Undoing Operational Steps of Domain-Specific Modeling Languages

Tim Hartmann   Daniel A. Sadilek

Humboldt-Universität zu Berlin
Unter den Linden 6
10099 Berlin, Germany
{hartmann|sadilek}@informatik.hu-berlin.de

Abstract

In this paper, we deal with the animated execution of domain-specific models (DSMs) that are expressed in domain-specific modeling languages (DSMLs) whose semantics are described in an operational fashion. We propose to support stepping back in the execution history of such DSMs. We argue that this eases debugging of the DSM itself and the DSML's operational semantics. As an example, we show animated model execution of Petri nets and identify the requirement to step back in their execution history. To accomplish this, we present an approach in which we apply principles for undoing user input in model editors to the animated execution of DSMs. Finally, we present an Eclipse-based implementation of our approach, which is an extension of the tool EPROVIDE.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors

General Terms Design, Languages

Keywords Domain-specific modeling languages, operational semantics, debugging, stepping back, Eclipse

1. Introduction

1.1 Domain-Specific Modeling Languages

In contrast to general-purpose languages like the Unified Modeling Language (UML), domain-specific modeling languages (DSMLs) are customized to a particular application domain. With DSMLs, domain experts can create models using a vocabulary they are used to. These models can be directly used in a software engineering process. For example, they can be interpreted or code can be generated from them.

When developing DSMLs, the requirements are not always clear from the start, and more than one development iteration is necessary. In such cases, prototyping of DSMLs is needed. In this paper, we deal with executable DSMLs. A prototyping process for such DSMLs requires that domain experts can create and execute example models expressed in the DSML.

1.2 Domain-Specific Model Execution using Operational Semantics

When a domain-specific model (DSM) is to be executed, the DSML it is expressed in must be given execution semantics. This can be done with a *translational* or an *interpretational* approach.

Using the translational approach, model execution is prepared by *translating* a model into an executable form (e.g., by code generation). But this shifts down the level of abstraction (e.g., into the realm of programming languages). It is, therefore, inappropriate for prototyping, especially when domain experts should contribute to the prototyping of the DSML.

Using the interpretational approach, models are executed by an interpreter. This interpreter can be hand-crafted or it can be based on an executable description of the DSML's operational semantics. The operational semantics of a language describes the meaning of a language instance as a sequence of execution steps (Plotkin 1981). Generally, a transition system $\langle \Gamma, \rightarrow \rangle$ forms the mathematical foundation, where Γ is a set of *configurations* and $\rightarrow \subseteq \Gamma \times \Gamma$ is a *transition relation*.

Model-Driven Approach to Operational Semantics. Wachsmuth (2008) applies the idea of structural operational semantics to model-driven language engineering. The configurations in Γ are represented as models, which are called *configuration models*. Hence, the space of all possible configurations is defined with a metamodel, which is called *configuration metamodel*; and the transition relation $\rightarrow$ is defined with a model-to-model transformation, which is called *transition transformation*. As the transition transformation is an executable specification of the transition relation, this kind of description can be directly used to interpret DSMs.

By describing operational semantics in terms of the language structure, the domain level of abstraction is kept so that domain experts can understand them. Thus, they can assess the operational semantics of a DSML by observing the execution of example DSMs. This allows to integrate them tightly into the prototyping process of DSMLs.

1.3 Debugging Executable Models

During execution, domain experts might observe erroneous behavior of models. The cause can lie inside the example DSM or in the prototypical semantics description of the DSML. To identify and correct such errors, domain experts and language engineers need debugging support.

Debugging means to control the execution process and to access and possibly modify the runtime state. Common features of debuggers are stepwise execution of programs and setting of breakpoints or, more generally, suspending and resuming the execution at designated execution states. Once the execution is halted, variable values can be inspected and modified. This type of debugging support for DSMLs can easily be achieved when using model-based operational semantics (Sadilek and Wachsmuth 2008b).

1.4 Undoing Model Execution

Based on these foundations for model execution and debugging, in this paper, we want to go further in providing debugging support.

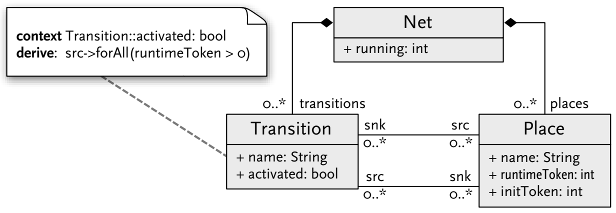


Figure 1. A Petri net metamodel with attributes for storing runtime states.

During DSML prototyping, we encountered the following problem. Assume, for example, a complex model and a configuration that can be reached only by a long series of execution steps that require user input (e.g., in the form of input data or by controlling the behavior of non-deterministic language concepts). Now assume that a domain expert finds an error in this configuration. The execution can be halted and the operational semantics can be changed by a language engineer. Afterwards, the execution has to be restarted and the whole series of execution steps, including every user input, has to be repeated.

To save this work, we propose to allow users to undo execution steps performed by the transition transformation. Thus, the configuration before the error occurred can be restored, and the execution can be resumed. The user can instantly review the effects of the changed operational semantics.

1.5 Structure of the Paper

The rest of this paper is structured as follows. In the following section, we give an example for animated model execution and identify the requirement to undo operational steps of DSMLs. In Sec. 3, we present our approach for undoing operational semantics, and in Sec. 4, we give a brief overview of our implementation. We discuss existing work about DSM execution in Sec. 5 and conclude with our contribution and future work in Sec. 6.

2. Animated Execution and Debugging of Petri Nets

In this section, we give an example of animated model execution and identify the requirement to undo operational steps of DSMLs. We use Petri nets as an example DSML and specify its operational semantics in Java. As execution engine, we use the tool EProvide, which our implementation will eventually be based on.

2.1 Animated Execution and Debugging with EProvide

In (Sadilek and Wachsmuth 2008b), operational semantics are combined with existing editor generation technology in order to support rapid prototyping of visual interpreters and debuggers. The runtime state of a DSM, its configuration, is represented as a model (as we described in Sec. 1.2). A graphical editor that is specific to the configuration metamodel is used to display and modify configurations. By visualizing successive configurations, the editor animates model execution. This approach is implemented in the Eclipse-based tool EProvide, which allows the use of different description languages (Java, QVT Relations, Scheme, Prolog, and Abstract State Machines) to describe operational semantics of metamodel-based languages (Sadilek and Wachsmuth 2008a).

2.2 Describing the Operational Semantics of Petri Nets

As an example, we describe operational semantics of Petri nets. Figure 1 shows a metamodel for Petri nets and their runtime configurations.[1] A Petri net consists of an arbitrary number of places and transitions. Places and transitions can have names and places are marked with a number of tokens. We distinguish the initial marking (`initToken`) and the runtime marking (`runtimeToken`) of places. The initial marking of a place is a "static" attribute that does not change when the model gets executed. The runtime marking is a "dynamic" attribute that encodes the runtime configuration and that changes during model execution.[2] When a model is reset to its initial state, the runtime marking of each place is set to its initial marking. Whether `initToken` or `runtimeToken` is displayed by the editor depends on the value of the Net's (`running`) attribute. Transitions have input (`src`) and output places (`snk`). Depending on its input places, a transition might be activated or not (`activated`).

```
public class PetriSemantics implements ISemanticsProvider {
    public void step(Resource model) {
        Net net = (Net) model.getContents().get(0);
        net.setRunning(true);
        fireTransition(net);
    }

    protected void fireTransition(Net net) {
        EList<Transition> ats = findActivatedTransitions(net);
        if (!ats.isEmpty()) {
            Transition t = choose(ats);

            Place p = choose(t.getSrc());
            consume(p);

            p = choose(t.getSnk());
            produce(p);
        }
    }

    protected T choose(List<T> list) {
        // Returns a randomly chosen member of list.
    }
// ...
}
```

Figure 2. Operational semantics for Petri nets described with Java.

Figure 2 shows a class implementing (erroneous) operational semantics for Petri nets in Java. The method `step()` implements the transition transformation. It is called repeatedly by EProvide to perform operational steps. The API used in `step()` is generated by the Eclipse Modeling Framework (EMF), which we use in our implementation (Sec. 4).

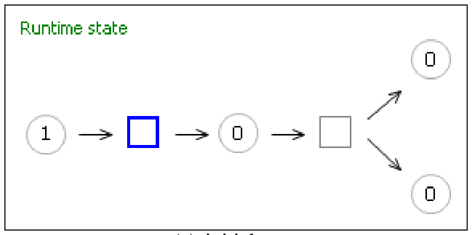
2.3 Animated Execution of a Petri Net

Figure 3 shows what animated execution with EProvide looks like. We use the standard notation for Petri nets: places are represented by circles, transitions by squares; the number of tokens on a place is shown inside the circle. Figure 3(a) shows the initial state of an example model. In Fig. 3(b), the first transition has fired and the token is now on the middle place. Fig. 3(c) shows the state after the second transition fired.

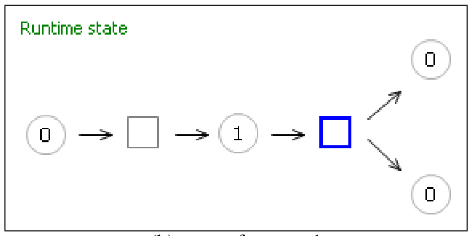
The final state in Fig. 3(c) contains an error that is easily identified by a domain expert. A token was produced at only one of the two output places, but a token should have been produced on all output places. Such errors can be caused by misunderstandings be-

[1] In general, one could use a separate metamodel for the runtime configurations. But to keep this example simple, we use an integrated metamodel.

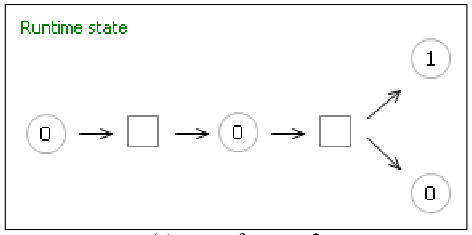
[2] In more complex cases, it is often not sufficient to add attributes to existing metamodel classes; instead, additional classes are necessary to model runtime configurations. This is the case, for example, for reentrant functions, which require that values of local variables are stored for each function call.



(a) initial state



(b) state after step 1



(c) state after step 2

Figure 3. Execution sequence of a Petri net model using erroneous operational semantics.

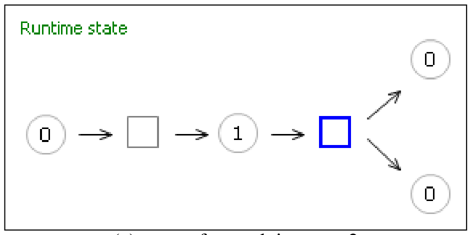
```
// ...
protected void fireTransition(Net net) {
    EList<Transition> ats = findActivatedTransitions(net);
    if (!ats.isEmpty()) {
        Transition t = choose(ats);

        for (Place p : t.getSrc())
            consume(p);

        for (Place p : t.getSnk())
            produce(p);
    }
}
// ...
```

Figure 4. Corrected operational semantics for Petri nets.

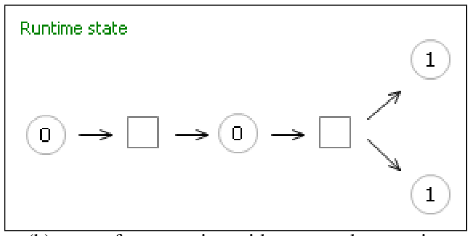
tween domain experts and language engineers. Here, the language engineer has assumed that exactly one token is transfered each time a transition fires. After finding the error by animated execution, the domain expert can explain to the language engineer that the correct behavior for a transition would be to consume one token from all its input places and produce one on all its output places. Now, the language engineer can correct the operational semantics. The corrected code is shown in Fig. 4.

2.4 Requirement to Undo Operational Steps

Now, the capability to undo model execution steps would come in handy. Via undo, the state before the last (incorrect) step could be reached, shown in Fig. 5(a), which is identical to the state shown in Fig. 3(b). From this point, the execution could be resumed, using



(a) state after undoing step 2



(b) state after resuming with corrected semantics

Figure 5. Execution sequence of the Petri net model after correcting the operational semantics.

the corrected operational semantics, which would lead to the state shown in Fig. 5(b).

To control such an undo feature and animated model execution in general, the user needs a graphical user interface. IDEs already provide debugging support for standard programming languages, including a debug user interface for stepwise execution. The undo feature for DSMs should integrate seamlessly into such an existing infrastructure.

3. An Approach for Undoing Operational Steps

We want to allow undoing of DSM execution steps. In each of the steps, the configuration is changed. To undo an execution step, the corresponding model changes have to be undone. A similar problem is already solved for model editors. Here, the model changes are not performed by an operational semantics but by a user. To make model changes undoable, they get encapsulated in undoable units of work that are managed on an undo stack. Our approach is to use similar techniques with the goal to reuse as much of an existing model editor implementation as possible.

3.1 Logging Model Changes

To be able to undo an execution step, it is necessary to know all changes that were performed by the transition transformation. Therefore, the execution engine needs to be notified of all model changes, and the changes have to be stored to create an undoable execution history. This notification problem does not occur with a model editor because the editor is implemented to encapsulate all model changes in undoable units of work. In contrast, with model-based operational semantics (Sec. 1.2), the semantics description does not contain such an encapsulation.

How notification of changes can be achieved is implementation dependent. Two possible ways are: (1) the operational semantics is not applied to the model directly but to a proxy that logs all model changes; (2) the observer pattern is used to receive and log notifications of model changes.

3.2 Integrating Model Execution and Editing

Another issue, which is related to the visualization approach introduced in Sec. 2.1, is the synchronization of editor and execution engine. The execution engine works on an instance of the configuration metamodel, transforming this instance with every execution step. The editor needs to work with the same configuration to visualize the current runtime state. Editor and execution engine access the configuration concurrently. This can cause inconsistent states. For example, the user could modify a value that the engine just read and still relies on during an execution step. To avoid such interferences, it is necessary to synchronize write access of editor and engine.

3.3 Composing Execution Steps

There is an important difference between changes made in an editor and those made by the execution engine. Generally, the user manipulates one model element at a time with an editor. But a single execution step will, in most cases, include a series of elementary changes of a configuration. Even in the simple Petri net example (Sec. 2), one step, i.e. firing a transition, comprises consuming one token from all input places and producing a token on all output places. All those changes have to be stored together and they must be associated with the execution step they belong to. Furthermore, either the complete step has to be performed or none of the elementary changes. Regarding the synchronization problem from the previous section, this means that the mutual exclusion of editor and execution engine has to span the whole execution step, not just elementary changes. Those requirements, namely atomicity and isolation, are met by transactions. Therefore, all changes belonging to one step must be wrapped into a single unit of work that is executed inside the scope of a transaction.

3.4 Managing Execution Steps

As we want to be able to undo more than one execution step, we have to manage the execution history on a stack. This is straightforward when taken for granted that one step is a single unit of work.

However, a consistency problem arises because the user can modify configurations with the editor. Assume a set M of possible configurations, a set C of possible model changes, and a transition transformation $\rightsquigarrow$: $M \rightarrow M \times C$. Let $m_1 \in M$ be an arbitrary configuration. Applying $\rightsquigarrow$ to m_1 results in a new configuration $m_2 \in M$ with the change c: $m_1 \overset{c}{\rightsquigarrow} m_2$. We can undo c and get back from m_2 to m_1 by reversing the transformation: $m_2 \overset{c^{-1}}{\rightsquigarrow} m_1$. If, however, the user changed m_2 to m_3 using the model editor, we cannot safely undo c in the current state m_3 anymore. That is because the user made the change $m_2 \overset{d}{\rightsquigarrow} m_3$ and generally, $c \neq d$. Therefore, undoing c in state m_3 could lead to an inconsistent state, which may even violate constraints of the metamodel.

To solve this problem, all model changes, regardless of their origin, have to be kept and undone in the order in which they were applied. This can be achieved by using the same stack for both editor and execution engine.

4. Implementation

This section sketches the implementation of our approach, which is based on the tool EPROVIDE[3], introduced in Sec. 2.1.

4.1 Implementation Basis

Important for our implementation are especially the Eclipse Modeling Framework (EMF), the Graphical Modeling Framework (GMF) and Eclipse's debug infrastructure. EMF uses Ecore as meta-metamodel, which is virtually an implementation of OMG's Essential MOF. EMF can generate a Java API for an Ecore metamodel. This API can be used to access metamodel instances programmatically. The Java-based semantics for Petri nets from Sec. 2.2 uses such an API. EMF uses `Resources`, which are an abstraction from physical files, to handle metamodel instances. Resources can reference, or be related to, other resources. They are managed in `ResourceSets`, which in turn are managed by `EditingDomains`. GMF allows graphical editors to be generated for EMF-based models from declarative descriptions. In EPROVIDE, such editors are used to visualize configurations (Sec. 2.1). Eclipse's debug infrastructure provides a language independent debug model and a user interface for common debug functionality.

4.2 Logging Model Changes

In Sec. 3.1, we presented two ways to get notified of changes applied to configurations. The second of them, the observer pattern, is already implemented in EMF: observed model elements are called notifiers and there is already a recording observer called `ChangeRecorder`. Consequently, we chose the observer pattern to get notified of configuration changes. EMF also provides a data structure to store changes, additions, and deletions of model elements: the `ChangeDescription`. All changes belonging to one execution step can be stored in one `ChangeDescription` by using a `ChangeRecorder`. The changes in a `ChangeDescription` can be undone by applying them in reverse to a configuration. Thus, execution steps can be undone.

4.3 Integrating Model Execution and Editing

Following our approach for undoing operational semantics, the EPROVIDE execution engine and the GMF generated editor have to share access to configurations (as was explained in Sec. 3.2). Configurations are EMF models, which are handled in the form of EMF `Resources`. EMF `Resources` can be shared, e.g., between different editors (Wegert and Shatalin 2008). This is achieved by using a shared `EditingDomain`. Following this principle, we use the same `EditingDomain` for editor and execution engine. This grants some additional benefits as we will see in Sec. 4.5.

To solve the problem of synchronizing editor and execution engine, we use EMF's transaction framework EMFT. EMFT is an extension of EMF that provides transactional access to EMF resources. GMF generated editors and EPROVIDE both use EMFT. Thus, synchronization is achieved.

4.4 Composing Execution Steps

As stated in Sec. 3.3, the elementary changes performed during an execution step have to be wrapped into a single unit of work. For this, we use EMFT's `RecordingCommand`, which internally uses a `ChangeRecorder`. For each execution step, a `RecordingCommand` is used that creates a transaction context for the configuration changes in that step. The `RecordingCommand`'s `ChangeRecorder` tracks all configuration changes of the step and stores them in a `ChangeDescription` so that the `RecordingCommand` can be undone. Thus, all model changes of one execution step can be undone as a whole.

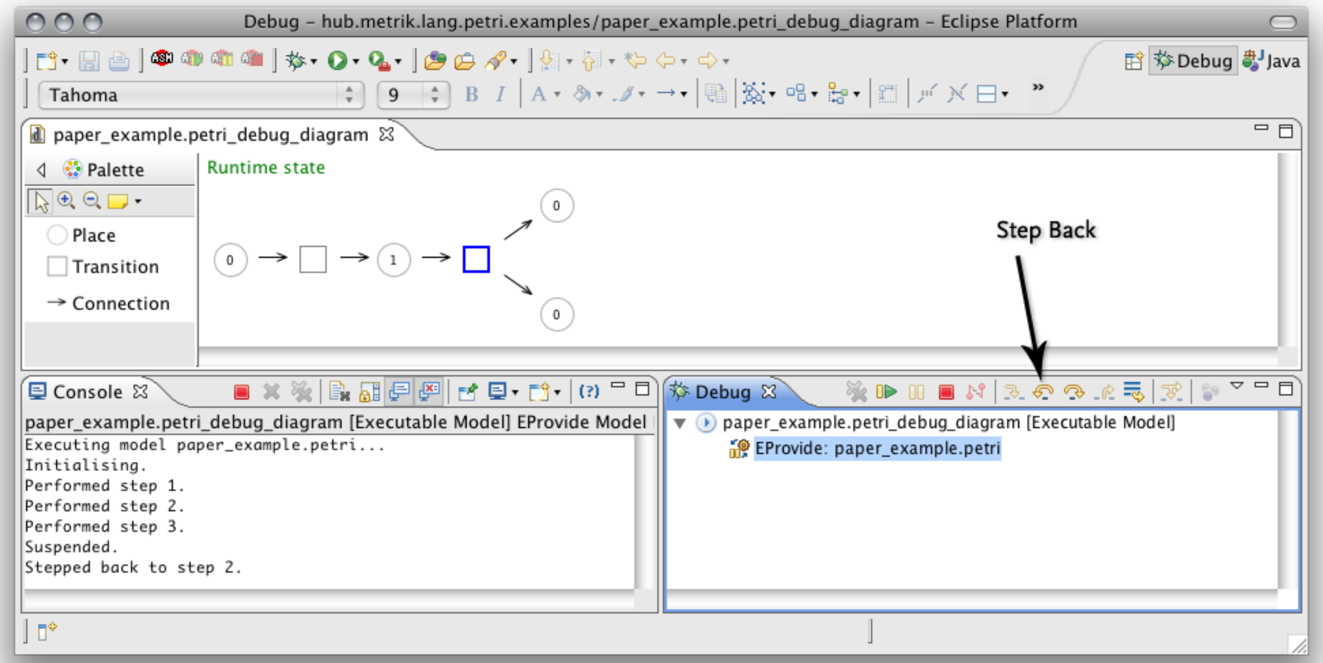
4.5 Managing Execution Steps

The shared `EditingDomain`, introduced in Sec. 4.3, also yields a shared `CommandStack`. The `RecordingCommands` from the execution engine and the commands from the editor are now executed and managed via this `CommandStack`. By this means, all commands are stored in the sequential order that was required in Sec. 3.4.

4.6 Integrating the User Interface

Eclipse's debug infrastructure can be extended with new debug functionality. We added support for stepwise DSM execution and

[3] EPROVIDE is open source software and available for download at `http://eprovide.sf.net`


Debug – hub.metrik.lang.petri.examples/paper_example.petri_debug_diagram – Eclipse Platform
Debug
Java
Tahoma
9
paper_example.petri_debug_diagram
Palette
Place
Transition
Connection
Runtime state
0
1
0
0
Step Back
Console
Debug
paper_example.petri_debug_diagram [Executable Model] EProvide Model
paper_example.petri_debug_diagram [Executable Model]
EProvide: paper_example.petri

```
Executing model paper_example.petri...
Initialising.
Performed step 1.
Performed step 2.
Performed step 3.
Suspended.
Stepped back to step 2.
```

Figure 6. Model debugging with EPROVIDE. The debug toolbar is extended with a button for stepping back in the execution history.

for stepping back in the resulting execution history. The Eclipse debug user interface already provides buttons to suspend, resume, terminate, and stepwise execute a program. We reuse them for model execution. For stepping back in the execution history, we added a "step back" button. Fig. 6 shows a screenshot of EPROVIDE with the extended debug interface.

5. Related Work

Runtime States as Models. Several approaches are based on metamodeling runtime states. All of them provide their own description languages, sometimes a variation or extension of existing modeling languages (Scheidgen and Fischer 2007), OCL (Muller et al. 2005; Clark et al. 2004) or Abstract State Machines (ASMs) (Di Ruscio et al. 2006). None of these approaches has support for stepping back in the execution history. Adding this feature would require solving the following problem: With EPROVIDE, the execution of the transition transformation is stateless and the complete runtime state is encoded in a model. The approaches above, in contrast, use description languages whose execution is stateful, i.e., between execution steps, a state is held inside the interpreter of the description language. For example, with the OCL-based, imperative description language of Muller et al. (2005), the interpreter state is a call stack with the local variables of procedure calls. Therefore, undoing an execution step in these approaches would require not only to undo the changes of the configuration but also to undo the changes of the description language's interpreter state.

Runtime States as Attributed Graphs. Graph transformations are a well-known technology to describe the operational semantics of visual languages (Engels et al. 2000; Ermel et al. 2005). In AToM3, de Lara and Vangheluwe (2004) use graph grammars to define the operational semantics of a visual modeling language. The Moses tool suite (Robert Esser 2001) provides a generic architecture to animate visual models, with execution semantics of models given as ASMs. As they are, these tools do not support stepping back in the execution history. But our approach to reuse model editing techniques could be applied to them, as well.

Hand-Crafted Interpreters. Interpreters for DSMLs can also be implemented manually. Ptolemy (Brooks et al. 2007) allows animated execution of hierarchically composed domain-specific models with different execution semantics. Adding a new DSML to Ptolemy requires that its syntax and its semantics are coded manually in Java. GME (Lédeczi et al. 2001) provides visualization of model interpretation and support for creating a DSML editor without manual coding. But as with Ptolemy, the interpreter semantics has to be implemented manually in Java or C++. The runtime states in these frameworks are encoded in data structures of the language used to implement the interpreter (Java, C++). In contrast to EMF, there is no editing framework with support for undoable commands already available. Therefore, if stepping back in the execution history is to be supported for manually implemented DSML interpreters, using a proxy-based approach like that sketched in Sec. 3.1 seems reasonable.

Animated Model Execution with Translational Semantics. Animated model execution can be achieved not only with an operational but also with a translational semantics description. For this, the DSM editor must provide an API with callback functions that are called by the generated code to reflect the current runtime state in the editor. This approach is supported, for example, by the tool MetaEdit+ (Tolvanen et al. 2007). Using this approach, an undo feature for execution steps cannot be provided generically. Whether and how execution steps can be undone is specific to the platform the generated code runs on.

6. Conclusion

Contribution. In this paper, we showed how prototyping of DSMLs with operational semantics can be improved by support for stepping back in the execution history of DSMs. We presented an approach for this support that is based on reusing model management techniques normally used in model editors. We proved the feasibility of our approach with an implementation based on the tool EPROVIDE.

Future Work. Execution control, such as support for breakpoints between DSL execution steps, can be achieved by extending the configuration metamodel with elements for controlling the execution process and by adapting the transition transformation to use these elements (Sadilek and Wachsmuth 2008b). For Petri Nets, this would mean to suspend the execution if a certain number of tokens on a specific place is reached or if a selected transition gets activated or deactivated. But at the moment, a user interface for managing breakpoints has to be programmed manually for each DSML. We want to investigate whether breakpoint support for DSMLs can be described declaratively.

Another common debug feature is source lookup. For model debugging, this means accessing and showing those parts of the operational semantics that are related to selected model elements. EPROVIDE is extensible to allow operational semantics descriptions in different languages. We want to investigate if and how generic source lookup for different semantics description languages can be provided.

Some debugging platforms include possibilities to change the structure of a debugged program at runtime. This is done, to some degree, with Java hot-code-replacement. In EPROVIDE, an editor is used for configuration visualization that allows a user to make arbitrary changes of configurations. To prevent a user from producing invalid configurations, the possible changes would have to be constrained. We want to investigate how such constraints can be described and implemented.

When a user finds an error in the operational semantics during execution, he can correct it and undo the step(s) in which he noticed the error (as described in Sec. 2). But maybe the error already occurred earlier and the user just did not notice. In this case, the current runtime state would have never been reached with the corrected operational semantics. EPROVIDE does not deal with this issue. To do so, it would be necessary to trace which parts of the operational semantics description are applied at which execution steps. This would enable EPROVIDE to automatically step back to the last state in the execution history that is consistent with the changed semantics. In principle, such a feature is possible (e.g., it was implemented for the visual, functional language Prograph (Cox and Pietrzykowski 1985)) but its feasibility strongly depends on the used description language.

Acknowledgments

We thank the anonymous reviewers for their valuable comments. This work is partially supported by grants from the Deutsche Forschungsgemeinschaft, Graduiertenkolleg METRIK (GRK 1324).

References

Christopher Brooks, Edward A. Lee, Xiaojun Liu, Steve Neuendorffer, Yang Zhao, and Haiyang Zheng. *Ptolemy II: Heterogeneous Concurrent Modeling and Design in Java*. UC Berkeley, 2007.

T. Clark, A. Evans, P. Sammut, and J. Willans. *Applied metamodelling: A foundation for language driven development*. `www.xactium.com`, 2004.

P. T. Cox and T. Pietrzykowski. Advanced programming aids in prograph. In *SIGSMALL '85: Proceedings of the 1985 ACM SIGSMALL symposium on Small systems*, pages 27–33, New York, NY, USA, 1985. ACM.

Juan de Lara and Hans Vangheluwe. Defining visual notations and their manipulation through meta-modelling and graph transformation. *Journal of Visual Languages & Computing*, 15(3-4):309–330, 2004.

Davide Di Ruscio, Frederic Jouault, Ivan Kurtev, Jean Bezivin, and Alfonso Pierantonio. Extending amma for supporting dynamic semantics specifications of dsls. Technical Report HAL - CCSd - CNRS, Laboratoire D'Informatique de Nantes-Atlantique, 2006.

G. Engels, J. H. Hausmann, R. Heckel, and S. Sauer. Dynamic metamodeling: A graphical approach to the operational semantics of behavioral diagrams in uml. In *UML'00*, volume 1939 of *LNCS*, pages 323–337. Springer, 2000.

Claudia Ermel, Karsten Hölscher, Sabine Kuske, and Paul Ziemann. Animated simulation of integrated uml behavioral models based on graph transformation. In *VL/HCC'05*, pages 125–133. IEEE Computer Society, 2005.

Ákos Lédeczi, Árpád Bakay, Miklós Maróti, Péter Völgyesi, Greg Nordstrom, Jonathan Sprinkle, and Gábor Karsai. Composing Domain-Specific Design Environments. *Computer*, 34(11):44–51, 2001. ISSN 0018-9162.

P.A. Muller, F. Fleurey, and J.M. Jézéquel. Weaving executability into object-oriented meta-languages. In *MoDELS'05*, volume 3713 of *LNCS*, pages 264–278. Springer, 2005.

Gordon D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

Jorn Janneck Robert Esser. Moses: A tool suite for visual modeling of discrete-event systems. In *HCC'01*, pages 272–279. IEEE Computer Society, 2001.

Daniel A. Sadilek and Guido Wachsmuth. Eprovide 2.0: Where grammarware meets modelware. Technical report, Humboldt-Universität zu Berlin, 2008a.

Daniel A. Sadilek and Guido Wachsmuth. Prototyping visual interpreters and debuggers for domain-specific modelling languages. In Ina Schieferdecker and Alan Hartman, editors, *ECMDA-FA 2008*, volume 5095 of *LNCS*, pages 63–78, Berlin, Germany, 2008b. Springer.

Markus Scheidgen and Joachim Fischer. Human comprehensible and machine processable specifications of operational semantics. In *ECMDA'07*, volume 4530 of *LNCS*, pages 157–171. Springer, 2007.

Juha-Pekka Tolvanen, Risto Pohjonen, and Steven Kelly. Advanced tooling for domain-specific modeling: MetaEdit+. In J. Sprinkle, J. Gray, M. Rossi, and J.-P. Tolvanen, editors, *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM'07)*, number TR-38 in Computer Science and Information System Reports. University of Jyväskylä, 2007.

Guido Wachsmuth. Modelling the operational semantics of domain-specific modelling languages. In *GTTSE 2007*, LNCS. Springer, 2008. to appear.

Volker Wegert and Alex Shatalin. *Integrating EMF and GMF Generated Editors*, March 2008. URL `http://www.eclipse.org/articles/article.php?file=Article-Integrating-EMF-GMF-Editors/index.html`.