Automated Testing Support for Reactive Domain-Specific Modelling Languages

Bart Meyers, Joachim Denil, István Dávid
Modelling, Simulation and Design Lab
University of Antwerp, Belgium
bart.meyers@uantwerpen.be
joachim.denil@uantwerpen.be
istvan.david@uantwerpen.be

Hans Vangheluwe
Modelling, Simulation and Design Lab
McGill University, Montréal, Canada
University of Antwerp, Belgium
hv@cs.mcgill.ca

Abstract

Domain-specific modelling languages (DSML) enable domain users to model systems in their problem domain, using concepts and notations they are familiar with. The process of domain-specific modelling (DSM) consists of two stages: a language engineering stage where a DSML is created, and a system modelling stage where the DSML is used. Because techniques such as metamodelling and model transformation allow for an efficient creation of DSMLs, and using DSMLs significantly increases productivity, DSM is very suitable for early prototyping. Many systems that are modelled using DSMLs are reactive, meaning that during their execution, they respond to external input. Because of the complexity of input and response behaviour of reactive systems, it is desirable to test models as early as possible. However, while dedicated testing support for specific DSMLs has been provided, no systematic support exists for testing DSML models according to DSM principles.

In this paper, we introduce a technique to automatically generate a domain-specific testing framework from an annotated DSML definition. In our approach, the DSML definition consists of a metamodel, a concrete syntax definition and operational semantics described as a schedule of graph rewrite rules, thus covering a large class of DSMLs. Currently, DSMLs with deterministic behaviour are supported, but we provide an outlook to other (nondeterministic, real-time or continuous-time) DSMLs. We illustrate the approach with a DSML for describing an elevator controller. We evaluate the approach and conclude that compared to the state-of-the-art, our testing support is significantly less costly, and similar or better (according to DSM principles) testing support is achieved. Additionally, the generative nature of the approach makes testing support for DSMLs less error-prone while catering the need for early testing.

Categories and Subject Descriptors D.2.2 [*Design Tools and Techniques*]: Domain-Specific Modelling; D.2.5 [*Testing and Debugging*]: Testing for DSM

Keywords Domain-Specific Modelling, Language Engineering, Model Testing

1. Introduction

In domain-specific modelling (DSM) (Kelly and Tolvanen 2008) the general goal is to provide means for domain users to model systems using concepts and notations they are familiar with, in their problem domain. Techniques such as metamodelling and model transformation enable modelling language engineers and domain experts to create domain-specific modelling languages (DSMLs) for the domain users. Because syntax and semantics of DSMLs are precisely defined by means of metamodelling and model transformation, models can be used for analysis, simulation, optimisation, documentation and even full code synthesis. This means that a DSM development process is split into two tasks: (i) the creation of a DSML known as *engineering a language*, by the *language engineer*, in consultation with a domain expert, and (ii) modelling the system using this DSML by a problem domain (but not solution domain) user, referred to as a *domain user*. It is shown that DSM significantly improves productivity, both for language engineering (Kelly and Tolvanen 2008) and for system modelling (Safa 2006). This makes DSM a suitable prototyping technique.

Although DSM prevents users from making certain mistakes by restricting the DSML's syntax, there is a need for testing more complex behaviour of the modelled system (Visser 2007; Safa 2006). DSMLs have been used to enhance testing processes (Kanstrén 2013), and extensive research has been carried out in testing exogenous model-to-model trans-

SLE'16, October 31 – November 1, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4447-0/16/10...
http://dx.doi.org/10.1145/2997364.2997367

formations (Baudry et al. 2010; Burgueño 2015). However, no systematic DSM solution exists for testing systems modelled using a DSML. Current resolutions are (a) manually executing a number of tests by executing the model under circumstances determined by the test, (b) testing in a different, less appropriate, formalism (*e.g.,* creating tests for code generated from the model) (King et al. 2014; Risoldi 2010), thus diverting from the user-friendly DSM principle, or (c) creating a DSML for testing, which needs to be developed and maintained (Smith et al. 2001; Puolitaival et al. 2011). All three resolutions severely hamper the usability for DSM as a domain user-friendly prototyping technique.

Problem statement. No systematic DSM solution exists for specifying tests at a domain level, matching the domain at which reactive design models are specified. Such domain-specific testing approach requires support for executing these tests, and reporting the test results. Additionally, the approach should be in line with early prototype development.

Research question. Can we find a systematic DSM solution with support for testing models in a reactive DSML? The solution should include the specification of tests, their execution and result reporting. The domain user should only be exposed to domain-specific concepts. The solution should weigh as little as possible on the lightweight DSM process, and should be an extension of the existing DSM techniques for design languages.

The contributions of the paper are twofold. Firstly, we define and generate a *testing DSML*. This testing DSML uses similar domain-specific concepts, familiar to domain users. The testing DSML is generated from the DSML definition, enriched with annotations (Meyers et al. 2014). These annotations provide sufficient information to generate a testing DSML. Secondly, annotations on the DSML's definition enable the automatic execution of modelled system and test model in concert. The approach is generic, and can be applied to any reactive, non-deterministic DSML of which the syntax is modelled as a metamodel, and the operational semantics modelled as a rule-based transformation that changes the system state. Thanks to the expressiveness of graph rewriting, this covers a large class of systems. The generated testing DSML and its semantics are explicitly modelled, allowing users to customise the testing DSML if needed.

This paper uses a DSML for elevator controllers as running example. We present an informal evaluation of the approach, discussing execution performance (in terms of computational complexity), and modelling performance (*i.e.,* how well the approach evaluates with respect to (a) adhering to DSM principles, and (b) being adequate as an early prototyping technique).

We use two different DSMLs in our approach, one for designing a system and one for modelling tests. Although these DSMLs are related, they are two separate languages. Throughout the paper, if confusion may arise about what kind of DSML or model we refer to, we use the term *dDSML* to denote a traditional DSML for designing a system, and *tDSML* to denote the related testing DSML that is presented in this paper. Analogously, we refer to *design models* and *test models*.

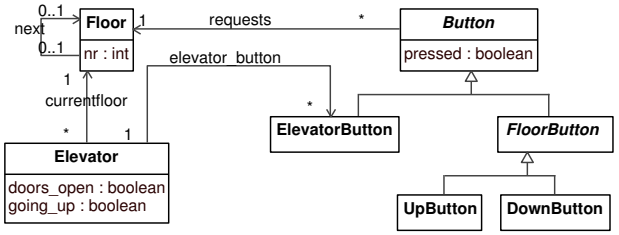

Figure 1. The metamodel of the Elevator DSML.

The paper is structured as follows. Section 2 introduces the necessary background and running example. In Section 3 the contribution of this paper is presented: an approach to automatically generate an operational testing framework from a DSML definition. The approach is evaluated informally in Section 4. Related work is discussed in Section 5 and a conclusion and outlook to future work are given in Section 6.

2. Background and Running Example

This section presents the prerequisites to discuss the approach, and introduces the running example, a DSML for elevator controllers.

2.1 DSML Engineering

The three main aspects of a DSML are its *abstract syntax* (describing the internal structure of a model, as a typed abstract syntax graph), the *concrete syntax* (describing how a model is represented, *e.g.,* in 2D vector graphics or in textual form) and its *semantics* (describing what a model means) (Mosterman and Vangheluwe 2004).

The Elevator DSML that will be used as running example enables modelling a building with floors, elevators and buttons. Additionally, it defines the operational semantics: the step-wise (discrete-time) behaviour of an elevator system, such as moving up or down to a different floor, closing or opening elevator doors, or pressing buttons.

Syntax. Figure 1 shows the metamodel of the Elevator DSML which we will call *E*. Elevators move between Floors, responding to Button press requests. A Button requests exactly one Floor. Floors are ordered by the next association and a derived attribute nr representing the Floor number. At any time, an Elevator is at exactly one Floor, modelled by the currentfloor association. An ElevatorButton is a button inside an Elevator, allowing a passenger to request going to a certain Floor. At every Floor, there can be an UpButton to request to go up and a DownButton to request to go down. An Elevator can have its doors open (in that case it cannot move) and has a direction (up or down).

Instance models of the DSML are said to *conform to* the metamodel. In (Kühne 2006), this relation is referred to as a *linguistic instance of*, and throughout this paper, *instance*

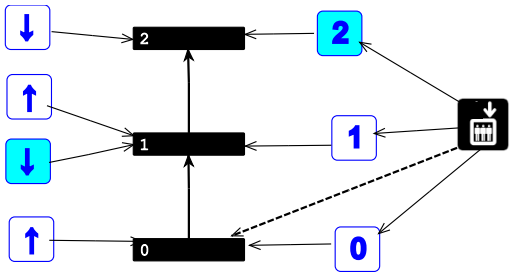

Figure 2. An Elevator model, instance of the Elevator DSML.

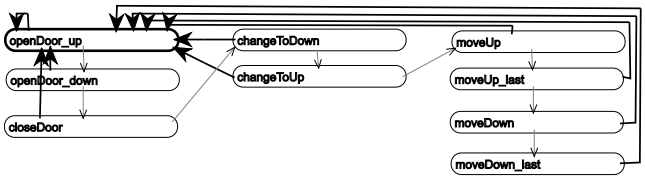

Figure 3. Rule schedule of the operational semantics $E_{[[.]]}$ of the Elevator example.

(of) will refer to this kind of instantiation. Figure 2 shows an instance model with three floors, one elevator and seven buttons, that uses the concrete syntax. As defined in the concrete syntax model (not depicted), pressed buttons have a blue fill, and they are connected to the floor they request. On the middle floor, a button is pressed by someone requesting to go down, and inside the elevator the button to go to the top floor has been pressed. The elevator is currently at the bottom floor. Its doors are open and its current direction is down. Note how all elements, including the links, conform to the metamodel of Figure 1.

Semantics. In this paper, semantics are described as operational semantics. Operational semantics capture explicitly how a model can be executed, which is effectively mapping a model onto an execution trace. In our approach, operational semantics are always formalised as transformations, that can transform (an) input model(s) to (an) output model(s), instances of the same or different languages. Since models are represented as graphs, a popular way to specify transformations is by means of graph rewrite *rules*.

The ordering of rules constituting the operational semantics are captured by the *rule schedule* of the transformation language. Rules consist of a *left-hand side* (LHS) containing a pattern representing a condition, and a *right-hand side* (RHS) containing a pattern representing an action (elements can be created, removed or updated). LHS and RHS are generic language constructs that can contain elements, each displayed as a differently shaped container. The contained elements form graph patterns that reuse concrete syntax taken from the input and output language. When a rule is *evaluated*, a *match* for the LHS is searched for in the input model. If a match is found, the RHS is *applied* to the input model, thus changing it. If the rule *fails* to match, the input model is left untouched. Depending on the outcome (failure or application), the next rule according to the rule schedule is evaluated. Figure 3 shows

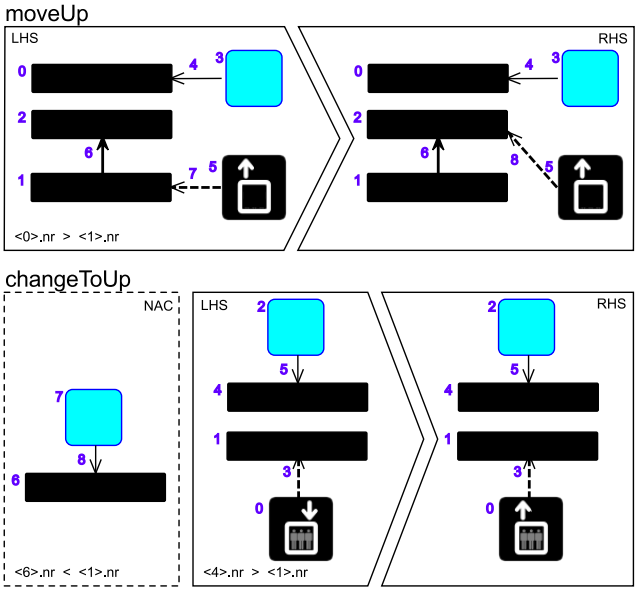

Figure 4. Two rules taken from the operational semantics of the Elevator example.

the rule schedule specifying the Elevator DSML's operational semantics, that determines how different rules are scheduled. Execution starts at openDoor_up (depicted as a rule with thick black stroke). In the case that a rule was applied, an outgoing success transition in the schedule (depicted as a black arrow) is followed. If no match is found, the input model is unchanged and an outgoing notApplicable transition (depicted as a thin grey arrow) is followed.

Inspired by a real elevator controller, the following rules implement how the elevator changes floors (one at a time), and opens and closes its door to honour the requests of users (modelled as pressed buttons). Two of the rules implementing this behaviour are shown in Figure 4. When a request for a floor is made for a different floor than the elevator's current floor, the doors close so that the elevator can start moving (the closeDoor rule). The case where the elevator is moving up (*i.e.,* changes its currentfloor link) is shown in the moveUp rule. The number labels on the top left of each pattern element serve as the relationship between LHS and RHS. Elements in the RHS with the same label as elements in the LHS, are the same elements. Differences between LHS and RHS represent the effects of a rule application. In the moveUp rule, the RHS depicts that the Elevator is now connected to another Floor. A condition is shown at the bottom of the LHS, stating (using EOL syntax (Kolovos et al. 2006), enriched with label placeholders) that the nr attribute of the Floor with label 0 must be larger than the nr attribute of the Floor with label 1. Related rules (not depicted) are moveDown (the dual of moveUp), moveUp_last (where the lit button is on the next floor), and moveDown_last (the dual of moveUp_last). Pressed buttons unlight when the door opens at a requested floor and the elevator goes in that direction in the openDoor_up rule (in the case the elevator is going up) and its dual, the

openDoor_down rule. The elevator only changes its direction if there are no more requests on its path. This can be seen in the changeToUp rule, which contains a negative application condition (NAC) (visualised as a dashed box) in addition to its LHS. The NAC states that the changeToUp rule matches only if no button on a lower floor is lit.

If the operational semantics are executed with the instance model of Figure 2 as input, then the first rule that is applicable is the changeToUp rule. Subsequently, the closeDoor, moveUp, openDoor, closeDoor, moveUp and openDoor are applied. If no rule is applicable, the transformation terminates.

A more detailed description of the Elevator running example can be found in (Meyers 2016).

2.2 Transformation Models and Higher Order Transformations

Transformations can be explicitly specified as transformation models (Bézivin et al. 2006), in a language that combines generic transformation concepts such as rules and a rule schedule, and concepts specific to the languages it transforms. A process called RAMification to generate such a rule-based domain-specific transformation language for given input and output DSMLs is presented by Kühne et al. (Kühne et al. 2009).

As can be seen from Figure 4, the modelling language for rules is composed from some generic language constructs for LHS, RHS and NAC, each displayed as a different shape of container. They include a constraint or action, and domain-specific language constructs that borrow syntax from the DSML. This transformation language (*i.e.,* its metamodel and concrete syntax model) can be generated using a transformation that takes the DSML metamodel and concrete syntax model as input and produces the metamodel and concrete syntax model of the transformation language as output, called the RAMification transformation (Kühne et al. 2009): Since the result of the RAMification transformation is a language definition with metamodel and concrete syntax model, the language engineer can adapt the concrete syntax model to make rules more appealing.

Because transformations themselves are explicitly modelled, and written in a modelling language, transformation models conform themselves to a transformation language, that has abstract syntax and concrete syntax as described above. Consequently, transformation models can be transformed in their own right, or can be generated. The transformations that have transformation models as input and/or output are called *higher order transformations* (HOTs). Similarly, abstract syntax and concrete syntax of a language are modelled as metamodels (in the class diagram language) and concrete syntax models (in *e.g.,* an icon language) respectively, and can thus be transformed as well.

A more detailed description of domain-specific modelling and model transformation in the DSM tool *AToMPM* can be found in (Mannadiar 2012).

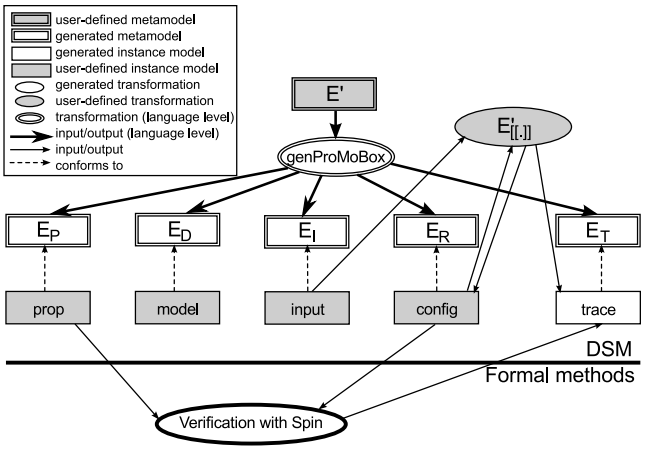

Figure 5. Overview of the *ProMoBox* approach.

2.3 The *ProMoBox* Approach for Language Engineering

The approach presented in this paper builds on the *ProMoBox* approach (Meyers et al. 2014, 2013b; Deshayes et al. 2014; Meyers 2016). *ProMoBox* stands for "Properties and (design) Models developed (Boxed) in concert". The language engineering support of *ProMoBox* consists of the following three parts, and is illustrated in Figure 5.

The *ProMoBox* Sublanguages. *ProMoBox* replaces the traditional DSML with five sublanguages (each DSMLs) for modelling all artifacts that are needed to specify and verify properties (Meyers et al. 2014). The five sublanguages are the following:

- A *design language* (E_D in Figure 5) for design modelling as supported by traditional DSMLs. With this language, the static structure (*i.e.,* language concepts that do not change at run-time) of the system is modelled. In case of Elevator, this includes all Elevator concepts, without the currentfloor association, nor the doors_open, going_up and pressed attributes;
- A *run-time language* (E_R in Figure 5) for run-time state representation. The run-time language always includes all elements of the design language, plus dynamic state information that can change at run-time. Run-time instances are always associated with a design instance with the same static structure. One design instance possibly has multiple run-time instances corresponding with it, representing all possible states of the model. Note that in traditional DSM, the DSML often includes run-time concepts, meaning that no distinction is made between static structure and dynamic state. The running example was also presented including dynamic state information. In fact, the metamodel shown in Figure 1 is the same as Elevator's run-time language, and its instance Figure 2 is a run-time model in *ProMoBox*. Hence, throughout this paper, a dDSML (*i.e.,* a DSML for designing a system) will refer to the run-time language in the context of *ProMoBox*;

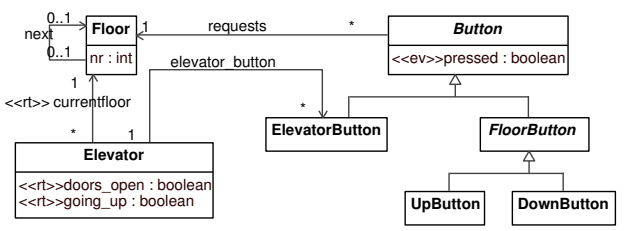

Figure 6. The annoted metamodel E' of the Elevator example.

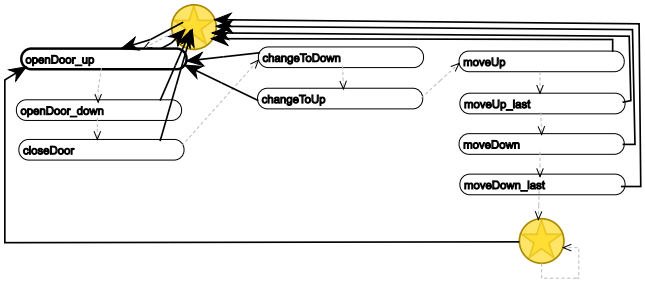

Figure 7. Annotated rule schedule of the operational semantics $E'_{[[.]]}$ of the Elevator example.

- An *input language* (E_I in Figure 5) to model event-based input (to model the environment in which the system operates). This language will not be relevant in this paper;
- A *trace language* (E_T in Figure 5) for state-based output representation (to model an execution trace of the system or verification counterexample). An execution trace is a sequence of run-time states connected with transitions that represent execution steps (*i.e.,* operational semantics' rule executions). The trace language can be used to represent execution traces of a simulation. A trace model is usually generated by a simulator or as a counterexample by a verification tool.
- A *property language* (E_P in Figure 5) for property specification (to model temporal or structural properties). This language will not be relevant in this paper.

Generating the Sublanguages. As the traditional DSML is replaced by five languages (*i.e.,* DSMLs), it would be time consuming to keep these intimately related sublanguages presented above consistent. Therefore, a fully automated method generates these sublanguages from a single DSML specification, keeping the five sublanguages consistent by construction. To be able to generate these sublanguages, we extend metamodelling and model transformation languages with annotations, to add necessary information for every language construct and to introduce the concept of a simulation step. The annotated metamodel of the Elevator example (E' in Figure 5) is shown in Figure 6.

Whereas the traditional operational semantics of Figure 3 has a DSML instance as input (*i.e.,* the model in its initial state), and produces a DSML instance (*i.e.,* the model in its final state), the operational semantics in *ProMoBox* ($E'_{[[.]]}$ in Figure 5) takes a run-time model (initial state, generated or manually created from the design model) and an input model (input events) as input, and generates a run-time model (the final state) and a trace model (simulation trace). This means that the operational semantics have to define *when* (*i.e.,* at what point in the rule schedule) a new input event is applied to the system, and a new state is added to the execution output trace. This is done by two types of annotations to the rule schedule in Figure 7: the input star representing a conceptual step and full or dashed transitions representing (not) writing to the output trace.

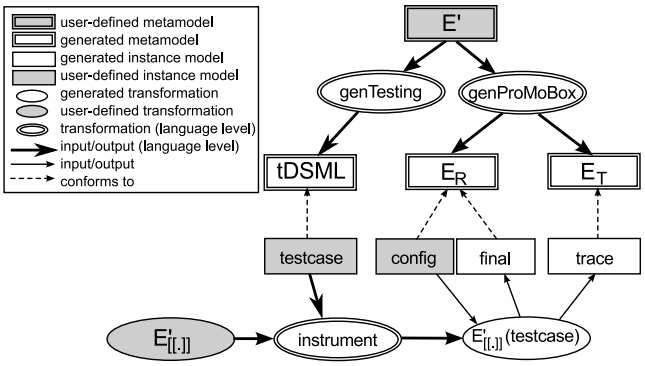

Figure 8. Overview of the generation and use of tDSML.

The Verification Backbone. The third part is the mapping to and from a verification backbone (Verification with Spin in Figure 5), making model checking automatically available for DSMLs. This part will not be relevant in this paper. We present testing as a valuable alternative to model checking, which suffers from inherent scalability issues. A more detailed description of the *ProMoBox* approach can be found in (Meyers 2016).

3. Support for Testing in DSM

In this section we introduce testing support for reactive DSMLs. In the resulting testing framework, the specification of testing scenarios is elevated to the domain-specific level, according to DSM principles. This way, similarly to designing systems with the dDSML, less technical domain users gain access to testing. Moreover, because of its generative nature, our approach does not require significant additional effort for the language engineer, both in terms of designing the framework as in keeping it consistent. The generated framework allows modelling and executing tests, and reporting test results. Throughout this section, we use the Elevator DSML as running example.

Figure 8 shows an overview of the testing framework as an extension of Figure 5, leaving out models that are irrelevant to our testing approach. Rectangles represent (meta)models, ovals represent transformation models. A shaded shape is manually created, an unshaded shape is automatically generated. Reflecting the two DSM phases, namely, language engineering and system modelling, thick arrows represent transformation input or output at the language engineering level, and thin arrows represent transformation input or output

at the system design level. Starting from the annotated metamodel E' (shown in Figure 6) and the dDSML's concrete syntax model, a testing DSML tDSML is generated. Subsequently, a testcase can be defined by the domain user as an instance of tDSML. From this testcase and the DSML's annotated operational semantics $E'_{[[.]]}$ (shown in Figure 7), an instrumented operational semantics $E'_{[[.]]}(testcase)$ is generated by the instrument transformation. The instrument transformation is a higher order transformation, as it produces a transformation model. $E'_{[[.]]}(testcase)$ contains the testcase, woven into the operational semantics. The testcase can now be executed with a tDSML model config as input, producing the system in its final state (final) as output after executing the testcase, and an execution trace. Note how in Figure 8, only the annotated metamodel E', operational semantics $E'_{[[.]]}$, the testcase and config models are user-defined (shaded). This means that in comparison to traditional DSM, in our approach, the only additional manual step is modelling a testcase. The generation of a tDSML, including its operational semantics, is automatic.

The remainder of this section is divided as follows. First, we introduce the tDSML for modelling testcases. Second, we explain how a tDSML is generated by the genTesting transformation. Third, we define the operational semantics of the tDSML. Fourth, we show how these semantics can be generated using the instrument HOT. Finally, we show how the generated framework supports execution of test suites using the generated $E'_{[[.]]}(testcase)$ transformation.

3.1 The Testing Language

The core of the testing framework is the testing language tDSML. Inspired by unit testing, the purpose of a tDSML is to provide a scenario, with a test goal (*i.e.,* an assertion). In the context of reactive systems, the scenario is driven by inputs at given points in time (*i.e.,* under specified circumstances). In our framework, a testcase is defined as a series of subsequent *input*s and *assert*s, which we call *phases*. A third kind of phase is a blocking *when* phase with condition, for which the test progresses only when the condition is satisfied. Branching is not allowed in a tDSML, because a testcase with branches represents in fact multiple testcases. Nevertheless, since the tDSML is explicitly modelled, such extensions can be made to the framework, and can be mapped to our syntax and semantics of a tDSML.

An instance of this tDSML is shown in Figure 9. It models a testcase called OpeningDoors, that contains four phases, and should be read from left to right. A testcase is executed together with the model, meaning that phases can be executed when the model accepts inputs (*i.e.,* when reaching an input star as shown in Figure 7). The testcase's execution semantics will be explained in more detail below. The first phase is an *input* event, depicted as a green circle. The input event models that the UpButton on the first floor is pressed. Note how the content of this phase, and all other phases, is a pattern much like a LHS pattern, and thus needs to be matched. In the case of an input event, the pattern is by convention matchable. Execution of this phase will result in setting the pressed attribute of the matched button to *true*. The second phase is a *when* phase, depicted as a diamond with "WHEN" at the top left corner. The *when* phase blocks further execution of the testcase until the condition is satisfied: the elevator has reached the first floor. The doors_open and going_up icons are greyed out, as their value does not matter for this condition. As such a condition might never be satisfied, a time-out is specified at the bottom right of the *when* phase: the testcase will fail if the testcase remains in this phase for 100 steps. After satisfying the condition, a new input event occurs *immediately* (denoted by the dashed transition), pressing an ElevatorButton at the second floor. The next time input is read, an *assert* phase is triggered. It states that the elevator must still be at the first floor, but with its doors open. The condition of the *assert* phase must be satisfied immediately upon reaching the phase, or the test case fails. Would the condition not have to be satisfied immediately, then the user can specify a time-out that defines the maximum time to meet the condition. The OpeningDoors testcase passes if the testcase runs to completion, including meeting the condition of the *assert* phase. The testcase fails if a time-out occurs during the *when* phase, or if the condition of the *assert* phase is not satisfied. Executing the testcase for the instance model of Figure 2 would result in the following sequence of rule or test phase applications: (1) the changeToUp rule, (2) the closeDoor rule, (3) the first phase of the testcase (because an input star is entered), (4) the moveUp rule, (5) the *when* phase of the testcase, (6) the third phase of the testcase, (7) the openDoor rule, (8) the *assert* phase, thus successfully ending the testcase.

Apart from the OpeningDoors testcase, the Elevator tDSML allows the user to model testcases with other than four phases, multiple assertions, long input sequences, etc.

Now that we introduced the Elevator tDSML by example, we introduce its definition. The metamodel of the Elevator tDSML is shown in Figure 10. It consists of a generic template (shaded), and a domain-specific part (unshaded). More in detail, it contains the following parts:

- *Test Engine:* A TestEngine runs multiple RunningTestCases, that keep track of the progress of testcases at runtime. The Status of a running testcase can be:
  - inprogress: the testcase is still running;
  - failed: the testcase finished and failed, either because of a time-out or a failed assert;
  - passed: the testcase finished successfully by passing the final assert phase;
  - error: an unexpected error occurred during execution, such as not matching an input event pattern;
  - notexecuted: the execution of the testcase did not yet start.

  The current phase, and optionally steps_spent and time_spent in the current phase are also tracked;

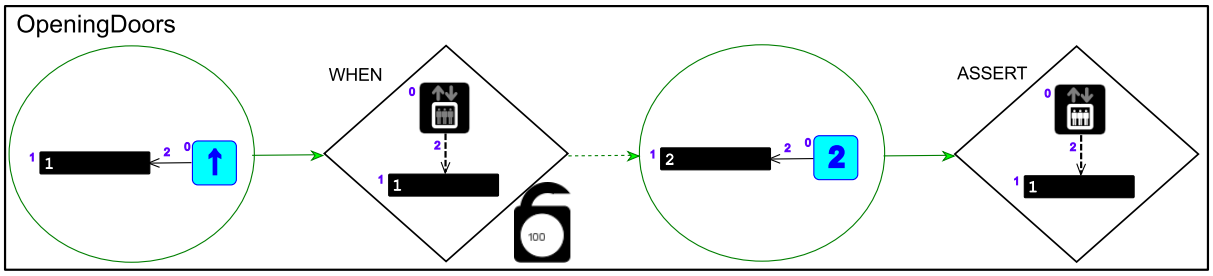

Figure 9. The OpeningDoors testcase, an instance of the tDSML.

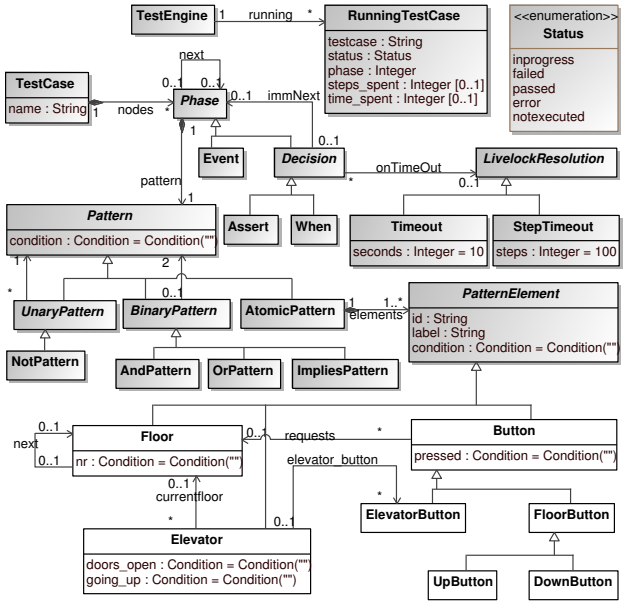

Figure 10. The generated metamodel of tDSML.

- *Testcase:* A TestCase contains a number of Phases, that may be connected with a next link to denote what Phase should be executed the next time input is read by the operational semantics. A Phase can be an input Event, an Assert or a blocking When phase. Assert and When may be connected to a Phase with a immNext link, denoting that the next input should be read immediately. Assert and When may have a LivelockResolution, which may be a real-time time-out, or a maximum number of steps as used in Figure 9;
- *Pattern:* Every Phase contains a Pattern, based on PaMoMo, a language for defining contracts (Guerra et al. 2010). A condition can be specified, which is by default the condition created from an empty string, that always returns *true*;
- *Pattern Elements:* the domain-specific part of the tDSML allows the user to specify patterns that can be matched in the dDSML model, similar to a transformation rule's LHS. All attribute types are Conditions, abstract classes are now concrete classes, and lower bounds of multiplicities are relaxed to 0.

Additional static semantics are defined for this metamodel:

- Event patterns are static, thus can always be matched. This means that they do not use run-time language elements apart from input language elements. In case of the Elevator tDSML, no conditions w.r.t. currentfloor, doors_open or going_up (all marked with *rt* in E') can be included in the pattern. pressed can be used as it is an input language element (marked with *ev* in E'), and has the special meaning of assigning the given value in the dDSML model.
- Phases can have maximally one outgoing and incoming next or immNext link.
- All Phases are connected, forming a step-by-step testcase.
- The final Phase is an Assert.

3.2 Generation of a tDSML

The genTesting transformation of Figure 8 generates the tDSML metamodel from an annotated metamodel using a template-based approach. The genTesting transformation takes the following steps to create the tDSML metamodel:

- apply the RAMification transformation (as explained in Section 2.2) to the dDSML metamodel of Figure 6 to create a pattern language of the dDSML;
- import the testing template (the shaded part of Figure 10);
- create an inheritance link from all top-level dDSML superclasses to the PatternElement class.

The generation of the concrete syntax (not visualised because of spatial constraints) follows a similar process. Step 3 is left out, so that the concrete syntax model of the tDSML is simply the union of the concrete syntax model of the RAMified dDSML and the concrete syntax template model.

3.3 The Operational Semantics of a tDSML

After informally introducing the semantics of a testcase in Section 3.1, conform to the tDSML, we present its complete operational semantics in this section.

The operational semantics of a testcase are modelled as an extension of the dDSML's operational semantics. The dDSML's operational semantics (shown in Figure 7) are "instrumented" so that the testcase runs in concert with the system. The instrumented operational semantics $E'_{[[.]]}(testcase)$ are shown in Figure 11. The main idea is that the input stars of Figure 7 are replaced with calls to a ProgressTestCase transformation, which takes one step of the testcase. Additionally, $E'_{[[.]]}(testcase)$ starts with two new rules. createTestEngine creates a TestEngine instance if none exists yet. initializeTestCase spawns a new RunningTestCase, in this example for

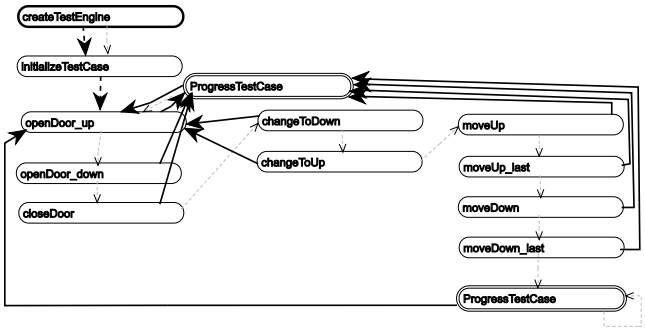

Figure 11. Rule schedule of the generated instrumented operational semantics $E'_{[[.]]}(testcase)$ of the tDSML.

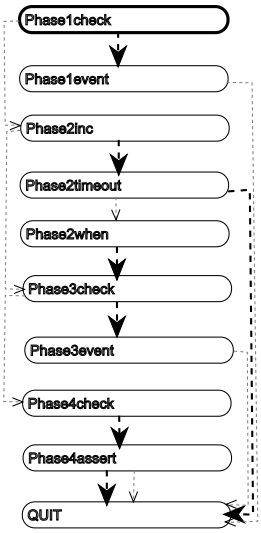

Figure 12. Rule schedule of the generated ProgressTestCase transformation.

the OpeningDoors testcase of Figure 9, and sets its testcase attribute to "OpeningDoors", status to inprogress, phase to 0, and optionally steps_spent or time_spent to 0.

After completing the above initialisation steps, the instrumented operational semantics are executed as usual. Upon entering a ProgressTestCase call, the ProgressTestCase transformation is executed from start to finish. ProgressTestCase executes one step of the OpeningDoors testcase, depending on the state (*i.e.,* current status, phase, steps and time spent) of its associated RunningTestCase. The schedule of the ProgressTestCase transformation is shown in Figure 12. All transitions are dashed, meaning that the execution of the testcase should not affect the system's execution trace. Its rules are shown in Figure 13:

- Phase1check: This rule checks whether the RunningTestCase must execute phase 1 next, corresponding to phase 1 of the OpeningDoors testcase shown in Figure 9. The LHS contains a TestEngine (visualised as cogwheels), and a connected RunningTestCase (with label e2). The condition at the bottom of the LHS states that the testcase attribute of the matched RunningTestCase instance must be "OpeningDoors", making sure that in case of multiple RunningTestCases, the right one is matched. Additionally, its current phase must be 0. If the rule matches, the first phase of the OpeningDoors testcase must be executed, so the success transition is followed and the Phase1event rule is evaluated. If the rule does not match, the Phase2inc rule is evaluated which checks whether the RunningTestCase is in phase 1.
- Phase1event: This rule implements phase 1 of the OpeningDoors testcase. Apart from the TestEngine and RunningTestCase, the LHS also contains a pattern similar to the first phase of OpeningDoors, thus matching the UpButton of the first Floor. The UpButton is half-shaded, to denote that its pressed attribute can have any value. If the rule matches (which should always be the case), the UpButton is pressed, and the RunningTestCase phase is set to 1. In that case, the ProgressTestCase transformation ends, because there is no outgoing success transition from Phase1event, and in Figure 11 the outgoing success transition of the respective ProgressTestCase call is

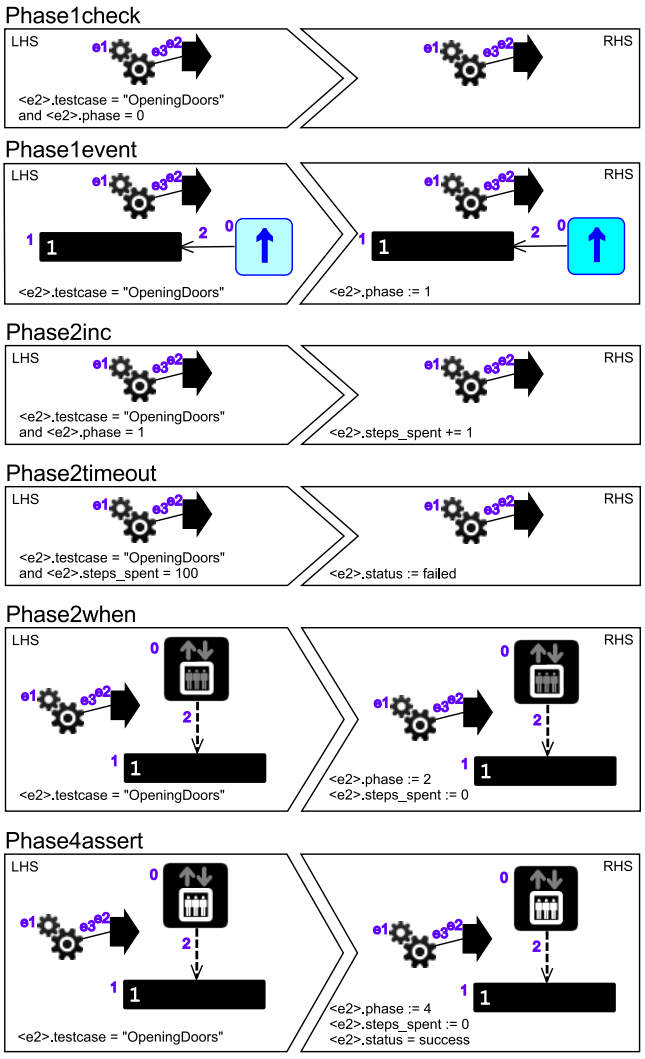

Figure 13. Generated rules of the ProgressTestCase transformation.

followed. The testcase has now completed phase 1, and phase 2 will be executed the next time ProgressTestCase is called. If the rule is not matched, it means that an erroneous pattern was given to the event (thus violating the tDSML's static semantics), and the QUIT rule is evaluated which terminates execution and sets the status to *error*.

- Phase2inc: This rule, and the next, manage the LivelockResolution (*i.e.,* time-out) of the *when* phase of Figure 9. This rule only matches if the current phase is set to 1. In that case, the steps_spent attribute is increased by 1, and next, the Phase2timeout is evaluated. If this rule is not matched (*i.e.,* the current phase of RunningTestCase is not 1), the Phase3event rule is evaluated.
- Phase2timeout: This rule checks whether the time-out condition of 100 steps has been reached. If so, the testcase has failed and the QUIT rule is evaluated, which terminates execution. If not, the Phase2when rule can be executed.
- Phase2when: This rule is the actual implementation of the *when* phase of Figure 9. If it matches (*i.e.,* the elevator is on the first floor), the RunningTestCase phase is set to 2 and the steps_spent attribute is reset. Then, the ModelPhase3check rule can be executed, because the third phase immediately follows the *when* phase. If no match is found, the ProgressTestCase ends.
- Phase3check, Phase3event and Phase4check: Similar to Phase1check and Phase1event, and not shown.
- Phase4assert: This rule implements the assert phase. It is similar to Phase2when, but if the rule matches, the RunningTestCase status is set to *success*. In any case, the QUIT rule will be evaluated.
- QUIT: This rule (not shown) always matches. If the status of the RunningTestCase is still *inprogress*, it sets it to *error* or *failed* (depending on the current phase), and terminates execution.

3.4 The Generation of a tDSML Operational Semantics

As explained above, a testcase can be described using operational semantics, represented as a model (as shown in Figure 11, Figure 12 and Figure 13). In this section, we describe how this transformation model can be fully automatically generated from the tDSML model of Figure 9, using a higher-order transformation (HOT) named instrument. This HOT is independent of the DSML for which the instrumented operational semantics are generated, so it is applicable to testcase models in any generated tDSML.

Figure 14 shows the rule schedule of the instrument HOT. In the left part of the instrument HOT, the annotated operational semantics are instrumented (resulting in Figure 11), the middle part is responsible for generating the ProgressTestCase rule schedule (Figure 12), and the rules (Figure 13) are generated by the right part. HOT rules with self loops use "for all" semantics, meaning that the rule should be applied for all matches. This is in contrast to a regular loop, where

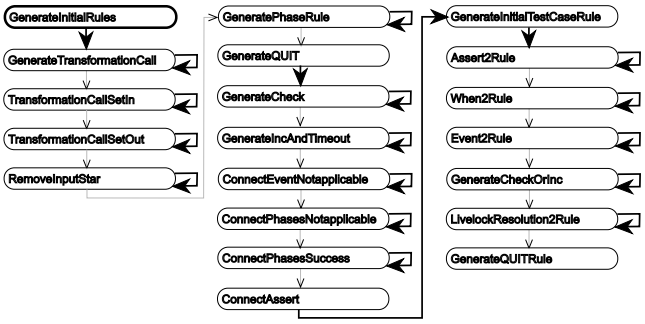

Figure 14. Rule schedule of the instrument HOT.

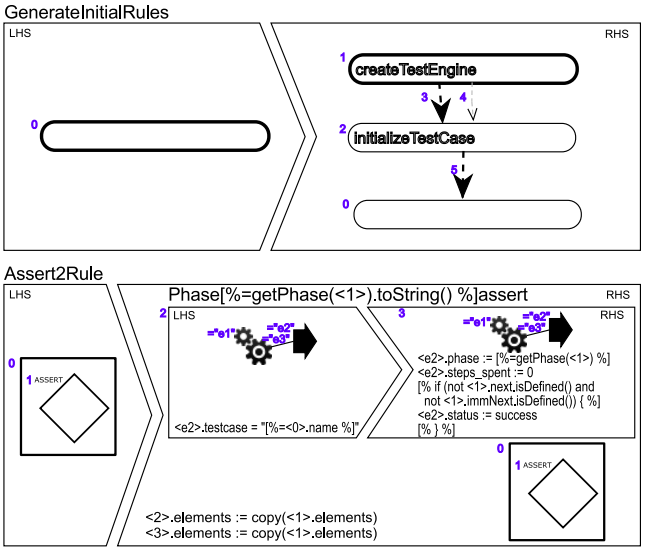

Figure 15. Rules of the instrument HOT.

after every rule application, the LHS is re-evaluated. The former can be easily emulated using a NAC that states that the RHS must not have been applied for the LHS match. Selected HOT rules are shown in Figure 15. For readability, NACs for emulating "for all" semantics are not shown. In the remaining of this section, we explain the rules shown in Figure 15[1].

GenerateInitialRules (shown in Figure 15): generates calls to the createTestEngine and initializeTestCase rules in Figure 11, and connected transitions. As shown in Figure 15, the initial rule is matched, and two new rule calls are generated, and connected to the existing rule. The existing rule is not the initial rule any more, but createTestEngine is;

Assert2Rule: generates the *assert* phase rules, and is shown in Figure 15. For example, the Phase4assert rule of Figure 13 is generated by this HOT rule. For every assert phase (label 1 in the LHS) in a tDSML testcase (label 0 in the LHS), a rule is generated consisting of LHS (label 2 in the RHS) and a RHS (label 3 in the RHS). The rule name and condition/action are expressed as Epsilon Generation Language (EGL) (Rose et al. 2008) code, a template language extension of EOL. The text should be interpreted as string,

[1] A full description of the HOT can be found at
http://msdl.cs.mcgill.ca/people/bart/sle2016.html

but the so-called *dynamic sections* between `[%=` and `%]` are replaced by its evaluated string value. Note that EGL is used for its simplicity and general applicability, but fully modelled HOT techniques should be preferred if possible as they take model conformance into account. The rule name uses a `getPhase` helper function that calculates the phase number:

```
operation getPhase(phase : Phase) {
  return Sequence{phase}.closure(e|A.allInstances()
                        .select(e2|e2.next=e)).size()+1;
}
```

The generated LHS condition and RHS action are similarly specified by EGL templates, including some dynamic sections. For instance, the statement setting the status to *success* is only generated if the *assert* phase does not have an outgoing link (*i.e.,* if it is the final phase). The pattern with TestEngine and RunningTestCase are generated in the generated LHS and RHS, and their label is set. Note how for these elements, for readability, the HOT labels are not visualised. Furthermore, the pattern of the matched *assert* phase (*e.g.,* the *assert* phase of Figure 9) needs to appear in the generated LHS and RHS. However, as the HOT can be applied to any tDSML, domain-specific concepts are unknown for the HOT. Therefore, it is impossible to include the pattern of the *assert* phase in the HOT rule. Instead, an action in the HOT RHS is added, stating that the elements in the matched *assert* phase (label 1) are copied and added to the elements of the generated LHS (label 2), and generated RHS (label 3);

3.5 Test Suite Execution

The $E'_{[[.]]}(testcase)$ transformation, generated from the testcase, can be executed to run the testcase. Optionally, the current state of a RunningTestCase can be visualised during execution by highlighting the current phase in the testcase model.

Multiple testcases can be collected in a test suite. A test suite consists of (runtime model, testcase) pairs, and can be executed automatically, by executing the instrument HOT for every (runtime model, testcase) pair, and executing the generated $E'_{[[.]]}(testcase)$ transformation. During this process, multiple RunningTestCases will be created and connected to one TestEngine, thus collecting all test results (status, phase, etc.). Individual tests can be inspected, and execution traces (generated by *ProMoBox*) can be consulted for each test case.

We modelled our approach in *AToMPM* (Syriani et al. 2013), a tool for modelling DSMLs and model transformations. *AToMPM* allows generating a fully functional test specification framework, including a modelling environment and execution environment, from language definitions and transformation models such as those described in this section.

4. Analysis of the Approach

In this section we analyse the approach by discussing the execution performance and the modelling process. Next, the prerequisites for implementing the framework are presented. Finally, assumptions and limitations of the approach are discussed.

4.1 Performance

We evaluated the performance of the approach in two ways: execution performance and modelling performance. Execution performance is evaluated in terms of computational complexity of the test execution process. Modelling performance is evaluated in terms of (a) adhering to DSM principles (*i.e.,* domain user-friendliness), and (b) increasing productivity, in order for the approach to be adequate as an early prototyping technique.

Execution performance. With n the number of test cases, s the number of steps (*i.e.,* rule evaluations) in the operational semantics of the dDSML and t the number of phases in the test model, executing a test suite as presented in this paper is in $\mathcal{O}(n \cdot s \cdot t)$. The complexity can be lowered easily to $\mathcal{O}(n \cdot s \cdot \log(t))$ if searching for the current test phase is implemented as a binary search. If the transformation schedule language allows transitions with guards, executing a test suite is in $\mathcal{O}(n \cdot s)$ (*i.e.,* the same complexity class as executing the operational semantics n times without testing), because the correct test phase can be chosen independently of the test model's number of steps or any other variable. Additionally, the performance of matching the testcase rules of Figure 13 can greatly benefit from using pivots (Kühne et al. 2009), which is a mechanism to pass variables (*i.e.,* previously matched objects) between rules. It can be concluded that our testing support is complementary to the model checking support presented in (Meyers et al. 2014), which has its limitations in scalability.

Modelling performance. The approach is designed to keep the manual language engineering input (creating a tDSML) as low as possible, as well as to support the domain user's work (using the tDSML). Comparing to state-of-the-art resolutions for testing design models (see Section 1), it can be concluded that our approach improves the modelling performance. **In comparison to manual testing:** When creating a DSML (in our approach this included the creation of a dDSML and a tDSML), the amount of manual work is slightly increased, since the metamodel and the operational semantics have to be annotated. This annotation process only requires a minimal amount of time, and has to be completed only once per DSML. When using a tDSML, the amount of manual work is decreased significantly as tests can be created in a dedicated language, automatically translated and executed. Regression testing is thus facilitated. **In comparison to testing using a different formalism:** When creating a DSML, the amount of manual work is decreased, because no mapping to a different formalism (*i.e.,* testing platform) needs to be implemented. In using a testing DSML, the difference in amount of manual work is difficult to assess. However, when testing is done in a different, already existing formalism (*e.g.,* in CO-OPN (Risoldi 2010)), it can be argued that this is not the most appropriate formalism for testing, as the dDSML's domain

concepts are not integrated in the language. This violates the very DSM principles that were followed when creating the dDSML. **In comparison to creating a DSML for testing:** When creating a DSML, the amount of manual work is dramatically decreased, as the tDSML is generated. When using a tDSML, it can be argued that the amount of manual work slightly increases, because a manually created DSML for testing can be tailored perfectly to the tester's needs whereas our tDSML covers generic testing needs. This gap can be closed however because the generated tDSML can be tailored itself, and might thus serve as a valuable starting point for a more specific testing DSML. In summary, our approach has clear advantages compared to the identified existing approaches in terms of creating and using a DSML for testing. As our approach focuses on automating the design of a tDSML, it is especially useful in quick prototyping of DSMLs.

4.2 Implementation Prerequisites

We presented a modelling approach in this paper that can be implemented in any modelling tool, provided some prerequisites. We aimed to limit the prerequisites for our approach, and identified the following:

- Metamodelling and rule-based model transformation has to be supported by the modelling tool, which is the case for most DSML modelling tools. Concrete syntax modelling, and explicit transformation rule schedules are not a strict prerequisite. Rule schedules can be emulated in rules by using variables, which might decrease the execution performance of the approach. The approach is currently only applicable for rule-based operational semantics, but the approach can be extended to other transformation tools if transformations can be generated. This requires a re-interpretation of the manual annotation step in order to identify conceptual steps in the DSML's behaviour (see Section 2.3. This is out of scope of this paper however.
- Metamodels and model transformations must be explicitly modelled, so that they can be input and/or output of model transformations themselves. This is a common prerequisite in modelling language engineering and is therefore one of the core features of *AToMPM* (Syriani et al. 2013).
- The technique of RAMification (Kühne et al. 2009), *i.e.,* generating a transformation language from (a) given metamodel(s), is a prerequisite. It can however be implemented in any tool that supports metamodelling, as illustrated by (Denil et al. 2014; Van Mierlo and Vangheluwe 2012).
- Not standard in graph rewrite rules, a copy operator must be available that allows you to copy an existing element in the RHS of a rule. The copy operator is used to generate the LHS as well as the RHS of a testing rule (see Figure 15).
- Not standard in graph rewrite rules, support for changing the type of an object to an equivalent (*i.e.,* structurally equivalent according to type theory (Pierce 2002), commonly known as "duck typing") type is a prerequisite. This technical requirement is used to migrate patterns that conform to the tDSML metamodel to the dDSML transformation language (see Figure 15). A pragmatic solution is a script that replaces the type name in the textual, saved model (*e.g.,* in xml-format).

4.3 Assumptions and Limitations

If the above prerequisites are met, a number of limitations apply. Firstly, nondeterministic dDSMLs are not supported. Our approach supports testing of nondeterministic models, as they are tested as regular models, which makes the test outcome nondeterministic. However, in case of nondeterminism in a design model, a tester is often interested in knowing whether the test passes for all possible execution paths of the system, or for at least one execution path. Model checking techniques as presented in (Meyers et al. 2014) can be used to verify all/any possible execution path(s) in the system. As an explicit input scenario is provided rather than having to cover all possible input scenarios, scalability will be less problematic as in (Meyers et al. 2014).

Secondly, real-time dDSMLs are not supported. Real-time models are models with discrete, timed events. In order to tackle this, time-outs may be added to rule transitions in the operational semantics (as *e.g.,* done in (Ölveczky 2014)), and time-out annotations may be added to next transitions in the test metamodel template. To execute both timed models (the design model and the test model) in concert, time calculus can be used (Boulanger et al. 2012).

Thirdly, continuous-time dDSMLs are not supported. Continuous-time models are models whose state continuously change over time. Graph rewriting is not appropriate for representing continuous-time behaviour. Therefore, the behaviour of the dDSML and the tDSML, represented by differential equations, needs to be combined. The technique of semantic adaptation may be used to combine these models (Meyers et al. 2013a).

In conclusion, we argue that variations of our approach can be applied to support different classes of reactive DSMLs. Still, because of the expressiveness of graph rewriting, the currently supported DSMLs cover a large class of reactive systems.

In comparison to the model checking approach in (Meyers et al. 2014) the limitations of boundedness and scalability are lifted. On the other hand, nondeterminism is not supported whereas it is supported in (Meyers et al. 2014), and the model checking approach provides certainty over a declarative property that spans several test cases, where in this approach, explicit test cases have to be modelled.

5. Related Work

Domain-specific modelling has been used in model-based testing (Kanstrén 2013). To address the needs of an agile development process, King *et al.* present Legend, a toolset of textual testing DSMLs (King et al. 2014). Similarly, Santiago

et al. use a textual DSML to tackle the inherent complexity of testing domain-intensive cloud applications (Santiago et al. 2013). In (Kloos and Eschbach 2010), Kloos and Eschbach use a DSML as domain experts need to describe tests for network-structured safety-critical systems.

Kanstrén and Puolitaival present a framework that involves the generation of a DSML from a test model to guide domain experts in creating test cases (Kanstrén and Puolitaival 2012). The generated DSML, or GUI alternative, can be used to express one test case, or multiple test cases. Similarly to our approach, the framework is generic, supporting languages with "transitions" and "guards", thus covering control-flow and data-flow test models. Contrary to our approach, the framework is textual, and a test model (that serves as metamodel) needs to be provided that acts as an API of the system under test. Moreover, the framework does not support assertions that are as complex as in our approach. The main focus is on providing input models.

In his master's thesis, ten Buuren introduces a generic approach for testing domain-specific models (ten Buuren 2015). Although the focus is on test case generation, the goal is similar: provide a generic framework for testing domain-specific models. In contrary to our work, ten Buuren assumes that Java code is generated from domain-specific models, and generates Java tests. This means that testing is done at the code level, meaning that unexpected behaviour has to be inspected at the code level. In our work, the testing process is completely pulled up to the domain-specific level, according to DSM principles.

Testing has been extensively researched in the context of model transformations. Al Mallah introduced a generic framework for model transformation testing (Al Mallah 2010). Guerra and Soeken introduced a generic visual language for testing model transformations (Guerra et al. 2010). Generating adequate input models from metamodels, which is of paramount importance for testing model transformations, is presented by Ehrig *et al.* (Ehrig et al. 2009), and a survey on the topic is given by (Wu et al. 2012). As this specific topic is well researched, covering all work is beyond the scope of this paper. Therefore, we redirect to a survey by Selim *et al.* (Selim et al. 2012) and challenges presented by Baudry *et al.* (Baudry et al. 2010). Contrary to our work, all of these approaches test model-to-model transformations. These can be considered functions that are expected to terminate, of which the output is tested using pre- and postconditions, and invariants. In our approach however, models with behaviour (*i.e.,* operational semantics implemented as transformation) are tested, meaning that tests cannot be simply expressed in terms of pre- and postconditions and invariants because behaviour is monitored at run-time.

Comparing the *ProMoBox* approach, Combemale *et al.* (Combemale et al. 2012) argue that executable DSMLs require languages similar to our design, runtime, input and trace languages. However, in their approach, the different metamodels must be specified explicitly by the language engineer, rather than using a generative approach that requires minimal annotation of the metamodel. Zalila *et al.* (Zalila et al. 2013) extend the approach by providing support for verification. Similar to the *ProMoBox* approach, the approach automatically verifies properties by mapping to a verification backbone, and also translates counterexamples to the domain-specific level. Contrary to our approach, a transformation for the latter has to be built by hand. Additionally, instead of using patterns as predicates like we propose, helper functions have to be written (requiring knowledge of OCL), while temporal properties in TOCL use these helper functions. No specific approach for testing is defined in this work.

Similar to our testing language, Runge *et al.* (Runge et al. 2013) use a visual language for testing. This language allows the user to model contracts to specify pre- and postconditions, from which test cases can be generated.

Geiger and Zündorf (Geiger and Zündorf 2005) introduce a generic testing language that can be used in a modelling context. Although this language is not domain-specific, its visual syntax can inspire to tailor the generic testing template of our approach (*i.e.,* the shaded part of Figure 10).

In the context of DSM, the so-called oracle issue (Mottu et al. 2008) needs to be resolved. It states that in model-driven engineering, a simple oracle model is hard to obtain and often insufficient. Three possible representations for oracles are introduced: model comparison, contracts and pattern matching. In light of this work, we employ pattern matching, which can also support model comparison. Contracts can be emulated in our language, but are not the main focus of our work because we focus on operational semantics rather than model-to-model transformations that behave like functions.

6. Conclusion and Future Work

In this paper, we presented a DSM solution for testing models that conform to a reactive DSML. From an annotated DSML specification, our solution generates a testing DSML, allowing domain users to specify tests using concepts and notations they are familiar with. Our approach includes generic support for executing the test models. Automation is key in our approach. We conducted an informal evaluation and concluded that with our approach, DSM can be used as an early prototyping technique that includes testing.

The main threads of future work are twofold. First, the approach can be extended to different classes of DSMLs, as discussed in Section 4.3. Second, we intend to extend the approach so that more declarative prototypes as defined in (Meyers et al. 2014) that span several test cases can be used to generate test models. This would result in a framework where domain users can model properties like "whenever I press a button, the elevator eventually arrives at the requested floor", which can be fully automatically verified using (a) model checking, or, in case model checking is infeasible, (b) model testing, according to some coverage criteria.

References

A. Al Mallah. Model-based testing of model transformations. Master's thesis, McGill University, Canada, 2010.

B. Baudry, S. Ghosh, F. Fleurey, R. B. France, Y. L. Traon, and J. Mottu. Barriers to systematic model transformation testing. *Commun. ACM*, 53(6):139–143, 2010.

J. Bézivin, F. Büttner, M. Gogolla, F. Jouault, I. Kurtev, and A. Lindow. Model transformations? transformation models! In *Lecture Notes in Computer Science*, volume 4199, pages 440–453, 2006.

F. Boulanger, C. Hardebolle, C. Jacquet, and I. Prodan. Modeling time for the execution of heterogeneous models. Technical report 2013-09-03-DI-FBO, Supélec E3S, 2012.

L. Burgueño. Testing M2M/M2T/T2M transformations. In *Proceedings of the ACM Student Research Competition at MODELS 2015, Ottawa, Canada, September 29, 2015.*, pages 7–12, 2015.

B. Combemale, X. Crégut, and M. Pantel. A design pattern to build executable dsmls and associated v&v tools. In K. R. P. H. Leung and P. Muenchaisri, editors, *19th Asia-Pacific Software Engineering Conference, APSEC 2012, Hong Kong, China, December 4-7, 2012*, pages 282–287. IEEE, 2012. ISBN 978-0-7695-4922-4. doi: 10.1109/APSEC.2012.79. URL `http://dx.doi.org/10.1109/APSEC.2012.79`.

J. Denil, P. J. Mosterman, and H. Vangheluwe. Rule-based model transformation for, and in simulink. In *Proceedings of the Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium*, page 4, 2014.

R. Deshayes, B. Meyers, T. Mens, and H. Vangheluwe. ProMoBox in practice : A case study on the GISMO domain-specific modelling language. In *CEUR Workshop Proceedings*, volume 1237, pages 21–30, 2014.

K. Ehrig, J. M. Küster, and G. Taentzer. Generating instance models from meta models. *Software and System Modeling*, 8(4):479–500, 2009.

L. Geiger and A. Zündorf. Story driven testing - SDT. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–6, 2005. doi: 10.1145/1082983.1083186. URL `http://doi.acm.org/10.1145/1082983.1083186`.

E. Guerra, J. de Lara, D. S. Kolovos, and R. F. Paige. A Visual Specification Language for Model-to-Model Transformations. In *VL/HCC*, pages 119–126, 2010.

T. Kanstrén. A review of domain-specific modelling and software testing. In *The Eighth International Multi-Conference on Computing in the Global Information Technology (ICCGI 2013)*, pages 51–56, 2013.

T. Kanstrén and O. Puolitaival. Using built-in domain-specific modeling support to guide model-based test generation. In *EPTCS*, pages 58–72, 2012.

S. Kelly and J.-P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. John Wiley & Sons, March 2008. ISBN 0470036664.

T. M. King, G. Nunez, D. Santiago, A. Cando, and C. Mack. Legend: an agile DSL toolset for web acceptance testing. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, pages 409–412, 2014.

J. Kloos and R. Eschbach. A systematic approach to construct compositional behaviour models for network-structured safety-critical systems. *Electr. Notes Theor. Comput. Sci.*, 263:145–160, 2010.

D. S. Kolovos, R. F. Paige, and F. Polack. The epsilon object language (EOL). In *Lecture Notes in Computer Science*, volume 4066, pages 128–142, 2006.

T. Kühne. Matters of (meta-)modeling. *Software and System Modeling*, 5(4):369–385, 2006. doi: 10.1007/s10270-006-0017-9. URL `http://dx.doi.org/10.1007/s10270-006-0017-9`.

T. Kühne, G. Mezei, E. Syriani, H. Vangheluwe, and M. Wimmer. Explicit transformation modeling. In *Models in Software Engineering, Workshops and Symposia at MODELS 2009, Denver, CO, USA, October 4-9, 2009, Reports and Revised Selected Papers*, volume 6002 of *Lecture Notes in Computer Science*, pages 240–255, 2009.

R. Mannadiar. *Multi-Paradigm Modelling Approach to the Foundations of Domain-Specific Modelling*. PhD thesis, McGill University, 6 2012.

B. Meyers. *A Multi-Paradigm Modelling Approach to the Design and Evolution of Domain-Specific Modelling Languages*. PhD thesis, University of Antwerp, 2 2016.

B. Meyers, J. Denil, F. Boulanger, C. Hardebolle, C. Jacquet, and H. Vangheluwe. A DSL for explicit semantic adaptation. In *Proceedings of the 7th Workshop on Multi-Paradigm Modeling co-located with the 16th International Conference on Model Driven Engineering Languages and Systems, MPM@MoDELS 2013, Miami, Florida, September 30, 2013.*, volume 1112 of *CEUR Workshop Proceedings*, pages 47–56, 2013a.

B. Meyers, M. Wimmer, and H. Vangheluwe. Towards domain-specific property languages: The ProMoBox approach. In *Proceedings of the 2013 ACM Workshop on Domain-specific Modeling*, pages 39–44. ACM New York, NY, USA, 2013b.

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

P. J. Mosterman and H. Vangheluwe. Computer automated multi-paradigm modeling: An introduction. *Simulation*, 80(9):433–450, 2004.

J. Mottu, B. Baudry, and Y. L. Traon. Model transformation testing: oracle issue. In *First International Conference on Software Testing Verification and Validation, ICST 2008, Lillehammer, Norway, April 9-11, 2008, Workshops Proceedings*, pages 105–112, 2008.

P. C. Ölveczky. Real-time maude and its applications. In *Lecture Notes in Computer Science*, volume 8663, pages 42–79, 2014.

B. C. Pierce. *Types and programming languages*. MIT Press, 2002. ISBN 978-0-262-16209-8.

O. Puolitaival, T. Kanstrén, V. matti Rytky, and A. Saarela. Utilizing domain-specific modelling for software testing. In *The Third International Conference on Advances in System Testing and Validation Lifecycle - VALID 2011*, pages 115–120. IARIA XPS Press, 2011.

M. Risoldi. *A Methodology For The Development Of Complex Domain Specific Languages*. PhD thesis, University of Geneva, 2010.

L. M. Rose, R. F. Paige, D. S. Kolovos, and F. Polack. The epsilon generation language. In *Lecture Notes in Computer Science*, volume 5095, pages 1–16, 2008.

O. Runge, T. A. Khan, and R. Heckel. Test case generation using visual contracts. *ECEASST*, 58, 2013. URL http://journal.ub.tu-berlin.de/eceasst/article/view/847.

L. Safa. The practice of deploying DSM Report from a Japanese appliance maker trenches. In J. Gray, J.-P. Tolvanen, and J. Sprinkle, editors, *Sixth Object-Oriented Programming, Systems, Languages and Applications Workshop on Domain-Specific Modeling*, pages 185–196. University of Jyväskylä, October 2006.

D. Santiago, A. Cando, C. Mack, G. Nunez, T. Thomas, and T. M. King. Towards domain-specific testing languages for software-as-a-service. In *CEUR Workshop Proceedings*, volume 1118, pages 43–52, 2013.

G. M. K. Selim, J. R. Cordy, and J. Dingel. Model transformation testing: The state of the art. In *Proceedings of the First Workshop on the Analysis of Model Transformations*, pages 21–26, 2012.

M. H. Smith, G. J. Holzmann, and K. Etessami. Events and constraints: A graphical editor for capturing logic requirements of programs. In *5th IEEE International Symposium on Requirements Engineering (RE 2001), 27-31 August 2001, Toronto, Canada*, pages 14–22, 2001.

E. Syriani, H. Vangheluwe, R. Mannadiar, C. Hansen, S. V. Mierlo, and H. Ergin. AToMPM: A web-based modeling environment. In *Joint Proceedings of MODELS'13 Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition co-located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013), Miami, USA, September 29 - October 4, 2013.*, pages 21–25, 2013.

R. ten Buuren. Domain-specific language testing framework. Master's thesis, University of Twente, the Netherlands, 2015.

S. Van Mierlo and H. Vangheluwe. Adding rule-based model transformation to modelling languages in metaedit+. *ECEASST*, 54, 2012.

E. Visser. WebDSL: A case study in domain-specific language engineering. In *Lecture Notes in Computer Science*, volume 5235, pages 291–373, 2007.

H. Wu, R. Monahan, and J. F. Power. Metamodel instance generation: A systematic literature review. *CoRR*, abs/1211.6322, 2012.

F. Zalila, X. Crégut, and M. Pantel. Formal verification integration approach for DSML. In A. Moreira, B. Schätz, J. Gray, A. Vallecillo, and P. J. Clarke, editors, *Model-Driven Engineering Languages and Systems - 16th International Conference, MODELS 2013, Miami, FL, USA, September 29 - October 4, 2013. Proceedings*, volume 8107 of *Lecture Notes in Computer Science*, pages 336–351. Springer, 2013. ISBN 978-3-642-41532-6. doi: 10.1007/978-3-642-41533-3_21. URL http://dx.doi.org/10.1007/978-3-642-41533-3_21.