Verification Support for a State-Transition-DSL Defined with Xtext

Thomas Baar(✉)

Hochschule für Technik und Wirtschaft (HTW) Berlin,
Wilhelminenhofstraße 75A, 12459 Berlin, Germany
thomas.baar@htw-berlin.de

Abstract. A Domain-Specific Language (DSL) allows the succinct modeling of phenomena in a problem domain. Modern DSL-tools make it easy for a language designer to define the syntax of a new DSL, to specify code generators or to build a new DSL on top of existing DSLs. Based on the language specification, the DSL-tool then generates rich editors. Often, these editors support features such as syntax highlighting, code completion or automatic refactoring.

In this paper, we describe an approach of adding verification support for DSLs defined within the Eclipse-framework Xtext. Xtext provides good support for checking the well-formedness rules of the DSL's syntax. In contrast, support for specifying the language's semantics as well as verification support have been rather neglected so far. Our approach of incorporating semantic verification techniques is illustrated by a very simple State-Transition-DSL, which has been fully implemented in Xtext. The DSL's editor verifies on the fly that the current model holds some semantic properties such as deterministic execution and invariant preservation. The verification services for this DSL are based on the theorem prover Princess.

Keywords: DSL · Model verification · Proof obligation · State machine

1 Motivation

20 years ago, the three amigos of the UML were proud to win the war of notation. They have successfully extracted the main modelling concepts of the predecessors of the UML, such as OMG, Booch, Shlaer-Mellor, and created a graphical language with 9 diagrams at the beginning. This language was supposed to be applicable in many domains.

The nice thing about UML was (and still is) that, while being an open standard, it triggered many research activities, e.g. research groups mainly from academia reported at the UML conference series (which was 2005 renamed into MoDELS) on progress when working with models, i.e. their efficient creation, understanding, reuse, and semantic analysis. These efforts enjoyed great interest in the scientific community due to the large user base of UML.

M. Mazzara and A. Voronkov (Eds.): PSI 2015, LNCS 9609, pp. 50–60, 2016.
DOI: 10.1007/978-3-319-41579-6_5

However, not only the general purpose modelling languages made progress, but also Domain-specific languages (DSLs). Tool support for DSLs has concentrated so far in the easy definition of abstract and concrete syntax, and how to define code generators that take a model as input (see [1] for an overview).

Tools for the semantic analysis of the model formulated in such a DSL have been rather neglected. A typical example is the tool Yakindu [2], which implements a statechart DSL and provides an editor for editing and even simulating statecharts. Internally, Yakindu is based on Xtext.

When editing a statechart, Yakindu's editor gives very valuable feedback if the user constructs a *syntactically ill-formed* model, e.g., if she adds a transition ending in the start state or refers to a non-existing variable or event. However, Yakindu's editor does not offer any support if the model is semantically incorrect. For example, it does not issue a warning if the model contains dead transitions. In the simplest case, a transition is dead if it is annotated with a guard `[false]`, but any other unsatisfiable guard (e.g. `[3 > 7]`) would also lead to a dead transition.

The detection of dead transitions goes beyond syntactic checks. It requires to interpret guards and - for example - to reduce a guard `[3 > 7]` to `[false]` due to the standard interpretation of mathematical symbols `3`, `7`, `>`.

In this paper, we develop a DSL for describing simple state machines using Xtext. The editor for this DSL offers semantic model checks, e.g. the detection of dead transitions. Our semantic model checks are internally based on the theorem prover PRINCESS, which supports reasoning on integer arithmetic and first-order logic. PRINCESS works very fast and discards each proof obligation from the example discussed in this paper within less than 10 ms. Thus, semantic checks are not more expensive than ordinary syntactic checks and can be executed on-the-fly by the DSL's editor while the user enters the model.

2 The Framework Xtext

Xtext is a framework for the development of textual DSLs. It is implemented in Java and is available in form of a plugin for the popular IDE *Eclipse*. Illustrative examples on what can be achieved via Xtext can be found on its homepage [3].

In order to define a DSL, the user first has to define a grammar in an EBNF-like syntax. Based on the grammar, the Eclipse plugin will then generate the so-called *language infrastructure*, consisting of a textual editor, the Ecore-metamodel, and the Java API for easy model access. This generation process as well as the resulting tools can be customized by the user in numerous ways, e.g. by implementing classes for presenting the syntax tree differently (so-called *label providers*) or by defining additional constraints on the syntax tree (implemented by a so-called *validator*).

The implementation of customization classes is done in Xtend, a Java-based language developed by the creators of Xtext. Using Xtend, many typical programming tasks such as traversing a syntax tree, creating a net of objects, or template-based string generation can be solved very elegantly.

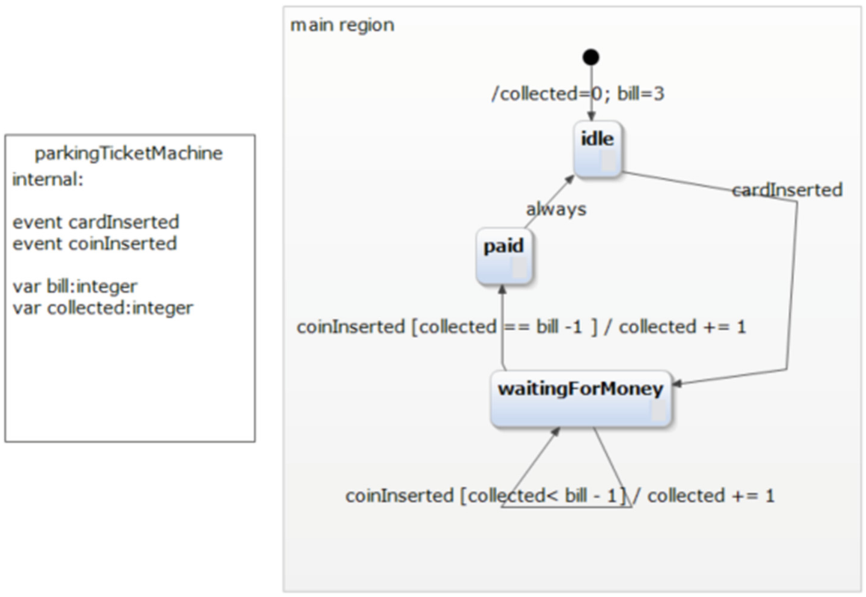


Fig. 1. ParkingTicketMachine (PTM) specified with Yakindu

2.1 Yakindu

Yakindu [2] is an open-source tool for the specification of statecharts [4]. It allows the user to *draw* states, nested states, and transitions; in this respect Yakindu is a tool for a graphical modelling language.

Since Yakindu is heavily based on Xtext, it is developed and maintained mainly by software engineers from *Itemis*, the company also standing behind Xtext. Yakindu allows the user to add to a transition an annotation consisting of a *guard*, an *event*, and an *action*. The formalism to express the annotation is internally defined as an Xtext-language.

An action can be a *named action* or an *update* of *state variables* with new values. Events, named actions, and state variables have to be declared separately. When specifying guards or updates, the user can take advantage of a pre-defined language for arithmetical and logical expressions.

Yakindu checks the type rules for expressions and sub-expressions instantly while a guard is typed in. Yakindu also executes additional *syntax checks* automatically, for example to ensure that each statechart has a start-state.

Yakindu does not offer *semantic checks*, which could help to prevent non-intended behaviour when executing the statechart. The only possibility for the user to analyze the run-time behaviour of the specified system is to execute the statechart in Yakindu's simulator.

2.2 Running Example ParkingTicketMachine (PTM)

In Fig. 1 we see a screenshot from Yakindu when editing a small statechart, which models a ticket machine in a parking deck (called *ParkingTicketMachine (PTM)*). The machine is waiting (state `idle`) for a customer inserting her park ticketing card (event `cardInserted`). Now, the ticket machine changes its state (to `waitingForMoney`) and the customer has to pay a certain amount of money (variable `bill`). The machine now expects coins to be inserted (event `coinInserted`) until the amount to be paid has been reached. For the sake of simplicity, we assume all coins to have the same value and that the amount to be paid is exactly the value of N many coins ($N > 0$). Once enough coins have been inserted, the machine changes its state first to `paid` and then immediately to `idle` (due to the pseudo-event `always`).

Unfortunately, the statechart has a semantic error: The machine collects the money correctly only from the first customer! After the machine has walked through the states `idle - waitingForMoney - paid - idle`, the state variable `collected` has the same value as `bill`. So, after the next customer has inserted her card, she can insert as many coins as she wants, but she will never reach the state `paid`!

3 Adding Verification Support for DSLs

In this section, we will demonstrate how one can implement verification support for DSLs implemented with Xtext. The verification support targets semantic properties of model written in the DSL. For the sake of illustration, we explain the verification techniques using the simple statechart introduced in Sect. 2.2.

Since our verification approach should serve as a blueprint for adding verification support to any textual DSL created with Xtext, we will start with the development of a DSL, which is able to denote simple forms of Yakindu's statecharts, but in a purely textual notation. Our language is called *SMINV* and models expressed in this language are called *state machines* in order to distinguish them easily from Yakindu's *statecharts*.

3.1 SMINV– A Textual DSL for Encoding Simple State Machines

A language to denote state machines can be basically subdivided into two parts: (1) The sublanguage for defining the 'infrastructure' of a state machine using concepts like *state*, *transition*, *event*, *state variable* and (2) the expression language for defining *guards* and *updates* of state variables.

Sublanguage for Infrastructure. The grammar definition for SMINV starts[1] with the start rule:

[1] Due to the paper's page limit, only the important parts of the grammar are presented here. The full grammar is available from [5].

```
SminvModel:
    vd=VarDecl
    sd=StateDecl
    ed=EventDecl
    td=TransDecl
```

A state machine is a sequence of declarations for variables, states, events, and transitions. While a variable, state or event is just declared by its name (which must be unique), the definition of a transition is a little bit more complex:

```
Transition:
    pre=[State] '=>' post=[State]
    (ev=[Event])? ('[' g=Term ']')? ('/' act+=Update+)? ';'
```

A transition connects two states (`pre`, `post`) and has optional annotations for event, guard, and updates.

Sublanguage for Expressions. The sublanguage for expressions is defined as usual in Xtext (see, for example, [6] for enlightening tutorial examples). Compared to the language supported by Yakindu, our expressions are simpler. We decided to allow only `INT` and `BOOL` as expression types. Thus, an expression is either a formula (boolean) or a term of type `INT`. Note that variables always have the type `INT`.

Running Example Formulated in SMINV. Despite its simplicity, our language allows to formulate many interesting models, including the running example PTM:

```
vars   collected bill
states start idle waitingForMoney paid
events cardInserted coinInserted

transitions
start => idle / collected = 0 bill = 3;

idle => waitingForMoney cardInserted;

waitingForMoney => waitingForMoney  coinInserted
        [collected < bill - 1 ] / collected += 1;

waitingForMoney => paid coinInserted
        [collected == bill - 1] / collected += 1;

paid => idle;
```

Adding Invariants. The semantic problem of the *PTM* was due to the fact, that it has been forgotten to set the value of `collected` to `0` when the transition from `paid` to `idle` is fired. In other words, the PTM only works correctly, if the variable `collected` has the value `0` when the system is in the state `idle`.

The statechart language of Yakindu does not offer the possibility to formulate invariants. In SMINV, the specification of state invariants is made possible by adding an additional element to the start rule of our grammar. The complete start rule of SMINV looks as follows:

```
SminvModel:
    vd=VarDecl
    sd=StateDecl
    ed=EventDecl
    td=TransDecl
    (id=InvDecl)?;
```

The definition of invariants is enabled by the following rules of the grammar:

```
InvDecl:
    {InvDecl} 'invariants' invs+=Inv*;

Inv:  state=[State] ':' inv=Term ';';
```

An invariant is an arbitrary term that has been attached to a state. However, this term must of type boolean, what is checked by an ordinary syntax check in SMINV's validator class.

Finally, we can now formulate the invariant for state `idle` formally.

```
invariants
idle : collected == 0;
```

However, in order to be able to prove the invariant, we should first fix the bug in PTM and add an update to the transition from `paid` to `idle`:

```
paid => idle / collected = 0;
```

3.2 Semantic properties to be verified

The verification of statecharts and related formalisms has been and still is a research topic for many authors [7,8]. The goal for this paper is to demonstrate how a language designer can implement verification support according to her needs for her user-defined DSL. For the language SMINV, we consider the following semantic model properties worth to be checked.

Invariant-Preserving Transitions. An state invariant is a boolean term expressing a constraint on the values of state variables (e.g. `v1 > 4`). By attaching a state invariant to a single state one claims, that a running state machine will satisfy the constraint on the values of state variables, whenever the machine is in the corresponding state. States without an attached invariant have always the implicit invariant `true`. The start state must not have any invariant attached.

By a simple induction argument, one can prove that all invariants are satisfied in all reachable states, if each transition establishes the invariant of the post-state.

Deterministic Transitions. It is also important to know whether the specified state machine works *deterministically* (recall that generation of implementation code from a state machine is in most cases meaningless when a specified state machine can behave non-deterministically). A non-determinism occurs when the system could change its state to more than one post-state upon receiving an event. This could happen, if a state exists that has at least two outgoing transitions that are triggered by the same event and whose guards overlap.

Alive Transitions. A state machine should have only *alive* transitions, because *dead* transitions are never executed. A transition is dead, if the constraints expressed by the transition's guard and the invariant of its pre-state are disjoint. In other words, it is not possible to find such values for the state variables that both the invariant of the pre-state and the guard are evaluated to *true*.

3.3 Proof Obligations

All semantic model properties described above can be formulated in form of proof obligations for model elements (for SMINV, either states or transitions). A proof obligation is a first-order formula with interpreted symbols for arithmetic operators. In the following, we formulate the proof obligation for each of the above described semantic model properties formally.

Invariant-Preserving Transitions. In order to prove that an invariant I_S for state S holds, one has to show that for all transitions t, which are incoming in state S, the update annotated on t is sufficient to establish the invariant I_S. Note that the start-state of the state machine must not be annotated by any invariant (start-states always have *implicitly* the invariant `true` annotated).

One can assume that the invariant annotated to the pre-state of t holds as well as the guard annotated to t. More formally:

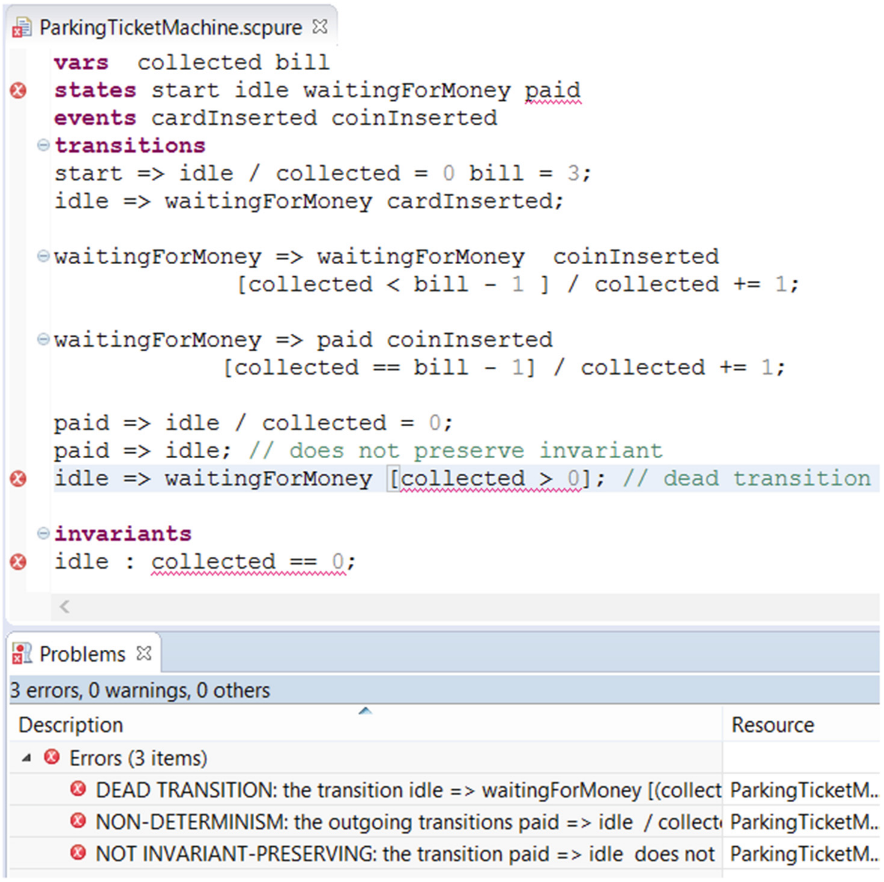
Let Σ be the set of all states and S an element from Σ. For any such S, I_S denotes the invariant annotated to state S and $I_S[v \leftarrow \mathit{update}(t)]$ denotes the substitution of all variables in I_S according to the update annotated on transition t. Furthermore, $\mathit{guard}(t)$ denotes the guard annotated to t (*true* is taken when no guard is specified). By $\mathit{pre}(t), \mathit{post}(t)$ the pre/post-state of transition t is denoted and $\forall\overrightarrow{v}$ means the all-quantification of all state variables *v1*, *v2*, ...

Then, one has to prove the following *proof obligation* for all transitions t:

$$\forall\overrightarrow{v}\ (I_{pre(t)} \wedge \mathit{guard}(t)) \longrightarrow I_{post(t)}[v \leftarrow \mathit{update}(t)] \tag{1}$$

Deterministic Transitions. In order to prove a state machine being deterministic, one has to show for all transitions t_1, t_2 that have the same pre-state and that are triggered by the same event:

$$\forall\overrightarrow{v}\ I_{pre(t_1)} \longrightarrow \neg(\mathit{guard}(t_1) \wedge \mathit{guard}(t_2)) \tag{2}$$



ParkingTicketMachine.scpure

```
vars  collected bill
states start idle waitingForMoney paid
events cardInserted coinInserted
transitions
start => idle / collected = 0 bill = 3;
idle => waitingForMoney cardInserted;

waitingForMoney => waitingForMoney  coinInserted
             [collected < bill - 1 ] / collected += 1;

waitingForMoney => paid coinInserted
            [collected == bill - 1] / collected += 1;

paid => idle / collected = 0;
paid => idle; // does not preserve invariant
idle => waitingForMoney [collected > 0]; // dead transition

invariants
idle : collected == 0;
```

Problems

3 errors, 0 warnings, 0 others

Description	Resource
Errors (3 items)	
DEAD TRANSITION: the transition idle => waitingForMoney [(collect	ParkingTicketM..
NON-DETERMINISM: the outgoing transitions paid => idle / collect	ParkingTicketM..
NOT INVARIANT-PRESERVING: the transition paid => idle does not	ParkingTicketM..

Fig. 2. Semantic error messages in editor for SMINV

Alive Transitions. In order to prove a transition t being alive, one has to show the satisfiability of its guard together with the invariant of the pre-state. More formally:

$$\exists \overrightarrow{v}\ I_{pre(t)} \wedge guard(t) \tag{3}$$

3.4 Implementation

The language SMINV as well as the corresponding toolset is free software, the sources of this software are made available on GitHub and can be downloaded from [5]. Figure 2 shows the editor running on the PTM-example, with some additional faulty transitions to cause some validation errors (note the error markers in the text as well as in the view *Problems*).

The semantic proof obligations discussed above have been implemented by `@Check`-annotated methods in the validator class of the language SMINV. The validator class is a standard class in the Xtext-framework to be implemented by each DSL separately. The validator also contains all syntax checks for the language.

The generation of proof obligations itself is delegated to another class, whose implementation in Xtend is surprisingly short. For example, the proof obligation for checking that transitions *t1* and *t2* (having the same pre-state and same event) do not cause an non-determinism (i.e. they cannot fire both in the same situation) is implemented as:

```
def getPO_NonDeterminism(Transition t1, Transition t2) {
  t1.pre.invariantsCopyConjunction
    .implies(neg(t1.guardCopy.and (t2.guardCopy)))
}
```

The generated proof obligation is passed to an automatic theorem prover in order to prove or disprove the obligation. Currently, SMINV supports only the prover Princess [9,10], but other suitable provers could be integrated as well.

Princess has been chosen for

- its excellent results for proof tasks on integer arithmetic
- its support of ordinary first-order predicate logic with uninterpreted symbols
- its ability to provide counterexamples for non-provable tasks

The input syntax of Princess is quite similar to the syntax for expressions in SMINV, only some logical operators have to be substituted (e.g. `&&` by `&`). An example for the Princess input from a proof obligation generated for the PTM-example is:

```
\universalConstants{ int collected; int bill;}
\problem{true -> ! ( collected < bill−1 & collected =
bill−1)}
```

The experimental results obtained when using Princess are very encouraging. All proof obligations generated for the PTM example could be proved/disproved within 2 ms – 8 ms on a Windows8 notebook (1.8 GHz, 8 GB RAM). Thus, the DSL editor can give instant feedback to the user also for semantic checks.

4 Discussion and Related Work

In the past, providing verification support for modelling languages was rather a task for either research groups or for companies selling tools for general purpose languages. The effort of implementing verification techniques only pays off when a language has a broad user base. In contrast, DSLs developed in industry for a very specific purpose often have very few users. One cannot expect tool builders to offer any dedicated support for single DSLs. Consequently, tool support has to be provided by the creator of the language herself.

There is another reason why verification support is less common for DSLs than for general purpose languages. It is not always clear what a (formal) semantics of a DSL could look like, though some DSLs has been recently made available, whose purpose is to define the semantics of other DSLs formally. For Xtext, such a DSL is Xsemantics [11]; the corresponding DSL for Spoofax [12] is called DynSem [13]). For general purpose languages, a common understanding of the language is evolving over time. This process eventually results in commonly accepted documents describing an informal or even formal semantics [14]. During this process, bugs, mistakes, and inconsistencies might be found by the broad user community. The semantics evolves over time. For DSLs, writing up the semantics might result in similar inconsistencies as in case of writing up the semantics of general purpose languages.

5 Conclusion and Future Work

DSLs allow to model succinctly for a certain purpose, but this flexibility also often means to abandon classical semantics for modelling languages. Consequently, semantic checks are currently neglected by DSL-definition frameworks such as Xtext, while syntax checks are very common and widely used.

The goal of this paper is to demonstrate that Xtext allows to implement DSL-specific semantic checks with moderate effort. Technically, semantic checks are realized analogously to syntax checks within the DSL's validator class. Thanks to the speed of the used theorem prover Princess, also semantic checks can give instant feedback to the user. From the user's perspective, semantic checks make an editor much more intelligent, since they can detect *semantic errors* the user has made.

In future, we plan to address the following issues. Firstly, an integration into the tool Yakindu would make this tool much more usable. One challenge here is to extend the verification support to the expression language used by Yakindu, which is more expressive than those of SMINV. Secondly, we plan to extend SMINV and to include other language constructs, e.g. nested states and `else`-guards for transitions. This will result in a more user-friendly DSL to describe state machines. Finally, besides Princess, other theorem provers supporting arithmetics and first-order logic should be integrated.

References

1. Aßmann, U., Bartho, A., Bürger, C., Cech, S., Demuth, B., Heidenreich, F., Johannes, J., Karol, S., Polowinski, J., Reimann, J., Schroeter, J., Seifert, M., Thiele, M., Wende, C., Wilke, C.: Dropsbox: the Dresden open software toolbox - domain-specific modelling tools beyond metamodels and transformations. Softw. Syst. Model. **13**(1), 133–169 (2014)
2. Yakindu: Homepage. http://statecharts.org/
3. Xtext: Homepage. http://www.eclipse.org/Xtext/
4. Harel, D.: Statecharts: a visual formalism for complex systems. Sci. Comput. Program. **8**(3), 231–274 (1987)
5. Baar, T.: SSMA - Simple State Machine Analyzer. https://github.com/thomas-baar/simplesma
6. Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend. Packt Publishing, Birmingham (2013)

7. Ghezzi, C., Menghi, C., Sharifloo, A.M., Spoletini, P.: On requirement verification for evolving statecharts specifications. Requir. Eng. **19**(3), 231–255 (2014)
8. Prashanth, C.M., Shet, K.C.: Efficient algorithms for verification of UML statechart models. JSW **4**(3), 175–182 (2009)
9. Rümmer, P.: Princess homepage. http://www.philipp.ruemmer.org/princess.shtml
10. Rümmer, P.: A constraint sequent calculus for first-order logic with linear integer arithmetic. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 274–289. Springer, Heidelberg (2008)
11. Bettini, L.: Xsemantics Documentation (2015). http://xsemantics.sourceforge.net/documentation/
12. Wachsmuth, G., Konat, G.D.P., Visser, E.: Language design with the Spoofax language workbench. IEEE Softw. **31**(5), 35–43 (2014)
13. Vergu, V.A., Neron, P., Visser, E.: Dynsem: a DSL for dynamic semantics specification. In: Fernández, M., (ed.) 26th International Conference on Rewriting Techniques and Applications, RTA 29 to 1 July 2015, Warsaw, Poland, vol. 36 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 365–378, June 2015
14. Object Management Group: Unified Modeling Language (UML), version 2.5, June 2015. http://www.omg.org/spec/UML/2.5/