Omniscient Debugging for Executable DSLs

Erwan Bousse, Dorian Leroy, Benoit Combemale, Manuel Wimmer, Benoit Baudry

Omniscient Debugging for Executable DSLs

Erwan Bousse*

TU Wien, Austria

Dorian Leroy

JKU Linz, Austria

Benoit Combemale

University of Toulouse, France

Manuel Wimmer

CDL-MINT, TU Wien, Austria

Benoit Baudry

KTH Royal Institute of Technology, Sweden

Abstract

Omniscient debugging is a promising technique that relies on execution traces to enable free traversal of the states reached by a model (or program) during an execution. While a few General-Purpose Languages (GPLs) already have support for omniscient debugging, developing such a complex tool for any executable Domain Specific Language (DSL) remains a challenging and error prone task. A generic solution must: support a wide range of executable DSLs independently of the metaprogramming approaches used for implementing their semantics; be efficient for good responsiveness. Our contribution relies on a *generic* omniscient debugger supported by *efficient* generic trace management facilities. To support a wide range of executable DSLs, the debugger provides a common set of debugging facilities, and is based on a pattern to define runtime services independently of metaprogramming approaches. Results show that our debugger can be used with various executable DSLs implemented with different metaprogramming approaches. As compared to a solution that copies the model at each step, it is on average six times more efficient in memory, and at least 2.2 faster when exploring past execution states, while only slowing down the execution 1.6 times on average.

Keywords: Software Language Engineering; Domain-Specific Languages; Executable DSL; Omniscient debugging; Execution trace

1. Introduction

A large amount of Domain-Specific Languages (DSLs) have been proposed and used to describe the dynamic aspects of systems [1, 2, 3, 4, 5]. In that context, early *dynamic* verification and validation (V&V) techniques, such as testing [6], semantic differencing [7] or runtime verification [8], are necessary to ensure that such executable models (or programs[1]) are correct. These techniques require models to be *executable*, which can be achieved by defining the execution semantics of the DSLs used to define them. To that effect, efforts have been made to provide facilities to design so-called *executable DSLs*. [9, 10, 11, 12, 13, 14, 15]. More precisely, two different approaches are commonly used to define the execution semantics of an executable DSL: operational semantics (*i.e.,* interpretation) and translational semantics (*i.e.,* compilation). We focus in this work on the case of operational semantics, *i.e.,* model interpretation.

Among dynamic V&V techniques, *interactive debugging* (ofter shortened to *debugging*) is a common facility. It enables the observation and control of an execution to better understand a certain behavior or to look for the cause of a defect. However, standard debugging only provides facilities to pause and step *forward* during an execution. This requires developers to restart the execution from the beginning to give a second look at a state of interest. To address this issue, *omniscient debugging* is a promising

*Corresponding author

Email addresses: erwan.bousse@tuwien.ac.at (Erwan Bousse), dorian.leroy@jku.at (Dorian Leroy), benoit.combemale@irit.fr (Benoit Combemale), wimmer@big.tuwien.ac.at (Manuel Wimmer), baudry@kth.se (Benoit Baudry)

[1]We arbitrarily use the term *executable model* since this work focuses on *metamodel-based DSLs*, but the term *program* can be used interchangeably.

technique that relies on execution traces to enable free traversal of the states reached by a model, thereby allowing developers to "go back in time" [16].

While most general-purpose languages (GPLs) already have their own efficient standard debugger or omniscient debugger, developing such a complex tool for any executable DSL remains a difficult and error prone task. In this context, a disciplined approach to obtain an omniscient debugger for any executable DSL with little cost would be very valuable. However, there are two main challenges to consider: (i) It must be *generic*, to be able to apply such an approach on a wide range of executable DSLs, independently of the metaprogramming approaches used for their implementation (*i.e.,* the patterns and languages used to implement their semantics); (ii) It must be *efficient*, because omniscient debugging is an interactive activity that may imply executing large and complex models and constructing large execution traces. In addition, there is necessarily a trade-off between both challenges, since supporting any executable DSL may require the use of expensive introspection, conditionals, or type checks to support a wide variety of syntax and runtime data structures.

To provide a *generic* approach, we make the following proposals. First, despite the specificities of each DSL, it is possible to identify a common set of debugging facilities (*e.g.,* breakpoints, stepping operators, etc.) that are compatible with a wide range of executable DSLs. Thus, to avoid the manual development of each debugger, we propose the definition of a *unique generic omniscient debugger* that provides this common set of debugging facilities. However, each metaprogramming approach that can be used to implement an executable DSL has its own characteristics. Common tasks may differ from one approach to another, including how to interrupt the execution of a model. Interruption is not only required for debugging, but also for any other *runtime service* (*e.g.,* trace construction, monitoring, etc.). Therefore, we propose the use of a pattern where an *execution engine* is responsible for applying the semantics and for interrupting the execution by sending notifications to *execution listeners* (*e.g.,* the debugger).

Omniscient debugging requires the construction of an execution trace, which can be costly to construct and to manipulate. Therefore, our contribution relies on *efficient and generic trace management facilities*. These facilities include: a *trace metamodel* that precisely captures the execution steps and the states of a model under execution; a *trace constructor* to construct execution traces conforming to this trace metamodel; a *model state manager* to restore an executed model to a state stored in an execution trace. These facilities are designed to avoid both redundant data in memory, and redundant trace manipulations.

We implemented our approach as part of the GEMOC Studio, a language and modeling workbench. The empirical evaluation we conducted evaluates the genericity of our approach: we tested our generic omniscient debugger with 10 different executable DSLs defined with three different metaprogramming approaches. Using example models, we successfully observed and controlled their executions with the omniscient debugger regardless of their executable DSLs and of the used metaprogramming approaches. We then evaluated the efficiency of our solution with regard to memory consumption and the time required to run omniscient debugging operations. As far we know, this is the first generic approach that brings omniscient debugging to a wide range of targeted executable DSLs. Consequently, we compared our approach with two other omniscient debuggers that we defined: one that simulates omniscient debugging by resetting the execution engine and re-executing until the target state is reached; one that copies the model at each execution step. The results show that our approach is on average six times more efficient in memory when compared to the second debugger, and at least respectively 60 and two times faster when exploring past states compared to the first and second debuggers, while only slowing down the execution 1.6 times on average.

This paper is a significant extension of our previous work [17, 18, 19]. The extension comprises (i) a pattern to interrupt model executions and to decouple runtime services from metaprogramming approaches, (ii) a fully generic set of trace management facilities, (iii) a revised generic omniscient debugger, (iv) an up-to-date description of the current tool-support, (v) a more thorough review of related work.

The remaining sections are organized as follows:

- Section 2 defines the required background, *i.e.,* the considered scope of executable DSLs and the notions of interactive and omniscient debugging.
- Section 3 gives an overview of our approach.
- Sections 4, 5, and 6 are our three main contributions:
  - Section 4 presents a method to interrupt the execution of models on execution steps, using a pattern that decouples runtime services (*e.g.,* debugging) from metaprogramming approaches.
  - Section 5 presents a set of efficient and generic trace management facilities for executable DSLs.
  - Section 6 presents how to apply the two previous contributions (Sections 4 and 5) to provide a generic omniscient debugger for a wide range of executable DSL.
- Section 7 describes our implementation.
- Section 8 presents the evaluation of our approach.
- Finally, Section 9 discusses related work and Section 10 concludes the paper.

2. Background

In this section, we introduce the background required by the approach we present afterwards. We first summarize the scope of considered executable DSLs. Then we briefly introduce interactive debugging and omniscient debugging.

2.1. Scope of Considered Executable DSLs

In this subsection, we summarize the scope of executable DSLs considered in our approach, *i.e.,* executable DSLs defined by a metamodel-based abstract syntax and discrete-event operational semantics. Note that we focus on *external* executable DSLs, *i.e.,* DSLs with their own syntax and not defined as part of a host language.

2.1.1. Abstract Syntax

There are two main approaches to define the abstract syntax of a Domain-Specific Language (DSL), namely *grammar-based* and *metamodel-based* approaches [20]. In this paper, we consider that an abstract syntax is defined using a metamodel. Yet, while the presented approach is not directly applicable to grammar-based DSLs, the general idea could be adapted to this technical space.

Metamodel-based approaches rely on a metamodeling language (*e.g.,* MOF [21] or Ecore [22]) to define the abstract syntax of a DSL in the form of a class-based object-oriented model, *i.e.,* a metamodel. In this paper, we define a metamodel and a model as follows:

Definition 1. *A* metamodel *is composed of:*

- *A set 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. A metaclass can be abstract, and may inherit from another metaclass.*
- Static semantics*, that are additional structural constraints that must be satisfied by conforming models (*e.g., *multiplicities, containment references, and invariants on the structure of models).*

Definition 2. *Given a metamodel, a* model *is a set of* objects *that are instances of the metaclasses of this metamodel, and that satisfy the static semantics of the metamodel. This model is said to be* conforming *to its metamodel. An object is composed of a set of* fields*, each matching a property of the corresponding metaclass.*

The top of Figure 1 depicts the abstract syntax of a Petri nets DSL. It is a metamodel composed of three metaclasses: Net, Place and Transition. Each metaclass contains properties; for instance Net contains a set of Place objects through the `places` reference.

Figure 2 shows the concrete syntax representation of a Petri net model conforming to the abstract syntax of Figure 1. It is composed of one object instance of Net (not explicitly shown), five objects instances of Place ($p1$, $p2$, etc.), and three objects instances of Transition ($t1$, $t2$, $t3$).

2.1.2. Execution Semantics

There are two general approaches to define execution semantics of executable DSLs, namely *translational* semantics (*i.e.,* compilation) and *operational* semantics (*i.e.,* interpretation) [23]. In this paper, we deal with operational

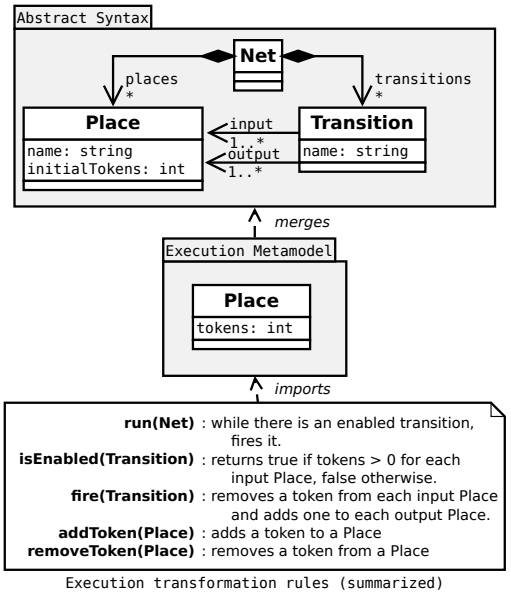


Figure 1: Petri net executable DSL.

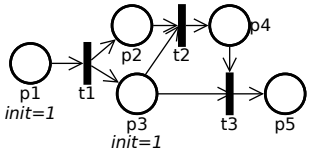


Figure 2: Concrete syntax representation of a Petri net model conforming to the abstract syntax of the DSL from Figure 1.

semantics, and we leave the case of translational semantics for future work. More precisely, we only focus on discrete-event operational semantics.

We distinguish three parts for an operational semantics. First, to define the state of an model under execution, we consider that the abstract syntax of an executable DSL can be extended into an *execution metamodel* with new properties and metaclasses. To this end, an extension mechanism equivalent to the well-known *package merge* operation of UML and MOF [24, 21] is used.

Second, we call *execution transformation* the transformation that changes the state of a model under execution. This transformation is composed of a set of *transformation rules*, each defining a subset of the changes performed on the model state. Depending on the metaprogramming approach used to define this transformation, rules can take different forms. In declarative languages [25, 26], a rule is composed of a source pattern and a target pattern, and pattern matching is used to schedule rule execution. In imperative languages [27, 10], a rule is an operation (or method) that can call other rules, one rule 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 model under execution 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

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 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)
```

for executable DSLs (*e.g.,* graphical animation).

Third and lastly, in order to execute a model originally expressed with the abstract syntax metamodel, the *initialization transformation* translates such a model into a model conforming to the execution metamodel, *i.e.,* extended with a transient dynamic state.

To summarize, the executable DSLs considered in the scope of this paper can be defined in the following way:

Definition 3. *An* executable DSL *is defined by:*

- *An* abstract syntax*, that is a metamodel.*
- *An* operational semantics*, composed of:*
  - *An* execution metamodel*, that defines the* state of executable models *by extending the abstract syntax with new dynamic properties and new dynamic metaclasses.*
  - *An* initialization transformation*, that transforms a model conforming to the abstract syntax into a model conforming to the execution metamodel.*
  - *An* execution transformation*, that modifies in-place a model conforming to the execution metamodel by changing values of dynamic fields and by creating/destroying instances of new metaclasses introduced in the execution metamodel.*

Figure 1 shows an example of a simple Petri net executable DSL. Next to the abstract syntax, the execution metamodel is shown. It extends the metaclass Place using *package merge* with a new dynamic property `tokens`. The initialization transformation (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 rules defined in the execution transformation are depicted. These rules are defined using an imperative approach. When called, these rules may change the `tokens` fields of the different Place objects, with *run* being the entry point of the transformation. Algorithms 1 and 2 show the definitions of both the *run* and *fire* rules.

2.2. Interactive and Omniscient Debugging

While the general goal of dynamic V&V is to check that a system fulfills its intended behavior, *debugging* is more specifically concerned with finding the cause of an unintended behavior (*i.e.,* a failure), and removing the defect responsible for this behavior [28, 29]. In particular, the term debugging is often directly associated with *interactive debugging* (shortened to *debugging* in most of the paper), which is a popular technique that allows developers to both control and observe an execution with the help of an *interactive debugger*.

With interactive debugging, controlling an execution means being able to *pause* and *unpause* it at will. Historically, interactive debuggers of programming languages (*e.g.,* `jdb`, `gdb`, etc.) have provided such control through *breakpoints*, which are conditions upon which the execution must be interrupted (*e.g.,* reach a specific statement). This implies that the execution is always explored in a *forward* fashion. When paused, observation is traditionally provided in the form of views, such as the current stack of method calls, or the values of all existing variables.

More recently, interactive debugging was extended into so-called *omniscient debugging* [30], which aims at exploring an execution in a *backward* fashion. This includes the possibility to define breakpoints in the past, execute the model backwards, or stepping back over a previously executed statement. Omniscient debugging avoids having to completely restart some execution to revisit a previously reached state. To be able to jump back in time, omniscient debugging approaches require storing the past states or events within an execution trace.

3. Approach Overview

Figure 3 shows an overview of the approach we propose. We build on existing approaches for model execution, and hence assume that: the considered executable DSL has been implemented by a DSL engineer in accordance with the definitions given in Section 2.1; the model under execution contains a transient dynamic state obtained after applying the *initialization transformation* of the DSL.

At the bottom left corner, the first part of the approach requires the annotation of a subset of the transformation rules of the execution transformation with *step annotations* (a). At the top left corner, these annotations are required by the *execution engine*, which is responsible both for running the execution transformation and for notifying *listeners* (*i.e.,* the trace constructor and the debugger) when rules with step annotations are being executed. Such

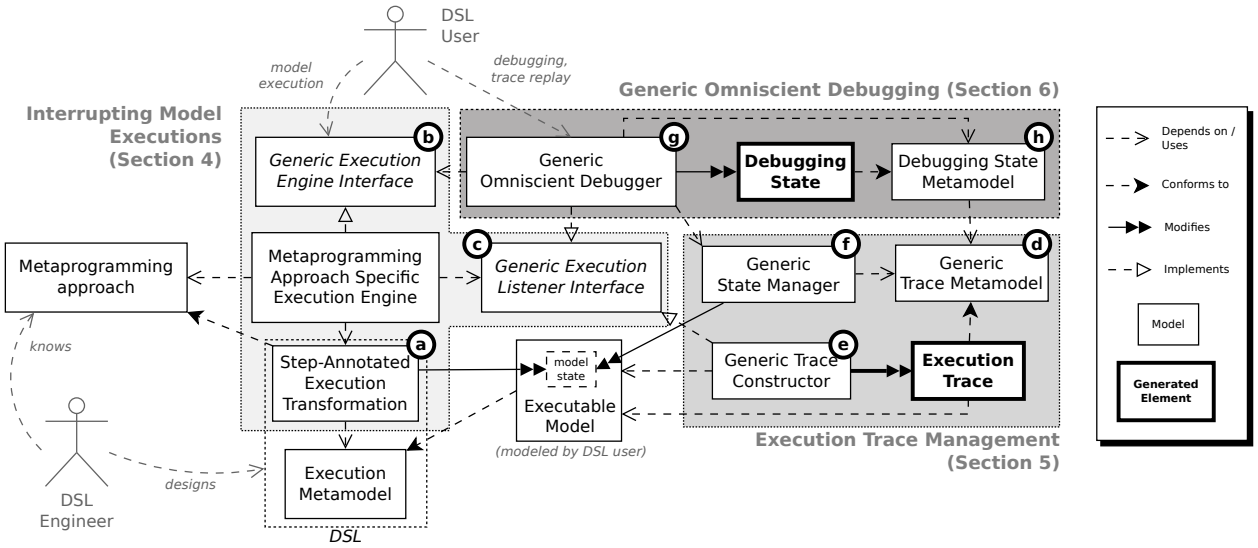


Figure 3: Architecture overview. Some dependency arrows are not shown for clarity.

an engine must comply with a *generic execution engine interface* (b) and likewise listeners must comply with a *generic execution listener interface* (c). We present this first part of the approach in Section 4.

At the bottom right corner, the second part of the approach is a set of generic trace management facilities, including a trace metamodel (d), a trace constructor (e), and a model state manager (f). They enable the construction and the manipulation of an execution trace for the omniscient debugger. We present this second part of the approach in Section 5.

At the top, the third part of the architecture is related to the execution and the debugging of the model. First, model execution can be performed using the execution engine. Next, the *generic omniscient debugger* (g) can be used both for debugging and trace replay. The debugging state conforms to the *debugging state metamodel* (h) and contains the internal state of the debugger. We present this third part of the architecture in Section 6.

4. Interrupting Model Executions

This section presents the first part of the approach, which is a method to interrupt the execution of models in between execution steps, where the granularity of execution steps are specified in the operational semantics of the DSL. Interruption of the execution is required for omniscient debugging both to probe the model regularly to construct an execution trace, and to be able to pause the execution at relevant instants, *e.g.,* when a breakpoint is enabled. We first motivate the need for such method. We then explain how to annotate operational semantics to define the execution steps of an executable DSL. Lastly, we describe a pattern to notify listeners when such execution steps occur and give an example of execution engine.

4.1. Motivation and Challenges

Given an executable DSL, executing a conforming model consists of the application of the execution transformation defined in the operational semantics: the rules of the transformation are applied one at a time and each may modify the model state (*i.e.,* the dynamic elements of the executed model). In parallel, an executable DSL may require the definition of *runtime services* (*e.g.,* debuggers, trace constructors, monitors) that may interrupt the execution transformation to observe or control the model. Such services must be connected to the execution transformation itself, which raises three main challenges.

First, to avoid interruptions in inconsistent states, it is required to specify *when* the execution can be interrupted. For example, observing the model state of a Petri net in the middle of firing a transition (*e.g.,* when input tokens have been removed, but output tokens have not been added) makes little sense, since firing a transition is supposed to be an atomic action.

Second, even after having specified when the execution can be interrupted, a mechanism is required to achieve such interruptions, *i.e.,* to allow runtime services to observe or control the execution.

Third, there are in practice many possible metaprogramming approaches to define the execution transformation of an executable DSL. Each approach has different characteristics, including different transformation languages, or different ways to structure the transformation. Hence, the task of interrupting the execution transformation may differ from one approach to another. One solution is to directly integrate the runtime service within the execution transformation itself. However, this situation leads to a dependency towards a metaprogramming approach, thus limitating its reuse by other executable DSLs defined using different metaprogramming approaches.

To address the first challenge, we propose to annotate

the operational semantics of the executable DSL with *step annotations*, which specify when can the execution be interrupted. To address the second and third challenges, we propose a simple pattern that relies on the notions of *execution engine* and *execution listener*. Note that the proposed interruption method only consists of informing listeners about occurring execution steps, and not about the possible changes in the model state.

4.2. Adding Step Annotations

To provide runtimes services, it is required to specify *when* the execution of a model can be safely interrupted and observed by these services. Since transformation rules are the main components of the execution transformation of an operational semantics, they represent good candidates for specifying when interruptions are possible. Therefore, the approach we propose relies on the use of *step annotations* on transformation rules. An annotation specifies that the execution can be interrupted both at the beginning and at the end of the application of an annotated rule. We call *step rule* a rule with a step annotation.

As an example, let us consider the Petri net executable DSL depicted in Figure 1. The semantics of Petri nets usually specify that firing a transition is an atomic operation, *i.e.,* we can only observe the marking of a Petri net before and after a transition has been fired. Therefore, we decide to annotate *fire* as a step rule, and we deliberately do not annotate *isEnabled*, *addTokens*, and *removeTokens* to forbid interruptions during *fire*. In addition, for illustration purposes, we choose to annotate *run* as a step rule. In summary, we have the following set of transformation rules $rules_{\text{PN}}$ for the Petri net executable DSL, with *run* and *fire* being annotated with the annotation «step»:

$$rules_{\text{PN}} = \{run_{\text{«step»}}, fire_{\text{«step»}}, isEnabled \; addTokens, removeTokens\}$$

We call *execution step* (or *step*) the application of a step rule. A step may contain multiple *nested* steps, which is only possible if the considered metaprogramming approach gives the possibility to call or use a transformation rule within another transformation rule.

As an example, when considering the Petri net executable DSL depicted in Figure 1, the definition of *fire* in Algorithm 2 shows that it never relies on another step rule. The definition of *run* in Algorithm 1 shows that it relies on the step rule *fire*. Hence, the application of *run* results in a step, composed of a number of nested *fire* steps.

4.3. A Pattern for Interrupting On Execution Steps

To provide runtimes services, a mechanism is required to interrupt the execution on execution steps. In addition, it must be possible to decouple runtime services from metaprogramming approaches. For these purposes, we propose a simple pattern relying on the notions of *execution engine* and *execution listener*.

4.3.1. Execution Engine

We propose a fixed generic interface that execution engines must comply with to facilitate their integration in a generic environment. A concrete engine that complies with this interface is necessarily specific to a given metaprogramming approach, as it should rely internally on specific ways to interact with transformations conforming to this meta-programming approach. We define this generic engine interface as the set of following services:

- ***initialize***: load an executable DSL and a conforming model, prepare the initial model state and the execution transformation.
- ***execute***: run the execution transformation.
- ***attachListener***: attach a new listener to the engine.
- ***detachListener***: detach a listener.

4.3.2. Execution Listener

To enable the reuse of runtime services among executable DSLs and metaprogramming approaches, we propose a fixed generic interface for interrupting to an execution. A listener that implements this interface is notified by the engine about the start and completion of each execution step, and can thereby provide runtime services. To provide a real interruption during the execution, these notifications must be made *synchronously* by the engine, *i.e.,* the execution must be suspended while the notifications are being handled by listeners. This allows a listener to safely access or modify the model state of the model while it is not being modified. We define the generic listener interface as the set of following services:

- ***executionStarting***: called when the execution starts.
- ***executionEnding***: called when the execution ends.
- ***stepStarting(Step)***: called when an execution step starts, and provide information on the starting step (*i.e.,* identifier and parameters).
- ***stepEnding()***: called when an execution step finishes.
- ***dependsOn(Listener)***: returns true if this listener depends on another listener, *i.e.,* if it should receive notifications after another listener.

Calling these services in the right order and at the correct instants is part of the *notification protocol*, which we present thereafter.

4.3.3. Notification Protocol

In addition to complying with the execution engine interface, an engine must follow a simple notification protocol that constrains when and how notifications must be sent to attached listeners. The notification protocol must be followed throughout the execution of the *execute* service of an execution engine, and consists of the following rules:

- Listeners must be notified in an order that satisfies their dependency relationships. These dependencies are available using the *dependsOn* service. If dependencies are conflicting (*e.g.,* two listeners declare depending on one another), the order is undefined.
- *executionStarting* must be called at the very beginning of execution transformation, *i.e.,* before any transformation rule has started.
- *executionEnding* must be called at the very end of the execution transformation, *i.e.,* after all transformation rules have ended.
- *stepStarting* must be called each time an execution step is *starting*.
- *stepEnding* must be called each time an execution step is *ending*.

4.3.4. Example of Execution Engine

Since an execution engine can be arbitrarily complex, we provide one example of execution engine to illustrate the protocol, which is an engine for transformations with a unique *entry point* rule. An example of engine requiring a scheduling strategy is described in Appendix A.

We consider a metaprogramming approach allowing the definition of rules that may call one another, and where one rule is considered to be the *entry point* of the transformation. Executing the complete transformation then simply consists in executing this entry point rule, which will by itself directly and indirectly execute other rules. This is how the semantics of the Petri net executable DSL presented in Section 2.1 is defined, where the *run* step rule acts as the entry point rule.

For such an engine, the main challenge is to know when each step is starting and ending, even though a rule is never directly triggered by the execution engine (except for the entry point). One solution is to rely on *callbacks* made by the transformation to the engine both when a step starts, and when a step ends. Enabling such callbacks can be achieved in different ways, such as:

- **Manual or automatic instrumentation**: The execution transformation can be modified to perform callbacks to the engine. For instance, the *fire* step rule shown in Figure 2 can be modified to perform a first callback at its very beginning, and a second at its the very end. This can be done manually or automatically through a model/program transformation.
- **Aspect weaving**: If the language supports it, aspect oriented programming (*e.g.,* AspectJ [31]) can be used to specify *pointcuts* at the beginning and at the end of all step rules to perform callbacks.
- **Dedicated metaprogramming feature**: If the considered metaprogramming language already provides a feature to perform callbacks before and after the applications of a rule (*e.g.,* xMOF virtual machine, `@Step` annotation in Kermeta), the engine can directly rely on such a feature to receive callbacks.

Figure 4 shows two sequence diagrams illustrating the behavior of an execution engine specific to an entry-point based metaprogramming approach. At the top, Figure 4a depicts how the engine is started by an external actor labeled "Engine caller". The *initialize* service is called to prepare the execution with a given executable DSL whose execution transformation conforms to the metaprogramming approach, and with a model conforming to the DSL. The *execute* service is then called to start the execution, which immediately triggers the *executionStarting* notification towards execution listeners. Continuing, the entry point rule is executed (described hereafter), and finally all listeners are notified with *executionEnding*.

At the bottom, Figure 4b depicts how a transformation rule is executed, including the entry point rule. First, if it is a step rule, a callback is made so that the engine notifies all listeners using *stepStarting*. Then the actual content of the rule is executed, which can include modifications to the model and calls to nested rules. If a nested rule is called the same Figure 4b can be read recursively. Finally, if it is a step rule, listeners are notified with *stepEnding*.

5. Efficient Generic Execution Trace Management

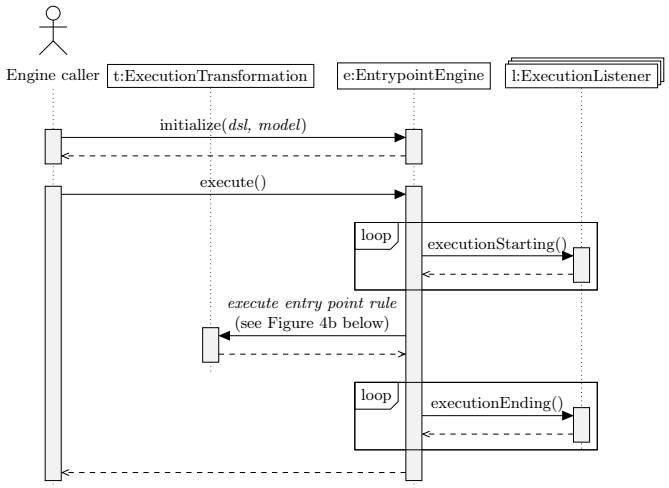
This section presents the second part of the approach, which is the definition of efficient and generic execution trace management facilities. Such facilities are required for providing omniscient debugging, as they give the possibility to revisit past execution states without restarting the execution of the model from the beginning.

We first define what we call an execution trace in the context of executable DSLs. We then present the three trace management facilities: the execution trace metamodel, the trace constructor, and the model state manager.

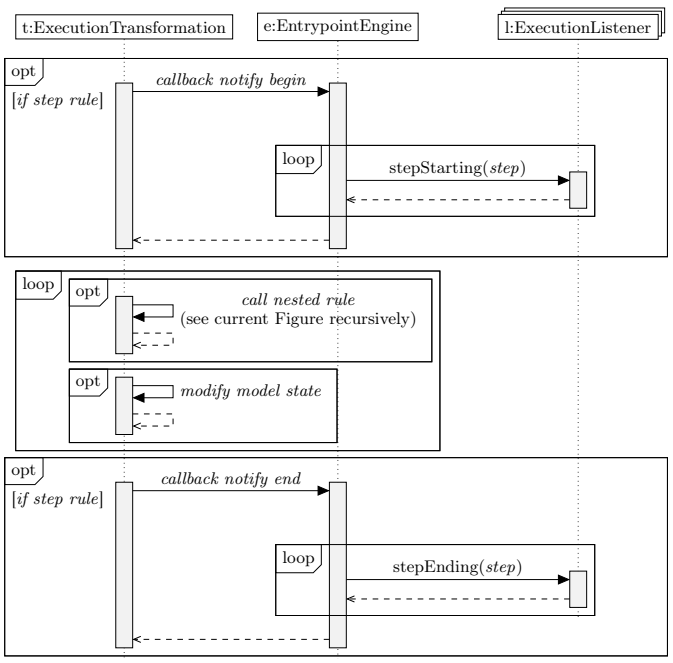
5.1. Definition

While in practice execution traces can take various forms (*e.g.,* logs, list of events, tree of method calls, memory dumps), we consider in this work that an execution trace is a sequence of model states and steps. We call *model 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 model state is changed by the application of the rules of the execution transformation. Note that an object created during the execution is indirectly part of the model state, since its fields are all dynamic.

Definition 4. *A* model state *is the set of the values of all dynamic fields of a model at a certain point in time of the execution. The model state is changed by the application of rules of the execution transformation.*



(a) Start of a model execution using the entry point rule of the execution transformation.



(b) Execution of a rule of the execution transformation, which may call nested rules.

Figure 4: Example of engine based on an entry point execution rule, and on callbacks triggered from the rules themselves.

We consider that an execution trace records all model states, as well as the execution steps causing changes on the model state. In other words, we consider a trace as a specific form of state-transition system. This generic definition is valid for any executable DSL.

Definition 5. *An* execution trace *is a sequence of model states and steps responsible for the model state changes.*

Figure 5 presents a trace from the execution of a Petri net conforming to the Petri net DSL shown in Figure 1. The trace is composed of four model states, on top of which the steps of the execution are depicted. States are separated by three steps that represent the applications of the *fire* step rule. A step goes from the first model state to the last model state to represent the application of the *run* rule. Execution states (*e.g.,* $\langle 1 \rangle$) are specific to debugging, and are therefore explained later in Section 6.1.

5.2. Execution Trace Metamodel

In this subsection, we present the generic trace metamodel used to represent the execution traces required by the omniscient debugger.

Figure 6 shows the generic trace metamodel. First, consider the left and middle parts of the metamodel. The root metaclass is called Trace. The states of the model under execution are stored as ModelState elements, while the steps are stored as Step elements. The relationships between ModelState and Step are required to know in which ModelState a Step is starting or ending. The Trace contains the sequence of all reached ModelState elements, and contains the root Step of the execution. Each other Step is contained in a parent Step (such as the root Step), thus execution steps are stored as a *call tree*.

Second, consider the right part of the metamodel. A TracedObject element corresponds to a dynamic object (*i.e.,* an object with dynamic fields) of the model being executed. Each TraceObject contains all the values reached by the corresponding dynamic object. This is achieved through the use of Dimension elements, each containing the sequence of values reached by one dynamic field of the corresponding dynamic object. These values are stored as elements typed by the Value abstract metaclass. On the Figure, three kinds of values are shown as subclasses of Value: ReferenceValue for references, ManyReferenceValue for references with multiple elements, and AttributeValue for primitive values. The subclasses of AttributeValue are not shown, such as IntegerAttributeValue or StringAttributeValue.

For example, consider using this trace metamodel to represent the Petri net execution trace shown in Figure 5. The Trace root contains one root Step for *run*, which itself contains three nested Step element for the multiple *fire* steps. The Trace also contains a sequence of four ModelState elements, and one TracedObject per Petri net Place. Each TracedObject then contains exactly one Dimension corresponding to the `tokens` dynamic field, and each Dimension contains each amount of `tokens` ever possessed by the corresponding Place during the execution.

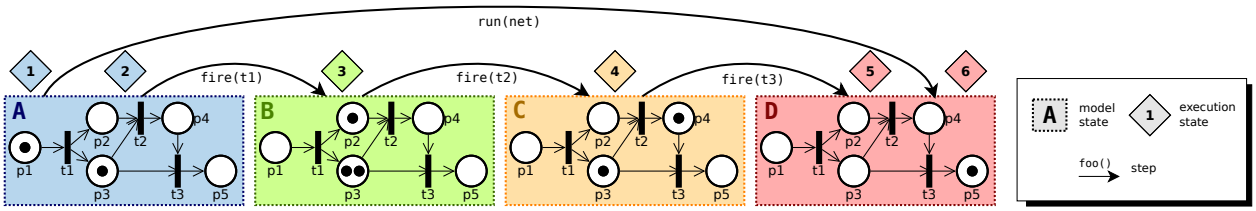


Figure 5: Example of Petri net execution trace, annotated with execution states.

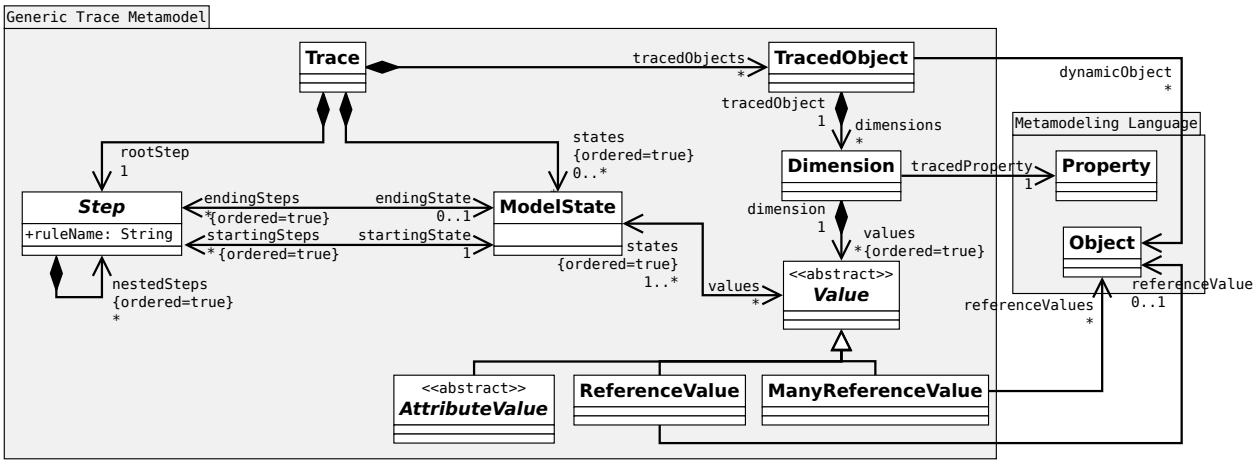


Figure 6: Generic execution trace metamodel

Efficiency of the metamodel. While being simple, the structure of the proposed generic trace metamodel is designed for efficient data capture. First, it focuses on the model state, and does not capture any superfluous information on the syntactic parts of the model under execution. This is the opposite from clone-based execution trace management approaches, where the complete model is cloned to store each model state. Second, when one value changes in the model state, only one new Value object is created. Then, the new ModelState references all previous Value elements that have not changed, and adds one reference to the new Value object. Thereby, values are shared among ModelState elements, and redundancies are avoided.

5.3. Execution Trace Constructor

In this subsection, we present the execution trace constructor used to obtain traces conforming to the generic trace metamodel. We first specify a set of generic trace construction services. Then we explain how to automatically call these services by interrupting the execution of a model using the method introduced in Section 4.

5.3.1. Specifying Generic Trace Construction Services

We specify a set of five generic services that can be used to perform trace construction tasks. The goal is to call these services during the execution of the model. It is assumed that the trace constructor has an internal state with a stack of the *in progress* steps, in order to keep track in which order steps must end in the trace.

- ***createRoot***: create the root Trace object.
- ***addInitialState***: create the first ModelState object of the model, with one TracedObject per initial dynamic object, each containing one Dimension object per dynamic field, each containing one Value object.
- ***addState***: add a new ModelState 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 ModelState object.
  - for each new dynamic object in the model, create a corresponding new TracedObject and corresponding Dimension objects, each with and initial Value object, and add these Value objects both to the corresponding TracedObject and to the new ModelState.
  - for each dynamic field that changed, create a Value object, and add it to both the corresponding Dimension and to the new ModelState.
  - for each dynamic field that did not change, add the last Value of this field to the ModelState.
  - for each dynamic object removed from the model, the corresponding Dimension will not be given new Value objects anymore, and new ModelState objects will not refer to any Value of the dynamic object anymore.
- ***addStep*** (*stepRuleID, stepRuleParams*): add a new Step in the Trace. This includes:

Listening services	Definition
executionStarting()	[1] createRoot() [2] addInitialState()
stepStarting(*step* : Step)	[1] addState() [2] addStep(*step*)
stepEnding()	[1] addState() [2] finishStep()

Table 1: Definition of the listening services of the generic trace constructor, using the trace construction services.

  - create a new Step object corresponding to the *stepRuleID* and containing the *stepRuleParams*,
  - if there was already a Step in progress, add the new Step object as a nested step of this Step,
  - set the current ModelState as the starting model state of the new Step.

- ***finishStep***: set the current ModelState as the ending model 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 metaprogramming approach (*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. However, the 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, our approach only looks 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.,* the value objects are not copied, 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.

5.3.2. Integration as an Engine Listener

We specified above a set of generic trace construction services. To construct a trace, these services must be called at relevant instants of the execution of a model. For such kind of purpose, we defined in Section 4 a method to interrupt the execution of models, where listeners receives notifications on occurring execution steps. Therefore, by complying with the listener interface, the trace constructor can call trace construction services when necessary.

Table 1 shows the listening services of the generic trace constructor. To describe these services, Figure 7 shows a trace construction scenario corresponding to the beginning of the execution of the Petri net from Figure 5.

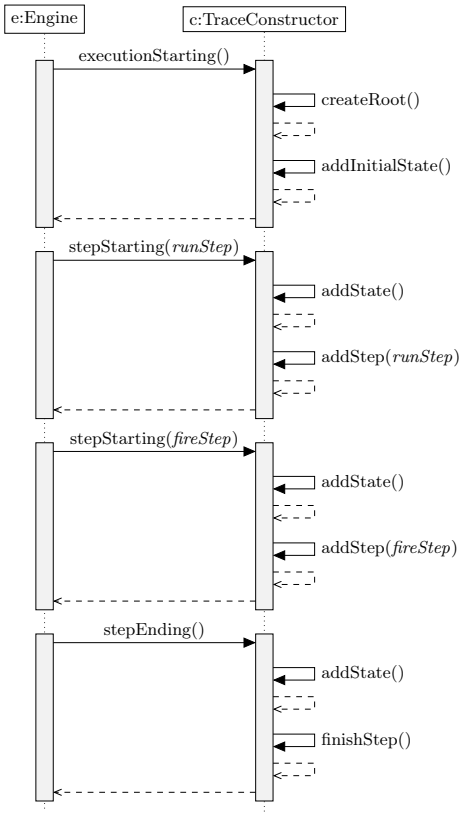


Figure 7: Trace construction during the beginning of the execution of the Petri net from Figure 5, with an engine sending notifications to the constructor, which reacts by calling trace construction services.

On the left, an engine is running the execution transformation, and sends notifications about occurring steps to the trace constructor on the right. First, the trace constructor receive the notification *executionStarting*. This triggers the call to *createRoot* to create the root Trace element of the trace, and *addInitialState* to create the initial ModelState object along with initial TracedObject and Dimension objects for storing the tokens of all places. Then the first step rule *run* starts, and thus the constructor receives *stepStarting*. This triggers a call to *addState*, which does nothing since there was no change in the model state yet. A call to *addStep* is also triggered, which adds a Step object for *run* in the trace. Then, *fire* is called, which again triggers *addState* and *addStep*, adding a new Step for *fire*. After *fire* has modified the model, the notification *stepEnding* triggers the third call to *addState*, which creates the second ModelState, with new Value objects for $p1$, $p2$ and $p3$, each insered in its corresponding Dimension. Lastly, *finishStep* is called, which sets the current ModelState as the ending state of the *fire* Step.

5.4. Model State Manager

This subsection presents the last trace management facility, which is the model state manager. An omniscient

Algorithm 3: *restoreModelToState*

Input:
state : the ModelState to restore in the executed model.

```
[1]  begin
[2]    foreach value ∈ state.values do
[3]      dimension ← value.dimension
[4]      tracedObject ← dimension.tracedObject
[5]      tracedProp ← tracedObject.tracedProperty
[6]      dynObject ← tracedObject.dynamicObject
[7]      if value is IntegerAttributeValue then
[8]        dynObject.set(tracedProp, value.intValue)
[9]      else if value is <other attribute types> then
[10]       ...
[11]     else if value is ReferenceValue then
[12]       dynObject.set(tracedProp, value.referenceValue)
[13]     else if value is ManyReferenceValue then
[14]       ...
```

debugger must be able to revisit a previous model state reached by an executed model. The goal of the model state manager is to read a model state stored in the constructed execution trace, and to transform back the executed model to this corresponding model state.

Algorithm 3 shows a partial description of *restoreModelToState*, which is the single service provided by the model state manager. It takes as an input an instance of the metaclass ModelState. Then, it loops over all Value elements of the input ModelState (line 2), and modifies the model under execution by setting each value into the corresponding dynamic object (*e.g.,* lines 8 or 12).

Efficiency of the model state manager. Thanks to the structure of the trace metamodel, the model state manager can directly access all the required information from one Value element, including the dynamic object that contained the value originally, and the property corresponding to the value. This avoid the need to query the model to find the object in which a value must be restored, and thence makes the *restoreModelToState* service efficient.

6. Generic Omniscient Debugging

This section presents the third part of our approach, which is the definition of a generic omniscient debugger that can be used with any executable DSL considered in our scope. The proposed omniscient debugger relies both on our first contribution for interruption the section when execution steps occur (see Section 4), and on our second contribution to construct and manipulate an execution trace in order to revisit past execution states (see Section 5).

In whats follows, we first define the notion of *execution state* and the set of services a generic omniscient debugger should provide to navigate among execution states. Then, we propose a metamodel to represent the *debugging state* of the omniscient debugger. Lastly, we present how the debugger is integrated in the proposed architecture as a listener of the execution engine, and how it provides all specified debugging services.

6.1. Execution State

As introduced in Section 5.1, at each instant of the execution, an executed model has a *state* containing dynamic objects and values of dynamic fields (*e.g.,* `tokens` values in a Petri net). Unfortunately, the model state does not reflect the entire context of a specific instant of the execution, as it does not contain the state of the execution transformation itself, *i.e.,* which transformation rules are being executed or are about to be executed. For debugging, being aware of current transformation rules is crucial to understand which parts of the semantics are responsible for changes in the model state. We call *execution state* the complete context at an instant in the execution, *i.e.,* the pair composed of the model state and the state of the execution transformation. Since we introduced in Section 4 that only execution steps are observable, we restrict the state of the execution transformation to the statuses of past and ongoing execution steps, and we call this information the current *stepping* of the execution state.

Definition 6. *The* execution state *of a model being executed is composed of:*

- *the* model state*,* i.e., *dynamic objects and values of dynamic fields (see Section 2.1),*
- *the* stepping*,* i.e., *the statuses of current or past execution steps. The status of a step can be:* starting *(the instant before it starts),* in progress *(after it has started and before it has ended),* ending *(the instant after it ends) or* ended *(instants after* ending*).*

Figure 5 shows an execution trace annotated with all possible execution states, when executing a Petri net model conforming to the DSL shown in Figure 1. There are two possible execution states for the first model state (A). First, at the very beginning of the execution, the execution state ⟨1⟩ has a stepping that contains a *run* step with the *starting* status. Second, right after run has started, the execution state ⟨2⟩ has the *run* step with the *in progress* status, and a *fire(t1)* step with the *starting* status. Continuing in the next model state (B), the only execution state ⟨3⟩ has an *ending fire(t1)* step, a *starting fire(t2)* step, and also has an *in progress run* step. Finally, in the last model state (D), each execution state has an *ending* step (*fire(t1)* for ⟨5⟩, *run* for ⟨6⟩), but has no *starting* step.

Note that the stepping can be used to represent the *stack* showed by traditional debuggers simply by displaying only steps that are *in progress*. For instance, if a programming language such as Java defines a "method call" as a step, its debugger would display a stack with all the method calls still in-progress. However, in the case of the simple Petri net DSL shown in Figure 1, such stack can only contain the

initial *run* step, since it is the only step which can contain nested steps.

6.2. Specifying Generic Omniscient Debugging Services

To provide a generic omniscient debugger that can be used with a wide range of executable DSLs, we must first specify the set of services that such debugger must provide. To increase usability, the approach we propose is based on a generalization of most of the well known concepts from programming languages debuggers, *e.g.,* breakpoints and step-wise operations. In the following, we first present standard interactive debugging services (*i.e.,* forward exploration), then we present omniscient debugging services (*i.e.,* backward operations). We rely on Figure 8 to illustrate these services using the same Petri net execution as Figure 5. Note that all these services are only valid when the execution is *paused* at a specific execution state.

Interactive Debugging. Most debuggers only provide standard *interactive debugging*, which includes the following well-known forward exploration services:

- ***breakpoint***: register a condition that will trigger a pause when met (*e.g.,* when a rule is applied on a specific model element).
- ***stepInto***: if a step is *starting*, resume execution and pause either when the step is *ending*, or when some step it contains is *starting*. In other words, pause as soon as possible.
- ***stepOver***: if a step is *starting*, resume execution and pause when this step is *ending* (includes the ending of the nested steps).
- ***stepOut***: if a step is *in progress*, resume execution and pause when the most recent *in progress* step is *ending*.
- ***play***: resume execution.
- ***examine model state***: provide the values of dynamic fields and objects (*e.g.,* for displaying these values in the debugger user interface).
- ***examine steps***: provide the statuses of all steps that occurred so far, *i.e.,* whether a step has ended or not (*e.g.,* for displaying a stack in the user interface).

The upper part of Figure 8 illustrates all possible uses of the standard debugging stepping services (*i.e.,* *stepInto*, *stepOver* and *stepOut*) to navigate forward in the Petri net execution shown in Figure 5. In the first execution state ⟨1⟩, *stepOut* cannot be used because there is no step *in progress*. On the contrary, *stepInto* and *stepOver* can be used because *run* is *starting*. From there, using *stepInto* would bring us "into" the *run* step, hence in ⟨2⟩, and *stepOver* would bring us at the end of the *run* step, hence in ⟨6⟩. If we are in ⟨2⟩, ⟨3⟩ or ⟨4⟩, using either *stepInto* or *stepOver* would always both bring us to the next execution state, because each of these three execution states is followed by a step. If we are in ⟨2⟩, ⟨3⟩, ⟨4⟩ or ⟨5⟩ using *stepOut* would bring us "out" of the *run* step *in progress*, hence in ⟨6⟩ Note that it is only possible to *stepOut* while in ⟨5⟩, because there is no *starting* step there

Omniscient Debugging. To provide exploration of previously visited execution states, *omniscient debugging* relies on the construction of an execution trace to extend standard debugging with the following additional services to go backward in the execution:

- ***jump***: revert to a specific execution state.
- ***backInto***: if a step is *ending*, revert to the previous execution state.
- ***backOver***: if a step is *ending*, execute backward until the execution state where this step was *starting*.
- ***backOut***: if step is *in progress*, execute backward until the execution state where the most recent *in progress* step was *starting*.
- ***examine the trace***: provide the list and content of the model states and steps contained in the trace, (*e.g.,* for displaying the trace and past step stacks).

The lower part of Figure 8 illustrates all possible uses of the omniscient debugging backward stepping services (*i.e.,* *backInto*, *backOver* and *backOut*) to navigate backward in the Petri net execution shown in Figure 5. Overall, the backward stepping services are equivalent to the forward ones, only they consider each execution step to be in the opposite direction, *i.e.,* with inversed starting and ending execution states. For instance, in the last execution state ⟨6⟩, using *backInto* would bring us back "into" the *run* step, hence in ⟨5⟩. Likewise, using *backOver* would bring us at the beginning of *run*, hence in ⟨1⟩.

6.3. Debugging State Metamodel

Debugging services not only need to manipulate the state of the model, but also require information about current execution steps, which we previously defined as the *stepping*, or about the *breakpoints* defined either internally or by the user. For instance, with the stepping, a debugger can know whether there currently is a *starting* step or not, or whether the current execution state is in the past or in the present. We call *debugging state* all the information required by the debugger for providing its services.

To make such information explicit and usable by the debugger, we defined a *debugging state metamodel* shown in Figure 9. First, a DebuggingState contains the current ExecutionState, which itself contains the current Stepping, and a reference to the current ModelState from the execution trace (see Figure 6). Likewise, the Stepping metaclass has several relationships with the Step metaclass. These

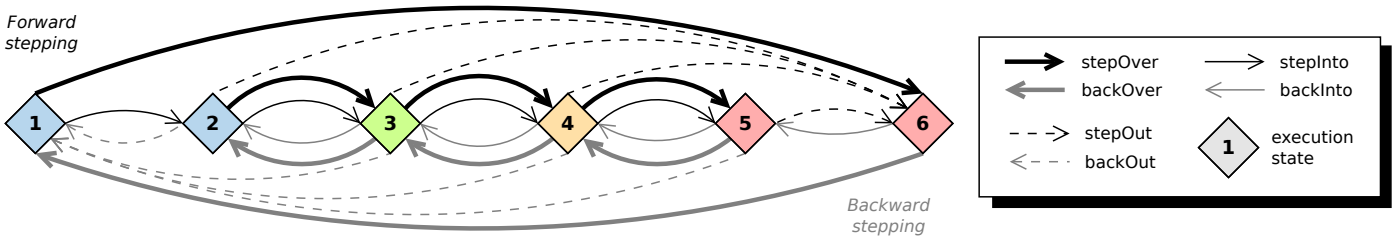


Figure 8: Use of all standard and omniscient debugging stepping services on each execution state of the Petri net model shown in Figure 5.

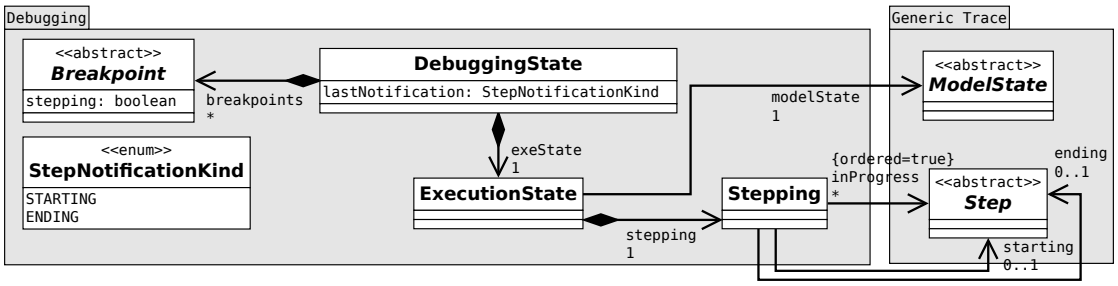


Figure 9: Debugging metamodel, linked to the generic trace metamodel from Figure 6. Note that containment references with a multiplicity of 1 are not structurally important, and are only present to show metaclasses aligned with the definitions of the paper.

relationships make it possible to directly access the execution trace, for instance to know whether a model state or a step is stored in the middle of the execution trace (*i.e.,* in the past) or a the very end of an execution trace (*i.e.,* in the present). In addition to an unbounded number of *in progress* steps, the stepping contains either one *starting* step, or one *ending* step, or one of each. Accordingly, we use three references to define the content of the Stepping: `starting`, `ending`, and `inProgress`. Note that we do not consider the *ended* steps, as they are required neither for observation, nor for the debugging services.

Next, a DebuggingState contains a set of Breakpoint objects, each specifying a condition. When a breakpoint condition is met, the execution should be paused in an execution state. If `stepping` is true, it means that the breakpoint was created by the debugger and is only required internally for stepping (*e.g.,* to pause at the end of a step when using *stepOver*). Stepping breakpoints do not have to be displayed, and must be discarded at each pause.A breakpoint can take many forms, from a simple reference to a syntactical element (*e.g.,* to pause when a specific Transition is fired) to a more complex predicate based on the content of the execution state (*e.g.,* when a specific Place has reached specific amount of tokens). Because of the many possible forms they can take, we specify the Breakpoint metaclass as *abstract*, and consider out the scope of this paper the definition of an appropriate data structure (*e.g.,* a property language) for specifying them.

Finally, a remaining piece of information required by the debugger is the last kind of step notification received from the execution engine. As we explain later in the section, such information is required in order to observe consistent future execution states. This is specified in the attribute `lastNotification`, which can have two values: STARTING if the last received notification was *stepStarting* or ENDING if it was *stepEnding*.

6.4. Design Recipe for Omniscient Debugging

A standard debugger only provide services to observe and control the *forward* execution of a model. In practice, with a standard debugger, a future execution state can always be observed by accomplishing the following tasks: (1) define a breakpoint; (2) resume the execution; (3) when a breakpoint is met, pause the execution engine; (4) access an observable representation of the execution state.

In the case of an omniscient debugger, services are provided to observe and control both the *forward* and the *backward* execution of a model. This has several impacts on the tasks performed by a standard debugger. First, backward execution must be possible, *i.e.,* to revisit all past execution states in the inverse order, during which breakpoints are expected to work as usual. Second, forward execution must still be possible even within past execution states. And third, during both forward or backward exploration of past execution states, the observable execution state must be updated based on the content of the execution trace. In summary, with an omniscient debugger, a past or future execution state can be observed by accomplishing the following tasks in order:

1. Define a breakpoint, either internally by the debugger for complex stepping actions (*e.g., stepOver* or *backOver*), or externally by the developer.
2. Execute forward or backward:
   (a) *Forward*:
      i. If the current execution state is in the middle of the trace (*i.e.,* in the past), execution

states of the trace are explored one by one in order, until the last execution state of the trace is reached (then switch to (ii)).

ii. If the current execution state is at the very end of the trace (*i.e.,* in the present), the execution engine is resumed.

(b) *Backward*: execution states of the execution trace are explored one by one in inverse order, until the beginning of the trace is reached.

3. When a breakpoint is encountered, automatically pause the forward or backward execution.

4. Access a representation of the execution state, *i.e.,* an instance of ExecutionState part of the DebuggingState of the debugger. This ExecutionState must be constantly updated by the debugger based either on the progress of the execution (for future states), or based on the content of the execution trace (for past states).

In the following, we first describe how the omniscient debugger is defined as an execution listener to control the forward execution performed with the execution engine (*i.e.,* item *2(a)ii* of the task list above) with breakpoint management. Then, we describe in detail all the omniscient debugging services and how they behave differently depending whether we are in the past or the present.

6.5. Integration as an Execution Listener

As we explained, the omniscient debugger must be able to control the forward execution performed by the execution engine. For such kind of purposes, we defined in Section 4 a pattern based on the notion of *execution engine* and *execution listener*, where a listener may receive notifications from an engine responsible for running the execution transformation. In particular, these notifications allow a listener to *react* when a step is *starting* or *ending*. The omniscient debugger can therefore be integrated in the proposed architecture as an execution listener connected to the execution engine, and react to notifications in order to pause the execution thread when it is required, *i.e.,* when a future execution state must be reached. In addition, the debugger can safely modify the model state during such pauses. In the following, we first explain the problem of correctly observing future execution states, then we present the definitions of the listening services to integrate the omniscient debugger with the execution engine.

Correct observation of future execution states. One of the tasks of the debugger is to update the observable ExecutionState element based on the progress of the forward execution performed by the execution engine. This update task can be accomplished by listening to the *stepStarting* and *stepEnding* notifications sent by the engine, as they inform of which execution steps are being performed, and they give the opportunity to inspect the current model state. In addition, when a breakpoint is defined to observe a future execution state, testing the breakpoint and pausing can be triggered during the handling of one of these notifications. Yet, when handling these notifications, it is important to pause at the right instants in order to only observe *complete* execution states.

For example, consider the execution trace shown previously in Figure 5 page 9, with six different execution states. The following problem could occur if a breakpoint is enabled (*i.e.,* its condition is true) in the execution state $\langle 3 \rangle$: if the pause is triggered during the *stepEnding* notification corresponding to the *ending* step (*i.e., fire(t1)*), then the observed execution state will not yet reflect the following *starting* step (*i.e., fire(t2)*), as it can only be known by the debugger after the corresponding *stepStarting* notification has been handled. The same situation would occur for $\langle 4 \rangle$. In other words, to always observe a complete ExecutionState, we cannot pause before having registered the *starting* step that may occur in this state.

To solve this problem, a solution is to manage three different cases for a breakpoint enabled in a future target execution state:

1. If there is a *stepStarting* notification during the target state, we pause with this notification, which makes sure no *starting* step remains unobserved. In Figure 5, this allows us to pause in $\langle 1 \rangle$, $\langle 2 \rangle$, $\langle 3 \rangle$ and $\langle 4 \rangle$.

2. If there is no *stepStarting* notification during the target execution state, and if this target is not the last execution state, we pause at the very beginning of the *next* *stepEnding* notification, *i.e.,* if there are two *stepEnding* notifications in a row. This pause must occur before the ExecutionState element gets updated due to this *stepEnding* notification. In Figure 5, this allows us to pause in $\langle 5 \rangle$ while handling the *stepEnding* notification of *run*.

3. If the target execution state is the last one, we pause with the *executionEnding* notification. In Figure 5, this allows us to pause in $\langle 6 \rangle$.

In the following, we define all the services to integrate the debugger with the execution engine, including the listening services that manage these three cases.

Definition of the integration services. To define the integration services, we consider that the debugger has always access to its internal state, which is identified by a variable named d_{state} typed by DebuggingState. We also consider that an operation *pauseIfBreakpoints* is available, which checks whether there is an enabled breakpoint, and which puts the execution thread asleep if this is the case. For more information, this operation is defined in Appendix B.

Algorithm 4 shows the definition of the *stepStarting* listening service. This service is called by the engine just before an execution step, and information on this step is provided in a parameter called s_{start}. We begin by updating the current model state and the current *starting*

Algorithm 4: *stepStarting* (omniscient debugger)

```
   Input:
   s_start: the Step that is starting.
[1] begin
[2]   d_state.exeState.modelState ← s_start.startingState
[3]   d_state.exeState.stepping.starting ← s_start
[4]   pauseIfBreakpoints()
[5]   d_state.exeState.stepping.starting ← null
[6]   d_state.exeState.stepping.ending ← null
[7]   d_state.exeState.stepping.inProgress.push(s_start)
[8]   d_state.lastNotification ← STARTING
```

Algorithm 5: *stepEnding* (omniscient debugger)

```
[1] begin
[2]   if d_state.lastNotification = ENDING then
[3]     pauseIfBreakpoints()
[4]   d_state.exeState.stepping.starting ← null
[5]   s_end ← d_state.exeState.stepping.inProgress.pop()
[6]   d_state.exeState.stepping.ending ← s_end
[7]   d_state.exeState.modelState ← s_end.endingState
[8]   d_state.lastNotification ← ENDING
```

step of the observable ExecutionState element (lines 2–3). Next, we check the breakpoints and pause if one is enabled (line 4). Note that we always make an attempt at pausing, since at this point the execution state is fully updated and hence complete. Continuing, we fully update the execution state: both the *starting* and the *ending* steps are discarded and s_{start} is pushed atop the stack of `inProgress` steps (lines 5–7). Finally, we update the `lastNotification` value to STARTING (line 8).

Algorithm 5 shows the definition of the *stepEnding* listening service. It must be called by the engine just after an execution step. First, if the last notification was *stepEnding* (line 2), we try to pause the execution (line 3). At this point, if there is a pause, the observed ExecutionState is aligned with the former encountered execution state (*e.g.,* in Figure 5, we observe $\langle 5 \rangle$ although we are at the *stepEnding* of *run*). Then, we fully update the observable ExecutionState element: the *starting* step is discarded (line 4), the upmost *in progress* step is popped in a variable named s_{end} (line 5), and both the *ending* step and the model state are set according to s_{end} (line 6–7). Finally, we update the `lastNotification` value to ENDING (line 8).

Lastly, we do not show algorithms for the last two services, since they can be defined in a single statement each. First the *executionEnding* listening service must be defined with a single call to *pauseIfBreakpoints* (*e.g.,* in Figure 5, to be able to pause in $\langle 6 \rangle$) Second, the *dependsOn* service must be defined to reflect the dependency between the omniscient debugger and the execution trace constructor. Indeed, the debugger constantly requires an up-to-date execution trace, and therefore it must always receive notifications after the trace constructor had finished its construction tasks.

6.6. Definition of the Omniscient Debugging Services

After having covered most of the architecture, we can now define the debugging services provided by the omniscient debugger. As before, we consider that the debugger has always access to its internal state, which is identified by a variable named d_{state} typed by DebuggingState. In addition, the model state manager is available through the variable *manager*, and the Trace root element of the execution trace is available through the variable *trace*. Below, we first present some internal services required by the debugger itself, then we cover jumping services for exploring past execution states, and finally we present the standard forward and backward services. Note that all these services can only be used when the execution is already paused.

Internal services of the omniscient debugger. We assume that the following internal services are available to the debugger: *restoreInProgressSteps* reconstructs the list of *in progress* steps of the current ExecutionState based on a given execution step, *getLastStep* retrieves the last execution step of the the execution trace, *inLastExeState* checks whether the last execution state of the trace was reached, *isInitialExeState* checks whether the initial execution state of the trace was reached, and *getCurrentValue* retrieves the current value of one the the dimensions of the executed model, *i.e.,* the value of the corresponding dynamic field. These services will be used for defining the debugging services thereafter. For more details, refer to Appendix C.

Trace exploration services. Table 2 shows the trace exploration services of the omniscient debugger, *i.e.,* the services available to restore the execution state to a specific point stored in the execution trace. They can be used either by the developer to explore specific points in the execution trace, or internally by the debugger to provide more high-level services. In fact, all the modifications made to the ExecutionState by the omniscient debugger are achieved using these trace exploration services.

Both *jumpToStartingStep* and *jumpToEndingStep* perform a similar task, which is to revert the current execution state to the point when a given step was either starting or ending, respectively. First, the model state manager is used to revert the executed model into the corresponding model state, and the ExecutionState element is updated to point to the corresponding ModelState element in the execution trace (lines 1–2). Then, the ExecutionState is updated regarding the *in progress* steps (line 3), and regarding both the *starting* and *ending* steps. If the target step is contained in a Step, we can find the preceding or following step using this container (lines 5–6). Else, it means the target step is the root step of the trace, which has no preceding nor following step (lines 7–8).

Lastly *jumpToModelState* is simply a wrapper to jump to the first execution state of a given model state, *i.e.,* when its first step is starting.

Trace Exploration Service	Definition
jumpToStartingStep(s_{start} : Step)	[1] *manager*.restoreModelToState(s_{start}.`startingState`) [2] d_{state}.`exeState.modelState` ← s_{start}.`startingState` [3] restoreInProgressSteps(s_{start}) [4] d_{state}.`exeState.stepping.starting` ← s_{start} [5] **if** s_{start}.`container` is Step **then** [6] d_{state}.`exeState.stepping.ending` ← s_{start}.`container.nestedSteps`.getBefore(s_{start}) [7] **else** [8] d_{state}.`exeState.stepping.ending` ← *null*
jumpToEndingStep(s_{end} : Step)	[1] *manager*.restoreModelToState(s_{end}.`endingState`) [2] d_{state}.`exeState.modelState` ← s_{end}.`endingState` [3] restoreInProgressSteps(s_{end}) [4] d_{state}.`exeState.stepping.ending` ← s_{end} [5] **if** s_{end}.`container` is Step **then** [6] d_{state}.`exeState.stepping.starting` ← s_{end}.`container.nestedSteps`.getAfter(s_{end}) [7] **else** [8] d_{state}.`exeState.stepping.starting` ← *null*
jumpToModelState(m : ModelState)	[1] jumpToStartingStep(m.`startedSteps`.get(0))

Table 2: Trace exploration services of the omniscient debugger

Forward debugging services. Table 3 shows the forward debugging services of the omniscient debugger. Most of the complexity relies in the definition of the *play* service, which can be used to start or resume the execution of the model. The first and main part handles forward execution within past execution states, by *replaying* the execution states in the execution trace. In a nutshell, a loop continuously jumps to the next available execution state (lines 2–11), and checks after each jump if a breakpoint is enabled (line 12). If a breakpoint is enabled, the stepping breakpoints are discarded and the operation is over (lines 13–15). Else, if we reached the end of the trace, we resume the execution engine using the *wakeUp* service (lines 16–17), which will start the exploration of new execution states and pass the relay to the listening services for managing breakpoints when notifications are received from the engine (see Section 6.5).

From this point, the stepping operators each consist in creating a specific breakpoint and in calling the *play* service. We use the notation [ ] for specifying stepping breakpoints in the form of closures. Each service starts with a condition: both *stepInto* and *stepOver* require a *starting* step, while *stepOut* require at least one *in progress* step. Then *stepInto* specifies a breakpoint to trigger a pause as soon as possible (*i.e.,* as soon as another step is encountered), *stepOut* when the *starting* step becomes *ending*, and *stepOut* when the last *in progress* step becomes *ending*.

Backward debugging services. Lastly, the backward debugging services of the omniscient debugger can be defined. These services are very similar to the forward ones, except they work in the opposite direction of the execution. For more information, refer to Appendix D.

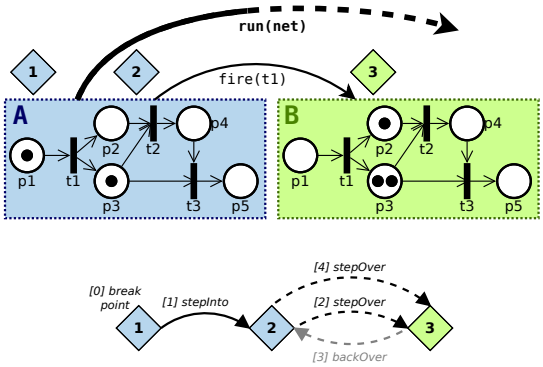


Figure 10: Omniscient debugging scenario based on the beginning of the execution shown in Figure 5.

6.7. Example of Omniscient Debugging Scenario

To illustrate an excerpt of the interactions between the omniscient debugger and other components of the architecture we propose, we consider in Figure 10 a debugging scenario based on the beginning of the execution previously shown in Figure 5 page 9. The relevant part of the execution is recalled at the top, and the considered sequence of debugging operations is shown at the bottom. Before starting the execution, a *breakpoint* is created to pause when the *run* step is *starting*. Once the breakpoint is reached in the first execution state, the engine is paused. Then *stepInto* and *stepOver* are used to reach the last execution state, followed by *backOver* to go back to the previous one, and finally by *stepOver* to go forward again.

Forward Debugging Service	Definition
play()	[1] $B_{enabled} = \varnothing$ [2] **while** ¬inLastExeState() $\wedge$ $B_{enabled} = \varnothing$ **do** [3] **if** d_{state}.exeState.stepping.starting $\neq$ *null* **then** [4] $s_{start} \leftarrow d_{state}$.exeState.stepping.starting [5] **if** s_{start}.nestedSteps $\neq \varnothing$ **then** [6] jumpToStarting(s_{start}.nestedSteps.first()) [7] **else** [8] jumpToEnding(s_{start}) [9] **else** [10] $s_{end} \leftarrow d_{state}$.exeState.stepping.ending [11] jumpToEnding(s_{start}.container) [12] $B_{enabled} = \{b \in d_{state}.\texttt{breakpoints} \mid \text{enabled}(b)\}$ [13] **if** $B_{enabled} \neq \varnothing$ **then** [14] $B_{enabledStepping} = \{b \in B_{enabled} \mid b.\texttt{stepping}\}$ [15] d_{state}.breakpoints.remove($B_{enabledStepping}$) [16] **else if** inLastExeState() **then** [17] wakeUp()
stepInto()	[1] **if** d_{state}.exeState.stepping.starting $\neq$ *null* **then** [2] d_{state}.breakpoints.add([*true*]) [3] play()
stepOver()	[1] **if** d_{state}.exeState.stepping.starting $\neq$ *null* **then** [2] $s_{over} \leftarrow d_{state}$.exeState.stepping.starting d_{state}.breakpoints.add([d_{state}.exeState.stepping.ending $= s_{over}$]) [3] play()
stepOut()	[1] **if** d_{state}.exeState.stepping.inProgress $\neq \varnothing$ **then** [2] $s_{out} \leftarrow d_{state}$.exeState.stepping.inProgress.peek() d_{state}.breakpoints.add([d_{state}.exeState.stepping.ending $= s_{out}$]) [3] play()

Table 3: Forward services of the omniscient debugger

Figure 11 depicts an excerpt of the interactions encountered in this scenario between the omniscient debugger, the execution engine, the model state manager, and the external control panel. We do not show the execution transformation, and only consider the notifications sent by the engine to the omniscient debugger about the progress of the execution. In addition, note that the trace constructor also receives each notification sent by the engine, only just before the debugger, which means that the execution trace is always up-to-date when the debugger is acting.

First, the engine uses *stepStarting* to notify the omniscient debugger that the *run* step is starting. Consequently, the debugger updates its internal state, then triggers a pause due to the breakpoint to stop when the *run* step is *starting*. This pause means that the execution thread (in light gray) is put asleep and will do nothing until it is awaken by debugging services. Then, a *stepInto* is triggered, which adds the corresponding breakpoint and starts *play*, which wakes up the execution engine. The execution thread then continues, allowing the debugger to update its internal debugging state again, and to finally end the handling of *stepStarting*, which means that the execution transformation will resume its normal progress.

Second, the engine uses *stepStarting* to notify the omniscient debugger that the *fire* step is starting for the $t1$ transition. This time the debugger triggers a pause because *stepInto* is finished, therefore it must wait for a new debugging service call. A *stepOver* is triggered, which again adds the corresponding breakpoint and wakes up the engine.

Third, the engine uses *stepEnding* to notify the omniscient debugger that the last step is now ending. Here, the debugger simply checks whether or not this is the second *stepEnding* in a row in order to trigger a pause accordingly (see Section 6.5), and then updates its internal state.

Lastly, the engine uses *stepStarting* to notify that the *fire* step is starting for the $t2$ transition. The debugger triggers a pause because *stepOver* is now finished. Then a *backOver* is triggered to undo this *stepOver*. A breakpoint is therefore added, and this time *playBackwards* is used to start a backward execution. This causes the debugger to explore the execution trace, and eventually to call *restoreModelToState* to restore the model state A in the

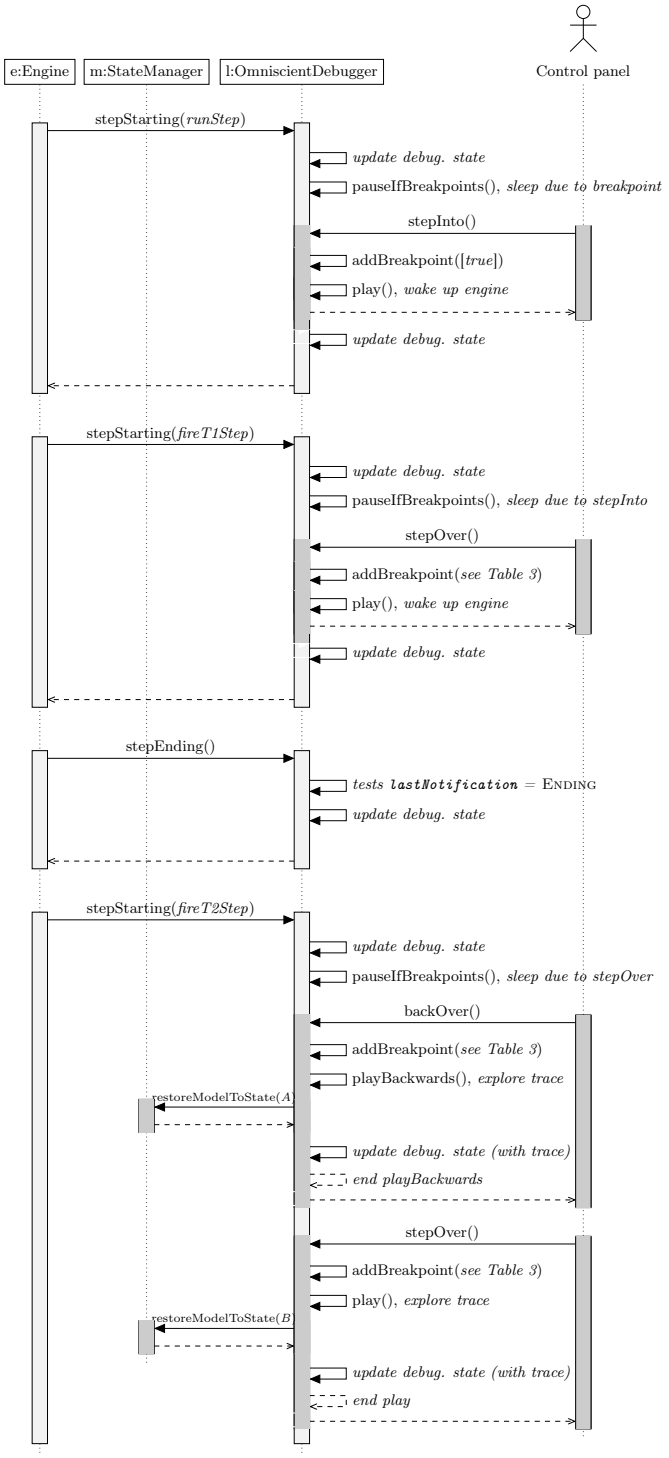


Figure 11: Excerpt of the interactions during of the omniscient debugging scenario shown in Figure 10. Parts in light gray are performed by the execution thread, and parts in dark gray are performed by the control panel thread.

executed model, and to update the debugging state accordingly. Then, *playBackwards* ends as soon as the breakpoint is met, which ends *backOver*.

At this point, the engine is still paused. A *stepOver* is used in order to reach again the third execution state. This time, when *play* is used for forward execution, it does not resume the engine because we are in the past. Instead, the execution trace is explored, using *restoreModelToState* to restore the model state B in the executed model, and the debugging state is updated accordingly.

7. Implementation

This section first presents the language and modeling workbench called GEMOC Studio, and then explains how we implemented the presented architecture to offer three tools: (1) an execution framework for integrating generic runtime services with different execution engines, based on the pattern we presented in Section 4; (2) generic trace management facilities, based on what we presented in Section 5; (3) a generic omniscient debugger, based on what we presented in Section 6. All the source code is available on Github[2].

7.1. Overview of the GEMOC Studio

The GEMOC Studio[3] is an open source (EPL 1.0) Eclipse package atop the Eclipse Modeling Framework (EMF), which includes:

- The *GEMOC language workbench*: used by language designers to build and compose new executable DSLs,
- The *GEMOC modeling workbench*: used by domain designers to create, execute and coordinate models conforming to executable DSLs.

The different concerns of a DSL, as defined with the tools of the language workbench, are automatically deployed into the modeling workbench.

In the GEMOC language workbench, for a given executable DSL, the operational semantics are defined using a specific metaprogramming approach. Different approaches are supported, including Java-based languages (*Kermeta* [27], Xtend, pure Java), xMOF [10], or a combination of a Java-based language with the MoCCML language [32]. Since the GEMOC Studio is based on EMF, Ecore is used to define the abstract syntax, and at runtime the model is a set of EMF objects. Either the Sirius toolkit can be used for defining a graphical concrete syntax, or Xtext can be used to define a textual concrete syntax.

In the GEMOC modeling workbench, the user can define an executable model conforming to an executable DSL,

[2] `https://github.com/eclipse/gemoc-studio-modeldebugging` for the execution framework and addons, `https://github.com/moliz/moliz.gemoc` for xMOF specific parts.

[3] `http://gemoc.org/studio`

and execute it using an execution engine and a selection of addons, such as graphical animation, debugging tools, trace and event managers, timeline visualizations, etc. In the following section, we present the execution framework that we implemented, and that can itself be used to implement such engines and addons

7.2. Execution Framework of the GEMOC Studio

We applied the pattern that we presented in Section 4, defining that an *execution engine* sends notifications to *execution listeners*, to implement a complete Java *execution framework* for the GEMOC Modeling Workbench. This framework was previously presented in detail in [19], and we summarize thereafter its main aspects and features.

Organization of the framework. The API of the GEMOC execution framework is organized around two core interfaces: `IExecutionEngine`, corresponding to the *engine* role of the pattern, and `IEngineAddon` corresponding to the *listener* role of the pattern.

First, an *engine* can be defined to integrate a metaprogramming approach in the modeling workbench, and must implement the `IEngineAddon` interface. When an engine is about to apply a step rule, it must create a step object containing information about the executed rule (identifiers of the rule, parameters given, etc.). For this purpose, the framework relies on a common interface called `Step`, which is defined in the generic trace metamodel of the framework. Then this step object must be used to notify addons of both the start and the end of a step.

Then, *addons* can be defined to provide runtime services, and must implement the `IEngineAddon` interface, which contains operations equivalent to what we presented in Section 4. Within the implementation of one of these operations, an addon can access the engine and its status, the executed model, or even the graphical interface of the studio. Therewith, an addon can accomplish a large diversity of tasks, such as changing the executed model, pausing or stoping the execution, displaying information, or sending some input data to the engine.

Available engines. Using the framework, we implemented four different execution engines. The Java engine is dedicated to operational semantics that are entirely defined using any Java-based language, such as Java, Xtend [33] or Kermeta [27]. The Java+MoCCML engine is dedicated to operational semantics where the (possibly concurrent and timed) control is described in the MoCCML formal language [32], while the execution rules are written in any Java based language (e.g., Kermeta) [34]. The xMOF engine supports the execution of operational semantics defined with xMOF [10]. The coordination engine supports the behavioral coordination of heterogeneous models, based on coordination patterns defined using BCOoL [35]. We plan to implement more execution engines to integrate more metaprogramming approaches in the future.

Available addons. Likewise, using the framework, we implemented several execution addons. Among many others, the graphical animator is an addon that updates different views in order to display the current execution state of the executed model, hence helping to understand models under execution. Animation is available for both graphical and textual concrete syntaxes (*e.g.,* highlighting the relevant line in the textual editor).

Regarding our approach, we implemented the set of generic trace management facilities that we presented in Section 4. The trace constructor was implemented as an addon, and the trace metamodel was defined using the Ecore metamodeling language. Lastly and most importantly, the omniscient debugger was also implemented as an addon, and is presented below.

7.3. Omniscient Debugging in the GEMOC Studio

We implemented the generic omniscient debugger that we presented in Section 4 as an addon for the GEMOC Modeling Workbench. The internal logic closely follows what we describe in Sections 6.5 and 6.6. To pause the execution, the debugger relies on the Java thread API to suspend the current thread. When debugging services are triggered to perform forward exploration without exploring the trace, the execution engine thread is awakened and the execution continues. When exploring the trace, the debugger modifies the state of the unique instance of the model loaded in memory for execution (*i.e.,* an EMF `Resource`). A screenshot of the user interface of the omniscient debugger is provided in Appendix E.

8. Evaluation

In this section, we present the design and results of an empirical evaluation of our approach. We first evaluate the genericity of our approach with respect to different executable DSLs and different metaprogramming approaches. We then evaluate both memory and time efficiency as compared to other implementation variants.

8.1. Genericity

To evaluate the genericity or our approach, we considered the following research questions

RQ#1: Can our omniscient debugger be used with *different executable DSLs*?

RQ#2: Can our omniscient debugger be used with executable DSLs implemented using *different metaprogramming approaches*?

To answer these questions, we tested our omniscient debugger on a set of different executable DSLs. Table 4 shows the 10 executable DSLs we have used, defined using three metaprogramming approaches: the xMOF [10] language, the Kermeta language [27], or a combination

Executable DSL	Link	Description	Metaprogramming Approach
Petri nets	*link*	Simple Petri nets (see Figure 1)	xMOF
fUML	*link*	Complete fUML	xMOF
fUML (subset)	*link*	Subset of fUML (activity diagrams)	Kermeta
ArduinoML	*link*	Simple Arduino hardware and software descriptions	Kermeta
IML	*link*	AutomationML Intermediate Modeling Layer	Kermeta
MiniTL	*link*	Mini declarative model transformation language	Kermeta
FSM	*link*	Finite state machines	Kermeta
fUML	*link*	fUML subset (activity diagrams)	Kermeta + MoCCML
SigPML	*link*	Signal Processing Modeling Language.	Kermeta + MoCCML
TFSM	*link*	Timed finite state machines	Kermeta + MoCCML

Table 4: Executable DSLs on which our implementation was tested.

of the Kermeta and MoCCML [32] languages. As we explained in Section 7, we implemented one execution engine per metaprogramming approach, resulting in three engines: one for xMOF-based semantics, one for Java-based (including Kermeta) semantics, and one for Java+MoCCML (including Kermeta+MoCCML) semantics. Note that our approach is not compatible with the *coordination engine* of the GEMOC Studio, as this last engine execute multiple models simultaneously, which breaks our hypothesis of having only a single model during execution.

For each executable DSL, we managed to start the execution of an example model with the omniscient debugger, and to use all the debugging facilities offered by the implemented omniscient debugger. Therefore, to answer RQ #1 and RQ #2, we observe that our approach is generic enough to support different executable DSLs, and different metaprogramming approaches to define these DSLs.

8.2. Efficiency

To evaluate the efficiency of our approach, we considered the following research questions:

RQ#3: What is the efficiency in *memory* of the generic trace management facilities of the approach? More precisely, how do they compare to clone-based trace management facilities?

RQ#4: What is the efficiency in *time* when exploring *past* execution states with our omniscient debugger based on generic trace management facilities? In other words, what is the time required for exploring the trace? More precisely, how does it compare:

- to an omniscient debugger without trace management facilities? (*i.e.,* which requires a restart of the execution for revisiting a past state)
- to an omniscient debugger based on clone-based trace management facilities?

RQ#5: What is the efficiency in *time* when exploring *future* execution states with our omniscient debugger based on generic trace management facilities? In other words, what is the execution time overhead? More precisely, how does it compare:

- to an execution without debugger nor trace management facilities?
- to an omniscient debugger based on clone-based trace management facilities?

Thus, our evaluation of efficiency is the comparison of three omniscient debuggers that we implemented, each with different trace management facilities. First, *NoTraceDebugger* is an omniscient debugger that does not construct any execution trace, and that restarts the execution each time it has to jump backward in time. Such a debugger is expected to be efficient in memory since there is no trace to store; efficient when exploring future states, since it does not construct any trace; and inefficient when exploring past states, because the execution engine must be restarted. Second, *CloneBasedDebugger* is a generic omniscient debugger that constructs a clone-based trace using *deep cloning* (*i.e.,* the complete model is copied at each step) and whose model state manager relies on the model differencing library EMF Compare[4]. Because this debugger relies on an execution trace, it is expected to be less efficient than *NoTraceDebugger* both in memory and when exploring future states, but more efficient when exploring past states thanks to the constructed trace. Third, *GenericDebugger* is an omniscient debugger relying on generic trace management facilities. More precisely, it uses a trace metamodel similar to the generic trace metamodel shown in Section 5.2, and for which we implemented both a generic trace constructor and a generic model state manager. All three debuggers were implemented in the GEMOC Studio.

Considered executable DSL. For this evaluation, we considered a subset of a real-world executable DSL, namely fUML [1]. The considered subset contains the *Activity Diagram* portion of the language. In summary, a model conforming to this executable DSL is made of an *activity*, which consists of *control nodes* and *action nodes*. Nodes are linked by *control flow links*, starting with an *initial node* and ending with a *final node*. Similarly to a Petri Net, *tokens* are passed along nodes to drive the execution. In

[4] `https://www.eclipse.org/emf/compare/`

addition, *variables* can be defined in activities and modified with actions. The executable DSL was implemented with GEMOC Studio using Ecore for the abstract syntax and Kermeta for the operational semantics.

Considered models. We considered two sets of fUML models. First, as in our previous work [18], we used models taken from the case study of Maoz et al. [36] [5]. This choice was made to help establish a benchmark, facilitate comparison with future work, and because the models were drawn from industrial sources. The dataset we obtain contains 42 models whose sizes range from 34 to 61 objects.

Second, to experiment with larger models, we implemented a random fUML activity diagram generator, which works in the following way. It first initializes a small valid activity diagram with four to six nodes. Then, one iteration consists in randomly selecting a sequence of two nodes, and replacing them with one fork node and one join node separated by two to four branches, each branch containing two to four nodes. The generator repeats this action until 250 to 500 nodes have been created. In the end, we generated 20 models whose sizes range from 500 to 1000 objects.

Data collection and analysis. To compare efficiency in *memory* of trace management facilities, instead of observing the memory usage of the complete environment (including the footprint of the execution engine and of the loaded model), we measured the memory used only for trace management. To do so, we executed each considered model, and constructed each time a corresponding execution trace in memory. At the end of each model execution, we performed a memory dump of the complete Java heap. To filter the noise due to the environment, we used the Object Query Language (OQL)[6] to define queries that only select elements that are part of the trace (*e.g.,* "select all instances of the metaclasses `my.trace.metamodel.*`"). We then used Eclipse MAT[7] to execute the OQL queries on each heap dump, and collected the precise amount of memory required to store each execution trace.

Unfortunately, Eclipse MAT was not able to analyze the heap dumps obtained with our larger models due to the too high amount of elements. Therefore, for the set of larger models, we serialized each of the execution traces as a file on the hard disk, and measured the size of each file. This serialization relies on the default *save* feature of the Eclipse Modeling Framework, which uses the standard XMI format (*i.e.,* an XML format) for saving models. Although these are not measurements of memory consumption, we believe that they can be used as useful approximations.

To compare efficiency in *time* when exploring *past* execution states, we measured the time required to execute

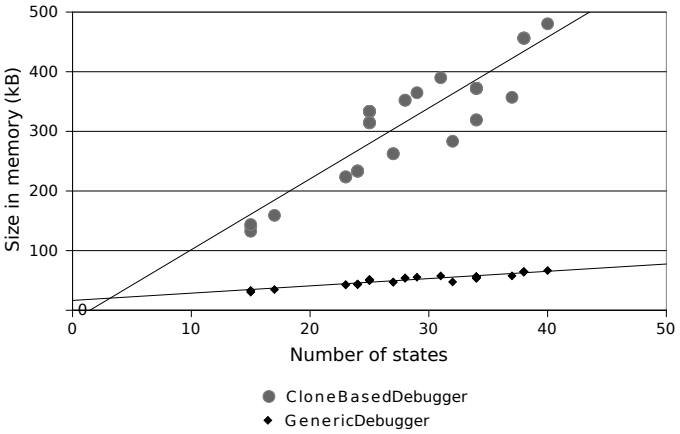


Figure 12: Memory used by the execution traces, with the set of 42 small models from industrial sources (34 to 61 objects per model)

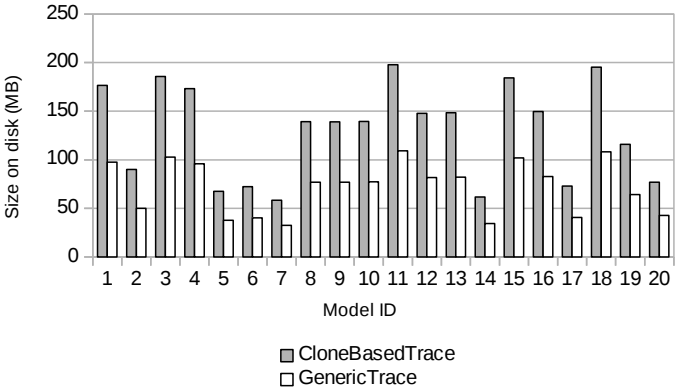


Figure 13: Size on disk of execution traces, with the set of 20 randomly generated large models (500 to 1000 objects per model)

the service *jumpToStartingStep*, which is always used by the debugger when exploring the past. More precisely, for each of the considered models, we measured the median amount of time required to perform a *jumpToStartingStep* jumping once to each previously visited execution state.

To compare efficiency in *time* when exploring *future* execution states we measured the total amount of time required to fully execute the model with a debugger attached, *i.e.,* with notifications sent from the engine to the debugger and with the contruction of an execution trace.

All time measurements were done using Java's operation `System.nanoTime`. Because the Java virtual machine tends to become faster over time when executing the same operation a large number of times, 10 warmups of the virtual machine were performed before each measurement, each warmup being a complete execution of the model and one *jumpToStartingStep* per execution state. Each result was computed from the average of 20 identical measurements made using an Intel i7-6600U CPU.All data was collected in a reproducible way through a programmatic use of GEMOC Studio's engine.

RQ #3: Efficiency in memory. Figure 12 shows the results obtained regarding the memory required to store an

[5] Available at `http://www.se-rwth.de/materials/semdiff/`

[6] OQL is a SQL-like language to query object-oriented data, see `https://en.wikipedia.org/wiki/Object_Query_Language`

[7] The Eclipse Memory Analyzer (Eclipse MAT) is a Java heap analyzer, see `https://www.eclipse.org/mat/`

execution trace for each considered omniscient debugger, when considering the set of smaller models. The x-axis shows the number of model states in the trace, while the y-axis shows the amount of memory used in kB. First, *NoTraceDebugger* does not use memory, because it does not store a trace. Therefore it does not appear on the figure. Then, we observe that *GenericDebugger* is always more efficient in terms of memory usage than the *CloneBasedDebugger* debugger, with 6.05 times improvement on average. This can be explained by the fact that traces obtained with our approach are designed to only contain the evolution of the dynamic fields of the model with minimal redundancy, whereas cloning implies significant redundancy.

Figure 13 shows the results obtained regarding the file size of each execution trace for each considered omniscient debugger, when considering the set of larger models. The x-axis shows the model ID (among the large models), while the y-axis shows the amount of hard disk space used in MB. Similarly to smaller models, we observe that *GenericDebugger* is always more efficient in terms of disk usage than the *CloneBasedDebugger* debugger, with 1.81 times improvement on average. The smaller improvement factor can be explained by the way traces are serialized on disk: our approach requires a large amount of links between objects, which are costly in text size when using the XMI format, while links use very little space in memory. Nonetheless, while the improvement factor is not representative of memory gain, we argue that it shows that our approach requires a lesser amount of memory with the set of large models.

To answer RQ #3, we observe that our approach is more efficient in memory than a clone-based approach.

RQ #4: Efficiency in time for exploring past states. Figures 14 and 15 present the time required to revisit one past execution state—*i.e.,* to perform a *jumpToStartingStep*—for each of the considered debuggers. The x-axis shows the identifier of the executed model, while the y-axis shows the amount of time in *ms* using a logarithmic scale. Figure 14 shows the results for the set of small models from industrial sources, while Figure 15 shows the results for the set of randomly generated large models.

First, we observe that trace-based debuggers are always better than *NoTraceDebugger*, with in particular *GenericDebugger* being 92.7 times faster than *NoTraceDebugger* with small models, and 335 with large models. This is explained by the time required to reset the execution engine, when no execution trace is used to reach a past execution state. Second, we observe that *GenericDebugger* is more efficient than *CloneBasedDebugger* with 3.5 times improvement on average with small models, and 2.4 with the set of large models. We believe this is due to the significant complexity of the model comparison operation required to implement *restoreModelToState* for *CloneBasedDebugger*.

To summarize and answer RQ #4, we observe that our approach is more efficient in time when exploring past execution states than two alternative omniscient debuggers: one without traces, one using clone-based traces.

RQ #5: Efficiency in time for exploring future states. Figures 16 and 17 present the results obtained regarding the median amount of time required explore all future execution states—*i.e.,* to fully execute a model with a debugger attached— with all three considered debuggers. The x-axis shows the identifier of the executed model, while the y-axis shows the amount of time in *ms* using a logarithmic scale.

First, we observe that trace-based debuggers are always slower than *NoTraceDebugger*, with in particular *GenericDebugger* being on average 1.59 times slower with small models, and 1.64 slower with large models. This is explained by the time required to construct a trace when exploring future execution states, while *NoTraceDebugger* does not construct any execution trace. Second, we observe that *GenericDebugger* is always faster than *CloneBasedDebugger*: it is on average 1.42 times faster with small models, and 1.81 times faster with large models. This is explained by the time required to make a complete copy of the model under execution in the case of *CloneBasedDebugger*, while our approach incrementally constructs a trace containing only the dynamic elements, and by only looking at the changes that occur in the model state.

To summarize and answer RQ #5, we observe that our approach is slower when exploring future execution states than an approach without traces, but faster than an approach using clone-based traces.

8.3. Threats to Validity

First, the considered set of models does not contain any large model of industrial or real-world source. This is compensated by having two sets of models that cover both characteristics: one with rather small models from industrial sources, and one with synthesized but large models.

Second, the approach is only compared with other omniscient debuggers variants that we developed internally for the GEMOC Studio. As there is no other approach providing generic omniscient debugging for executable DSLs, we could not find any external solution to use as a suitable baseline. In addition, our implementation is based on the GEMOC Studio, which may sometimes be a cause of overhead when initialing or execution models, and therefore making it difficult to make a fair comparison with implementations not based on GEMOC.

9. Related Work

In this section, we overview existing work on standard debugging, omniscient debugging, execution trace management, and on the integration of runtime services with operational semantics.

9.1. Standard Debugging for Executable DSLs

We present thereafter some existing standard debugging approaches (*i.e.,* not omniscient) for executable DSLs: domain-specific debuggers, generic debuggers, and approaches to define domain-specific debuggers.

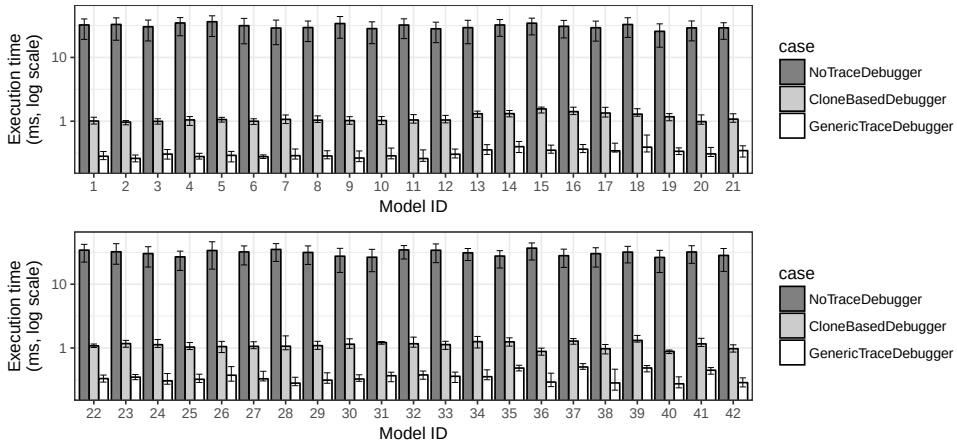


Figure 14: Time required to explore one past state with the set of 42 small models from industrial sources (34 to 61 objects per model).

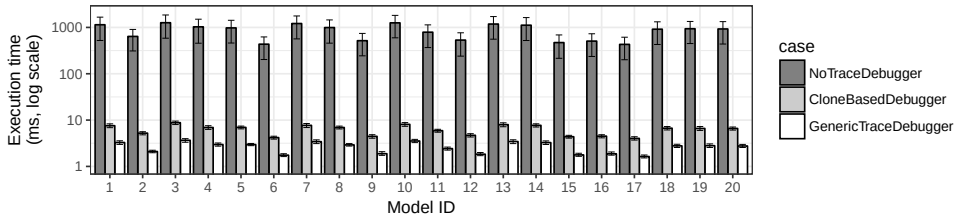


Figure 15: Time required to explore one past state with the set of 20 randomly generated large models (500 to 1000 objects per model).

Domain-Specific Standard Debugging. Many approaches provide standard debuggers that are domain-specific, *i.e.,* specific to a single executable DSL.

Van Mierlo et al. [37] defined a debugger for the *Parallel DEVS* executable DSL, which is an extension to *DEVS*, a formalism for modeling complex dynamic systems. They developed a specific interpreter using Statechart models, in which they defined debugging-specific operations such as pausing, breakpoints, and state manipulation.

Mayerhofer et al. [38] extended the standard fUML operational semantics in order to support debugging of fUML models. This includes the definition of a control API to pause or execute single steps, and an observer pattern to follow model changes.

Lastly, many approaches provide debugging for several parts of the UML [39, 40, 41, 42, 43], each considered as an independent executable DSL.

None of these debuggers are omniscient, and each is only specific to a single executable DSL, while the approach we propose is omniscient and valid for any executable DSL.

Generic Standard Debugging for Executable DSLs. Previously, we proposed a model simulator in the TopCased toolkit [44]. This simulator can execute models, and provides a GUI for interactive simulation that can be considered as debugging. While the approach is generic, the presented prototype is specific to an ad-hoc simulation engine for UML state machines.

Ráth et al. [45] propose an approach based on the Viatra language to execute and debug models conforming to executable DSLs. The execution can be paused in between steps, and the model can be edited on-the-fly during pauses (similarly to "hot code replace" proposed by some GPLs debuggers). Visualization of the execution state is provided by the graphical editor used to edit models.

Bandener et al. [12] propose a tool called the *Dynamic Meta Modeling (DMM) Player.*A debugger is provided on top of the execution engine responsible for executing the model transformation. Similarly to the approach we propose, a subset of the rules are chosen as *visual steps* that update the concrete syntax representation of the models. When debugging, the execution can be paused before or after the application of any rule.

While these debugging approaches can be used for any executable DSL, some with advanced features, none of them provide any form of omniscient debugging.

Domain-Specific Debugger Definition Approaches. Several approaches have been proposed regarding how to define a domain-specific debugger for an executable DSL.

Wu et al. [46] propose a generative approach for grammar-based executable DSLs with translational semantics where a debugger is already available for the target language (*e.g.,* a GPL such as Java). Traceability links are required between the executed model and the target model. Debugging components are generated to implement the debugging interface of the Eclipse IDE. Using these traceability links, debugging services at the DSL level (*e.g.,* step forward) are translated into orders for the target language debugger (*e.g.,* set a breakpoint and continue).

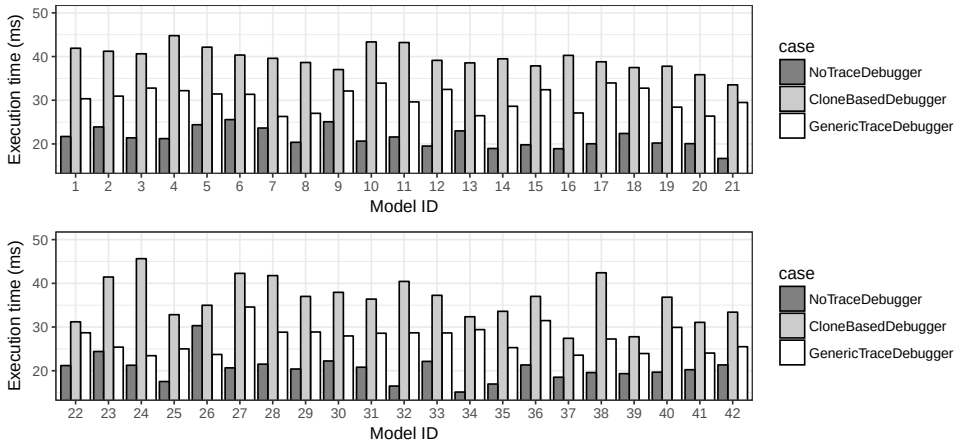


Figure 16: Time required to explore all future states with the set of 42 small models from industrial sources (34 to 61 objects per model).

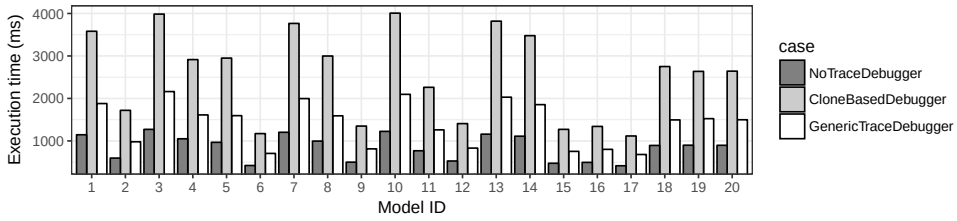


Figure 17: Time required to explore all future states with the set of 20 randomly generated large models (500 to 1000 objects per model).

Lindeman et al. [47] present a generative approach for grammar-based executable DSLs targeting both translational and operational semantics. A language called the *debugger specification language* is used to specify when debugging events occur during the execution of the model. Such specification is used to automatically instrument the executable model to send such events to an external component. This requires the executable DSL to be expressive enough to make such event management possible.

Chiş et al. [48, 49] proposed the Moldable Debugger framework for developing domain-specific debuggers. They provide a framework to develop *domain-specific extensions*, each being composed of a set of *domain-specific debugging operations* and a *domain-specific debugging* view.

While these approaches can be applied to any executable DSL, and sometimes provide domain-specific debugging operations, none provide omniscient debugging services.

9.2. Omniscient Debugging for Executable DSLs

In the last decades, significant work has been done to provide omniscient debuggers for GPLs, such as for C/C++ [50], Java [51, 52, 30], or Smalltalk [16]. A recent example is the work of Barr and Marron [53] on the TARDIS debugger which provides very efficient omniscient debugging for C#. While most research on omniscient debugging is being done for GPLs, little work has been done to provide omniscient debugging for executable DSLs. In the following, we present some domain-specific omniscient debuggers for executable DSLs, and we discuss generic omniscient debugging for executable DSLs.

Domain-specific Omniscient Debugging. A few approaches provide omniscient debuggers that are domain-specific, *i.e.,* specific to a single executable DSL.

Krasnogolowy et al. [54] manually mapped GPL debugging concepts (*e.g.,* step, instruction, variable, stack, scope) to a *story diagram* executable DSL, and proposed a debugger following this mapping. In addition to breakpoints and step-wise execution, the resulting debugger provides advanced facilities such as, control flow visualization, variable modification, remote debugging and omniscient debugging. Back-stepping is achieved by creating an execution trace containing all the changes made to the execution state, hence making possible to undo these changes.

Laurent et al. [55] extended the standard fUML operational semantics in order to support debugging of fUML models. They proposed an extension to fUML operational semantics to make debugging possible, which includes the definition of a *controller* that centrally manages all model modifications as steps. This controller is extended to implement a debugger, with facilities such as breakpoints, stepwise execution and back stepping. They provide the possibility to roll-back the execution by relying on an execution trace containing the previous positions and contents of all the fUML tokens.

Maoz [56], Maoz et al. [36] worked on exploration of execution traces of scenario models. Since such approaches allow to explore previous states of an executed model, they are very similar to omniscient debugging, although it does not provide control over a running execution.

While these approaches do provide omniscient debugging, none of them can be used in a generic fashion for any executable DSL, contrary to the approach we propose.

Generic Omniscient Debugging for Executable DSLs. To our knowledge, no approaches aim at providing omniscient debugging to any executable DSL. We discuss below two approaches that share similarities with generic omniscient debugging for executable DSLs.

Corley et al. [57] propose omniscient debugging facilities for the cloud-based modeling solution AToMPM, in order to step both forward and backward in model transformations executed in an AToMPM runtime. Yet it does not support languages that are not model transformation languages.

Hegedüs et al. [58] propose an execution trace management approach for executable DSLs. In addition to an extensible execution trace metamodel, the approach includes a detailed way to *replay* execution traces obtained from previous executions or from counter-examples generated of a model-checker. While being able to step forward and backward according an execution trace is very similar to omniscient debugging, trace replay is only offline and it is not possible to step backwards during a model execution.

9.3. Integration of Runtime Services

While the approach we propose focuses on omniscient debugging, it also deals with concerns related to the integration of runtime services with operational semantics. We review two recent approaches that each propose how to achieve such integration.

Meyers et al. [6] propose an approach to test executable models, using test models conforming to a domain-specific test language automatically generated for a given executable DSL. To execute a test jointly with an executable model, the operational semantics of the considered executable DSL is instrumented specifically for this single test. In particular, this instrumentation adds the call of a new transformation rule *ProgressTestCase* in between each execution step.This implies that the instrumentation transformation is specific to the metaprogramming approach used to define the execution transformation of the semantics, and that a variant of the operational semantics must be generated for each test model. Moreover, this instrumentation is only valid for testing models, and not for other activities such as debugging. By contrast, the approach we propose is based on a pattern to decouple any sort of runtime service from the considered metaprogramming approach, and does not necessarily require the instrumentation of the operational semantics.

Drey and Teodorov [59] propose an object-oriented design pattern for defining monitoring services (*i.e.,* runtime services) for an executable DSL. This pattern can be used to define runtime monitors for operational semantics based on a visitor design pattern.This is accomplished by subtyping classes of the abstract syntax and of the operational semantics, which implies that a monitor is always specific to the considered executable DSL. While at design time a monitor defined with this pattern can be defined without altering the operational semantics, at runtime the object graph of the executable model is altered to add the new nodes specific to monitoring. By contrast, the pattern we propose to integrate runtime services is independent of metaprogramming approaches (*e.g.,* a visitor is not required), allows the definition of generic runtime services independent from a specific executable DSL, and does not alter the object graph of the executable model.

10. Conclusion and Future Work

Omniscient debugging is a promising dynamic V&V approach for executable DSLs that enables free traversal of the execution of a model. While most GPLs already have efficient debuggers, bringing omnicient debugging to any executable DSL is a tedious and error-prone task. A generic solution must support a wide range of executable DSLs independently of the metaprogramming approaches used for their implementation, and must be efficient to ensure the responsiveness of the debugger.

We presented an approach based on a *generic* omniscient debugger, defined independently of any metaprogramming approach, and supported by efficient generic trace management facilities. We provide an implementation for the GEMOC Studio, a language and modeling workbench, and an empirical evaluation based on a set of executable DSLs. We showed that the debugger can be used with different DSLs regardless of the metaprogramming approches used for their implementations, and we observed an improvement regarding both the memory consumption and the time to explore past and future states, when compared to two omniscient debuggers variants.

The direct perspectives of this work include: the definition of a property language to let the developer define complex breakpoints; and adapting omniscient debugging to support both external stimuli and the concurrency model in operational semantics [34], *e.g.,* to explore the possible executions traces of a single executable model.

Acknowledgments

This work was supported by: the Austrian Science Fund (FWF) P 28519-N31; the ANR INS Project GEMOC (ANR-12-INSE-0011); the French LEOC Project Clarity; the Austrian Federal Ministry of Science, Research and Economy and the National Foundation for Research, Technology and Development.

References

[1] Object Management Group, Semantics of a Foundational Subset for Executable UML Models, V 1.1, 2013.

[2] R. Bendraou, B. Combemale, X. Crégut, M. P. Gervais, Definition of an executable SPEM 2.0, in: Proceedings of the 14th Asia-Pacific Software Engineering Conference (APSEC'07), IEEE, ISBN 0769530575, ISSN 15301362, 390–397, 2007.

[3] T. Fischer, J. Niere, L. Torunski, A. Zündorf, 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, ISBN 3-540-67203-6, ISSN 16113349, 157–167, 2000.

[4] D. Harel, H. Lachover, A. Naamad, A. Pnuelli, M. Politi, R. Sherman, A. Shtull-trauring, M. Trakhtenbrot, STATEMATE: a working environment for the development of complex reactive systems, IEEE Transactions on software engineering 16 (4) (1990) 403–414.

[5] OASIS, Web Services Business Process Execution Language Version 2.0, URL `https://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`, 2007.

[6] B. Meyers, J. Denil, I. Dávid, H. Vangheluwe, Automated testing support for reactive domain-specific modelling languages, in: International Conference on Software Language Engineering (SLE), ISBN 9781450344470, 181–194, URL `http://dl.acm.org/citation.cfm?doid=2997364.2997367`, 2016.

[7] P. Langer, T. Mayerhofer, G. Kappel, Semantic Model Differencing Utilizing Behavioral Semantics Specifications, in: 17th Int. Conf. on Model Driven Eng. Lang. and Sys., vol. 8767 of *LNCS*, Springer, 116–132, 2014.

[8] M. Leucker, C. Schallhart, A brief account of runtime verification, Journal of Logic and Algebraic Programming 78 (5) (2009) 293–303.

[9] B. Combemale, X. Crégut, M. Pantel, A Design Pattern to Build Executable DSMLs and Associated V&V Tools, in: 19th Asia-Pacific Soft. Eng. Conf. (APSEC), vol. 1, IEEE, 2012.

[10] T. Mayerhofer, P. Langer, M. Wimmer, G. Kappel, xMOF: Executable DSMLs based on fUML, in: 6th Int. Conf. on Soft. Lang. Eng. (SLE), vol. 8225 of *LNCS*, Springer, 2013.

[11] G. Engels, J. H. Hausmann, R. Heckel, S. Sauer, Dynamic Meta-Modeling: 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), vol. 1939 of *LNCS*, Springer Berlin Heidelberg, ISBN 978-3-540-41133-8, 323–337, 2000.

[12] N. Bandener, C. Soltenborn, G. Engels, 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, Springer Berlin Heidelberg, ISBN 9783642194399, ISSN 03029743, 357–376, 2010.

[13] Á. Hegedüs, G. Bergmann, I. Ráth, D. Varró, 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), IEEE, 145–155, 2010.

[14] M. Soden, H. Eichler, 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.

[15] J. Tatibouët, A. Cuccuru, S. Gérard, F. Terrier, Formalizing Execution Semantics of UML Profiles with fUML Models, in: 17th Int. Conf. on Model Driven Eng. Lang. and Sys. (MODELS), vol. 8767 of *LNCS*, Springer, 2014.

[16] A. Lienhard, T. Gîrba, O. Nierstrasz, Practical Object-Oriented Back-in-Time Debugging, in: 22nd Eur. Conf. on Object-Oriented Programming (ECOOP), vol. 5142 of *LNCS*, Springer, 2008.

[17] E. Bousse, J. Corley, B. Combemale, J. Gray, B. Baudry, Supporting Efficient and Advanced Omniscient Debugging for xDSMLs, in: International Conference on Software Language Engineering (SLE), ACM, 2015.

[18] E. Bousse, T. Mayerhofer, B. Combemale, B. Baudry, A Generative Approach to Define Rich Domain-Specific Trace Metamodels, in: 11th Eur. Conf. on Modelling Foundations and Applications (ECMFA), vol. 9153 of *LNCS*, Springer, 2015.

[19] E. Bousse, T. Degueule, D. Vojtisek, T. Mayerhofer, J. Deantoni, B. Combemale, Execution Framework of the GEMOC Studio (Tool Demo), in: Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2016, Amsterdam, Netherlands, 8, URL `https://hal.inria.fr/hal-01355391`, 2016.

[20] R. F. Paige, D. S. Kolovos, F. A. C. Polack, Metamodelling for grammarware researchers, in: International Conference on Software Language Engineering (SLE), vol. 7745 of *LNCS*, Springer, ISBN 9783642360886, ISSN 03029743, 64–82, 2013.

[21] Object Management Group, Meta Object Facility (MOF) Core Specification, V 2.5, `http://www.omg.org/spec/MOF/2.5`, 2016.

[22] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks, EMF: Eclipse Modeling Framework, 2nd Edition, Eclipse Series, Addison-Wesley Professional, ISBN 0321331885, 2008.

[23] B. Combemale, X. Crégut, P.-L. Garoche, X. Thirioux, Essay on Semantics Definition in MDE - An Instrumented Approach for Model Verification, Journal of Soft. 4 (9) (2009) 943–958.

[24] Object Management Group, OMG Unified Modeling Language (OMG UML), V 2.5, `http://www.omg.org/spec/UML/2.5`, 2013.

[25] G. Csertan, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, D. Varro, 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), IEEE, ISBN 0-7695-1736-6, ISSN 1527-1366, 267 – 270, 2002.

[26] F. Jouault, I. Kurtev, Transforming models with ATL, in: Proceedings of the Workshop on Model Transformations in Practice (MTiP'05), vol. 3844 of *LNCS*, Springer Berlin Heidelberg, 128–138, 2006.

[27] J.-M. Jézéquel, B. Combemale, O. Barais, M. Monperrus, F. Fouquet, Mashup of metalanguages and its implementation in the Kermeta language workbench, Software & Systems Modeling (SoSyM) 14 (2).

[28] Peggy Aldrich Kidwell, Stalking the Elusive Computer Bug, IEEE Annals Of The History Of Computing 20 (4) (1998) 3–7.

[29] A. Zeller, Why Program Fail – 1st Edition, Elsevier, ISBN 9780080481739, URL `http://www.whyprogramsfail.com/`, 2004.

[30] G. Pothier, É. Tanter, Back to the future: Omniscient debugging, IEEE Software 26 (6).

[31] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold, An Overview of AspectJ, in: J. L. Knudsen (Ed.), 15th European Conference on Object-Oriented Programming (ECOOP), Springer Berlin Heidelberg, Berlin, Heidelberg, ISBN 978-3-540-45337-6, 327–354, URL `https://doi.org/10.1007/3-540-45337-7_18`, 2001.

[32] J. Deantoni, P. Issa Diallo, C. Teodorov, J. Champeau, B. Combemale, Towards a Meta-Language for the Concurrency Concern in DSLs, in: Design, Automation and Test in Europe Conference and Exhibition (DATE), Grenoble, France, URL `https://hal.inria.fr/hal-01087442`, 2015.

[33] S. Efftinge, M. Eysholdt, J. Köhnlein, S. Zarnekow, R. von Massow, W. Hasselbring, M. Hanus, Xbase: Implementing Domain-specific Languages for Java, in: 11th International Conference on Generative Programming and Component Engineering (GPCE), ACM, ISBN 978-1-4503-1129-8, 112–121, URL `http://doi.acm.org/10.1145/2371401.2371419`, 2012.

[34] B. Combemale, J. Deantoni, M. V. Larsen, F. Mallet, O. Barais, B. Baudry, R. France, Reifying Concurrency for Executable Metamodeling, in: 6th Int. Conf. on Soft. Lang. Eng. (SLE), vol. 8225 of *LNCS*, Springer, 2013.

[35] M. E. Vara Larsen, J. Deantoni, B. Combemale, F. Mallet, A Behavioral Coordination Operator Language (BCOoL), in: International Conference on Model Driven Engineering Languages and Systems (MODELS), 18, ACM, 462, URL `https:`

//hal.inria.fr/hal-01182773, 2015.
[36] S. Maoz, J. O. Ringert, B. Rumpe, ADDiff: Semantic Differencing for Activity Diagrams, in: 19th ACM SIGSOFT Symposium and the 13th Eur. Conf. on Foundations of Soft. Eng. (ESEC/FSE), ACM, 2011.
[37] S. Van Mierlo, Y. Van Tendeloo, H. Vangheluwe, Debugging Parallel DEVS, SIMULATION 93 (4) (2017) 285–306, URL http://dx.doi.org/10.1177/0037549716658360.
[38] T. Mayerhofer, P. Langer, G. Kappel, A runtime model for fUML, in: Workshop on Models@run.time (MRT), ACM, 53–58, 2012.
[39] D. Riehle, S. Fraleigh, D. Bucka-Lassen, N. Omorogbe, The architecture of a UML virtual machine, in: International Conference on Object Oriented Programming Systems Languages and Applications (OOSPLA), ACM, 327–341, 2001.
[40] A. Kirshin, D. Dotan, A. Hartman, A UML Simulator Based on a Generic Model Execution Engine, in: Workshop on Multi-Paradigm Modeling (MPM), vol. 4364 of *LNCS*, Springer Berlin Heidelberg, 324–326, 2006.
[41] D. Dotan, A. Kirshin, Debugging and testing behavioral UML models, in: International Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM, ISBN 9781595938657, 838–839, 2007.
[42] M. L. Crane, J. Dingel, Towards a UML Virtual Machine: Implementing an Interpreter for UML 2 Actions and Activities, in: Conference of the center for advanced studies on collaborative research: meeting of minds (CASCON), ACM, 96, 2008.
[43] L. Fuentes, J. Manrique, P. Sánchez, Execution and simulation of (profiled) UML models using pópulo, in: International workshop on Models in Software Engineering (MiSE), ACM, ISBN 9781605580258, 75–81, 2008.
[44] B. Combemale, X. Crégut, J.-P. Giacometti, P. Michel, M. Pantel, Introducing Simulation and Model Animation in the MDE Topcased Toolkit, in: European congress on Embedded Real Time Software and Systems (ERTS), 2008.
[45] I. Ráth, D. Vágó, D. Varró, Design-time simulation of domain-specific models by incremental pattern matching, in: Symposium on Visual Languages and Human-Centric Computing (VL/HCC), IEEE, 219–222, 2008.
[46] H. Wu, J. Gray, M. Mernik, Grammar-driven generation of domain-specific language debuggers, Software: Practice and Experience 38 (10) (2008) 1073–1103.
[47] R. T. Lindeman, L. C. L. Kats, E. Visser, Declaratively Defining Domain-Specific Language Debuggers, in: International Conference on Generative Programming and Component Engineering (GPCE), ACM, 127–136, 2012.
[48] A. Chiş, T. Gîrba, O. Nierstrasz, The Moldable Debugger: a Framework for Developing Domain-Specific Debuggers, in: 7th Int. Conf. on Soft. Lang. Eng. (SLE), vol. 8706 of *LNCS*, Springer, 2014.
[49] A. Chiş, T. Gîrba, O. Nierstrasz, M. Denker, Practical domain-specific debuggers using the Moldable Debugger framework, Computer Languages, Systems & Structures 44 (Part A) (2015) 89–113, ISSN 14778424.
[50] B. Boothe, Efficient algorithms for bidirectional debugging, in: Conference on Programming Language Design and Implementation (PLDI), ACM, ISBN 1-58113-199-2, ISSN 03621340, 299–310, 2000.
[51] B. Lewis, Debugging backwards in time, in: International Workshop on Workshop on Automated and Algorithmic Debugging (AADEBUG), 2003.
[52] A. Georges, M. Christiaens, M. Ronsse, K. De Bosschere, JaRec: a portable record/replay environment for multi-threaded Java applications, Software: Practice and Experience 34 (6) (2004) 523–547, ISSN 0038-0644.
[53] E. T. Barr, M. Marron, TARDIS: Affordable Time-Travel Debugging in Managed Runtimes, in: International Conference on Object Oriented Programming Systems Languages & Applications (OOSPLA), ACM, ISBN 9781450325851, ISSN 15232867, 2014.
[54] A. Krasnogolowy, S. Hildebrandt, S. Wlitzoldt, Flexible Debugging of Behavior Models, in: International Conference on Industrial Technology (ICIT), IEEE, 331–336, 2012.
[55] Y. Laurent, R. Bendraou, M. Gervais, Executing and debugging UML models: an fUML extension, in: Symposium on Applied Computing (SAC), ACM, 1095–1102, 2013.
[56] S. Maoz, Model-based traces, in: Workshop on Models@runtime (MRT), vol. 5421 of *LNCS*, Springer Berlin Heidelberg, 109–119, 2009.
[57] J. Corley, B. P. Eddy, J. Gray, Towards Efficient and Scalabale Omniscient Debugging for Model Transformations, in: Proceedings of the 14th Workshop on Domain-Specific Modeling (DSM'14), ACM, 13–18, 2014.
[58] Á. Hegedüs, I. Ráth, D. Varró, Back-annotation framework for Simulation Traces of Discrete Event-based Languages, Tech. Rep., BME, URL https://inf.mit.bme.hu/sites/default/files/publications/Hegedus_TechRep_201004.pdf, 2010.
[59] Z. Drey, C. Teodorov, Object-oriented design pattern for DSL program monitoring, in: Intern. Conf. on Software Language Engineering (SLE), ISBN 9781450344470, 70–83, URL http://dl.acm.org/citation.cfm?doid=2997364.2997373, 2016.

```
[1] begin
[2]   B_enabled ← {b ∈ d_state.breakpoints | enabled(b)}
[3]   if B_enabled ≠ ∅ then
[4]     B_enabledStepping ← {b ∈ B_enabled | b.stepping}
[5]     d_state.breakpoints.remove(B_enabledStepping)
[6]     sleep()
```

Algorithm 6: *pauseIfBreakpoints*

Appendix A. Example of a Scheduling Execution Engine

We consider a different metaprogramming approach that does *not* allow the definition of rules that may call other step rules. Instead, the order of execution of step rules is provided by a scheduling policy that may be imposed by the metaprogramming language itself, or may be explicitly defined next to the rules of the transformation (*e.g.,* in the form of a model of computation). This approach implies that an execution would never lead to nested steps, and hence does not require the use of any intrusive callback mechanism to inform an engine of starting and ending steps. Instead, an execution engine specific to this approach can be responsible for directly triggering each execution step, which gives it the opportunity to notify listeners before and after each step.

Figure A.18 shows a sequence diagrams illustrating the behavior of an execution engine specific to a scheduling based metaprogramming approach. The beginning of the diagram is identical to Figure 4a, with the initialization of the engine and the *executionStarting* notification sent to listeners. Then, the engine enters a loop where all required execution steps are performed, based on a scheduling policy. Since each execution step is triggered by the engine, and since an execution step may not trigger another execution step, the engine can directly send the *stepStarting* notification to listeners before executing the next step rule, and *stepEnding* after having executed it. Hence, no callbacks from the execution transformation are required.

Appendix B. Integration Services of the Omniscient Debugger

Algorithm 6 shows the *pauseIfBreakpoints* internal service, which is used to put the execution thread to sleep if breakpoints are enabled. First, we suppose a service called *enabled* is available to check whether or not the condition of a breakpoint is true, and we use it to gather all enabled breakpoints (line 2). Then, if such breakpoints exist (line 3), we discard the `stepping` breapoints (lines 4–5), and we pause the execution until some debugging service is called (line 6). For this last task, we suppose that two services are available: *sleep* to suspend the current execution thread of the engine, and *wakeUp* (used later for debugging services) to resume the execution thread of the engine.

Appendix C. Internal Services of the Omniscient Debugger

Table C.5 shows internal services required by the debugger, and used for defining the debugging services thereafter. Without detailing, the services are the following: *restoreInProgressSteps* reconstructs the list of *in progress* steps of the current ExecutionState based on a given execution step, *getLastStep* retrieves the last execution step of the the execution trace, *inLastExeState* checks whether the last execution state of the trace was reached, *isInitialExeState* checks whether the initial execution state of the trace was reached, and *getCurrentValue* retrieves the current value of one the the dimensions of the executed model, *i.e.,* the value of the corresponding dynamic field.

Appendix D. Backward Services of the Omniscient Debugger

Table D.6 shows the backward debugging services of the omniscient debugger. These services are very similar to the forward ones, except in the opposite direction of the execution: *playBackwards* continuously jumps to the previous available execution state instead of the following state, and each stepping operator (*backInto*, *backOver* and *backOut*) define a breakpoint to pause when the relevant step is *starting* instead of *ending*. The main difference is that once the first execution state of the trace is reached, it is not required to resume the execution engine.

Appendix E. Screenshot of the Implemented Omniscient Debugger

Figure E.19 shows a screenshot of the GEMOC modeling workbench during the debugging of an activity diagram model. In the middle, two representations of the model are shown: a graphical representation on the left including a representation of the model state in the form of tokens atop the different activity nodes, and textual representation on the right. At the top right, the variable view shows a complementary representation of the model state in the form of a list of all dynamic fields with their values. At the top left, the call stack shows both the *in progress* and *starting* execution steps. The debugger is integrated with the Eclipse debug API, which means that the Eclipse toolbar can directly be used for stepping into, over or out. Next to this bar, we added an set of buttons for the backwards stepping services. At the bottom, a representation of the execution trace is shown in the form of a *timeline*. Each model state is shown as a circle, and each execution step is shown as a line. Below the row of model states, each row is dedicated to a dynamic field of the model, and each bar represents a single value reached by this field. The current execution state is shown by highlighting in yellow the current model state, the current execution steps, and the current values in the dimensions. Jumping services can be triggered by simply double-clicking either on the model states.

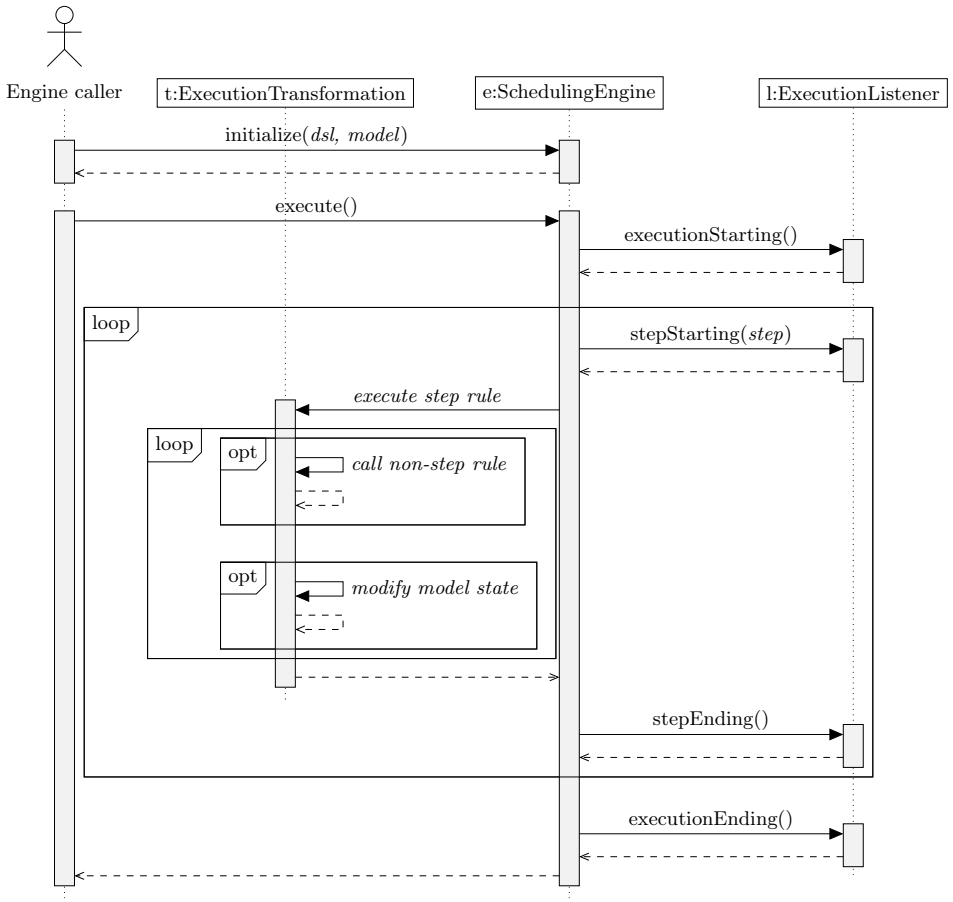


Figure A.18: Example of engine that schedules itself the execution of step rules, without nested step rules and without any callbacks.

Internal Service	**Definition**
restoreInProgressSteps(s)	[1] d_{state}.exeState.stepping.inProgress $\leftarrow \varnothing$ [2] *parent* $= s$.container [3] **while** *parent* $\neq$ *null* $\wedge$ *parent* is Step **do** [4] $\quad d_{state}$.exeState.stepping.inProgress.push(*parent*) [5] $\quad$ *parent* $= s$.container
getLastStep()	[1] $s \leftarrow$ *trace*.rootStep [2] **while** s.nestedSteps $\neq \varnothing$ **do** [3] $\quad s \leftarrow s$.nestedSteps.last() [4] **return** s
inLastExeState()	[1] $s_{start} \leftarrow d_{state}$.exeState.stepping.starting [2] $s_{end} \leftarrow d_{state}$.exeState.stepping.ending [3] $s_{last} \leftarrow$ getLastStep() [4] **return** $(s_{end} = s_{last}) \vee (s_{last}$.endingState $=$ *null* $\wedge\ s_{start} = s_{last})$
inInitialExeState()	[1] **return** d_{state}.exeState.stepping.starting $=$ *trace*.rootStep
getCurrentValue(*dim* : Dimension)	[1] $v :\in (d_{state}$.exeState.modelState.values $\cap$ *dim*.values$)$ [2] **return** v

Table C.5: Internal services of the omniscient debugger

Backward Debugging Service	Definition
playBackwards()	[1] $B_{enabled} \leftarrow \varnothing$ [2] **while** ¬inInitialExeState() $\wedge$ $B_{enabled} = \varnothing$ **do** [3] **if** d_{state}.exeState.stepping.ending $\neq$ *null* **then** [4] $s_{end} \leftarrow d_{state}$.exeState.stepping.ending [5] **if** s_{end}.nestedSteps $\neq \varnothing$ **then** [6] jumpToEndingStep(s_{end}.nestedSteps.last()) [7] **else** [8] jumpToStartingStep(s_{end}) [9] **else** [10] $s_{start} \leftarrow d_{state}$.exeState.stepping.starting [11] jumpToStartingStep(s_{start}.container) [12] $B_{enabled} \leftarrow \{b \in d_{state}$.breakpoints $\mid$ enabled(b)$\}$ [13] **if** $B_{enabled} \neq \varnothing$ **then** [14] $B_{enabledStepping} \leftarrow \{b \in B_{enabled} \mid b$.stepping$\}$ [15] d_{state}.breakpoints.remove($B_{enabledStepping}$)
backInto()	[1] **if** d_{state}.exeState.stepping.ending $\neq$ *null* **then** [2] d_{state}.breakpoints.add([*true*]) [3] playBackwards()
backOver()	[1] **if** d_{state}.exeState.stepping.ending $\neq$ *null* **then** [2] $s_{over} \leftarrow d_{state}$.exeState.stepping.ending [3] d_{state}.breakpoints.add([d_{state}.exeState.stepping.starting $= s_{over}$]) [4] playBackwards()
backOut()	[1] **if** d_{state}.exeState.stepping.inProgress $\neq \varnothing$ **then** [2] $s_{out} \leftarrow d_{state}$.exeState.stepping.inProgress.peek() [3] d_{state}.breakpoints.add([d_{state}.exeState.stepping.starting $= s_{out}$]) [4] playBackwards()

Table D.6: Backward services of the omniscient debugger

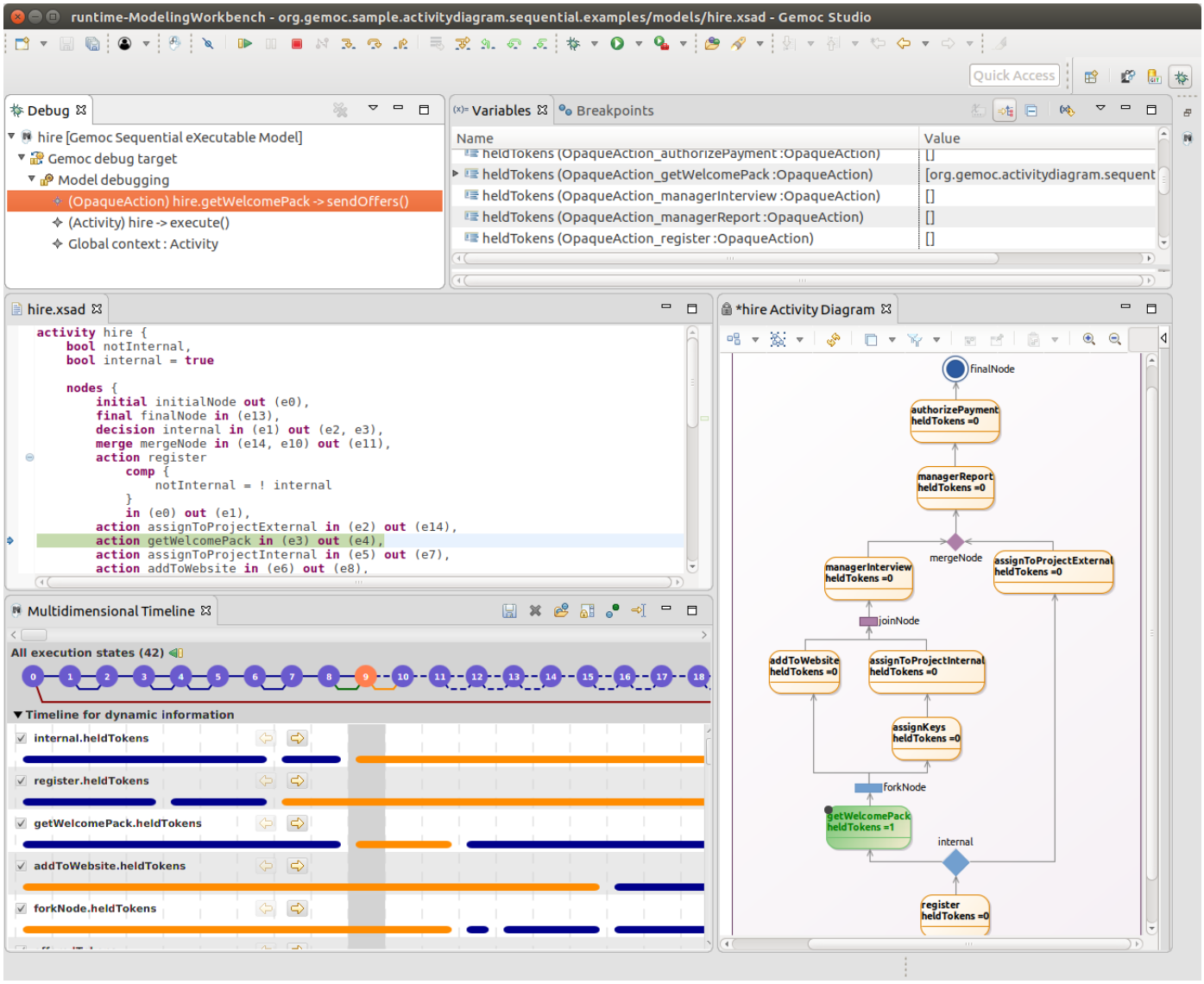


Figure E.19: Omniscient debugger of the GEMOC modeling workbench during the execution of an fUML activity diagram.