representation format that models only the information we need from operational semantics. This intermediate representation contains simplified descriptions of the elements introduced in both the execution metamodel (e.g., dynamic properties) and the execution transformation (e.g., step rules). We use this representation to handle different formalisms for the specification of operational semantics in source metamodels. Our generators rely on the intermediate representation to actually generate the trace management tooling according to the Algorithms 3 and 4 of Sect. 5 of our approach. Thereby, to apply our approach to any EMF-based model transformation language, the only requirement is to implement an extractor that produces the intermediate representation.

Our generators can be called from the graphical user interface to trigger the generation for an xDSML defined in the language workbench of the studio.

Implementation of generated trace constructors. As we have mentioned previously, a trace constructor generated by our implementation takes the form of an engine add-on that is deployed in the GEMOC Studio modeling workbench. This greatly simplifies the integration of the trace constructor with the execution transformation, as the engine only is responsible for notifying add-ons about the progress of the execution. Therefore, the trace construction add-on must simply be enabled, and no modifications of the execution transformation are required for enabling the construction of traces.

Implementation of tracing annotations. To represent tracing annotations as defined in Sect. 7.3, we implemented a simple metamodel using Ecore, and we use the generic tree-based editor provided by EMF to manually create tracing annotations conforming to this metamodel. Like explained in Sect. 7.4, such tracing annotations can be provided to our generators to produce customized trace metamodels.

The automated generation of a static metamodel footprint described in Sect. 7.5 has been implemented with Java to analyze model operations specified with the Epsilon Comparison Language (ECL) [42]. ECL is employed by the semantic model differencing framework considered in the evaluation of our approach (cf. Sect. 9). The footprint metamodel needed for the generation of tracing annotations was implemented using Ecore. The automatic transformation of footprints into tracing annotation was implemented with Java, and the generated annotations are directly suitable for generating customized trace metamodels.

Examples of generated trace metamodels. We used our implementation on a selection of xDSMLs that had previously been developed using the GEMOC Studio. Table 1 lists all the xDSMLs considered so far, along with links to their source material, metrics on their size, links to their generated trace metamodels, and metrics on the generated trace metamodels. Trace construction using the generated trace constructors was also successfully tested for these languages.

9 Evaluation

To answer the posed research questions RQ1–RQ4, we have evaluated our approach in a case study on the xDSML *fUML* and the trace analysis activity *semantic model differencing*. In this section, we first provide background information on semantic model differencing, then give the setup of our case study, and finally discuss the results of the evaluation.

9.1 Semantic model differencing

Managing the evolution of models is a crucial concern in model-driven engineering since models constitute the central development artifacts. One important technique in this area is *model differencing*, which is concerned with the identification of differences among two models. Most of the existing approaches in model differencing [1,45] apply a purely syntactic approach meaning that they analyze two models in order to identify syntactic differences among them, such as additions, removals, and modifications of model elements. However, since syntactic differences among executable models may also result in differences among their behaviors, it is also of interest to identify these semantic differences. The process of identifying semantic differences among models is called *semantic model differencing* [46].

In previous work [43], we have proposed a semantic model differencing framework that relies on domain-specific analyses of execution traces. In particular, the framework performs a comparison of the execution traces of two models for identifying semantic differences among them. The comparison algorithm is implemented with so-called *semantic differencing rules* that indicate which syntactic differences among the execution traces constitute semantic differences among the models. The semantic differencing rules are not only specific to the used xDSML, but also specific to the semantic equivalence criterion suitable for the particular application context of the xDSML.

The definition of the semantic differencing rules depends directly on the employed trace format. In its original ver-

Table 1 xDSMLs on which our implementation was tested, and the resulting generated trace metamodels

xDSML	Link	Description	Semantics	C.	DC.	DP.	SR.	TMM	TMM C.	TMM P.
Petri nets	*link*[a]	Simple Petri nets (see Fig. 1)	xMOF	3	0	1	2	*link*[b]	14	16
fUML	*link*[c]	Complete fUML	xMOF	107	58	56	178	*link*[d]	776	747
ArduinoML	*link*[e]	Simple Arduino hardware and software descriptions	Kermeta	58	6	5	8	*link*[f]	50	73
TFSM	*link*[g]	Timed finite state machines	Kermeta	11	3	5	7	*link*[h]	32	56
IML	*link*[i]	AutomationML Intermediate Modeling Layer	Kermeta	18	10	4	10	*link*[j]	51	62
MiniTL	*link*[k]	Mini declarative model transformation language	Kermeta	11	2	5	7	*link*[l]	29	44
Activity diagrams	*link*[m]	Activity diagrams, inspired by fUML, from TTC'15 [50]	Kermeta	29	23	11	10	*link*[n]	61	100

Trace metamodels sizes may differ from results obtained with formulas from Sect. 5.4 due to implementation-specific concerns, inheritance relationships, and overridden rules
Semantics Language used to define the operational semantics, *C.* Number of static metaclasses, *DC.* Number of dynamic metaclasses, *DP.* Number of dynamic properties. *SR.* Number of step rules, *TMM* Generated trace metamodel, *TMM C.* Number of metaclasses in the generated trace metamodel, *TMM P.* Number of properties in the generated trace metamodel

[a] https://github.com/moliz/moliz.gemoc/tree/master/examples/petrinet/language_workbench
[b] https://github.com/moliz/moliz.gemoc/tree/master/examples/petrinet/language_workbench/org.modelexecution.xmof.examples.petrinet.trace/model/petrinetConfigurationTrace.ecore
[c] https://github.com/moliz/moliz.gemoc/tree/master/examples/fuml/language_workbench
[d] https://github.com/moliz/moliz.gemoc/tree/master/examples/fuml/language_workbench/org.modelexecution.xmof.examples.fuml.trace/model/fumlConfigurationTrace.ecore
[e] https://github.com/gemoc/arduinomodeling/tree/master/dev/language_workbench_sequential
[f] https://github.com/gemoc/arduinomodeling/tree/master/dev/language_workbench_sequential/org.gemoc.arduino.sequential.xarduino.trace/model/arduinoTrace.ecore
[g] https://github.com/gemoc/gemoc-studio/tree/master/official_samples/TFSM_PlainK3/language_workbench
[h] https://github.com/gemoc/gemoc-studio/tree/master/official_samples/TFSM_PlainK3/language_workbench/org.gemoc.sample.tfsm.sequential.xtfsm.trace/model/tfsmTrace.ecore
[i] https://github.com/moliz/moliz.gemoc/tree/master/examples/iml/language_workbench
[j] https://github.com/moliz/moliz.gemoc/tree/master/examples/iml/language_workbench/org.modelexecution.xmof.examples.iml.sequential.sequentialiml.trace/model/imlTrace.ecore
[k] https://github.com/tetrabox/minitl/tree/master
[l] https://github.com/tetrabox/minitl/tree/master/plugins/org.tetrabox.example.minitl.trace/model/minitlTrace.ecore
[m] https://github.com/gemoc/activitydiagram/tree/master/dev/gemoc_sequential/language_workbench
[n] https://github.com/gemoc/activitydiagram/tree/master/dev/gemoc_sequential/language_workbench/org.gemoc.activitydiagram.sequential.xactivitydiagram.trace/model/activitydiagramTrace.ecore

sion, our semantic model differencing framework utilizes a *generic and clone-based trace metamodel* for representing execution traces. More precisely, a trace conforming to this metamodel is a sequence of clones of the executed model that capture the model's execution states after each state change.[15] The usage of a generic and clone-based trace metamodel has two key implications on the trace analysis implemented in semantic differencing rules:

1. The generic and clone-based trace metamodel lacks *usability* for defining domain-specific trace analyses, since a state is a collection of objects of any type requiring type checking and casting for the analyses. This implies complex rules that are hard to read and comprehend.
2. *Scalability in time* is an issue when employing the generic and clone-based trace metamodel, because comparing two traces requires the traversal of all execution states, even if the trace comparison considers only a subset of the execution states.

To overcome these issues, we propose to enhance semantic model differencing by relying on multidimensional and domain-specific execution traces as presented in this work. By doing so we expect both *better usability* of traces for defining semantic differencing rules resulting in less complex semantic differencing rules that are easier to comprehend, as well as *better scalability in time* of the trace comparison performed by the semantic differencing rules. Furthermore, by relying on a multidimensional domain-specific trace metamodel instead of a generic clone-based one, we expect a *lower memory consumption* and a *lower runtime overhead* of the trace construction.

9.2 Case study

As explained above, we have evaluated our approach on multidimensional and domain-specific execution traces in a case study on semantic model differencing. For this, we have adapted our semantic model differencing framework [43] to analyze multidimensional domain-specific execution traces instead of generic clone-based ones.[16]

Considered xDSML. Based on this adapted semantic differencing framework, we have performed a case study on the real world xDSML *foundational UML (fUML)* [55] that we presented in Sect. 7.2. For this, we have first defined the execution semantics of fUML in an operational way with xMOF [49]. The execution metamodel of fUML extends one metaclass of the language's abstract syntax metamodel and defines 58 dynamic metaclasses, as well as 56 dynamic properties. The execution transformation contains 178 step rules.

Trace metamodels. Based on the definition of fUML developed with xMOF, we have generated a multidimensional and domain-specific trace metamodel for fUML, as well as an accompanying trace constructor by applying our proposed approach. The generated trace metamodel comprises 56 metaclasses for capturing values of dynamic fields, 300 metaclasses for capturing object states, and 415 metaclasses for capturing execution steps. The accompanying trace constructor comprises 16 196 lines of Java code.

In addition, we have defined tracing annotations for fUML that narrow the scope of traces down to the data actually needed for performing the semantic differencing of fUML models. The tracing annotations select one metaclass and two dynamic properties of the execution metamodel, as well as three step rules of the execution transformation of fUML for tracing. As a consequence of the defined tracing annotations, the size of the generated multidimensional and domain-specific trace metamodel is reduced to two metaclasses for capturing values of dynamic fields, ten metaclasses for capturing object states,[17] and 39 metaclasses for capturing execution steps.[18] The associated generated domain-specific trace constructor is composed of 832 lines of Java code.

Semantic differencing rules. In our previous work [43], we had defined a set of semantic differencing rules for fUML based on a generic and clone-based trace metamodel. These rules were defined using the Epsilon Comparison Language (ECL) [42]. We have adapted these rules to carry out the semantic differencing on multidimensional and domain-specific traces, instead of generic and clone-based ones. Note that these adapted rules can also be directly used for the customized fUML trace metamodel obtained with tracing annotations described above.

9.3 Experiments

To answer all research questions RQ1–RQ4 that we introduced in Sect. 4, we have conducted the following series of experiments and measures with the considered use case.

[15] The trace constructor associated to the generic and clone-based trace metamodel relies on the class EcoreUtil.Copier provided by EMF to create clones of the executed model.

[16] All evaluation material may be found at https://github.com/moliz/moliz.gemoc.

[17] This high amount of metaclasses for capturing object states is due to the types of the annotated dynamic properties and the types of the parameters of the annotated step rules, which require additional metaclasses for tracing.

[18] The high amount of metaclasses for representing only three annotated step rules is due to inheritance relationships in the execution metamodel of fUML. In particular, the metaclasses for which the annotated step rules are defined have many subtypes and for each subtype, a metaclass has to be produced in the trace metamodel.

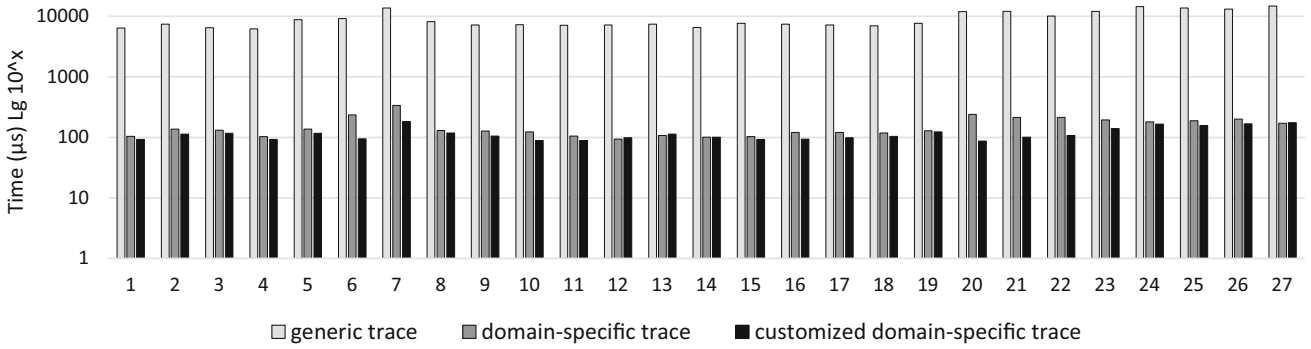

Fig. 11 Execution time of the semantic differencing rules of fUML for generic and domain-specific traces. Each identifier is a different pair of models whose traces have been compared

For answering RQ3 (usability), we have compared the complexity of the two variants of semantic differencing rules defined for fUML, *i.e.,* the semantic differencing rules defined based on the multidimensional and domain-specific trace metamodel generated with the approach proposed in this work, and the semantic differencing rules defined based on the generic and clone-based trace metamodel originally employed by the semantic model differencing framework. We measured the complexity by performing a static analysis of the defined semantic differencing rules.

For the other research questions, we have used the GEMOC Studio to execute a set of real world fUML models taken from a semantic model differencing case study realized by Maoz et al. [47] in four different configurations:

- *C1* without the construction of execution traces,
- *C2* with the construction of generic and clone-based execution traces,
- *C3* with the construction of multidimensional and domain-specific execution traces, and
- *C4* with the construction of multidimensional and domain-specific execution traces that are customized to the developed semantic differencing rules by means of tracing annotations.

For answering RQ1 (scalability in time), we have measured the time needed for executing the semantic differencing rules on the three different types of execution traces constructed for the considered set of fUML models, *i.e.,* the traces constructed with the configurations *C2–C4*. The measurements where repeated ten times and the arithmetic mean execution time was analyzed for answering the research question.

For answering RQ2 (memory), we have measured and compared the memory used by the execution traces constructed in the configurations *C2–C4*. More precisely, for each execution and constructed trace, we produced a memory dump of the heap of the Java Virtual Machine, and we analyzed it using the Eclipse Memory Analyzer.[19] Thanks to precise queries, our analysis only measures the exact memory usage of an execution trace, leaving aside the memory used by the environment or by the loaded metamodels.

Finally, for answering RQ4 (trace construction overhead), we have compared the runtime overhead induced by the construction of the different types of execution traces in the configurations *C2–C4*, by comparing each execution time to the configuration *C1* where no trace was constructed. The measurements where repeated five times and the arithmetic mean was analyzed for answering the research question.

The experiment conducted for answering research question RQ1 was performed on an Intel Core i7-4600U CPU, 2.10 GHz, 2.69 GHz, with 12 GB RAM, running Windows 8.1 Pro 64-Bit. The experiments conducted for answering the research questions RQ2 and RQ4 were performed on an Intel Core i5-4210U CPU, 1.70 GHz, with 12 GB RAM, running Fedora 23 64 bits.

9.4 Results

In the following, we present the results of the evaluation and discuss the answers to the research questions RQ1–RQ4 that follow from these results.

RQ1: Reduction of trace manipulation time. Figure 11 shows the execution times measured for applying the semantic differencing rules on the traces of the considered example models. The *X*-axis shows the identifier of the compared model-pair and the *Y*-axis shows the measured execution time in microseconds on a logarithmic scale.

As can be seen from the measurements, the rules analyzing traces conforming to the multidimensional domain-specific trace metamodel outperform the rules analyzing generic and clone-based traces. They are executed between 40 and 87 times faster with an average of 62. The main hypothesis for

[19] http://www.eclipse.org/mat/.

explaining this result is that the multidimensional structure of the domain-specific trace metamodel allows to efficiently explore the trace through dedicated navigation paths related to specific model elements. The usage of a customized domain-specific trace metamodel specific to the semantic differencing rules brought no major performance improvements for the performed trace analysis itself. However, the time needed for loading the analyzed traces was significantly reduced due to the reduced size of the traces. Please note that the time needed for loading the traces is not included in the execution times shown in Fig. 11.

These results allow us to answer RQ1 as follows: multidimensional domain-specific trace metamodels provide a smaller trace manipulation time compared to a generic clone-based trace metamodel.

RQ2: Reduction of memory consumption. Figure 12 shows the memory consumption of the execution traces obtained with the generic and the domain-specific trace metamodels, while Fig. 13 shows the memory consumption of the customized domain-specific trace metamodel. Each data point represents the amount of memory used by an execution trace obtained from execution of one particular model. The X-axis shows the amount of execution states captured by the considered trace. The Y-axis shows the amount of memory used by the trace in kilobytes (kB). The generic and domain-specific traces are put on the same graph because they both contain the same amount of execution states, while the customized domain-specific traces contain less execution states due to the performed customizations.

Generic traces require 4.1 to 4.5 times more memory than domain-specific traces with an average of 4.3. Domain-specific traces require 20.4 to 25.4 times more memory

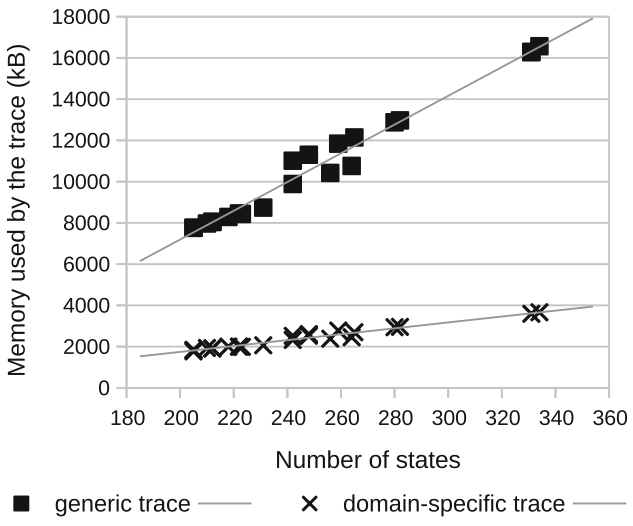

Fig. 12 Memory consumption of generic and domain-specific traces, ordered by the amount of execution states constructed

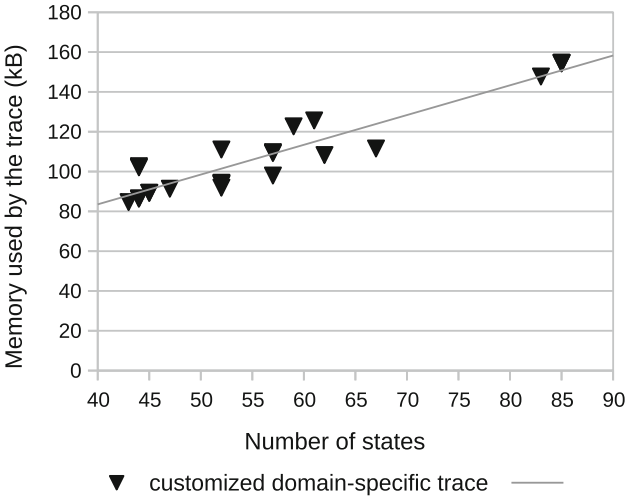

Fig. 13 Memory consumption of customized domain-specific traces, ordered by the amount of execution states constructed

Table 2 Complexity of the semantic differencing rules of fUML defined for the generic (G) and domain-specific (DS) trace metamodel

Elements	G	DS	Reduction (%)
Lines of code	109	45	59
Statements	62	27	56
Operations	15	6	60
Operation calls (analysis-specific operations)	19	6	68
Loops	5	4	20
Conditions	11	2	73
Type checks	4	0	100
Feature accesses	18	10	44

than customized domain-specific traces with an average of 22.6. In addition, we observe that the memory consumption of domain-specific traces increases less rapidly with an increased number of execution states than the memory consumption of generic traces, which suggests better scalability in memory for large executions. Our main hypothesis for explaining this result is that domain-specific traces are designed to only contain the evolution of dynamic objects with minimal redundancy and, hence, consume less memory than clone-based ones that contain significant redundancy.

To summarize and answer RQ2, we observe that domain-specific traces are more efficient in memory than generic and clone-based traces. Moreover, customized domain-specific traces reduce even more significantly the memory consumption for a specific application context.

RQ3: Complexity reduction of trace analyses. Table 2 compares the complexity of the semantic differencing rules defined for fUML based on the generic and clone-based trace metamodel, and the multidimensional and domain-specific

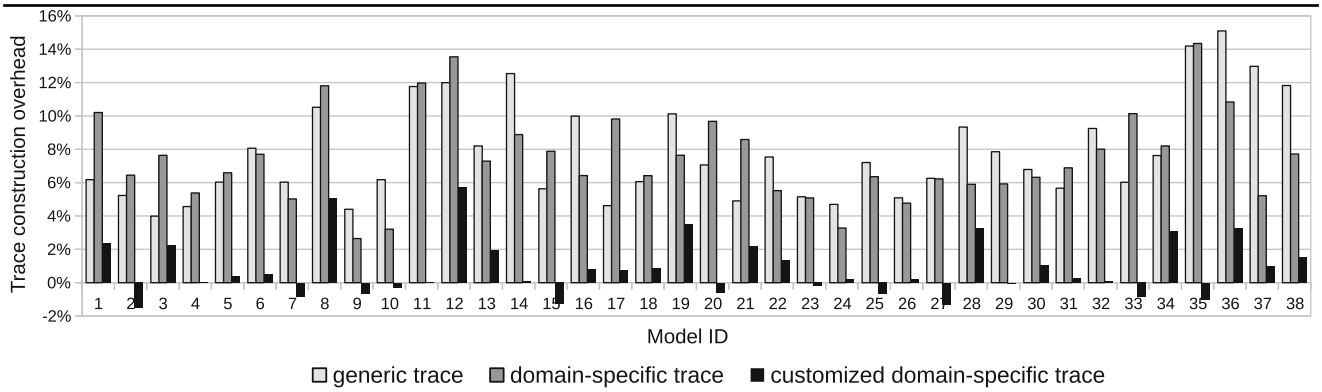

Fig. 14 Runtime overhead due to the trace construction, for each executed model and each trace metamodel

trace metamodel. For all elements, we observe a significant reduction of the complexity of the rules ranging from 20 to 100%.

The trace analysis defined on traces conforming to the generic and clone-based trace metamodel comprises 109 lines of code distributed in 15 operations defining 62 statements in total. Five loops are required for traversing the states captured in the traces, as well as the quite complex data structure of the execution metamodel of fUML. Furthermore, four type checks and 18 feature navigation expressions are required during these traversals to extract the data necessary for the trace comparison.

The trace analysis defined based on trace models conforming to the generated multidimensional and domain-specific trace metamodel consists of 45 lines of code, six operations, and 27 statements. This reduction of 56% in terms of overall statements is mainly due to the multidimensional structure of the generated domain-specific trace metamodel enabling a more efficient traversal of the execution traces. 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 objects relevant for performing the trace comparison can be directly accessed. Furthermore, due to the domain-specific nature of the trace metamodel, type checks become obsolete.

These results allow us to answer RQ3 as follows: multidimensional and domain-specific trace metamodels simplify the definition of domain-specific trace manipulations compared to a generic and clone-based trace metamodel.

RQ4: Reduction of runtime overhead for trace construction. Figure 14 shows the runtime overhead induced by constructing execution traces, *i.e.,* the percentage of additional execution time spent on constructing a trace, using the three considered trace metamodels and corresponding trace constructors. The X-axis shows the identifiers of each executed model, and the Y-axis shows the percentage of runtime overhead due to trace construction. Please note that some results are negative in the case of customized domain-specific traces because the execution times are overall very close to the case without trace construction, and because execution time measurements are subject to fluctuations.

On average, the runtime overhead comprises 7.81% for constructing a generic trace, 7.25% for construction a domain-specific trace, and 0.86% for constructing a customized domain-specific trace. Overall, we observe that the overhead for constructing domain-specific traces is similar to the overhead for constructing generic traces, and heavily depends on the considered execution.

Our hypothesis for explaining these results is that despite the fact that a domain-specific trace constructor has much less elements to create, it contains a significant amount of conditional expressions to analyze the execution state of the executed model, while a generic trace constructor can simply copy each object of the executed model into the trace. Constructing a customized domain-specific trace, on the other hand, always induces a smaller overhead on the execution, which is due to both the very small amount of calls made to the trace constructor (since few step rules are traced) and the small amount of data stored in each execution state (since fewer dynamic properties are considered).

To summarize and answer RQ4, our approach does not significantly reduce the runtime overhead induced by constructing domain-specific traces as compared to generic clone-based traces, although the overhead remains quite low and under 10%. However, the construction of a trace conforming to a customized domain-specific trace metamodel tailored for a considered application context has a negligible overhead on the execution, even below 1%.

10 Related work

In this section, we present existing approaches that are comparable to our solution. We first focus on methods for defining domain-specific trace metamodels, then we look at existing work on multidimensional trace data structures and finally we examine how self-defining trace formats can be related to our work.

10.1 Defining domain-specific trace data structures

Hegedüs et al. [34] propose a generic execution trace metamodel that must be manually extended into a domain-specific trace metamodel using inheritance relationships. They consider a trace to be a sequence of both changes and snapshots of objects of the model, with no representation of the complete execution state. We can summarize three main differences with our approach. First, the structure is different from ours, both because we take into account the complete execution states of the model, and because we only consider high-level changes (*i.e.,* execution steps) corresponding to a relevant subset (*i.e.,* domain-specific) of the execution semantics. Second, their approach consists in extending a generic execution trace metamodel using inheritance, while we generate a complete metamodel with customized metaclasses and properties for the considered xDSML. Thereby, we aim to avoid both type checks and casting, and to provide trace metamodels that are closer to the domain of the considered xDSML. Lastly, their approach is manual, while ours is generative and automatized.

In the context of the TOPCASED project [13,14,17], we proposed the definition of a *trace management metamodel* specific to the model of computation of an xDSML. Such a trace metamodel is only concerned with event occurrences in an execution (corresponding to execution steps), while our present approach considers also execution states. In addition, like the approach by Hegedüs et al. [34], their approach is manual while ours is generative and automatized.

Gogolla et al. [27] generate *filmstrip models* from UML metaclass diagrams to represent the evolution of a system's state. Such filmstrip models incorporate the original UML metaclasses, and thus match what we call domain-specific trace metamodels. However, the generated metaclasses are almost identical to the ones from the input UML metaclass diagrams, hence leading to a trace metamodel equivalent to a clone-based one. Yet, as we have previously explained and shown with our evaluation, clone-based approaches do not scale in memory and offer poor usability.

Meyers et al. [51] propose the *ProMoBox* framework, which generates a set of metamodels from an annotated xDSML, including a trace metamodel. More precisely, they provide a clone-based generic execution trace metamodel that is extended into a domain-specific metamodel by their generative approach. While being generative like ours, their approach differs from our approach in multiple aspects. First, they consider an abstract syntax whose properties are annotated either as *runtime* or *event* to identify dynamic 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. Furthermore, it facilitates a fully automated derivation of the concepts that have to be introduced into a domain-specific trace metamodel for capturing the execution states and execution steps of a model. Second, the obtained trace metamodel is clone-based, since each recorded execution state is a complete snapshot of the executed model with the same limitations as the approach proposed by Gogolla et al. [27]. Finally, similarly to the approach by Hegedüs et al. [34], inheritance relationships are used to extend a base trace metamodel, while we generate new metaclasses to avoid having to rely on introspection and casting when manipulating traces.

10.2 Multidimensional trace data structures

Few approaches propose multidimensional facilities to follow the evolution of specific model elements within a trace.

Filmstrip models from Gogolla et al. [27]—mentioned above for their domain-specific aspect—provide a structure that makes possible to follow the evolution of a single object of a model, which facilitates the analysis of specific elements. This is very similar to the *dimensions* we propose in our approach. However, because a new snapshot of an object is created at each execution step, following such navigation path requires as many iterations as browsing the complete execution trace of the model. Moreover, we consider a dimension to be at the level of a *field*, while they consider the level of an *object*.

KMF runtime versioning [32] stores the versions of each object of a model separately, allowing to enumerate the states of a specific object of the executed model. Their approach does allow to navigate among the states of a model from the perspective of a specific element of the model, hence with much fewer iterations. However, their approach is generic and does not capture a domain-specific execution trace metamodel. Moreover, similarly to Gogolla et al. [27], they consider changes at the level of an *object*.

10.3 Self-defining trace formats

Self-defining trace formats are formats allowing to define the data structure of a trace within the metadata of the trace itself. This can be compared to a model that would embed its own metamodel. Examples of such trace formats include Pablo SDDF [3], Pajé [60] and the trace metamodel of the SOCTrace project [57]. For the tracing of embedded systems and operating systems, a well-known self-defining trace format is the Common Trace Format (CTF) [20].

While a self-defining trace format cannot directly be used to construct the traces of an xDSML, it could potentially be an interesting alternative to MOF for the definition of trace metamodels. For instance, adapting our approach to generate domain-specific metadata for the Common Trace Format (CTF) [20] would make possible to benefit from a very memory efficient binary format. However, since we con-

sider xDSMLs to be defined using metamodels defined with MOF, using such meta-formats for execution traces would make it difficult to properly define links within the executable model at both the metamodel and the model level. Moreover, to our knowledge, self-defining trace formats do not provide multidimensional navigation facilities.

11 Conclusion

Dynamic V&V of models requires the ability to model executions traces. We identified three important requirements regarding the definition of a *trace metamodel* for an xDSML: It must provide good *usability* for the development of domain-specific trace analyses, good *scalability in time* for manipulating traces, and good *scalability in memory*. Existing trace formats are not adequate because of their distance to the domain of an arbitrary xDSML and because of their lack of alternative trace exploration means. The approach we presented consists in automatically generating a *multidimensional* and *domain-specific* trace metamodel of an xDSML, using the xDSML's definition of what the execution state of a model is, and which steps may occur during an execution. We reify the dynamic properties of the execution metamodel into metaclasses, allowing both to reduce redundancy of data captured in traces, and to narrow the scope of the trace metamodel to the considered xDSML. We also provide navigation paths both to follow the evolution of each dynamic field of the model over time, and to follow the occurrences of each step definition. This allows an efficient navigation of traces, *i.e.,* an exploration without visiting each state of the trace. Furthermore, we propose the concept of tracing annotations, which can be used to further reduce the scope of the trace metamodel to the dynamic data and execution steps that are relevant for a particular application context of the considered xDSML. This improves the usability of trace metamodels and traces, reduces the memory footprint of traces, and improves the performance of the trace construction and analyses. 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. Furthermore, we observed a reduced memory footprint of traces as well as a reduced runtime overhead induced by the trace construction during the execution of models.

12 Perspectives

There are many direct perspectives for this work. We present a selection of them in the following paragraphs.

Interactions with the execution environment. A model may interact with its execution environment for allowing an external source (e.g., the user, a solver, sensors) to provide input data or to resolve non-deterministic situations. Therefore, if the operational semantics of an xDSML explicitly defines the possible external stimuli that a conforming model can handle, this information could be taken into account in the corresponding domain-specific trace metamodel. This would allow to capture even more precise execution traces from models conforming to an xDSML. For instance, a step could contain information on the external stimulus responsible for its application.

Execution branches. Depending on the operational semantics, executing a model can be non-deterministic. This means that a single model may yield different executions, and therefore different execution traces. In particular, the semantics may handle input stimuli (see previous paragraph), or may contain a concurrency model which introduces non-determinism [15]. Yet, different executions may share a common prefix before diverging. Consequently, it would be possible to use a single execution trace model to represent a set of executions sharing a common prefix, which would result in a reduction of the memory footprint, and give the possibility to explore these different executions when analyzing a trace. Such *branches* in an execution trace could be used by trace analyses to efficiently look at multiple similar scenarios. This would require generating appropriate trace metamodels that allow the construction of different branches, each starting from an execution state of another branch.

Common generic structural interface. Although defining a generic trace manipulation for a domain-specific trace metamodel is possible, it is not possible to directly reuse it for another xDSML. A solution to manipulate any domain-specific trace in a generic way is the definition of a common generic structural interface, such as a generic trace metamodel. We already proposed such idea and metamodel in [9]. In that paper, given an xDSML, we proposed to automatically generate a one-way model transformation to obtain a generic trace from a domain-specific one, in order to enable the generic *analysis* of traces. However, other solutions can be considered for also enabling the generic *modification* of traces, such as a bidirectional model transformation between the generic and domain-specific trace metamodels, or metamodel substitutability techniques such as model subtyping [28].

Compression of traces. To further reduce memory consumption of execution traces, it is possible for certain types of value changes to store the *delta* between a former value and a new value, instead of a complete new value. For instance, if a dynamic field is a collection of elements, an execution state can contain additions and removals of elements instead of a complete new collection. Likewise, if a dynamic field is a

string, the same can be accomplished by looking at additions and removals of substrings. However, for a given execution state, such storage model would require reading a complete sequence of deltas to reconstruct the actual value of such dynamic field. Another perspective is the detection of redundant *patterns* within execution steps, as proposed in [30], in order to store patterns only once and to be able to create simpler pattern instances inside the trace.

Extended notion of dimension. Our approach defines a dimension as the sequence of values reached by a single dynamic field of an executed model. We chose this low level of granularity so that even the smallest change in the execution state may be tracked efficiently when reading a trace. However, trace analyses might require to follow the evolution of multiple dynamic fields *at the same time*, which may require some non-intuitive queries when using our generated trace metamodels. Therefore, coarse-grained possibilities could also be explored, such as having one dimension per dynamic object (similarly to [27]), each containing the sequence of states reached by an object. And going further, in order to consider any level of granularity, a dimension could be more generally based on a *tuple* of dynamic fields, possibly scattered among different dynamic objects. Defining what dimensions should be considered for a given set of trace analyses could also be part of an extension of the customization process that we proposed.

Acknowledgements Open access funding provided by TU Wien (TUW). This work is supported by the ANR INS Project GEMOC (ANR-12-INSE-0011), the COST Action MPM4CPS (IC1404), the Christian Doppler Forschungsgesellschaft CDL-Flex, the BMWFW, the Austrian Science Fund (FWF): P 28519-N31, the Austrian Agency for Cooperation in Education and Research (OeAD) under the grand number FR 08/2017 and by the French Ministries of Foreign Affairs and International Development (MAEDI) and the French Ministry of Education, Higher Education and Research (MENESR).

References

1. Alanen, M., Porres, I.: Difference and union of models. In: Proceedings of the 6th International Conference on the Unified Modeling Language (UML'03), LNCS, vol. 2863, pp. 2–17. Springer (2003). doi:10.1007/978-3-540-45221-8_2
2. Alawneh, L., Hamou-Lhadj, A.: MTF: a scalable exchange format for traces of high performance computing systems. In: Proceedings of the 19th International Conference on Program Comprehension (ICPC'11), pp. 181–184. IEEE (2011). doi:10.1109/ICPC.2011.15
3. Aydt, R.: The Pablo Self-Defining Data Format. Technical Report, Department of Computer Science at the University of Illinois at Urbana-Champaign (1992)
4. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, Cambridge (2008)
5. Bandener, N., Soltenborn, C., Engels, G.: Extending DMM behavior specifications for visual execution and debugging. In: Proceedings of the Third International Conference on Software Language Engineering (SLE'10), vol. 6563 LNCS, pp. 357–376. Springer, Berlin (2010). doi:10.1007/978-3-642-19440-5_24
6. Bendraou, R., Combemale, B., Crégut, X., Gervais, M.P.: Definition of an executable SPEM 2.0. In: Proceedings of the 14th Asia-Pacific Software Engineering Conference (APSEC'07), pp. 390–397. IEEE (2007). doi:10.1109/APSEC.2007.38
7. Bousse, E., Combemale, B., Baudry, B.: Scalable armies of model clones through data sharing. In: Proceedings of the 17th International Conference on Model Driven Engineering Languages and Systems (MODELS'14), LNCS, vol. 8767, pp. 86–301. Springer (2014). doi:10.1007/978-3-319-11653-2_18
8. Bousse, E., Combemale, B., Baudry, B.: Towards scalable multidimensional execution traces for xDSMLs. In: Proceedings of the 11th Workshop on Model-Driven Engineering, Verification and Validation (MoDeVVa'14), CEUR-WS, vol. 1235, pp. 13–18. CEUR (2014). http://ceur-ws.org/Vol-1235/paper-03.pdf
9. Bousse, E., Corley, J., Combemale, B., Gray, J., Baudry, B.: Supporting efficient and advanced omniscient debugging for xDSMLs. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering (SLE'15), pp. 137–148. ACM (2015). doi:10.1145/2814251.2814262
10. Bousse, E., Mayerhofer, T., Combemale, B., Baudry, B.: A generative approach to define rich domain-specific trace metamodels. In: Proceedings of the 11th European Conference on Modelling Foundations and Applications (ECMFA'15), LNCS, vol. 9153, pp. 45–61. Springer (2015). doi:10.1007/978-3-319-21151-0_4
11. Burgueño, L., Troya, J., Wimmer, M., Vallecillo, A.: Static fault localization in model transformations. IEEE Trans. Softw. Eng. **41**(5), 490–506 (2015). doi:10.1109/TSE.2014.2375201
12. Combemale, B., Crégut, X., Garoche, P.L., Thirioux, X.: Essay on semantics definition in mde—an instrumented approach for model verification. J. Softw. **4**(9), 943–958 (2009). doi:10.4304/jsw.4.9.943-958
13. Combemale, B., Crégut, X., Giacometti, J.P., Michel, P., Pantel, M.: Introducing simulation and model animation in the MDE topcased toolkit. In: Proceedings of the 4th European Congress on Embedded Real Time Software and Systems (ERTS'08) (2008)
14. Combemale, B., Crégut, X., Pantel, M.: A design pattern to build executable DSMLs and associated V&V tools. In: Proceedings of the 19th Asia-Pacific Software Engineering Conference, pp. 282–287. IEEE (2012). doi:10.1109/APSEC.2012.79
15. Combemale, B., Deantoni, J., Larsen, M.V., Mallet, F., Barais, O., Baudry, B., France, R.: Reifying concurrency for executable metamodeling. In: Proceedings of the 6th International Conference on Software Language Engineering (SLE'13), LNCS, vol. 8225, pp. 365–384. Springer (2013). doi:10.1007/978-3-319-02654-1_20
16. Corley, J., Eddy, B.P., Gray, J.: Towards efficient and scalabale omniscient debugging for model transformations. In: Proceedings of the 14th Workshop on Domain-Specific Modeling (DSM'14), pp. 13–18. ACM (2014). doi:10.1145/2688447.2688450
17. Crégut, X., Combemale, B., Pantel, M., Faudoux, R., Pavei, J.: Generative technologies for model animation in the TopCased platform. In: Proceedings of the 6th European Conference on Modeling Foundations and Applications (ECMFA'10), LNCS, vol. 6138, pp. 90–103. Springer, Berlin (2010). doi:10.1007/978-3-642-13595-8_9
18. Csertan, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., Varro, D.: VIATRA-visual automated transformations for formal verification and validation of UML models. In: Proceedings of the 17th International Conference on Automated Software Engineering (ASE'02), pp. 267–270. IEEE (2002). doi:10.1109/ASE.2002.1115027

19. DeAntoni, J., Mallet, F., Thomas, F., Reydet, G., Babau, J.P., Mraidha, C., Gauthier, L., Rioux, L., Sordon, N.: RT-simex: retro-analysis of execution traces. In: Proceedings of the 18th International Symposium on the Foundations of Software Engineering (FSE'10), pp. 377–378. ACM (2010). doi:10.1145/1882291.1882357
20. Desnoyers, M.: Common Trace Format (CTF) Specification (v1.8.2) (2013). http://git.efficios.com/?p=ctf.git;a=blob_plain;f=common-trace-format-specification.md;hb=master
21. Eichler, H., Soden, M.: An Approach to Behaviour Comparison Using Execution Traces. Technical Report, Department of Computer Science, Humboldt University Berlin (2009)
22. Engels, G., Hausmann, J.H., Heckel, R., Sauer, S.: Dynamic metamodeling: a graphical approach to the operational semantics of behavioral diagrams in UML. In: Proceedings of the Third International Conference on the Unified Modeling Language (UML'00), LNCS, vol. 1939, pp. 323–337. Springer, Berlin (2000). doi:10.1007/3-540-40011-7_23
23. 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: Proceedings of the 14th International Conference on Parallel Computing, Advances in Parallel Computing, vol. 22, pp. 481–490. IOS Press (2011). doi:10.3233/978-1-61499-041-3-481
24. Esmaeilsabzali, S., Day, N.A.: Prescriptive semantics for big-step modeling languages. In: Proceedings of the 13th International Conference on Fundamental Approaches to Software Engineering (FASE'10), LNCS, vol. 6013, pp. 158–172. Springer, Berlin (2010). doi:10.1007/978-3-642-12029-9_12
25. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story diagrams: a new graph rewrite language based on the unified modeling language and java. In: Proceedings of the 6th International Workshop Theory and Application of Graph Transformations (TAGT'98), vol. 1764, pp. 157–167 (2000). doi:10.1007/978-3-540-46464-8_21
26. Geimer, M., Wolf, F., Wylie, B.J.N., Ábrahám, E., Becker, D., Mohr, B.: The Scalasca performance toolset architecture. Concurr. Comput. Pract. Exp. **22**(6), 702–719 (2010). doi:10.1002/cpe.1556
27. 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)
28. Guy, C., Combemale, B., Derrien, S.: On Model Subtyping. In: Proceedings of the 8th European Conference on Modeling Foundations and Applications (ECMFA'12), LNCS, vol. 7349, pp. 400–415. Springer (2012). doi:10.1007/978-3-642-31491-9_30
29. Haerder, T., Reuter, A.: Principles of transaction-oriented database recovery. ACM Comput. Surv. **15**(4), 287–317 (1983). doi:10.1145/289.291
30. Hamou-Lhadj, A., Lethbridge, T.C.: A metamodel for the compact but lossless exchange of execution traces. Softw. Syst. Model. **11**(1), 77–98 (2010). doi:10.1007/s10270-010-0180-x
31. Harel, D., Lachover, H., Naamad, A., Pnuelli, A., Politi, M., Sherman, R., Shtull-trauring, A., Trakhtenbrot, M.: Statemate: a working environment for the development of complex reactive systems. IEEE Trans. Softw. Eng. **16**(4), 403–414 (1990). doi:10.1109/ICCSSE.1988.72235
32. Hartmann, T., Fouquet, F., Nain, G., Morin, B., Klein, J., Barais, O., Traon, Y.L.: A native versioning concept to support historized models at runtime. In: Proceedings of the 17th International Conference on Model Driven Engineering Languages and Systems (MODELS'14), LNCS, vol. 8767, pp. 252–268. Springer (2014). doi:10.1007/978-3-319-11653-2_16
33. Hegedüs, Á., Bergmann, G., Ráth, I., Varró, D.: Back-annotation of simulation traces with change-driven model transformations. In: Proceedings of the 8th International Conference on Software Engineering and Formal Methods (SEFM'10), pp. 145–155. IEEE (2010). doi:10.1109/SEFM.2010.28
34. Hegedüs, Á., Ráth, I., Varró, D.: Back-Annotation Framework for Simulation Traces of Discrete Event-based Languages. Technical Report, BME (2010). https://inf.mit.bme.hu/sites/default/files/publications/Hegedus_TechRep_201004.pdf
35. Hegedüs, A., Ráth, I., Varró, D.: Replaying execution trace models for dynamic modeling languages. Period. Polytech. Electr. Eng. **56**(3), 71–82 (2012)
36. Hutchinson, J., Whittle, J., Rouncefield, M., Kristoffersen, S.: Empirical assessment of MDE in industry. In: Proceedings of the 33rd International Conference on Software Engineering (ICSE'11), pp. 471–480. ACM (2011). doi:10.1145/1985793.1985858
37. Jeanneret, C., Glinz, M., Baudry, B.: Estimating footprints of model operations. In: Proceedings of the 33rd International Conference on Software Engineering (ICSE'11), pp. 601–610. ACM (2011). doi:10.1145/1985793.1985875
38. Jeanneret, C., Glinz, M., Baudry, B.: Footprinting Operations Written in Kermeta. Technical Report IFI-2011.0002, University of Zurich, Department of Informatics (IFI) (2011). https://www.merlin.uzh.ch/contributionDocument/download/2130
39. Jouault, F., Kurtev, I.: Transforming models with ATL. In: Proceedings of the Workshop on Model Transformations in Practice (MTiP'05), LNCS, vol. 3844, pp. 128–138. Springer, Berlin (2006). doi:10.1007/11663430_14
40. Jézéquel J.M., Combemale B., Barais O., Monperrus M., Fouquet F. (2013) Mashup of metalanguages and its implementation in the Kermeta language workbench. Softw. Syst Model. doi:10.1007/s10270-013-0354-4
41. Kelly, S., Pohjonen, R.: Worst practices for domain-specific modeling. IEEE Softw. **26**(4), 22–29 (2009). doi:10.1109/MS.2009.109
42. Kolovos, D., Rose, L., García-Domínguez, A., Paige, R.: The epsilon book (2016). https://www.eclipse.org/epsilon/doc/book
43. Langer, P., Mayerhofer, T., Kappel, G.: Semantic model differencing utilizing behavioral semantics specifications. In: Proceedings of the 17th International Conference on Model Driven Engineering Languages and Systems (MODELS'14), LNCS, vol. 8767, pp. 116–132. Springer (2014)
44. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Log. Algebr. Program. **78**(5), 293–303 (2009). doi:10.1016/j.jlap.2008.08.004
45. Lin, Y., Gray, J., Jouault, F.: DSMDiff: a differentiation tool for domain-specific models. Eur. J. Inf. Syst. **16**(4), 349–361 (2007)
46. Maoz, S., Ringert, J.O., Rumpe, B.: A manifesto for semantic model differencing. In: Models in Software Engineering: Workshops and Symposia at MODELS 2010, Reports and Revised Selected Papers, LNCS, vol. 6627, pp. 194–203. Springer (2011). doi:10.1007/978-3-642-21210-9_19
47. Maoz, S., Ringert, J.O., Rumpe, B.: ADDiff: Semantic differencing for activity diagrams. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, pp. 179–189. ACM (2011). doi:10.1145/2025113.2025140
48. Mayerhofer, T., Langer, P., Kappel, G.: A runtime model for fUML. In: Proceedings of the 7th Workshop on Models@run.time (MRT'12), pp. 53–58. ACM (2012). doi:10.1145/2422518.2422527
49. Mayerhofer, T., Langer, P., Wimmer, M., Kappel, G.: xMOF: executable DSMLs based on fUML. In: Proceedings of the 6th International Conference on Software Language Engineering (SLE'13), LNCS, vol. 8225, pp. 56–75. Springer (2013). doi:10.1007/978-3-319-02654-1_4
50. Mayerhofer, T., Wimmer, M.: The TTC 2015 model execution case. In: Proceedings of the 8th Transformation Tool Contest (TTC'15),

CEUR Workshop Proceedings, vol. 1524, pp. 2–18. CEUR-WS.org (2015). http://ceur-ws.org/Vol-1524
51. Meyers, B., Deshayes, R., Lucio, L., Syriani, E., Vangheluwe, H., Wimmer, M.: ProMoBox: a framework for generating domain-specific property languages. In: Proceedings of the 7th International Conference on Software Language Engineering (SLE'14), LNCS, vol. 8706, pp. 1–20. Springer (2014). doi:10.1007/978-3-319-11245-9_1
52. OASIS: Web Services Business Process Execution Language Version 2.0 (2007). https://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html
53. Object Management Group: OMG Unified Modeling Language (OMG UML), V 2.5 (2013). http://www.omg.org/spec/UML/2.5
54. Object Management Group: UML Testing Profile (UTP), V 1.2 (2013). http://www.omg.org/spec/UTP/1.2/
55. Object Management Group: Semantics of a Foundational Subset for Executable UML Models (fUML), V 1.2.1 (2015). http://www.omg.org/spec/FUML/1.2.1
56. Object Management Group: Meta Object Facility (MOF) Core Specification, V 2.5 (2016). http://www.omg.org/spec/MOF/2.5
57. Pagano, G., Dosimont, D., Huard, G., Marangozova-Martin, V., Vincent, J.M.: Trace management and analysis for embedded systems. In: Proceedings of the 7th International Symposium on Embedded Multicore Socs (MCSoC'13), pp. 119–122. IEEE (2013). doi:10.1109/MCSoC.2013.28
58. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science (SFCS'77), pp. 46–57. (1977). doi:10.1109/SFCS.1977.32
59. Sánchez Cuadrado, J., Guerra, E., de Lara, J.: Reverse engineering of model transformations for reusability. In: Proceedings of the 7th International Conference on the Theory and Practice of Model Transformations (ICMT'14), LNCS, vol. 8568, pp. 186–201. Springer (2014). doi:10.1007/978-3-319-08789-4_14
60. Schnorr, L.M., Stein, O., Chassin, J.: Paje trace file format, V 1.2.5 (2013). http://paje.sourceforge.net/download/publication/lang-paje.pdf
61. Schürr, A.: Specification of graph translators with triple graph grammars. In: Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science (WG'94), pp. 151–163. Springer (1995)
62. Seidewitz, E., Cuccuru, A.: Agile programming with executable models: an open-source, standards-based eclipse environment. In: Proceedings of the 2014 Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH'14), Companion Volume, pp. 39–40. ACM (2014). doi:10.1145/2660252.2664664
63. Soden, M., Eichler, H.: Towards a model execution framework for Eclipse. In: Proceedings of the 1st Workshop on Behaviour Modelling in Model-Driven Architecture (BD-MDA'09). ACM (2009). doi:10.1145/1555852.1555856
64. STMicroelectronics: KPTrace Specification (2012). http://www.stlinux.com/stworkbench/interactive_analysis/stlinux.trace/kptrace_traceFormat.html
65. Tatibouët, J., Cuccuru, A., Gérard, S., Terrier, F.: Formalizing execution semantics of UML profiles with fUML models. In: Proceedings of the 17th International Conference on Model Driven Engineering Languages and Systems (MODELS'14), LNCS, vol. 8767, pp. 133–148. Springer (2014). doi:10.1007/978-3-319-11653-2_9
66. Whittle, J., Hutchinson, J., Rouncefield, M.: The state of practice in model-driven engineering. IEEE Softw. **31**(3), 79–85 (2014). doi:10.1109/MS.2013.65

Erwan Bousse is a postdoctoral researcher at the Business Informatics Group at TU Wien. He has obtained his Ph.D. in France in 2015 at the University of Rennes 1 for his work on execution traces and omniscient debugging of executable models. His current research interests include model transformation testing, language engineering, model execution, and efficient execution trace management for executable models. For more information, please visit http://www.big.tuwien.ac.at/staff/ebousse.

Tanja Mayerhofer is a postdoctoral researcher at the Business Informatics Group at TU Wien. She has received her Ph.D. in 2014 from TU Wien for her work on executable modeling based on fUML. Her current research interests focus on model-driven engineering, specifically on modeling language engineering, model execution, and model analysis. For more information, please visit http://www.big.tuwien.ac.at/staff/tmayerhofer.

Benoit Combemale received his Ph.D. in Computer Science from the University of Toulouse in 2008, and his habilitation in Computer Science from the University of Rennes in 2015. He first worked at Inria before joining the University of Rennes 1 in 2009. He is now associate professor of Computer Science at the University of Rennes 1, specializing in software engineering, and a member of both the IRISA and Inria Labs. His research interests include model-driven engineering (MDE), software language engineering (SLE), and validation & verification (V&V). For more information, please visit http://people.irisa.fr/Benoit.Combemale/.

Benoit Baudry is a research scientist at INRIA since 2004 and leads the DiverSE research group (EPI) since 2013. His research focuses on software design, testing and analysis. Most of his work is driven by empirical observations about software and how quality can be improved. He contributes in the areas of software testing, model-driven engineering, software metrics, and automatic software diversification. For further information, please visit http://people.rennes.inria.fr/Benoit.Baudry/.