HAL
open science

Supporting Efficient and Advanced Omniscient Debugging for xDSMLs

Erwan Bousse, Jonathan Corley, Benoit Combemale, Jeff Gray, Benoit Baudry

To cite this version:

Erwan Bousse, Jonathan Corley, Benoit Combemale, Jeff Gray, Benoit Baudry. Supporting Efficient and Advanced Omniscient Debugging for xDSMLs. 8th International Conference on Software Language Engineering (SLE), Oct 2015, Pittsburg, United States. hal-01182517v1

HAL Id: hal-01182517

https://inria.hal.science/hal-01182517v1

Submitted on 31 Jul 2015 (v1), last revised 21 Aug 2015 (v2)

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

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

Supporting Efficient and Advanced Omniscient Debugging for xDSMLs

Erwan Bousse
Irisa
Rennes, France
erwan.bousse@irisa.fr

Jonathan Corley
University of Alabama
Tuscaloosa, AL, USA
corle001@crimson.ua.edu

Benoit Combemale
Inria
Rennes, France
benoit.combemale@inria.fr

Jeff Gray
University of Alabama
Tuscaloosa, AL, USA
gray@cs.ua.edu

Benoit Baudry
Inria
Rennes, France
benoit.baudry@inria.fr

Abstract

Omniscient debugging is a promising technique that relies on execution traces to enable free traversal of the states reached by a system during an execution. While some General-Purpose Languages (GPLs) already have support for omniscient debugging, developing such a complex tool for any executable Domain-Specific Modeling Language (xDSML) remains a challenging and error prone task. A solution to this problem is to define a *generic* omniscient debugger for all xDSMLs. However, generically supporting any xDSML both compromises the efficiency and the usability of such an approach. Our contribution relies on a *partly generic* omniscient debugger supported by *generated domain-specific* trace management facilities. Being domain-specific, these facilities are tuned to the considered xDSML for better efficiency. Usability is strengthened by providing *multidimensional* omniscient debugging. Results show that our approach is on average 3.0 times more efficient in memory and 5.03 more efficient in time when compared to a generic solution that copies the model at each step.

Keywords Omniscient Debugger, xDSML, Execution Trace

1. Introduction

Many recent efforts aim at providing facilities to design executable Domain-Specific Modeling Languages (xDSMLs) [4, 13, 17]. Among others, xDSMLs allow system engineers to analyze behavioral properties early in the development process. In particular, *debugging* is a common dynamic facility to observe and control an execution in order to better understand a behavior or to look for the cause of a defect. However, standard debugging only provides facilities to pause and step *forward* during an execution, hence requiring to go back from the start to give a second look at a situation. To cope with this issue, *omniscient debugging* is a promising technique that relies on execution traces to enable free traversal of the states reached by a system, thereby allowing developers to "go back in time." [10]

While most general-purpose languages (GPLs) already have their own efficient standard debugger (*e.g.,* Java[1]) or omniscient debugger (*e.g.,* also Java [16]), developing such a complex tool for any xDSML remains a difficult and error prone task. Despite the specificities of each xDSML, it is possible to identify a common set of debugging facilities for all xDSMLs. Thus, to avoid manual creation of each debugger, a possible solution is to define a generic omniscient debugger that would work for any xDSML. However, handling any xDSML has two main consequences: (1) There is necessarily a trade-off between genericity and efficiency of the debugging operations, since supporting any xDSML requires the use of expensive introspection, conditionals, or type checks to support a wide variety of abstract syntax and runtime data structures. Moreover, since debugging is an interactive activity, *responsiveness* is of primary importance. Hence, a first concern is the *efficiency* of a generic debugger. (2) The execution data structure defined in an xDSML can be arbitrarily complex (*e.g.,* a large object-oriented structure), and therefore difficult to comprehend in a debugging session, especially if the execution leads to a large amount of states. Hence, a second concern is the *usability* of omniscient debugging for xDSMLs; *i.e.,* specific *advanced* facilities are required to manage the complexity and size of the executions. To summarize, the following are key objectives that drive the focus of this paper:

O#1: Providing efficient omniscient debugging facilities, to ensure responsiveness of the debugger.

[1] http://docs.oracle.com/javase/8/docs/technotes/tools/unix/jdb.html

O#2: Offering advanced omniscient debugging facilities, to improve the usability.

To address O#1, we propose to go from a *generic* omniscient debugger to a *generic meta-approach* to define omniscient debuggers. Such a generative approach can provide an efficient and finely tuned omniscient debugger for any xDSML. Yet, considering a generic set of debugging services for all xDSMLs, both the interface and some underlying logic of a debugger can remain generic without compromising efficiency. Hence, our contribution relies on a *partially generic* omniscient debugger supported by *generated domain-specific* trace management facilities. The trace management facilities include a *domain-specific trace metamodel* that precisely captures the execution state of a model conforming to the xDSML, and a *domain-specific trace manager* providing all the required services to manipulate the execution trace generically. Because the trace manager is domain-specific, it is finely tuned to the considered xDSML and to the generated trace metamodel, and hence more efficient than a generic one. To address O#2, our contribution provides *multidimensional omniscient debugging* services, which mix both omniscient debugging services, and advanced facilities to navigate among the *values* of specific elements of the executed model.

We implemented our approach as part of the GEMOC Studio, a language and modeling workbench; and we conducted an empirical evaluation. To evaluate the efficiency of our solution, we assessed its quality with regard to both memory consumption and the time required to run omniscient debugging operations. We compared our approach with two generic omniscient debuggers: one that simulates omniscient debugging by resetting the execution engine and re-executing until the target state is reached, and one that copies the model at each execution step. Obtained results show that our approach is on average 3.0 times more efficient in memory when compared to the second debugger, and respectively 54.1 and 5.03 times more efficient in time when compared respectively to the first and the second debugger.

The remaining sections are as follows. Section 2 defines the considered scope of model execution and model debugging. Section 3 presents our generative approach to provide generic multidimensional omniscient debugging. Section 4 describes a prototype supporting the technique in the GEMOC Studio. Section 5 discusses the evaluation of our approach. Finally, Section 6 discusses related work and Section 7 concludes the paper.

2. Model Execution and Model Debugging

In this section, we precisely define the scope of the considered executable models, their specification into xDSMLs, and the resulting execution trace corresponding to the application of the latter to the former. Then we define the scope of the advanced omniscient debugging facilities proposed to analyze executable models.

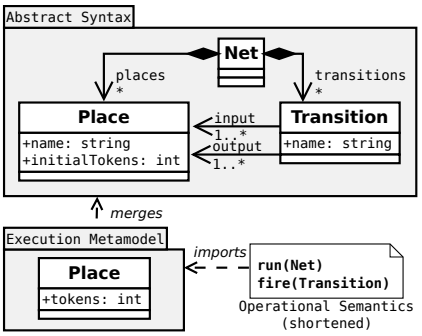

Figure 1: Petri Net xDSML

2.1 Model Execution

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

While the purpose of metamodeling is to define languages, *executable* metamodeling also includes defining execution semantics within the language definition. This is done through xDSMLs, languages that include the definitions of the *execution state* of a model conforming to the language, and *execution semantics* that operate on this state.

There are two different approaches to define execution semantics: *translational* semantics and *operational* semantics. We focus in this paper on the case of operational semantics, and we plan to consider translational semantics in the future.

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

- *An* abstract syntax*, that is a metamodel. We call* immutable property *a property introduced in this metamodel. At the model level, we call* immutable field *an object's field based on an immutable property.*
- *An* execution metamodel*, that extends the abstract syntax by package merge* [2]*. We call a property introduced in this metamodel a* mutable property*. In a model, a* mutable field *is an object's field based on a* mutable property.
- Operational semantics*, that are a set of transformation rules that modify a model conforming to the execution metamodel by changing values of mutable fields and creating/destroying instances of classes introduced in the execution metamodel.*

[2] Note that in practice, existing tools and approaches use different but similar extension mechanisms; *e.g.,* Kermeta [9] uses aspect weaving and xMOF [13] uses generalization.

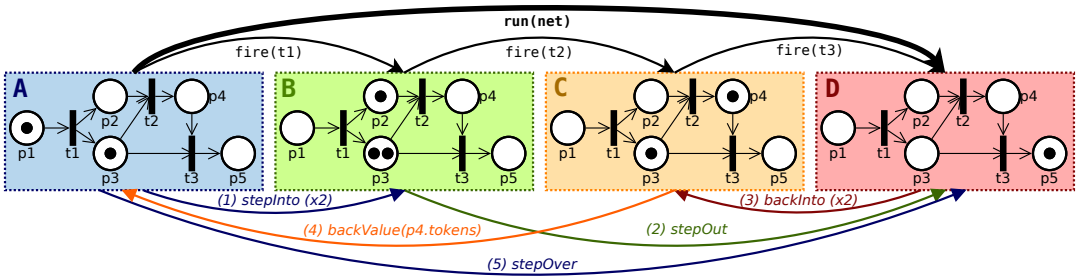

Figure 2: Example of Petri Net execution trace annotated with the use of a selection of debugging services

- *An* initialization function *that, when given a model conforming to the abstract syntax, returns a model conforming to the execution metamodel.*

Figure 1 shows an example of a Petri Net xDSML. At the left, its abstract syntax is depicted with three classes Net, Place and Transition. Next to the abstract syntax is the execution metamodel, that extends the class Place using *package merge* with a new mutable property `tokens`. The initialization function (not shown) transforms each original object into an executable object (*e.g.,* a Place object gains a `tokens` field) as defined in the execution metamodel. The function also initializes each `tokens` field with the value of `initialTokens`. At the right are the signatures of the rules defined in the operational semantics. The rules are defined as follows: `run` continuously looks for enabled transitions, and uses `fire` to transfer tokens from input to output places of such transitions.

Model Execution Given an xDSML, executing a conforming model consists of the application, on demand, of the transformation rules that define its operational semantics.

Execution Trace Execution traces aim at *capturing* both the changes in the mutable fields of the model, and the applied transformation rules that led to these changes. While traces can take various forms, this work considers that an execution trace is a sequence of *states* capturing the values of the mutable fields and *steps* capturing the application of transformation rules.

Definition 2 *An* execution trace *is defined by:*

- *A sequence of* execution states*, each capturing the values of all mutable fields in the model.*
- *A set of* steps*, each being the application of a transformation rule of the operational semantics. A step going from a state to its following state is called a* small step*, while a step containing several steps is called a* big step*.*

Figure 2 presents a trace from the execution of a Petri Net model conforming to the Petri Net xDSML shown in Figure 1. The trace is composed of four states, on top of which are depicted the steps of the execution. States are separated by three *small steps* that represent the applications of the *fire* transformation rule. A *big step* goes from the first state to the last state to represent the application of the *run* rule. Note that the bottom part of the figure (*i.e.,* the colored arrow) is described later in Section 2.2.

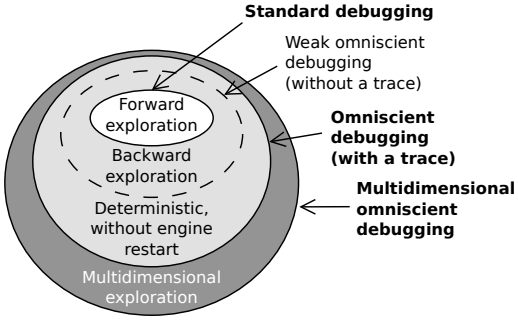

Figure 3: Feature comparison of the debugging approaches

2.2 Model Debugging

Debugging Approaches Debugging an executable model involves *controlling* the model's execution and observing the states traversed. Figure 3 shows four approaches to achieve this, with different levels of control over the execution. First, *standard debugging* traverses forward the states reached by the model through the application of the operational semantics rules. Second, we call *weak omniscient debugging* the possibility to go backward in the exploration of the states through a restart of the execution engine and a re-execution until the target state is reached. Note that this can be accomplished manually with any standard debugger. Third, *omniscient debugging* relies on an execution trace to revert the executed model into a prior state. Using a trace makes the procedure deterministic (*i.e.,* the exact same states are visited) even if the model or the operational semantics are non-deterministic. Finally, our proposal relies on *multidimensional omniscient debugging*, which adds facilities to navigate among the values of mutable fields of the model. In the remainder of this section, we present these debugging approaches as sets of provided services. Note that all these services are only valid when the execution is *paused*, *i.e.,* when the execution engine waits for instruction before the application of a transformation rule.

Standard Debugging Most debuggers only provide *standard debugging*, which includes the following forward exploration services:

- ***breakpoint***: pause the execution when a specified condition is true (*e.g.,* a transformation rule is reached).
- ***stepInto***: resume execution and pauses after either executing a single *small step* or moving to the next step encountered in the following *big step*.
- ***stepOver***: resume execution and pause when the next step is completed (including the contained steps, if this is a *big step*).
- ***stepOut***: resume execution and pause when the first step not contained within the current *big step* is reached.
- ***play***: resume execution.
- ***visualization of the current state***: display the values of relevant mutable elements.

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

- ***jump***: revert the model to a specified state.
- ***backInto***: revert a single *small step* or moves to the last step encountered in a *big step*.
- ***backOver***: revert the last encountered step (including the contained steps, if the last step is a *big step*).
- ***backOut***: revert all the remaining steps within the current *big step*.
- ***playBackwards***: continuously revert execution until the execution is paused or the initial state is reached.
- ***visualization of the trace***: display an interactive representation of the reached execution states and show which state is current.

Multidimensional Omniscient Debugging With the ability to go both forward and backward, a developer can explore any state of a model's execution. Yet, large traces are difficult to navigate practically, and information stored within a state can be arbitrarily complex, compromising usability (O#2). To cope with this issue, we investigate *multidimensional omniscient debugging*; *i.e.,* facilities to navigate among the *values* of the mutable fields of the model:

- ***jumpValue***: jump to the first state in which a given mutable field has a given value.
- ***stepValue***: given a mutable field, jump to the next value of this field.
- ***backValue***: given a mutable field, jump to the previous value of this field.
- ***visualization of the value sequences***: display an interactive representation of the reached values of the mutable fields and show which values are the current ones.

Example Debugging Scenario Consider a complete execution and debugging scenario with a Petri Net model conforming to the xDSML, as shown in Figure 1. The initial state of the considered Petri Net model is depicted at the left of Figure 2 with the label A. First, we set a *breakpoint* in order to pause the execution right after it starts. Then we start the execution and reach the first A state. From there, the next step is an application of `run`. We perform a first *stepInto* (1), which does not change the current state, but presents us with a new next step, which is an application of `fire` on `t1`. We then use *stepInto* a second time (1), which applies the `fire` *small step* and brings us to the B state. From there, we use *stepOut* (2) to get out from the current *big step* (*i.e.,* `run`), which brings us to the D state. At this point, the execution trace is fully constructed, and no additional transformation rules will be applied.

Then, similar to the beginning of the scenario, we apply twice *backInto* (3) to reach the C state. We then use *backValue* (4) to go back to the previous value reached by the `tokens` field of the p4 Place object. While p4 has one token in the C state, its previous amount was zero, which started in the A state. Hence, we reach the A state again. Finally, this time we use *stepOver* (5) to directly follow the first step (*i.e.,* `run`) and we reach the D state again. Note that in this case, *stepOver* should not apply any transformation rules, but simply read information from the execution trace to directly revert the executed model into the stored D state.

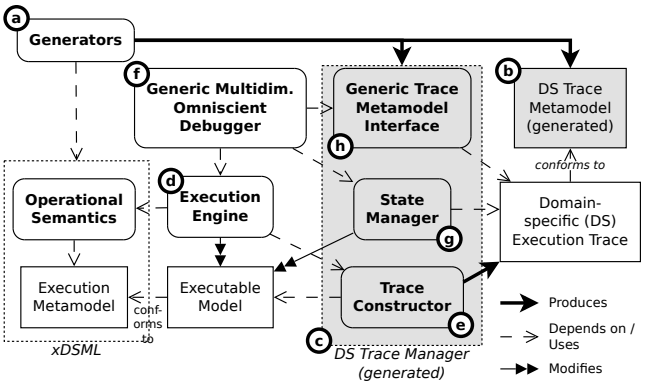

Figure 4: Overview of the approach

3. A Generative Approach for Multidimensional Omniscient Debugging

This section presents our approach that provides efficient and advanced omniscient debugging for xDSMLs using a partially generic, multidimensional omniscient debugger supported by generated domain-specific trace management facilities.

3.1 Overview of the Approach

Defining an xDSML implies the definition of a number of domain-specific facilities to edit or analyze a model conforming to the language. In particular, one method to provide a visual animation of a model execution is to observe

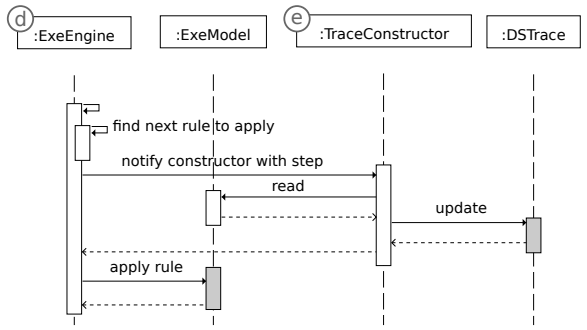

Figure 5: Interactions when a *small step* is to be computed and added to the trace

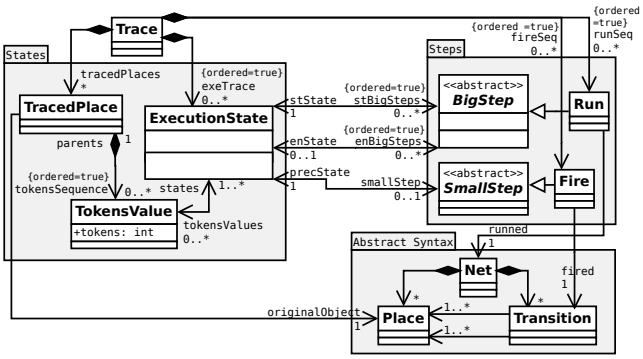

Figure 6: Petri net rich domain-specific trace metamodel

the model and react to changes. Because such a pattern is common when defining tools for xDSMLs, our approach is designed to have a *single instance* of the executed model loaded at any given time that can be modified throughout the execution and the debugging session.

Figure 4 shows an overview of our approach. We consider that the *initialization function* of the xDSML was already applied to an input model, creating the *executable model*. The first step of our approach relies on *generators* (a), which take the considered xDSML as input to produce two domain-specific components: a *trace metamodel* (b) and a *trace manager* (c). The second step is the execution and the debugging of the model. The *execution engine* (d) applies the *operational semantics* to change the model and uses the *trace constructor* (e) from the trace manager to construct a domain-specific trace. The *generic multidimensional omniscient debugger* (f) provides all the services described in Section 2 by controlling the execution engine and relying on the *state manager* (g) to revert the model into previous states. Additionally, the debugger relies on the *generic trace metamodel interface* (h) to manipulate the trace.

To illustrate a subset of the interactions between the components shown in Figure 4, Figure 5 shows a sequence diagram that sketches what happens when a *small step* must be computed and stored in the trace. Duration bars depicted in gray represent changes made in the affected element. First, the *engine* (d) determines the next rule to apply then notifies the *trace constructor* (e) that a small step will occur. As a result, the trace constructor reads the *executed model*, and updates the *domain-specific trace* with new elements accordingly (*e.g.,* add a new *small step* and, if the model was altered, a new state). Finally, the execution engine applies the rule and modifies the executed model accordingly.

We present all of these components in more detail in the remainder of this section.

3.2 Execution Engine

First and foremost, an omniscient debugger must provide precise control over the execution of a model, such as the ability to pause during execution or traverse the trace in a controlled manner. For this to be possible, the *execution engine* (d) must adhere to certain specifications. The engine must be able to drive the execution of the model (*i.e.,* initialization, start, stop), and to provide to the debugger some control over the execution. This includes the ability to pause the execution at a specific state during execution, and the ability to resume the execution from a paused state. We assume that the engine provides at least the following services:

- ***pauseWhen***: order to suspend the execution in between two transformation rules applications as soon as a given predicate is `true`.
- ***isPaused***: return `true` if the engine is paused.
- ***resume***: resume execution (*i.e.,* cancel a pause).

As presented in Section 4.1, we developed an execution engine that encompasses the aforementioned services.

3.3 Domain-Specific Trace Metamodel

In our prior work [2], we presented a generative approach that automatically provides a rich domain-specific trace metamodel for an xDSML. Instead of relying on clones of the executed model to construct a trace, the metamodel precisely captures its execution state through an efficient object-oriented structure based on the mutable properties of the xDSML. In addition, the structure provides rich navigation facilities to browse a trace according to the values reached by the mutable fields of the model. To benefit from such efficient trace structures (O#1), we rely on this approach for the *automatic generation* (a in Figure 4) of the *domain-specific trace metamodel* (b in Figure 4).

To support omniscient debugging, we extended this generative approach with the notion of *big step*, in addition to *small step*. This included adding a BigStep class to the base classes that are generated, the derivation of domain-specific *big step* classes from the xDSML operation definitions, and extending the generation algorithm to create inheritance links from *big step* classes to BigStep.

Figure 6 shows an example of a rich domain-specific trace metamodel for Petri Nets using our approach. As an overview, the state of the model is captured in the Execution-

State class, which is composed of a tuple of TokensValue objects, each of which represents the value of a `tokens` field at a given point in time. Each TracedPlace object captures all the values of the mutable fields of a specific Place object of the executed model. On the right, steps are represented by Run and Fire, which inherit respectively BigStep and SmallStep.

3.4 Trace Constructor

To provide omniscient debugging, we must construct an execution trace during the execution of the model. We have defined the following set of operations to be provided by the *trace constructor* (e in Figure 4):

- ***initialize***: create the base elements of the trace.
- ***addState***: add a new state in the trace if a mutable field of the model changed, or if instances of classes introduced in the execution metamodel are created/deleted;
- ***addSmallStep:*** add a *small step* in the trace.
- ***bigStepStarted***: notify that a *big step* has started.
- ***bigStepEnded***: notify that a *big step* has ended.

As explained in Section 2.1, the execution of a model consists in the application of a sequence of transformation rules. To capture an execution state that matches a model conforming to the execution metamodel, the operation *addState* must be called just before or after the transformation rule.

Since a *big step* is simply a sequence of *small steps*, we only need to capture states before and after *small steps*. However, we also need to capture when steps occur, hence *addSmallStep* must be called at each transformation rule that matches a *small step*, while *bigStepStarted* and *bigStepEnded* must be called before and after a rule matching a *big step*, respectively. In summary, all the calls required to construct the trace are as follows:

- Just before the first *small step*: *initialization*
- Just before a *small step*: *addState*, *addSmallStep*
- After the last *small step*: *addState*
- Just before a *big step*: *bigStepStarted*
- Just after a *big step*: *bigStepEnded*

3.5 Generic Trace Metamodel

Our approach relies on the generation of a domain-specific trace metamodel for the considered xDSML. Since the debugger is generic, an interface must also be defined to manipulate traces in a generic way despite their various possible data structures. We defined this structural interface as a *generic trace metamodel* (h in Figure 4) specifying all the information that should be accessible within a domain-specific trace. Thus, it has a similar structure to generated domain-specific trace metamodels, except it contains less classes and properties.

Figure 7 shows the generic trace metamodel interface. To summarize, we have the same base classes (Trace, ExecutionState, TracedObject) as generated domains specific trace metamodels (*e.g.,* the Petri net trace metamodel shown in Figure 6), and generic classes to represent both steps (ExecutionStep) and values (ValueSequence, Value). Primitive types that extend the Value class are not shown due to space limitations. We also use references to elements of the xDSML execution metamodel, operational semantics, and executed model: `appliedRule` to specify which rule was applied, `originalObject` to specify which object of the original model is traced by a TracedObject, and `tracedProperty` to specify the property traced by a ValueSequence. Also note that derived properties are defined to facilitate the navigation among the trace, such as the `nextState` of an ExecutionState. Finally, ExecutionStep objects are ordered either by starting time, or by ending time, hence the derived properties `nextStarting` and `previousStarting` for the starting time then `nextEnding` and `previousEnding` for the ending time.

In order to go back and forth through the execution states and steps, a Trace has a reference `currentStepForward` to the ExecutionStep object that represents the next forward execution step, and a similar reference `currentStepBackward` for the next backward step (*e.g.,* to *backOver* the last that step handled by the debugger). The current state is accessible with `currentState`, which is a property derived from `currentStepForward`. Similarly, the property `currentValue` of ValueSequence is indirectly derived from `currentState`.

To provide this interface, our solution relies on the generation of a one-way transformation from the domain specific trace metamodel to the generic trace metamodel. This provides a read-access to the trace. Regarding the debugging state, it is stored and modified generically in a separate structure of the trace manager.

3.6 State Manager

An omniscient debugger must be able to revisit a previous state by reverting the executed model into the state stored in the execution trace. The operation enabling a debugger to return to a past state is provided by the *state manager* (g in Figure 4), which we specified with a single service:

- ***restoreModelToState***: restore the executed model into a given execution state.

The idea is similar to the well-studied *memento* design pattern, albeit at the model level. The *originator* is the model being executed; the *memento* is an execution state of the trace; and the *caretaker* is both the execution trace and the trace manager.

3.7 Domain-Specific Trace Manager

To implement both the trace constructor and the state manager and to generically expose as much information as stated

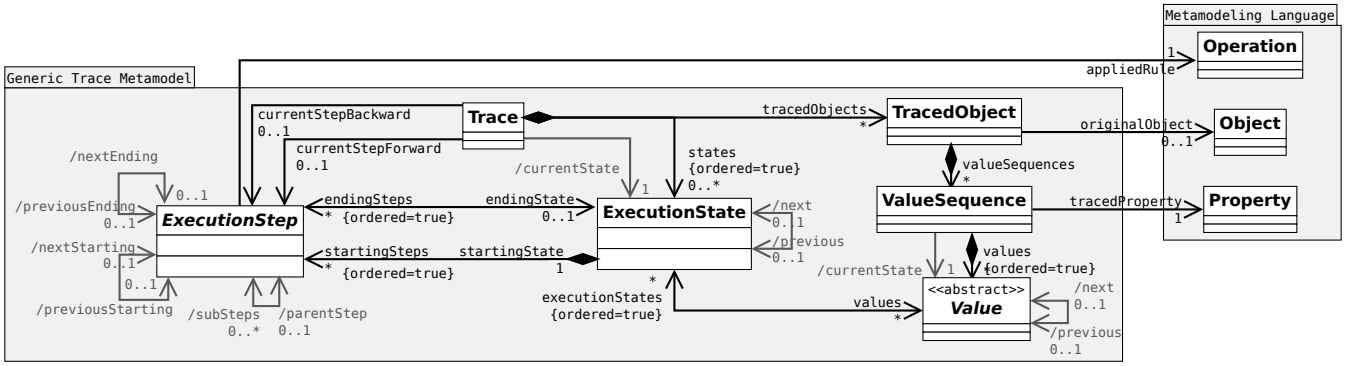

Figure 7: Generic Trace Metamodel Interface

in the generic trace metamodel, our approach relies on the generation of a *domain-specific trace manager* (c in Figure 4). The reason for generating this component is *efficiency* (O#1), because trace manipulations can be tuned for both the considered xDSML and the generated domain-specific trace metamodel (introduced in Section 3.3).

Consequently, the domain-specific trace manager generation is coupled with the domain-specific trace metamodel generation. Since all generated operations manipulate a trace conforming to this metamodel, a set of traceability links obtained from the generation of the domain-specific trace metamodel is provided to the generator. From there, the main steps of the generation are as follows:

1. Since the systematic base structure of the generated trace metamodels is known from the domain-specific trace metamodel generator, *initialize* can be generated;
2. Since the mutable fields of the execution metamodel and the corresponding classes in the trace metamodel are known, *addState* can be generated. An implementation of this service includes looking for changes among mutable fields then creating a state and new values if any change is detected. Likewise, *revertModelToState* can be generated, which can rely on links from the trace to the model to restore values and re-create objects.
3. Since the operational semantics and the corresponding step classes in the trace metamodel are known, step creation can be generated. While *addSmallStep* is straightforward, *bigStepStarted* requires stacking big steps that are in progress, and to unstack them in *bigStepEnded*.
4. Finally, since the systematic shape of generated trace metamodels is known, a generic trace metamodel interface can be provided, as defined in Section 3.5.

3.8 Generic Multidimensional Omniscient Debugger

The last component to define is the *generic multidimensional omniscient debugger* (f in Figure 4) that relies on the execution engine to control the current execution, on the state manager to restore previous states, and on the generic trace metamodel interface to manipulate traces.

Table 1 provides a precise definition of each service required for multidimensional omniscient debugging using the services of the three aforementioned required components. These components are represented by three singletons: *engine* represents the execution engine, *trace* represents the root element of a model conforming to the generic trace metamodel, and *manager* represents the state manager. In the following paragraphs, we explain the definitions of all the services provided by the debugger defined in Table 1.

Jump services Table 1 starts with the definition of the most important omniscient debugging service, *jump*. Jumping consists of going back to a chose state in the execution trace, and is accomplished via the *jumpToState* service. First, it uses the *restoreModelToState* service from the state manager to modify the model, then updates the debugger state represented by `currentForwardStep` and `currentBackwardStep`. Additionally, we need to be able to jump back either right *before* or *after* an execution step, which is provided by the services *jumpBeforeStep* and *jumpAfterStep*.

Other omniscient debugging services Next, we define the remaining omniscient debugging services. The definitions of *backInto*, *backOver* and *backOut* directly rely on *jumps* to reach the correct state. The last service, *playBackwards*, is a loop backwards until either the initial state is reached or the engine is paused.

Standard debugging services The second tiers of Table 1 define the standard debugging services; *i.e.,* breakpoints and forward stepping. *toggleBreakpoint* provides a generic way to define a breakpoint through a predicate, that can be defined on the model state (*e.g.,* watching for a specific instruction to be reached) or on the trace (*e.g.,* verifying a temporal property or watching for a specific step to be applied). It is defined using the *pauseWhen* service that must be provided by the execution engine. The next services are the standard step operations: *stepInto*, *stepOver*, and *stepOut*. There are

Omniscient Debugging Service	**Definition**

jumpToState(*state* : ExecutionState)	**if** *state* $\neq$ *trace*.currentState **then**
	  manager.restoreModelToState(*state*);
	if *state*.*startingSteps* $\neq \varnothing$ **then**
	  trace.currentForwardStep $\leftarrow$ *state*.startingSteps.first();
	else
	  trace.currentForwardStep $\leftarrow$ *null*;
	if *state*.*endingSteps* $\neq \varnothing$ **then**
	  trace.currentBackwardStep $\leftarrow$ *state*.endingSteps.last();
	else
	  trace.currentBackwardStep $\leftarrow$ *null*;

jumpBeforeStep(*step* : ExecutionStep)	jumpToState(*step*.startingState); *trace*.currentForwardStep $\leftarrow$ *step*;

jumpAfterStep(*step* : ExecutionStep)	**if** *step*.endingState $\neq$ *null* **then**
	  jumpToState(*step*.endingState);
	  trace.currentBackwardStep $\leftarrow$ *step*;

backInto()	jumpAfterStep(*trace*.currentBackwardStep.previousEnding)
backOver()	jumpBeforeStep(*trace*.currentBackwardStep)

backOut()	**if** *trace*.currentBackwardStep.parent $\neq$ *null* **then**
	  trace.currentBackwardStep $\leftarrow$ *trace*.currentBackwardStep.parent;
	  backOver();

playBackwards()	**while** *trace*.currentBackwardStep.previousEnding $\neq$ *null* $\wedge$ $\neg$*engine*.isPaused() **do** backInto()
Standard Debugging Service	**Definition**
toggleBreakpoint(p : Predicate)	*engine*.pauseWhen(p)

stepInto()	**if** *trace*.currentForwardStep.nextStarting $=$ *null* **then**
	  steppedInto $\leftarrow$ *trace*.currentForwardStep;
	  engine.pauseWhen(*steppedInto*.nextStarting $\neq$ *null*);
	  engine.resume();
	else
	  jumpBeforeStep(*trace*.currentForwardStep.nextStarting);

stepOver()	**if** *trace*.currentForwardStep.endingState $=$ *null* **then**
	  steppedOver $\leftarrow$ *trace*.currentForwardStep;
	  engine.pauseWhen(*steppedOver*.endingState $\neq$ *null*;
	  engine.resume();
	else
	  jumpAfterStep(*trace*.currentForwardStep);

stepOut()	**if** *trace*.currentForwardStep.parent $\neq$ *null* **then**
	  trace.currentForwardStep $\leftarrow$ *trace*.currentForwardStep.parent;
	  stepOver();

play()	**while** *trace*.currentForwardStep.nextStarting $\neq$ *null* $\wedge$ $\neg$*engine*.isPaused() **do** stepInto() *engine*.resume()
Multidim. Omniscient Debugging Service	**Definition**
jumpToValue(v : Value)	jumpToState(v.executionStates.first())

stepValue(*valueSeq* : ValueSequence)	**if** *valueSeq*.current.nextValue $=$ *null* **then**
	  previousValue $\leftarrow$ *valueSeq*.current;
	  engine.pauseWhen(*previousValue*.nextValue $\neq$ *null*;
	  engine.resume();
	else
	  jumpToValue(*valueSeq*.current.nextValue);

backValue(*valueSeq* : ValueSequence)	jumpToValue(*valueSeq*.current.previousValue)

Table 1: Definition of the Generic Multidimensional Omniscient Debugger

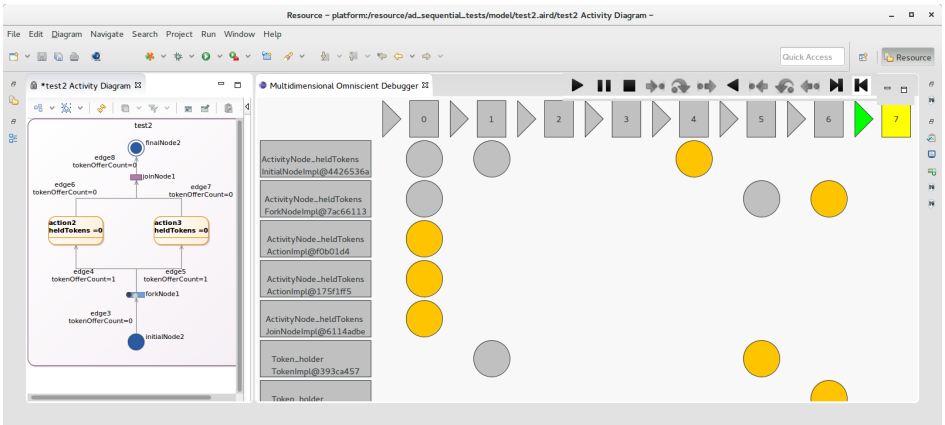

Figure 8: GEMOC Studio with the multidimensional omniscient debugger prototype running an fUML activity.

two cases to consider: (1) When the current step is at the end of the trace, we rely on *pauseWhen* and *resume* to apply the operational semantics up until the correct situation is reached (*e.g.,* waiting for the current big step to be finished with *stepOver*). (2) When the execution state is at a past state (*e.g.,* after a jump backwards), *jump* services are called (even though these step services are not specific to omniscient debugging) while the engine remains paused.

Multidimensional omniscient debugging services The last tiers of Table 1 define the final set of services providing multidimensional omniscient debugging facilities. The goal of these services is to provide the capacity to debug a model by following the sequences of values of specific mutable fields, thereby improving the usability of omniscient debugging for xDSMLs (O#2). Implementing these services is simplified by the structure of the trace metamodel providing access to each of the value sequences. Thus, *jumpToValue* is a use of *jumpToState*, while *stepValue* is very similar to *stepOver* and *backValue* directly uses *jumpValue*.

4. Tooling for Omniscient Debugging

This section presents the language and modeling workbench called GEMOC Studio and explains how we applied a subset of our approach (*i.e.,* the generative part and a debugger with basic operations) to offer a proof-of-concept prototype for a multidimensional omniscient debugger.

4.1 The GEMOC Studio

The GEMOC Studio[3] is an Eclipse package atop the *Eclipse Modeling Framework* (EMF) that includes both a language workbench to design and implement tool-supported xDSMLs as well as a modeling workbench where the xDSMLs are automatically deployed to allow system designers to edit, execute, simulate, and animate their models. The modeling workbench includes an advanced generic execution engine that can used to execute any model conforming to an xDSML defined within the language workbench. An API is available to extend the engine with *addons* through the use of an observer pattern: the engine sends notifications to all its addons to inform them about the execution progress (*e.g.,* a step is starting/ending). Lastly, it supports pausing and resuming the execution between steps.

Our prototype focused on the operational semantics implementation language *Kermeta* [9] that relies on aspects to modularly implement the operational semantics and weave them into the provided metamodel.

4.2 Omniscient Debugging in the GEMOC Studio

We propose a prototype that implements our approach to support multidimensional omniscient debugging in the GEMOC Studio. This implementation required the creation and integration of the following components.

Trace addon generator The generative part of our approach takes the form of a *trace addon generator*, that takes as input an xDSML composed of an Ecore abstract syntax and Kermeta aspects. Note that Kermeta aspects function both as an execution metamodel and operational semantics, because aspect weaving allows the definition of new mutable properties in classes and operations used as transformation rules. The generator produces a GEMOC engine addon with a domain-specific trace metamodel and a domain-specific trace manager. Using the addon mechanism, the manager reacts to the execution steps of the engine to construct the trace accordingly. It also provides an interface to revert the state of the model (*i.e.,* a state manager), and an interface to query the trace (*i.e.,* a generic trace metamodel interface).

Generic debugger logic The generic part of our approach includes multidimensional omniscient debugging services within the GEMOC studio. Our prototype provides *toggleBreakpoint* with only one kind of predicate (*i.e.,* a model element is targeted by a step), *stepInto*, *jumpToState*, *jumpToValue*, and visualization (see next paragraph).

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

Graphical interface The graphical user interface of the debugger includes a prototypical graphical widget that shows both the execution trace and the value sequences of all mutable fields. Figure 8 shows the GEMOC Studio with a model animator on the left and the omniscient debugging widget on the right. Double clicking on a model element triggers a *toggleBreakpoint*, stating that the execution must pause when this element is targeted by a step. In the right widget showing the trace, the first row of numbered squares represents all the execution states. Each subsequent row represents the value sequence of a specific mutable field. Yellow circles indicate the current execution state and the current values of the mutable fields that changed with the most recent *small step*, and orange circles indicate current values originating in past states (*i.e.,* value generated as part of a previous *small step*). Double-clicking on a state (numbered squares) triggers a *jumpToState* to the corresponding execution state. Doing similarly on one of values (circles) triggers a *jumpToValue* to the corresponding value. Since the GEMOC Studio provides animation of the executed model, the left view showing the model is updated at each action.

Additional information about our prototype can be found on the companion web page of this paper:
`http://gemoc.org/sle15-omniscientdebugging/`.

5. Evaluation

In this section, we first present the design and results of an empirical study providing an initial evaluation of the efficiency of our approach. Then, we discuss the benefits of multidimensional omniscient debugging.

5.1 Efficiency of the Approach

To evaluate the efficiency of our approach (O#1), we considered the following research questions:

RQ#1: Is our approach more efficient in memory as compared to a clone-based omniscient debugger?

RQ#2: Is our approach more efficient in time for omniscient debugging services as compared to a weak omniscient debugger and to a clone-based omniscient debugger?

Thus, our evaluation of efficiency is the comparison of three omniscient debuggers as presented in Section 2.2 and in Figure 3. First, *WeakDebugger* is a weak generic omniscient debugger. Such a debugger is expected to be efficient in memory, because there is no trace to store; and inefficient in time, because the execution engine must be restarted at each jump backward. Second, *CloneBasedDebugger* is a clone-based generic omniscient debugger, that constructs a generic trace using *deep cloning* (*i.e.,* the complete model is copied at each step) and implements *jumps* using the model differencing library EMF Compare [3]. Because this debugger relies on an execution trace, it is expected to be less efficient in memory and more efficient in time than *WeakDebugger*. Finally, *MultiDimDebugger* is the multidimensional omniscient debugger obtained with our approach. All three debuggers were implemented in the GEMOC Studio.

We applied our approach to a subset of a real-world xDSML, namely fUML [15]. The considered subset contains the *Activity Diagram* portion of the language. In summary, a model conforming to this xDSML 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 addition, *variables* can be defined in activities and modified with actions. The xDSML was implemented with GEMOC Studio using Ecore for the abstract syntax and Kermeta for both the execution metamodel and the operational semantics.

We used models taken from the case study of Maoz et al. [12] for evaluating their semantic differencing approach[4]. This choice was made to help establish a benchmark and facilitate comparison with future work, and because the models were drawn from industrial sources. The dataset we obtain contains 40 models whose sizes range from 36 to 51 objects. We plan to integrate larger models to the dataset for a future study, but are confident in the current ones to provide initial meaningful comparison material.

Data Collection and Analysis To compare efficiency in *memory*, instead of observing the memory usage of the complete environment (*e.g.,* execution engine and loaded model), we measured the memory used only by the debugger. More precisely, for each of the considered models, we collected the amount of memory required to store the execution trace at the end of its execution by making precise memory measurements using heap dumps and Eclipse MAT[5].

To compare efficiency in *time*, we focused on the main operation used by all omniscient debugging services: *jumpToState*. More precisely, for each of the considered models, we measured the average amount of time required to perform a *jumpToState* by jumping to each previously visited state once and in a random order. Measures were done using Java's operation `System.nanoTime`.

Data was collected in a reproducible way through a programmatic use of GEMOC Studio's engine (see the companion webpage). Each result is an average value computed from five identical measurements made using an Intel i7-3720QM CPU with 8GB of RAM.

RQ #1: Efficiency in memory Figure 10 shows the results obtained regarding the memory required to store an execution trace. The x-axis shows the number of elements in the trace, while the y-axis shows the amount of memory used in kB. First, *WeakDebugger* does not use memory, because it does not store a trace. Second, we observe that our approach is always more efficient in terms of memory usage than the *CloneBasedDebugger* debugger with 3.0 times im-

[4] Models available at `http://www.se-rwth.de/materials/semdiff/`

[5] `https://www.eclipse.org/mat/`

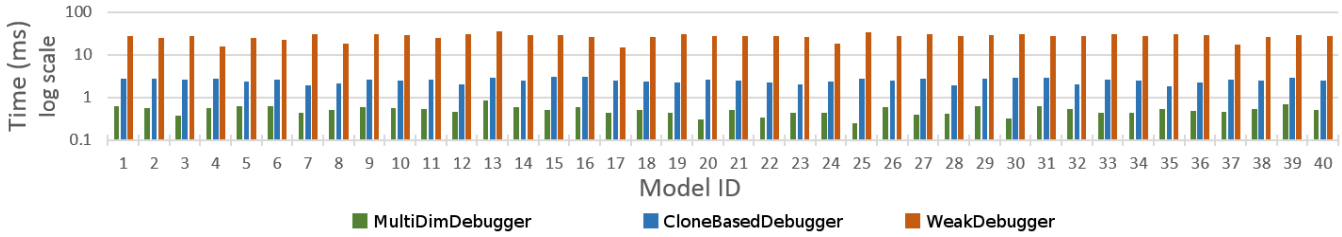

Figure 9: Time required to perform a *jumpToState*

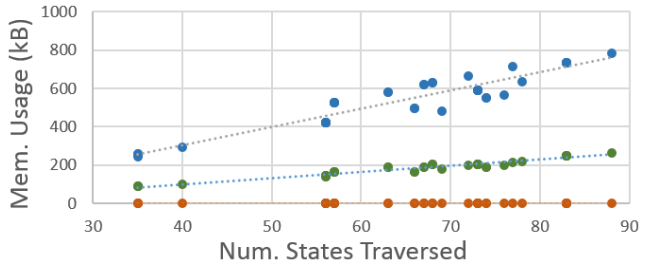

Figure 10: Memory used by the execution trace

provement on average. We hypothesize this is due to the domain-specific traces obtained with our approach that are designed to only contain the evolution of the mutable fields of the model with minimal redundancy, whereas cloning implies significant redundancy. In addition, we note that our approach has a gentler slope than *CloneBasedDebugger*, which suggests better scalability with large traces. To summarize and answer RQ #1, we observe that our approach is more efficient in memory than a clone-based approach.

RQ #2: Efficiency in time Figure 9 presents the results obtained regarding the average amount of time required to perform a *jumpToState*. The x-axis shows the identifier of the executed model, while the y-axis shows the amount of time in *ms*. First, we observe that trace-based debuggers are always better than *WeakDebugger*, with in particular *MultiDimDebugger* being 54.1 times faster than *WeakDebugger*. This is explained by the time required to reset the execution engine. Second, we observe that *MultiDimDebugger* is more efficient than *CloneBasedDebugger* with 5.03 times improvement on average. We hypothesize this is due to the generated trace manager, which contains code specific and tuned to both the xDSML and the generated domain-specific trace metamodel. To summarize and answer RQ #2, we observe that our approach is more efficient in time than both a traceless approach and a clone-based approach.

5.2 Benefits of Multidimensional Facilities

To ensure the usability of omniscient debugging (O#2), our approach provides *multidimensional omniscient debugging*; *i.e.,* facilities to navigate among the values of the mutable fields of the executed model. Since the structure of the mutable fields can be arbitrarily complex, our idea is to *project* the trace on a chosen specific dimension, thereby making possible the visualization and the exploration of the trace according to this axis only. In essence, we believe that providing explicit visualization of the dimensions of a trace (see Figure 8) and means to traverse such trace according to specific dimensions (*e.g., stepValue*), has a significant positive impact on usability (O#2). The perspectives of this work include user experiments to empirically assess the expected benefits of multidimensional facilities.

6. Related Work

In this section, we overview the existing work in the MDE literature on omniscient debugging, followed by a discussion of alternative execution trace structures.

6.1 Omniscient Debugging in MDE

In our previous work [5], we explored applying omniscient debugging to model transformations within the context of AToMPM, a multi-paradigm modeling tool. AToMPM provides two basic transformation languages, but we are concerned with xDSMLs. Furthermore, we provide a similar, but more advanced trace structure in this work. We are not aware of any other literature that applied omniscient debugging in the context of MDE.

6.2 Execution Traces and Trace Exploration in MDE

Meyers et al. introduced the ProMoBox framework [14], which generates a set of metamodels from an annotated xDSML, including a property metamodel and a trace metamodel. Their trace metamodel generation has multiple differences with our approach. Among others, they consider an abstract syntax whose properties are annotated either as runtime or event to identify mutable elements and event-related elements, while we consider the abstract syntax and the execution metamodel to be separated. Indeed, such separation makes possible a better separation of concerns and interchangeability of semantics. Also, they use generalization to extend a base trace metamodel, while we generate new classes to avoid having to rely on introspection and casting when manipulating traces. In addition, they do not provide alternative ways to explore a trace, while we provide various navigation paths. Gogolla et al. [8] generate filmstrip models from UML class diagrams. Such filmstrip models match what we call domain-specific trace metamodels, and also

provide some navigation paths among objects states. However, they do not tackle redundancy since object states are always recreated at each model change, and they do not consider only storing only mutable fields values.

Maoz and Harel [11] and Hegedüs et al. [7] present trace exploration tools which contain similar facilities to an omniscient debugger. However, these techniques are defined for post-mortem analysis rather than use during live sessions, whereas our technique supports live debugging sessions. Additionally, neither work supports providing domain-specific execution traces for xDSMLs.

7. Conclusion and Future Work

Omniscient debugging is a promising dynamic V&V approach for xDSMLS that enables free traversal of the execution of a system. While most GPLs already have efficient debuggers, bringing omnicient debugging to any xDSML is a tedious and error-prone task. A solution is to define a purely generic debugger, but this requires managing both *efficiency* and *usability* issues that emerge. The approach we presented relies on generated domain-specific trace management facilities for improved efficiency and provides multidimensional omniscient debugging facilities for improved usability. The debugger relies on an execution engine to control the execution and a generated domain-specific trace manager to provide omniscient services. The states reached during an execution are stored in an trace conforming to a generated domain-specific trace metamodel. We provide a prototype within GEMOC Studio, a language and modeling workbench, and an evaluation performed using the fUML language. We observed an improvement regarding both the memory consumption and the time to perform a *jump*, when compared to two generic omniscient debugger variants.

The direct perspectives of this work include taking advantage of the underlying domain-specific trace metamodel to supplement the debugger with domain-specific tooling, and user experiments to empirically assess the expected benefits of multidimensional facilities.

Acknowledgment

This work is partially supported by the ANR INS Project GEMOC (ANR-12-INSE-0011).

References

[1] E. Bousse, B. Combemale, and B. Baudry. Scalable Armies of Model Clones through Data Sharing. In *17th Int. Conf. on Model Driven Engineering Languages and Systems*, volume 8767 of *LNCS*, pages 86–301. Springer, 2014.

[2] E. Bousse, T. Mayerhofer, B. Combemale, and B. Baudry. A Generative Approach to Define Rich Domain-Specific Trace Metamodels. In *ECMFA*, LNCS. Springer, 2015.

[3] C. Brun and A. Pierantonio. Model differences in the eclipse modeling framework. *UPGRADE, The European Journal for the Informatics Professional*, 9(2):29–34, 2008.

[4] B. Combemale, X. Crégut, and M. Pantel. A Design Pattern to Build Executable DSMLs and Associated V&V Tools. In *19th Asia-Pacific Software Engineering Conference*, pages 282–287. IEEE, 2012.

[5] J. Corley, B. Eddy, and J. Gray. Exploring omniscient debugging for model transformations. In *14th Workshop on Domain-Specific Modeling*, 2014.

[6] C. Guy, B. Combemale, and S. Derrien. On Model Subtyping. In *8th Europ. Conf. on Modeling Foundations and Applications*, volume 7349 of *LNCS*, pages 400–415. Springer, 2012.

[7] Á. Hegedüs, I. Ráth, and D. Varró. Replaying execution trace models for dynamic modeling languages. *Periodica Polytechnica-Electrical Engineering*, 56:71–82, 2012.

[8] F. Hilken, L. Hamann, and M. Gogolla. Transformation of uml and ocl models into filmstrip models. In *Theory and Practice of Model Transformations*, volume 8568 of *Lecture Notes in Computer Science*, pages 170–185. Springer International Publishing, 2014.

[9] J.-M. Jézéquel, B. Combemale, O. Barais, M. Monperrus, and F. Fouquet. Mashup of metalanguages and its implementation in the Kermeta language workbench. *Software & Systems Modeling*, pages 1–16, 2013.

[10] B. Lewis. Debugging backwards in time. *CoRR*, cs.SE/0310016, 2003. URL `http://arxiv.org/abs/cs.SE/0310016`.

[11] S. Maoz and D. Harel. On tracing reactive systems. *Software & Systems Modeling*, 10(4):447–468, 2011.

[12] S. Maoz, J. O. Ringert, and B. Rumpe. ADDiff: Semantic Differencing for Activity Diagrams. In *19th ACM SIGSOFT Symposium and 13th Europ. Conf. on Foundations of Software Engineering*, pages 179–189. ACM, 2011.

[13] T. Mayerhofer, P. Langer, M. Wimmer, and G. Kappel. xMOF: Executable DSMLs based on fUML. In *6th Int. Conf. on Software Language Engineering*, volume 8225 of *LNCS*, pages 56–75. Springer, 2013.

[14] B. Meyers, R. Deshayes, L. Lucio, E. Syriani, H. Vangheluwe, and M. Wimmer. Promobox: A framework for generating domain-specific property languages. In *Software Language Engineering*, volume 8706 of *Lecture Notes in Computer Science*, pages 1–20. Springer International Publishing, 2014.

[15] Object Management Group. Semantics of a Foundational Subset for Executable UML Models (fUML), V 1.1, August 2013. `http://www.omg.org/spec/FUML/1.1`.

[16] G. Pothier and E. Tanter. Back to the future: Omniscient debugging. *IEEE Software*, 26(6):78–85, nov 2009.

[17] J. Tatibouët, A. Cuccuru, S. Gérard, and F. Terrier. Formalizing Execution Semantics of UML Profiles with fUML Models. In *17th International Conference on Model Driven Engineering Languages and Systems*, volume 8767 of *LNCS*, pages 133–148. Springer, 2014.