Prototyping Visual Interpreters and Debuggers for Domain-Specific Modelling Languages

Daniel A. Sadilek and Guido Wachsmuth

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

Abstract. This paper is about visual and executable domain-specific modelling languages (DSMLs) that are used at the platform independent level of the Model-Driven Architecture. We deal with DSMLs that are new or evolve rapidly and, as a consequence, have to be prototyped cheaply. We argue that for prototyping a DSML on the platform independent level, its semantics should not only be described in a transformational but also in an operational fashion. For this, we use standard modelling means, i.e. MOF and QVT Relations. We combine operational semantics descriptions with existing metamodel-based editor creation technology. This allows for cheap prototyping of visual interpreters and debuggers. We exemplify our approach with a language for Petri nets and assess the manual work necessary. Finally, we present *EProvide*, an implementation of our approach based on the Eclipse platform, and we identify missing features in the Eclipse tools we used.

1 Introduction

Developing a software system for a specific domain requires knowledge from the practitioners of that domain, the so-called *domain experts*. In the *Model Driven Architecture* (MDA), domain experts should be involved in creating the *computation independent model* (CIM) [1]. The CIM captures the requirements for the system—the "know-what". Additionally, domain experts do also have knowledge about how the system can fulfil the requirements—the "know-how". This knowledge is needed for the *platform independent model* (PIM). A PIM can be expressed in a general-purpose modelling language like the UML. But domain experts like seismologists or meteorologists are not used to the modelling concepts and notation used in the UML.[1] In contrast, a *domain-specific modelling language* (DSML), specific for the application domain, provides domain experts with concepts they know and with a special notation that matches their intuition. Thus, DSMLs allow domain experts to provide their know-how on the PIM level.

[1] UML can be customised with its profiling mechanism. However, this is not adequate in all cases because a UML profile can only introduce new concepts as specialisation of existing UML concepts.

A DSML can have a very narrow application domain. When building a new system, no existing DSML may be appropriate and a new one may be necessary. Usually, the concepts the new DSML should offer are not clear in the first place and multiple development iterations are necessary. Furthermore, when requirements for a system change, the DSML may need to be adapted. Therefore, a prototyping process for the DSML is needed. To evaluate prototypical versions of a DSML, the domain expert should be able to *use* the DSML: that is to create example models expressed in the DSML and to execute them.

To enable *model creation*, editors for DSMLs can already be generated from a declarative description [2]. The typical way in MDA to *execute PIMs* is to translate them (by model transformation or code generation). But such a translation does not have a proper level of abstraction for prototyping: language semantics is intermingled with platform-specific details. This inhibits understanding, troubleshooting, and adaptation of evolving language semantics—especially for domain experts.

Therefore, we argue that for prototyping a DSML, its semantics should be described on the platform independent level in an operational fashion. For this, we use standard model transformation techniques. By combining this approach with existing metamodel-based editor creation technology, we enable cheap prototyping of visual *interpreters* and *debuggers*.

In the following section, we introduce necessary vocabulary and technology. We present our approach and exemplify it with a language for Petri nets in Sect. 3. In Sect. 4, we present *EProvide*, an Eclipse-based implementation of our approach. In Sect. 5, we show the manual work necessary for developing a Petri net debugger and we discuss missing features we encountered in Eclipse EMF, GMF, and in the QVT implementation we use. We discuss related work in Sect. 6 and conclude in Sect. 7.

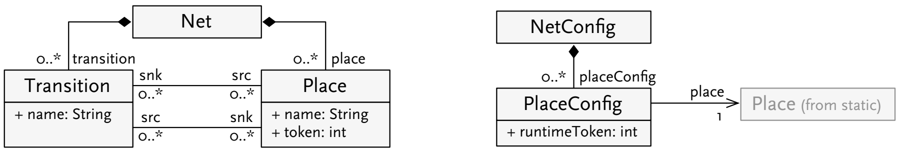
2 Preliminaries

2.1 Model-Driven Language Engineering

In the MDA, it is common practice to specify modelling languages by modelling means. A metamodel is an object-oriented model of the abstract syntax of a modelling language. With its *Meta Object Facility* (MOF) [3], the OMG provides standard description means for metamodels.

Example 1 (Petri net metamodel). Figure 1(a) provides a MOF compliant metamodel for Petri nets. A Petri net consists of an arbitrary number of places and transitions. Transitions have input (src) and output places (snk). Places are marked with a number of tokens. Places and transitions can have names.

Model transformations take a central place in MDA. They are used to translate models to other models, e.g. PIMs into PSMs (*platform-specific models*). Often they are used to translate a model into an executable form, e.g. to Java. From a



(a) Static metamodel. (b) Separate configuration metamodel.

Fig. 1. Petri net metamodels

language perspective, model transformations define translational semantics for a modelling language.

In this paper, we use the high-level declarative language QVT Relations for transformations. It is part of the OMG standard Query/View/Transformation (QVT) [4] and heavily relies on OCL. In QVT Relations, a *transformation* consists of queries and relations. A *query* returns the result of an OCL expression. A *relation* relates model elements in the domains of the transformation by patterns. All relations of a transformation which are declared as *top* need to hold. If a relation does not hold, the transformation tries to satisfy it by changing model elements in domains declared as *enforce*. Relations can have two kinds of clauses: *Where clauses* must hold in order to make a relation hold. A *where* clause can contain OCL queries and invocations of other relation. *When clauses* act as preconditions. A relation must hold only if its *when* clause already holds.

Example 2 (Queries and relations in QVT Relations). Figure 2 shows a transformation in QVT Relations. It relates the two domains `input` and `output`, which both contain Petri net models. The transformation contains the queries `isActivated` and `getActivated`. The first query returns whether a given transition in a Petri net is activated. For this, it checks if all the transition's input places are marked with tokens. The second query returns an activated transition of a given Petri net. Here, the OCL predicate `any` ensures a non-deterministic choice.

Furthermore, the transformation contains several relations. The relation `run` relates a Petri net model from the `input` domain with a Petri net model from the `output` domain. The first pattern of `run` binds the variable `net` to a Petri net in the input domain. The second pattern *enforces* the same Petri net to occur in the output domain, i.e., it will be created if it not already exists.

The relation obtains an activated transition of the net by calling the query `getActivated`. To make the relation `run` hold, the unary relation `fire` needs to hold. `fire` simply checks if the transition can be found in the input domain.

The first relation `run` is declared as *top*. Therefore, it has to hold to process the transformation successfully. Since the relation `fire` is not declared as *top*, it only has to hold to fulfil other relations calling it from their *where* clauses. We will discuss the remaining relations `produce`, `consume`, and `preserve` later on.

```
transformation petri_sos(input:petri, output:petri) {

    top relation run {
        trans: Transition;
        checkonly domain input net:Net{};
        enforce domain domain output net:Net{};
        where { trans = getActivated(net); fire(trans); }
    }

    query isActivated(trans: Transition): Boolean {
         trans.src -> forAll(place | place.token > 0)
    }

    query getActivated(net: Net): Transition {
         net.transition -> any(trans | isActivated(trans))
    }

    relation fire {
        checkonly domain input trans:Transition{};
    }

    top relation produce {
        checkonly domain input place:Place{
            src = trans:Transition{}, token = n:Integer{}
        };
        enforce domain output place:Place{ token = n+1 };
        when { fire(trans); trans.src -> excludes(place); }
    }

    top relation consume {
        checkonly domain input place:Place{
            snk = trans:Transition{}, token = n:Integer{}
        };
        enforce domain output place:Place{ token = n-1 };
        when { fire(trans); trans.snk -> excludes(place); }
    }

    top relation preserve {
        checkonly domain input place:Place{ token = n:Integer{} };
        enforce domain output place:Place{ token = n-1 };
        when { not produce(place, place); not consume(place, place); }
    }
}
```

Fig. 2. Operational semantics for Petri nets

2.2 Structural Operational Semantics

The operational semantics of a language describes the meaning of a language instance as a sequence of computational steps. 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*.

Plotkin pioneered this approach. In his work on structural operational semantics [5], he proposed to describe transitions according to the abstract syntax of the language. This allows for reasoning about programs by structural induction and correctness proofs of compilers and debuggers [6]. The structural operational semantics of a language defines an abstract interpreter for this language working on its abstract syntax. It can be used as a reference to test implementations of compilers and interpreters.

3 Platform Independent Model Semantics

3.1 Abstract Interpretation

In this paper, we apply the idea of structural operational semantics to model-driven language engineering. For this, we rely on standard modelling techniques only: MOF and QVT Relations.

We represent the configurations in Γ as models, which we call *configuration models*. Hence, we define the space of all possible configurations with a metamodel, which we call *configuration metamodel*; and we define the transition relation $\rightarrow$ with a model-to-model transformation, which we call *transition transformation*.

For some simple languages, computational steps can be expressed as mere syntactic manipulation of the program/model. A well-known example for this is the lambda calculus [7]. Another example are Petri nets.

Example 3 (Syntactic manipulation of Petri nets). A configuration of a Petri net is simply its current marking. Therefore, we can use the static DSML metamodel from Fig. 1(a) as configuration metamodel, as well. A computation step in a Petri net chooses an activated transition non-deterministically and fires it. The marking of a place connected to the fired transition is

(i) increased by one token iff it is only an output place,
(ii) decreased by one token iff it is only an input place,
(iii) preserved otherwise.

The transformation given in Fig. 2, specifies these semantics in QVT Relations. It contains the relations `run` and `fire` as well as the queries `isActivated` and `getActivated`, which we already discussed in Ex. 2. The relations `produce` and `consume` adapt the token count of places that are connected to fired transitions:

(i) The relation `produce` matches a place `place` in the input, an incoming transition `trans` of this place, and its number of tokens `n`. It enforces an increase of one in the number of tokens in the output. The relation needs to hold only if the matched transition is fired and if the matched place is not an input place of this transition.
(ii) The relation `consume` works similarly. It matches a place in the input, an outgoing transition of this place, and its number of tokens. The number of tokens in the output is decreased by one. The relation has to hold only if the transition is fired and if the place is not an output place of the transition.
(iii) If neither `produce` nor `consume` hold for a place, its token count is preserved by the relation `preserve`.

Describing operational semantics by mere syntactic manipulation works only for simple languages. In general, runtime states cannot be expressed with instances of the static language metamodel. Hence, a separate configuration metamodel is needed to represent runtime states. In addition, an *initialisation transformation* is needed that creates an initial configuration from a static model.

```
transformation init(net: petri, config: petriCfg) {
    top relation initNet {
        checkonly domain net n:Net{};
        enforce domain config nc:NetConfig{net = n};
    }

    top relation initPlace {
        checkonly domain net n:Net {place = p: Place{token = i:Integer{}}};
        enforce domain config nc:NetConfig{
            placeConfig = pc:PlaceConfig{place = p, runtimeToken = i}
        };
        when { initNet(n, nc); }
    }
}
```

Fig. 3. QVT transformation to initialise the runtime configuration of a Petri net

Example 4 (Advanced semantics of Petri nets). The Petri net semantics presented in Ex. 3 destroys the initial marking of a Petri net model by the execution. A separate configuration metamodel solves this problem (Fig. 1(b)). To preserve the initial marking of a Petri net, we distinguish `Net` and its configuration `NetConfig`. A net configuration contains a configuration `PlaceConfig` for each place in the net. The marking of a place is stored in the attribute `PlaceConfig.runtimeToken`.

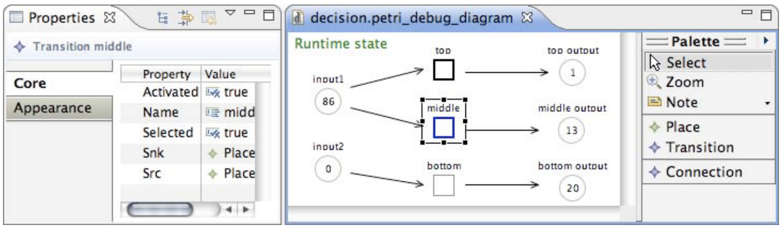
With a separate configuration metamodel, we need an initial configuration to run the net. The QVT transformation given in Fig. 3 specifies this initialisation. It enforces a net configuration to contain a configuration for each place in the net. For a place configuration, the attribute `runtimeToken` is initialised with the initial marking from the attribute `token` of the configured place. Furthermore, we need to adapt the semantics description to act on the configuration instead of the net. We will see the adapted description in Ex. 6.

3.2 Visual Interpretation

Given a visualisation of the runtime state, a visual interpreter comes for free: PIMs are executed stepwise with the transition transformation and each step can be visualised. We give the user the option to control how many execution steps are performed before the visualisation gets updated. Thus, one can control how fast the interpretation process is animated.

Because we represent the runtime state of a PIM (its current configuration) as a model, it can easily be visualised reusing existing metamodel-based technology for creating model editors. There are two options for visualising runtime elements in a visual interpreter:

(i) runtime elements are visualised as additional graphical entities,
(ii) runtime elements affect the visualisation of static model elements, e.g. existing labels are extended with runtime values or the colour of existing graphical entities is controlled by runtime values.



Property	Value
Activated	true
Name	midd
Selected	true
Snk	Place
Src	Place

Fig. 4. Debugger for the Petri net DSML prototyped with *EProvide*

Example 5 (Petri net visualisation). Figure 4 shows an editor for the runtime state of a Petri net. Following Petri net conventions, circles and boxes represent places and transitions respectively. Incoming and outgoing arcs determine the input and output places of transitions. A number inside a circle shows the runtime marking of the place—not the initial (static) marking. This corresponds to option (ii).

3.3 Visual Debugging

Debugging means to control the execution process and to access and possibly modify the runtime state of a program. In usual debuggers, a program can be executed stepwise and breakpoints can be set to halt the execution at a specified point of execution. Whenever a program is halted, the user can access and modify variable values.

With our approach, a PIM can be executed stepwise by setting the number of execution steps to one. Further control on the execution, like support for breakpoints, can be achieved by extending the configuration metamodel with elements for controlling the interpretation process and adapting the transition transformation to use these elements. Accessing and modifying runtime state between interpretation steps is possible via the model editor that visualises the configuration model.

As for visual interpretation, visual debugging is achieved by applying existing editor creation technology.

Example 6 (Debugging Petri nets). When a user debugs Petri nets, he wants to know if a transition is currently activated or not. Furthermore, he wants to control which transition fires in the next execution step. We extend the configuration metamodel from Fig. 1(b) with a class `RuntimeTransition` that includes appropriate attributes. One is the derived attribute `activated`, which indicates the activation of a transition. We specify its value with an OCL query, similar to the body of `isActivated` given in Fig. 2. Additionally, we introduce the attribute `selected`, which can be set by the user to select transitions for execution.

Breakpoints are another mandatory feature for debugging. For Petri nets, we introduce two kinds of breakpoints in our metamodel referring places and transitions respectively. A breakpoint for places defines a number of tokens.

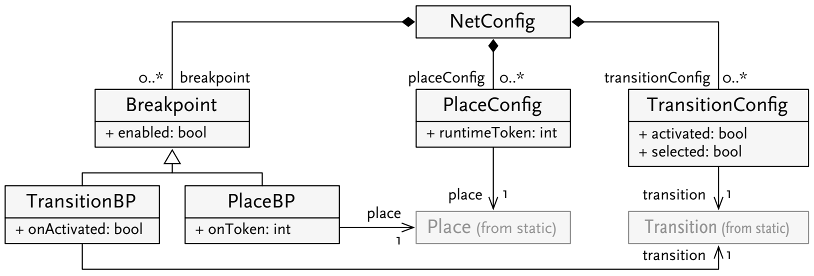


Fig. 5. Configuration metamodel extended with additional classes and attributes for debugging

If the place is marked with the defined number of tokens, the execution will be interrupted. A breakpoint for transitions refers to a transition and defines whether it should become activated or deactivated in order to interrupt the execution. Both kinds of breakpoints can be disabled or enabled.

Figure 5 shows the configuration metamodel extended for debugging. It includes the class `TransitionConfig` as well as the classes `Breakpoint`, `PlaceBP`, and `TransitionBP` for breakpoints. To achieve the expected behaviour of the debugger, we adapt the transition transformation. The adapted version is shown in Fig. 6. The new relations `breakPlace` and `breakTransition` check whether enabled breakpoints trigger an interruption of the execution. Furthermore, the relation `run` is only established if no breakpoint meets its breaking condition. To fire only selected transitions, the query `getActivated` is redefined to use `TransitionConfig`'s new attributes `activated` and `selected`. The query `is-Activated` is obsolete. All other relations remain unchanged.

4 Implementation

4.1 Base Technologies

In this section, we describe the implementation of our approach. We based our implementation on Eclipse modelling technologies. As a metamodelling framework, we use the Eclipse Modeling Framework (EMF) [8]. In EMF, metamodels need to comply to the Ecore meta-metamodel which is very similar to Essential MOF. Based on the Ecore-based metamodel of a DSML, Java code for a *DSML plugin* can be generated with EMF. This plugin provides the infrastructure to create, access, modify, and store models that are instances of the DSML.

For the creation of graphical editors, we use Eclipse's Graphical Modeling Framework[2] (GMF). In GMF, an editor is described with a set of declarative models, which we call *editor definition* in the following. Amongst other things, those models define which graphical elements are used to represent which elements of the DSML metamodel. The editor definition is used by the GMF code

[2] `http://www.eclipse.org/gmf/`

```
transformation petri_debug(input:petriCfg, output:petriCfg) {

    top relation breakPlace {
        checkonly domain input nc:NetConfig{
            breakpoint = bp:PlaceBP{
                enabled = true, onToken = n:Integer{}, place = p:Place{}
            },
            placeConfig = pc:PlaceConfig{place = p, runtimeToken = n}
        };
    }

    top relation breakTransition {
        checkonly domain input nc:NetConfig{
            breakpoint = bp:TransitionBP{
                enabled = true, onActivated = b:Boolean{}, transition = t:Transition{}
            },
            transitionConfig = tc:TransitionConfig{transition = t, activated = b}
        };
    }

    top relation run {
        trans: Transition;
        checkonly domain input nc:NetConfig{};
        enforce domain output nc:NetConfig{};
        when { not breakPlace(nc); not breakTransition(nc); }
        where { trans = getActivated(nc); fire(trans); }
    }

   query getActivated(netCfg: NetConfig): Transition {
         netCfg.transitionConfig -> any(selected & activated).trans;
    }

    top relation produce {
        checkonly domain input placeCfg:PlaceConfig{
            place = place:Place{ src = trans:Transition{} },
            runtimeToken = n:Integer{}
        };
        enforce domain output placeCfg:PlaceConfig{ runtimeToken = n+1 };
        when { fire(trans); trans.src -> excludes(place); }
    }
    ...
}
```

Fig. 6. Operational semantics for debugging Petri nets

generator to generate Java source code for a *graphical editor*. The generated code can be modified by hand afterwards. As we will see in Sect. 5, this was necessary for the Petri net example.

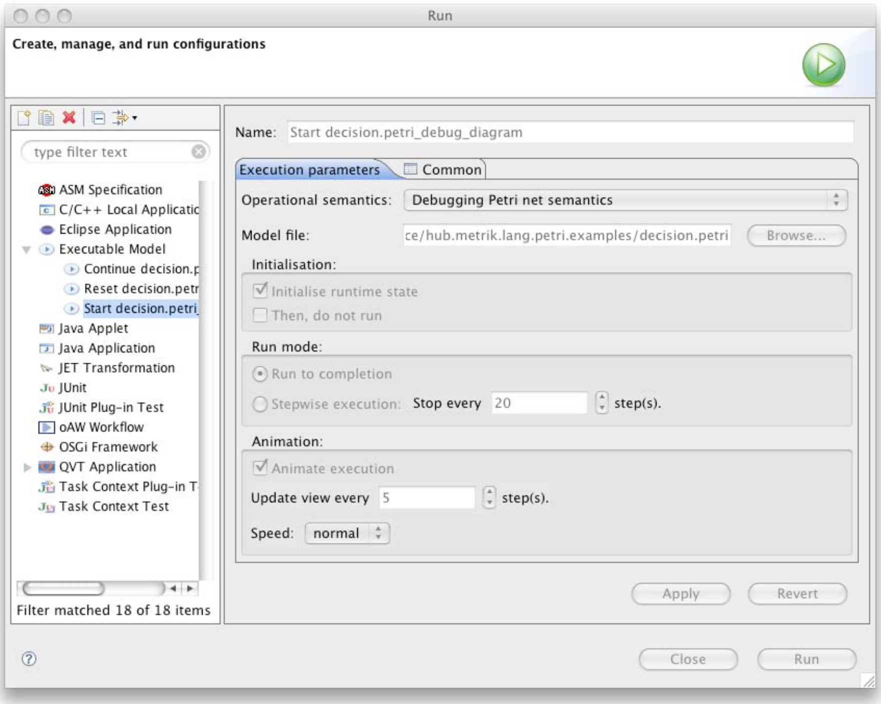
For model-to-model transformations, we use ikv's medini QVT[3], an implementation of QVT Relations that can work with Ecore compliant metamodels.

4.2 *EProvide*

EProvide[4] is an implementation of our approach. It is an Eclipse plugin that plugs into the Eclipse execution infrastructure to make domain-specific

[3] http://projects.ikv.de/qvt

[4] *E*clipse plugin for *PRO*totyping *V*isual *I*nterpreters and *DE*buggers; available at http://eprovide.sf.net



Name: Start decision.petri_debug_diagram

Operational semantics: Debugging Petri net semantics

Model file: ce/hub.metrik.lang.petri.examples/decision.petri

Initialisation:
[✓ Initialise runtime state]
Then, do not run

Run mode:
[✓ Run to completion]
Stepwise execution: Stop every 20 step(s).

Animation:
[✓ Animate execution]
Update view every 5 step(s).
Speed: normal

Fig. 7. *EProvide*'s run dialogue

PIMs—or domain-specific models in general—executable. It provides an extension point that DSML plugins can use to register themselves with *EProvide*.

The Language User Perspective. *EProvide* offers a run dialogue in which the user can configure model execution (Fig. 7): The user can choose whether the model should be initialised and whether it should be executed to completion or stepwise. If he selects "Run to completion", the transition transformation gets repeatedly executed until the model reaches a fixed-point state in which it persists. If he selects "Stepwise execution", the execution stops after the given number of steps (if no fixed-point state is reached before). Furthermore, *EProvide* allows to animate the execution process. The user can activate and configure this feature in the run dialogue, as well. He can specify how often the editor gets updated (e.g. every 5-th transformation step) and how fast the animation is (i.e. how long each visualised configuration is shown). If multiple semantics are available for a DSML, the run dialogue allows the user to select which semantics should be applied.

The Language Engineer Perspective. To achieve execution support for a new DSML, the DSML plugin has to provide an extension for *EProvide*'s extension point. In this extension, the following information is given: a name for the semantics that is shown in the run dialogue, the file extension of the model files that the semantics should be applied to, the path to the QVT files with the transition and the initialisation transformation, and the full qualified Java name of the DSML's metamodel package.

5 Case Study: A Debugger for the Petri Net DSML

In this section, we show the manual work necessary for developing a Petri net debugger with *EProvide*. We develop the debugger in five steps, starting with a simple editor and ending with the full debugger. In some steps, manual modifications of Java code generated with EMF or GMF are necessary. These manual modifications could be avoided if EMF and GMF had some missing features that we will identify in the following. Table 1 summarises the work necessary for each step and also shows which work could be saved with which additional features.

Step 1: Editor. The first step is to create a new DSML plugin that contains an EMF compliant metamodel for the Petri net DSML (cf. Fig. 1(a) from Ex. 1) and a GMF-definition of the graphical editor. This step is independent of *EProvide*. The editor definition defines which graphical elements should be used (graphical definition), for which elements of the metamodel creation tools exist (tooling definition), and how metamodel elements are mapped to tooling elements and graphical elements. Figure 4 shows a screenshot of the final debugger; the editor actually does not yet highlight activated and selected transitions.

Step 2: Visual Interpreter without Separate Runtime Attributes. As second step, we create an interpreter that uses the static metamodel to save the runtime states as explained in Ex. 3. For this, we define a new extension for *EProvide*'s extension point in the DSML plugin and we implement the transition transformation as shown in Fig. 2 from Ex. 3. This transition transformation uses the OCL-function `any`, which should behave non-deterministically.

Missing Feature 1: Non-deterministic Choice in QVT Implementation. Unfortunately, the QVT implementation we use lacks non-determinism. Therefore, we have to extend the class `Net` with a method `choose()` implementing a non-deterministic choice. The signature of this method is specified in the metamodel while its implementation is given in Java. We adapt the transition transformation to use `Net.choose()` instead of the OCL-function `any`.

Step 3: Visual Interpreter with Separate Runtime Attributes. In this step, we extend the interpreter so that interpretation does not destroy the initial marking of the Petri net. As described in Ex. 4, this requires to store the configurations in separate metamodel elements and to provide an initialisation transformation. We suggested to define a separate runtime metamodel (Fig. 1(b)) that augments the static metamodel. This would result in the architecture shown in Fig. 8 and would have the advantage that the static metamodel is not polluted with runtime elements. In this architecture, the visual interpreter would be an extension of the model editor and would show the runtime marking instead

Table 1. Manual work required for the development of the Petri net debugger. The work that could be saved with additional EMF/GMF features is marked with the number of the corresponding feature in parentheses.

1. Editor		
Metamodel	3 classes (`Net`, `Place`, `Transition`), 5 associations, 3 attributes	
Editor definition	35 + 13 + 35 XMI nodes (graphical, tooling, mapping)	
2. Visual interpreter without separate runtime attributes, destroys initial state		
Metamodel	1 operation (`Net::choose`) for non-deterministic choice	(1)
DSML plugin	6 lines Java code for implementing `Net.choose()`	(1)
	8 lines in the plugin.xml for plugging into *EProvide*	
Semantics	31 lines QVT Relations code	
3. Visual interpreter with separate runtime attributes		
Metamodel	3 attributes (`Net::running`, `Place::initToken`, `Place::runtimeToken`)	
DSML plugin	23 lines Java code for `Net.running`-switch	(2)
Semantics	17 lines QVT Relations code for initialising runtime model	
Graphical editor	12 lines Java code for showing a running label	(2)
4. Debugger without breakpoints		
Metamodel	2 attributes (`Transition::activated`, `Transition::selected`)	
Semantics	1 line QVT Relations code for using the new attributes	
Graphical editor	27 lines Java code for visualising transition states	(3)
	8 lines Java code for change notif. for `Transition.activated`	(4)
5. Debugger with breakpoints		
Metamodel	3 classes (`Breakpoint`, `PlaceBP`, `TransitionBP`),	
Semantics	17 lines QVT Relations code to take breakpoints into account	
Editor definition	— (breakpoints can only be defined in generic tree editor)	

of the initial marking of a Petri net. In GMF, the layout information (element positions, size, etc.) of a diagram is stored in a separate graphical model. In the shown architecture, the graphical runtime model would augment the graphical static model so that the layout information from the static model would be reused for visual interpretation.

Missing Feature 2: Editor extension in GMF. GMF does not support this architecture. An editor definition cannot extend another one and a graphical model cannot reference another one. Therefore, we cannot use a separate runtime model with a separate visual interpreter but have to extend the static model to an *integrated model* with runtime elements. The corresponding *integrated metamodel* has three additional attributes and one with changed semantics. `Place.token` now delegates to one of the two new attributes `Place.initToken` and `Place.runtimeToken`. Which it delegates to is controlled by the third new attribute, `Net.running`. In order to show the user whether the static model

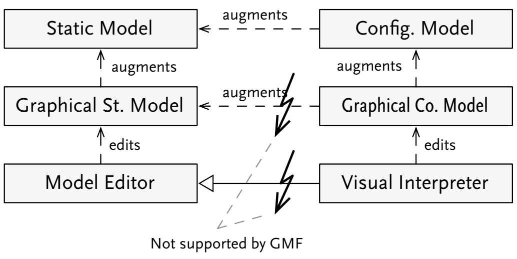


Fig. 8. The editor/interpreter architecture as it should be

or the runtime model is visualised, we manually add a “running” label to the generated GMF editor code that is shown only for the runtime model (cf. Fig. 4).

Step 4: Debugger without Breakpoints. In this step, we add features that allow the user to control the execution of a Petri net model. Stepwise execution of Petri net models is already provided by the *EProvide* infrastructure (cf. Sect. 4.2). In addition, we want to visualise activated transitions with a thick, black border (transition “top” in Fig. 4) and transitions selected by the user to fire with a thick, blue border (transition “middle” in Fig. 4). To implement this, we add two attributes (cf. Ex. 6) to the integrated metamodel: `Transition.activated` is a derived attribute and implemented in OCL, and `Transition.selected` is an ordinary attribute that can be set by the user. Furthermore, we adapt the transition transformation to use the new attributes.

Missing Feature 3: Visual attributes in GMF derived from model attributes. GMF’s editor definition does not support to derive the visualisation of a transition’s border from the transition’s model attributes. Therefore, we manually add the visualisation in the generated GMF editor code.

Missing Feature 4: Change notifications for derived attributes in EMF. EMF allows to define derived attributes in OCL. The Java code generated for the metamodel then contains code that interprets the OCL expression. If an attribute that the derived attribute depends on changes, the getter for the derived attribute returns an updated value when it is called the next time. But the generated code does not automatically send a change notification for the derived attribute. Therefore, we manually modify the generated code to send change notifications for `Transition.activated`.

Step 5: Debugger with Breakpoints. In the last step, we add support for breakpoints. For this, we add the three classes `Breakpoint`, `PlaceBP`, and `TransitionBP` to the integrated metamodel. Furthermore, we adapt the transition transformation to use the new classes (cf. Ex. 6). In this case study, we do not support graphical specification of breakpoints; they have to be specified in the generic EMF tree editor. (If we wanted to support graphical breakpoint specification, we had to extend the editor definition.)

6 Related Work

Ptolemy [9] allows animated interpretation of hierarchically composed domain-specific models with different execution semantics. Adding a new DSML to Ptolemy requires a lot of work as both its syntax and its semantics have to be coded manually in Java. GME [10] provides visualisation of the 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++.

Several approaches address a metamodel-based formalisation of language semantics. Sunyé et al. recommend UML action semantics for executable UML models [11]. Furthermore, they suggest activities with action semantics for language modelling. Scheidgen and Fischer follow this suggestion and provide description means for the operational semantics of MOF compliant metamodels [12]. Muller et al. integrate OCL into an imperative action language [13] to provide semantics description means for the Kermeta framework.In a similar way, OCL is extended with actions to provide semantics description means in the Mosaic framework [14]. The AMMA framework integrates Abstract State Machines for the specification of execution semantics [15]. Furthermore, the model transformation language ATL [16] can be applied to specify operational semantics. These approaches lack visualisation of the interpretation but those based on EMF can be integrated into *EProvide* easily.

Graph transformations are a well-known technology to describe the operational semantics of visual languages. Engels et al. describe the operational semantics of UML behaviour diagrams in terms of collaboration diagrams which represent graph transformations [17]. Similarly, Ermel et al. translate UML behaviour diagrams into graph transformations to simulate UML models [18]. Since they provide domain-specific animation views, the runtime state can not be changed by the user which inhibits debugging.

The Moses tool suite [19] provides a generic architecture to animate and debug visual models, which are represented as attributed graphs. The runtime state is visualised by so called animation decorators added to the attributed graph. The difference to our approach is that in Moses the execution semantics of models is given as an abstract state machine description and the runtime state of a model is encoded in ASM functions. In contrast, we store the runtime state itself as a model, which allows us to reuse the same editor creation technology for animation as for editing.

In AToM3, de Lara and Vangheluwe use graph grammars to define the operational semantics of a visual modelling language [20]. In contrast, we rely on OMG standard means to define metamodel-based operational semantics. Thus, operational semantics can be integrated in an MDA process more naturally.

7 Conclusion

Contribution. We showed how operational semantics of a DSML can be specified at the PIM level with standard modelling techniques. Furthermore, we combined this approach with existing metamodel-based editor creation technology. This enables rapid prototyping of visual interpreters and debuggers. We illustrated our approach with a DSML for Petri nets. Thereby, we identified desirable features missing in base technologies we used. With *EProvide*, we offer an implementation of our approach as an Eclipse plugin.

Our approach minimises the effort for MDA practitioners to define prototypical language semantics. They can rely on standard means and tools they are used to. This facilitates short iteration circles for language engineering, early integration of domain experts, higher quality of DSMLs and the system under development, and thus minimises development costs.

Future Work. In the Petri net case study, we did not distinguish between static model and configuration model but used an integrated model. Although we did this because of the missing editor extension feature in GMF, it allows for another nice feature: a user can modify static model elements at runtime. As we argued, we would prefer not to pollute the static model with configuration elements. But without an integrated model, allowing the user to modify static model elements at runtime would require additional work. For example, if a user creates a new element in the static model, the corresponding element in the configuration model must be created. This adaptation needs additional specification. Here, the question is whether the adaptation must be programmed manually or if it is possible to describe it declaratively in a model.

Typically, a DSML metamodel evolves over time. Prototypical tool support as mentioned in this paper can help to reveal necessary changes. Automated adaptation of metamodels [21] helps to control metamodel evolution. Automated co-adaptation helps to keep related artefacts like instances or semantic descriptions in sync with the evolving metamodel. For the approach presented in this paper, automated co-adaptation of declarative editor models is needed.

Acknowledgement. We are thankful to Hajo Eichler, Omar Ekine, and Jörg Kiegeland from ikv for help with medini QVT. Also thanks to the anonymous reviewers for valuable comments. This work is supported by grants from the DFG (German Research Foundation, Graduiertenkolleg METRIK).

References

1. Miller, J., Mukerji, J.: MDA Guide Version 1.0.1. Technical report, Object Management Group (OMG) (2003)
2. The Eclipse Foundation: Eclipse Graphical Modeling Framework (GMF) (2007), http://www.eclipse.org/gmf/
3. Object Management Group: Meta Object Facility 2.0 Core Specification (2006)
4. Object Management Group: Meta Object Facility 2.0 Query/View/Transformation Specification (2007)
5. Plotkin, G.D.: A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus (1981)

6. da Silva, F.Q.B.: Correctness Proofs of Compilers and Debuggers: an Approach Based on Structural Operational Semantics. PhD thesis, University of Edinburgh (1992)
7. Barendregt, H.P.: The Lambda Calculus its Syntax and Semantics, 2nd edn. North Holland, Amsterdam (1987)
8. Budinsky, F., Merks, E., Steinberg, D.: Eclipse Modeling Framework, 2nd edn. Addison-Wesley, Reading (2006)
9. Brooks, C., Lee, E.A., Liu, X., Neuendorffer, S., Zhao, Y., Zheng, H.: Ptolemy II: Heterogeneous Concurrent Modeling and Design in Java. UC Berkeley (2007)
10. Lédeczi, Á., Bakay, Á., Maróti, M., Völgyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G.: Composing Domain-Specific Design Environments. Computer 34(11), 44–51 (2001)
11. Sunyé, G., Pennaneach, F., Ho, W.M., Guennec, A.L., Jéquel, J.M.: Using uml action semantics for executable modeling and beyond. In: Dittrich, K.R., Geppert, A., Norrie, M.C. (eds.) CAiSE 2001. LNCS, vol. 2068, pp. 433–447. Springer, Heidelberg (2001)
12. Scheidgen, M., Fischer, J.: Human comprehensible and machine processable specifications of operational semantics. In: Akehurst, D.H., Vogel, R., Paige, R.F. (eds.) ECMDA-FA. LNCS, vol. 4530, pp. 157–171. Springer, Heidelberg (2007)
13. Muller, P., Fleurey, F., Jézéquel, J.: Weaving executability into object-oriented meta-languages. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 264–278. Springer, Heidelberg (2005)
14. Clark, T., Evans, A., Sammut, P., Willans, J.: Applied metamodelling: A foundation for language driven development (2004), http://www.xactium.com
15. Di Ruscio, D., Jouault, F., Kurtev, I., Bezivin, J., Pierantonio, A.: Extending amma for supporting dynamic semantics specifications of dsls. Technical Report HAL - CCSd - CNRS, Laboratoire D'Informatique de Nantes-Atlantique (2006)
16. Jouault, F., Kurtev, I.: Transforming models with atl. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
17. Engels, G., Hausmann, J.H., Heckel, R., Sauer, S.: Dynamic meta-modeling: A graphical approach to the operational semantics of behavioral diagrams in uml. In: Evans, A., Kent, S., Selic, B. (eds.) UML 2000. LNCS, vol. 1939, pp. 323–337. Springer, Heidelberg (2000)
18. Ermel, C., Hölscher, K., Kuske, S., Ziemann, P.: Animated simulation of integrated uml behavioral models based on graph transformation. In: VL/HCC 2005, pp. 125–133. IEEE Computer Society, Los Alamitos (2005)
19. Robert Esser, J.J.: Moses: A tool suite for visual modeling of discrete-event systems. In: HCC 2001, pp. 272–279. IEEE Computer Society, Los Alamitos (2001)
20. de Lara, J., Vangheluwe, H.: Defining visual notations and their manipulation through meta-modelling and graph transformation. Journal of Visual Languages & Computing 15(3-4), 309–330 (2004)
21. Wachsmuth, G.: Metamodel adaptation and model co-adaptation. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609. Springer, Heidelberg (2007)