Towards Domain-Specific Property Languages: The ProMoBox Approach

Bart Meyers
Modeling, Simulation and Design Lab (MSDL)
University of Antwerp
Antwerp, Belgium
bart.meyers@uantwerp.be

Manuel Wimmer
Business Informatics Group (BIG)
Vienna University of Technology,
Vienna, Austria
wimmer@big.tuwien.ac.at

Hans Vangheluwe
Modeling, Simulation and Design Lab (MSDL)
University of Antwerp
Antwerp, Belgium
McGill University, Montréal, Canada
hans.vangheluwe@uantwerp.be
hv@cs.mcgill.ca

Abstract

Domain-specific modeling (DSM) is one major building block of model-driven engineering. By moving from the solution space to the problem space, systems are designed by domain experts. The benefits of DSM are not unique to the design of systems, the specification and verification of desired properties of the designed systems by the help of DSM seems the next logical step. However, this latter aspect is often neglected by DSM approaches or only supported by translating design models to formal representations on which temporal properties are defined and evaluated. Obviously, this transition to the solution space is in contradiction with DSM.

To shift the specification and verification tasks to the DSM level, we extend traditional Domain-Specific Modeling Languages (DSMLs) for design with *ProMoBox*, a language family comprising three DSMLs covering design, property specification, and verification results. By using *ProMoBox*, temporal properties can be described for finite-state systems and verified by the SPIN model checker, by compiling them to Promela and Linear Temporal Logic (LTL). For specifying properties we present a DSML that is based on Dwyer's specification patterns and mash it up with adapted versions of the design DSML to formulate structural patterns. In particular, we show that a *ProMoBox* can be generated from a single root meta-model and we demonstrate our approach with a *ProMoBox* for statecharts.

Categories and Subject Descriptors D.2.1 [*Software Engineering*]: Requirements/Specifications—languages, methodologies, tools; D.2.4 [*Software Engineering*]: Software/-Program Verification—model checking; D.2.13 [*Software Engineering*]: Reusable Software—domain engineering; I.6.5 [*Simulation and Modeling*]: Model Validation and Analysis; I.6.5 [*Simulation and Modeling*]: Model Development—Modeling methodologies

Keywords ProMoBox; Domain-Specific Modeling; Language Engineering; Statecharts; Linear Temporal Logic; Spin; Model Checking

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
DSM '13, October 27, 2013, Indianapolis, Indiana, USA.
Copyright © 2013 ACM 978-1-4503-2600-1/13/10. . . $15.00.
http://dx.doi.org/10.1145/2541928.2541936

1. Introduction

Domain-specific modeling (DSM) is one major building block of model-driven engineering (MDE) [8]. By providing languages that are specific to the problem space and no longer to the solution space, systems are designable by domain experts while model transformations are taking care of achieving the transition to the solution space. Although, most DSM approaches focus on generating efficiently and effectively production code from high-level models, the benefits of DSM are not unique to the design task. In similar ways, other engineering tasks may benefit from DSM. In particular, the specification and verification of desired temporal properties for the designed systems is becoming more and more important to develop high-quality systems in general and in particular in MDE [7]. Thus, supporting these tasks by DSM seems the next important step to provide a holistic DSM experience to domain engineers. However, specifying and verifying properties is often neglected by current DSM approaches or only supported by translating system models to representations executable by model checkers on which temporal properties are defined and analyzed. Obviously, this transition to the solution space is in contradiction with DSM. The need to raise the level of abstraction for specification and verification tasks is also stressed by a recent publication by Visser et al. [24] on how to successfully apply model checking. The authors stress that the effective application of model checking requires that the input/output models that are consumed/produced by a model checker should be hidden from domain engineers.

To shift the specification and verification tasks to the DSM level, we propose *ProMoBox* for generating and operationalizing a family of domain-specific modeling languages (DSMLs) for a given DSML, covering not only design modeling as supported by traditional DSMLs, but also property specification and verification results representation. Property languages generated by *ProMoBox* are specifically tailored to ease the development of temporal patterns as well as structural patterns needed to describe the desired properties

of the system's design by domain engineers in the DSML's concrete syntax. Furthermore, to allow to formulate temporal properties at a high-level of abstraction, we formalize Dwyer's specification patterns [4] for defining temporal patterns as a DSML. With the help of this DSML, domain engineers are able to express temporal properties for finite state verification such as absence, existence, or universality. To ease the development of structural patterns to be checked on snapshots of the system's execution states, we propose an automated technique based on [16, 23] that is able to produce a specialized language from a given DSML tailored to express structural patterns. The language for defining structural patterns is inspired by *PaMoMo* [10, 11], a language for defining structural patterns on models supporting several pattern kinds such as enabling, positive, and negative patterns. Finally, we introduce the possibility to define quantifiers for temporal properties to express complex properties in a more concise manner, *e.g.*, every element of a certain type has to fulfill a certain property. Properties expressed in *ProMoBox* generated languages are automatically translatable to Promela and Linear Temporal Logic (LTL) and checked by SPIN [12]. By this approach, the modeler is shielded from the complexity of formal representations, because not only the system's design is given as a domain-specific model, but also the properties as well as the verification results. The latter are produced in a form that allows to replay the counterexample computed by SPIN on the DSM level if a property is violated.

We demonstrate *ProMoBox* for statecharts and show based on an elevator example how properties are defined and verification results are represented. Note that the current focus of *ProMoBox* is on partially ordered finite state transition systems. Integrating the notion of time in *ProMoBox* to support the specification of real-time properties, or supporting continuous-value systems, is left as subject to future work.

The rest of the paper is structured as follows. Section 2 gives an overview on the state-of-the-art of property languages used in MDE and motivates our approach. In Section 3, we introduce the running example for the subsequent sections. Section 4 introduces our approach *ProMoBox*. Section 5 shows for the running example how properties can be specified and verified by a generated *ProMoBox* for statecharts. Finally, Section 6 concludes this paper with an outlook on future work.

2. Related Work

With respect to the contribution of this paper, we distinguish three threads of related work. First, we consider approaches that translate models to formal representations to specify and verify properties that are created specifically for one modeling language. Second, we discuss approaches that have a more general view on providing specification and verification support for modeling languages. Third, we shortly elaborate on approaches that have inspired our concrete transformation from statecharts to Promela.

Specific Solutions. In the last decade, a plethora of language specific approaches have been presented to define properties and verification results for different kind of design-oriented languages. For instance, Cimatti et al. [2] have proposed to verify component-based systems by using scenarios specified as Message Sequence Charts. Li et al. [18] also apply Message Sequence Charts for specifying scenarios for verifying concurrent systems. The CHARMY approach [20] offers amongst other features, verification support for architectural models described in UML. Collaboration and sequence diagrams have been applied to check the behavior of systems described in terms of state machines [1, 14, 22]. Rivera et al. [21] map the operational semantics of DSMLs to Maude, and thus, benefit from analyzing methods provided out-of-the-box of Maude environments such as checking of temporal properties specified in LTL.

These mentioned approaches are just a few examples that aim at specifying temporal properties for models and verifying them by model checkers. However, these approaches offer language-specific property languages or LTL properties have to be defined directly on the formal representation. Thus, these approaches are not aiming to support DSMLs designers in the task of building domain-specific property languages.

Generic Solutions. There are some approaches that aim to shift the specification and verification tasks to the model level in a more generalized manner. First of all, there are approaches that use an extended version of OCL, called Temporal OCL (TOCL) [26], for defining temporal properties on models. As OCL can be used in combination with any modeling language, TOCL can be seen as a generic model-based property language as well. In [3, 25] the authors discuss and apply a pattern to extend modeling languages with events, traces, and further runtime concepts to represent the state of a model's execution and to use TOCL for defining properties that are verified by mapping the system models as well as the properties expressed in TOCL to formal domains that provide verification support. In addition, not only the input for model checkers is automatically produced, but also the output, *i.e.*, the verification results, is translated back to the model level. The authors explain the choice of using TOCL to be able to express properties at the business domain level, because TOCL is close to OCL and should be therefore familiar to domain engineers. However, they also state that early feedback of applying their approach has shown that TOCL is still not well suited to many domain engineers and they state in future work that more tailored languages may be of help for the domain engineers. The work of this paper goes directly in this direction. We are employing specification patterns to ease the development of temporal patterns and we allow to model structural patterns as model fragments using the notation of the modeling language. Thus, the domain engineers are able to use the notation they are familiar with for defining properties and exploring the verification results.

Another approach that aims to define temporal properties on the model level in a generic way is presented in [13]. The authors extend a language for defining structural patterns based on Story Diagrams [6] to allow for modeling temporal patterns as well. The resulting language allows to define conditionally timed scenarios stating the partial order of structural patterns. They authors argue that their language is more accessible for domain engineers, because their language allow to decompose more complex temporal properties into smaller ones by if-then-else decomposition and quantification over free variables is possible. Their approach is tailored to engineers that are familiar to work with UML class diagrams and UML object diagrams as their notation is heavily based on the concepts of these two languages. Furthermore, they explain how the specification patterns of Dwyer et al. [4] are encoded in their language, but there is no language-inherent support to explicitly model these patterns. In this work, we tackle these two issues in the context

of DSM, namely to explicitly model the specification patterns instead of encoding them in a more general language as well as to allow to reuse the notation of the domain engineers even for specifying the temporal properties.

Formalization of statecharts in SPIN. There exists a huge body of knowledge about giving semantics to statecharts by mapping to SPIN, *e.g.*, [1, 17, 19, 22] to name just a few. In this paper, we follow the existing work on how to map statecharts to SPIN in order to end up with efficiently analyzable specifications.

3. Motivating Example

This section introduces a running example, namely an elevator that will subsequently illustrate our approach. The system is implemented as a statechart shown in Fig. 1. Transitions are triggered by events and/or guards (in square brackets) that take the currently active state into account. The system we aim to verify is an elevator with three floors. As usual, the elevator has three buttons in its cabin, for requesting to go to each floor. Additionally, on each floor there are buttons to request going up or down, but on the bottom/top floor there is no button to request to go down/up. Button presses are implemented as events, and are handled by an orthogonal component for each button. In these components the state for requesting a floor becomes active if the button is pressed and if the lift does not have its doors open (*i.e.*, `idle`) at that floor. When the lift opens its door at a requested floor, associated requests are voided. The internal controller of the elevator is represented by the `ElevatorState` composite state, containing three orthogonal components denoting the elevator's position, activity and direction. The elevator can change `floors` if it is `moving` and goes `up` or `down`. The behaviour for activity and direction is modeled as follows:

- guard [*i2m*]: the elevator starts moving when there is a floor request that is not for the current floor, modeled by the guard;
- guard [*m2i*]: the elevator stops moving when it is at a floor for which there was a request in the cabin, or when there was a request outside the cabin to go in the same direction the elevator was going;
- guards [*u2d*] and [*d2u*]: the elevator changes direction when there is no request following the direction the elevator is heading, but there is a request in the opposite direction.

Note that this model is verbose and not easily scalable, *e.g.*, if an additional floor is added. We argue that such a model can be automatically generated from a DSML for elevators, which is exactly what we intend to do in further research, where we want to pull up the *ProMoBox* further to higher abstraction levels.

For this system, we want to be able to verify the following requirements:

- *ReachesFloor*: when a request for any floor is made, the elevator eventually opens its doors at that floor;
- *SkipFloorOnce*: when a request for any floor is made, the elevator opens its doors at the latest the second time it passes that floor.

It is clear that it cannot be immediately concluded whether the system satisfies these requirements by just looking at Fig. 1. The requirements and a possible counter-example should be modeled and visualized as properties at the correct level of abstraction (*i.e.*, statecharts).

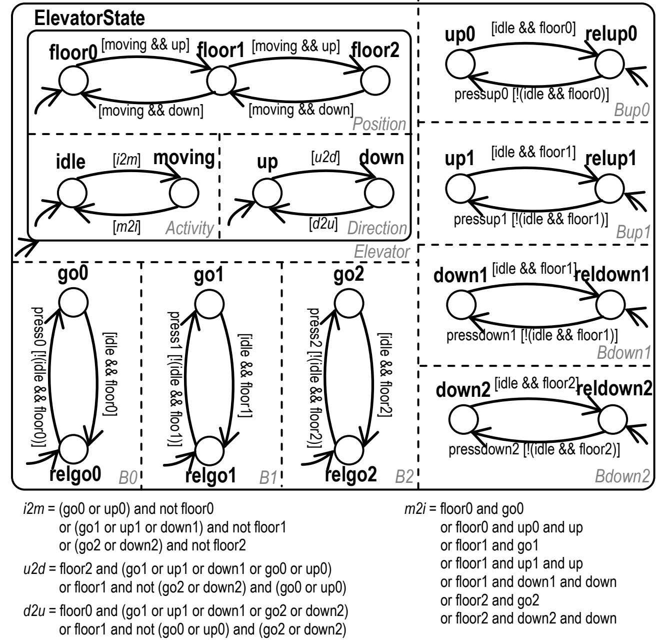

Figure 1. The Elevator Statechart Model.

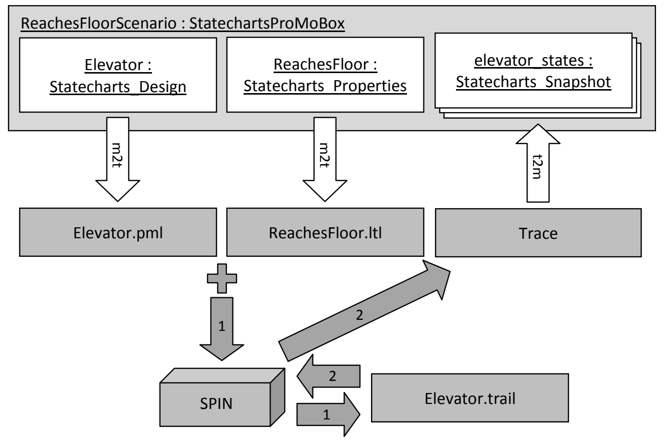

Figure 2. The *ProMoBox* Approach at a Glance.

4. The *ProMoBox* Approach

In this section, we present and motivate the *ProMoBox* approach. A *ProMoBox* is a language box that allows a modeler to design the system, specify properties, and visualize traces of possible counter-examples for failing properties. The contribution of this paper can be split up into two parts: the introduction of the *ProMoBox* and the automatic support for generating such a *ProMoBox* from a given meta-model. After giving an architectural overview in the following, we introduce the Property Language and Snapshot Language that will form, together with the Design Language, the *ProMoBox*. Finally we present how to generate a such a *ProMoBox*.

4.1 Overview

The Statecharts *ProMoBox*, and its translation to Promela models as well as the translation of traces back from SPIN to

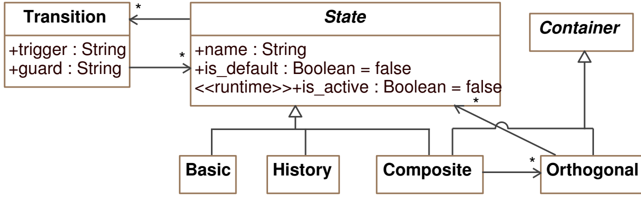

Figure 3. The Statechart Meta-model.

statecharts, is presented in Fig. 2. An architectural overview is given of the scenario where the *ReachesFloor* property is verified on the elevator model. A *ProMoBox* (the grey box on top) consists of three languages, the Design Language, the Properties Language and the Snapshot Language. The Design Language allows the modeler to design the system in the traditional way. The Properties Language allows the modeler to define properties over the designed system by using a combination of quantifiers, temporal patterns and structural patterns on the model's state. The Snapshot Language allows to visualize a run-time state of the system, thus allowing an execution trace to be visualized as a sequence of snapshots. A key aspect of the *ProMoBox* is that the concrete syntax of these languages is as close as possible to the modeling language, so that the modeler can optimally use them.

The design model is transformed by a model-to-text transformation to Promela code. To this end, we used an extension of the algorithm introduced by Latella et al. [17]. We extended this translation in two ways. Firstly, we generate print statements that print out system states, system events and input events. We will use these later to generate traces. Secondly, we added the notion of an environment that automatically generates input events. These event generators are implemented the same way as orthogonal regions, and it is made sure that every event has an equal occurrence probability, which can be influenced by a compiler flag for "slowing down" the generation of events. Two modes exist for the transformation: one generates a process type for each orthogonal region, and one introduces determinism by including all orthogonal regions in a single process type, thus achieving verification speed-ups.

A property is transformed to an LTL formula. This process is presented more in detail below. The SPIN tool is used to verify the low-level Promela model using the LTL formula. If the Promela code satisfies the property, the modeler is "just" notified with a message. In case of finding a counter-example, a SPIN trail file is used that contains the steps that were followed to obtain the counter-example. Consequently, the Promela code is executed following the steps of trail file, and as a side-effect, all states, system events and input events are printed out. The resulting trace can be filtered per type of state/event. The trace that was produced when executing the counter-example is filtered for system states, which are converted to a sequence of models in the Snapshot Language. These can be displayed by automatically transforming the design model to a snapshot model (thus maintaining the model layout the modeler is familiar with, while adding run-time information) and showing system states step-by-step.

4.2 The Languages of the *ProMoBox*

As mentioned before, all three languages of the *ProMoBox* should be familiar to the modeler. Consequently, the ab-

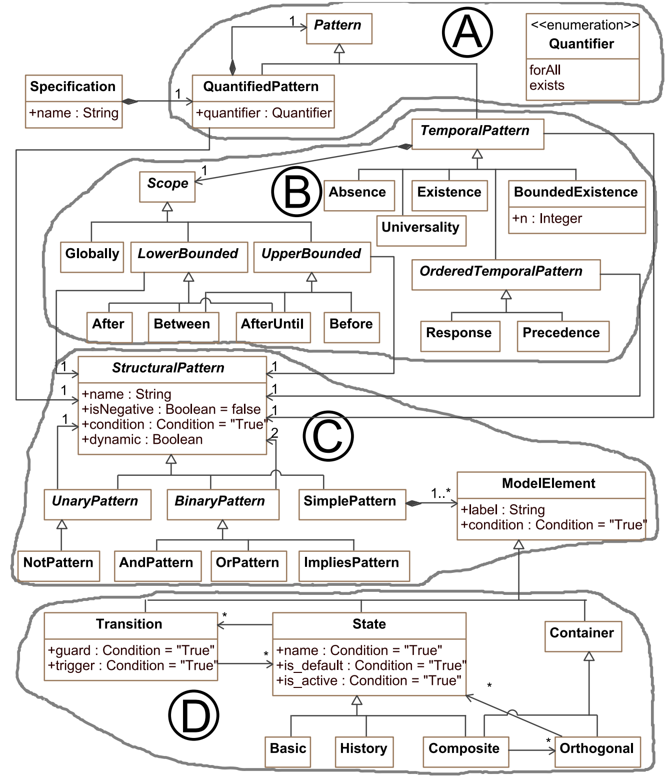

Figure 4. The Meta-model of the Statechart Property Language.

stract and concrete syntax is re-used when possible. We therefore start with one *root meta-model* of the statecharts language, shown in Fig. 3. This meta-model comprises more than just the Design Language, as it also contains language constructs for expressing the *dynamic state* of the model as also proposed by dynamic metamodeling approaches [5]. The parts of a meta-model that are dynamic are annotated with the *«runtime»* stereotype. In the case of the statecharts language, the `is_active` attribute would not be present in the static Design Language. On the contrary, the Snapshot Language includes all *«runtime»* constructs, as its purpose is visualizing the dynamic state of the model. The concrete syntaxes of the Design Language and the Snapshot Language are very similar. An instance of the Statechart Design Language is shown in Fig. 1, whereas a Snapshot instance looks the same, but can have certain states flagged as active, which is showed by coloring the state grey.

Property Languages are built out of four components as shown in Fig. 4. These are discussed below.

[A] the quantification of the formula by (*i*) *forAll* or *exists* clause(s), and (*ii*) corresponding structural pattern(s). The modeler can choose to model a property for all elements that match the associated structural pattern. This structural pattern is evaluated on the static model (*i.e.*, in the case of statecharts, without the `is_active` attribute). Consequently, the property must be satisfied for all, or for one (depending on the quantifier) match(es) of the structural pattern. The resulting matches can be re-used as bound variables in the property. The quantification is statically compiled to Promela and-/or-conjuctions of temporal patterns,

taking the design model into account. Quantification patterns can be nested, or can contain a temporal pattern.

[B] the temporal pattern, based on Dwyer's specification patterns [4]. The temporal pattern allows the user to specify a pattern over a given scope, *e.g.*, "the absence of P, after the occurrence of Q", or "P is responded by S, between occurrences of Q and R" (with proposition variables P, S, Q and R). Over 90% of the properties that were investigated by Dwyer et al. can be expressed in this simple framework [4]. Temporal patterns are compiled to Promela LTL formulas according to [4], with empty spots for proposition values. Up to four proposition variables of the temporal patterns are expressed as structural patterns, that represent patterns on one snapshot of the system.

[C] the structural pattern, based on PaMoMo [11], for both static (when attached to a quantification) as well as dynamic (when attached to a temporal pattern) models. Using a structural pattern, a query can be defined on a model, in case of a static pattern returning all bound variables in found matches, and in case of a dynamic pattern returning true if at least one match is found or false when no match is found. In our current approach, we use an ad-hoc matching algorithm, but we intend to re-use the matching algorithm presented in [23]. Only a small part of PaMoMo's expressiveness is displayed in Fig. 4, but this suffices for defining most properties. Of course, the meta-model can be extended with additional language constructs. A `StructuralPattern`, and a `ModelElement` can hold a condition, which returns true by default and is in our current approach modeled as a string. A model element serves as the superclass for pattern elements, that are specific to the modeling language. Structural patterns are compiled to Promela boolean expressions.

[D] the pattern elements, based on the RAM process. The elements of a structural pattern are based on the root meta-model but need to be changed in several ways in order to allow the modeler to specify patterns that are match in model fragments. A similar problem exists when constructing a pattern language for creating a meta-model for transformation rules, and is formalized by the RAM process [16, 23]. In this process, all classes are subclasses of `ModelElement` and have a label (for binding variables) and a condition, attribute types are now conditions, no more classes are abstract classes, and all lower bounds of association multiplicities are set to 0. Pattern elements are compiled to their corresponding Promela variable names, which can be used in the Promela boolean expression of the structural pattern.

Note that only component D depends on (and can be automatically generated from) the meta-model of the design language, while components A, B and C are generic. Consequently, our approach is applicable to other meta-modelled languages.

4.3 Generating a *ProMoBox*

The three languages in the *ProMoBox* are automatically generated from a root meta-model like the one shown in Fig. 3. The generation of the Design Language is done by taking the root meta-model and removing all language constructs that are flagged *«runtime»*. The generation of the Snapshot Language is done by taking the root meta-model and keeping all constructs, but simply removing the *«runtime»* flags. Generating the Properties Language requires merging the generic properties template (parts A, B and C in Fig. 4) with a language for structural patterns (part D in Figure 4) that is based on the root meta-model. Using the

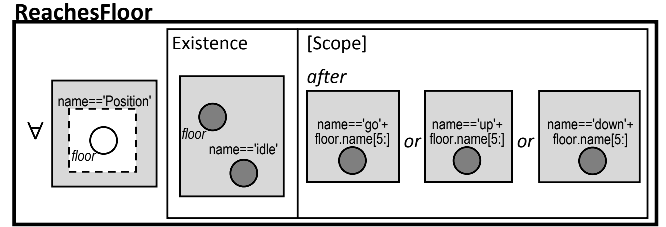

Figure 5. The *ReachesFloor* Property: when a request for any floor is made, the elevator eventually opens its doors at that floor.

RAM process [23], the meta-model of Fig. 3 can be automatically transformed to a pattern meta-model (part D in Fig. 4).

Our approach is realized in the tool $AToM^3$, which allowed us to specify transformations on meta-models and model-to-text transformations. It is provided on our website at `http://msdl.cs.mcgill.ca/people/bart/promobox`. We consider this work to be a prelude to the more specific case of generating a *ProMoBox* for any DSML meta-model, including its translation to a semantic domain such as the statecharts *ProMoBox* presented here. In that way, the modeler can design and verify systems at an even more appropriate level of abstraction, while, as in traditional DSML engineering, only one root meta-model and a semantic mapping has to be modeled.

5. The Statechart *ProMoBox*

In this section, the statecharts *ProMoBox* is applied for the motivating example. The *ReachesFloor* property is expressed in the Property Language in Fig. 5. The concrete syntax is a combination of a generic syntax for properties, and re-used syntax of the statecharts DSML. The *ReachesFloor* property contains a *forAll* quantifier (on the left side), with an associated structural pattern, visualized by the leftmost grey box. This static pattern binds all states to the variable name `floor`, that are inside an orthogonal region named `Position` (modeled as a constraint over the name attribute). Thus, the property will be evaluated "for all floors", and in the case of the Fig. 1 model, `floor0`, `floor1` and `floor2` will be bound one after the other, yielding three LTL formulas, concatenated by an and-clause. On the right side, the temporal pattern is shown. We will verify the "existence" of the lift being at that floor (represented by the bound `floor` state being active - colored grey), and its doors being open (represented by the `idle` state being active), after one of the buttons for that floor is pressed (represented by the states `goN`, `upN` or `downN` being active, where `N` is the same number as the bound `floor` state's name ending number). Note that in the case where `floor0` is bound, no `down0` state exists, and that or-option will be automatically discarded. Similarly, there is no "up"-option for `floor2`.

Next to this visual concrete syntax, we built an alternative user interface for defining properties in the form of a wizard menu, where one can select the desired temporal pattern, and structural patterns are still modeled explicity.

The LTL formula for the *ReachesFloor* property contains of 31 operators and 20 proposition variables, and nests brackets up to 7 levels deep. The *SkipFloorOnce* property even results in a formula of 107 operators and 82 proposition variables, and brackets depth of 11. Clearly writing or maintaining such a LTL formula is unpractical. Verifying

whether the model shown in Fig. 1 satisfies the *ReachesFloor* or *SkipFloorOnce* property with a maximum search depth of 10000 (which is sufficient to explore the full state space), is done within minutes.

6. Conclusions and Future Work

In this paper we have presented *ProMoBox* to generate a family of languages for a given DSML to allow domain engineers to specify and verify properties on the DSM level. With the support of the family of languages and model transformations, domain engineers specify properties that are verifiable by current model checkers with the help of Property Languages. Furthermore, the result of the verification, *i.e.*, the counter-example in case a property is violated, is reviewed/replayed on the DSM level. With a case study of modeling an elevator system in terms of statecharts we have demonstrated how properties can be defined on this level of abstraction and translated them to Promela and LTL for verification in SPIN. The model checker results, most interestingly the counter-examples in case properties are not fulfilled, are used to animate the trace by a snapshot sequence language on the DSM level.

While in this paper we have presented an important step towards *ProMoBox*es, there are several open issues for future work. First, our plan is to go one step higher in the abstraction level, *e.g.*, providing an elevator DSML on top of statecharts, and reuse the current framework as an intermediate layer to provide model checking possibilities for several DSML that fit their semantics on state/transition systems. Second, in our current *ProMoBox*es, we have no means to specify real-time properties. This problem may be tackled in a similar way as we did it for partial order state properties, namely to make use of documented patterns for the real-time property domain [9, 15] and formulate them as DSML.

7. Acknowledgments

This work has been partially funded by Research Foundation Flanders under grant FWO-K219913N, Vienna Science and Technology Fund (WWTF) under grant ICT10-018, and Austrian Research Promotion Agency (FFG) under grant 832160.

References

[1] P. Brosch, U. Egly, S. Gabmeyer, G. Kappel, M. Seidl, H. Tompits, M. Widl, and M. Wimmer. Towards Scenario-Based Testing of UML Diagrams. In *TAP*, pages 149–155, 2012.

[2] A. Cimatti, S. Mover, and S. Tonetta. Proving and Explaining the Unfeasibility of Message Sequence Charts for Hybrid Systems. In *FMCAD'11*, pages 54–52, 2011.

[3] B. Combemale, X. Crégut, and M. Pantel. A Design Pattern to Build Executable DSMLs and Associated V&V Tools. In *APSEC*, pages 282–287, 2012.

[4] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *ICSE*, pages 411–420, 1999.

[5] G. Engels, J. H. Hausmann, R. Heckel, and S. Sauer. Dynamic Meta Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML. In *UML*, pages 323–337, 2000.

[6] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In *TAGT'98*, pages 296–309, 2000.

[7] R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. In *FOSE'07*, pages 37–54, 2007.

[8] J. Gray, J.-P. Tolvanen, S. Kelly, A. Gokhale, S. Neema, and J. Sprinkle. Domain-specific modeling. *Handbook of Dynamic System Modeling*, 2007.

[9] V. Gruhn and R. Laue. Patterns for timed property specifications. *ENTCS*, 153(2):117–133, 2006.

[10] 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.

[11] E. Guerra, J. de Lara, M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, and W. Schwinger. Automated verification of model transformations based on visual contracts. *ASE*, 20(1):5–46, 2013.

[12] G. J. Holzmann. The Model Checker SPIN. *TSE*, 23(5):279–295, 1997.

[13] F. Klein and H. Giese. Joint structural and temporal property specification using timed story scenario diagrams. In *FASE'07*, pages 185–199, 2007.

[14] A. Knapp and J. Wuttke. Model checking of UML 2.0 interactions. In *MoDELS'06*, pages 42–51, 2006.

[15] S. Konrad and B. H. C. Cheng. Real-time specification patterns. In *ICSE'05*, pages 372–381, 2005.

[16] T. Kühne, G. Mezei, E. Syriani, H. Vangheluwe, and M. Wimmer. Explicit transformation modeling. In *MoDELS Workshops*, pages 240–255, 2009.

[17] D. Latella, I. Majzik, and M. Massink. Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker. *Formal Asp. Comput.*, 11(6):637–664, 1999.

[18] X. Li, J. Hu, L. Bu, J. Zhao, and G. Zheng. Consistency Checking of Concurrent Models for Scenario-Based Specifications. In *SDL'05*, pages 1171–1180. 2005.

[19] J. Lilius and I. P. Paltor. vUML: A Tool for Verifying UML Models. In *ASE'99*, 1999.

[20] P. Pelliccione, P. Inverardi, and H. Muccini. CHARMY: A Framework for Designing and Verifying Architectural Specifications. *TSE*, 35(3):325–346, 2008.

[21] J. E. Rivera, E. Guerra, J. de Lara, and A. Vallecillo. Analyzing Rule-Based Behavioral Semantics of Visual Modeling Languages with Maude. In *SLE*, pages 54–73, 2008.

[22] T. Schäfer, A. Knapp, and S. Merz. Model Checking UML State Machines and Collaborations. *ENTCS*, 55(3):357–369, 2001.

[23] E. Syriani. *A Multi-Paradigm Foundation for Model Transformation Language Engineering*. PhD thesis, McGill University Montreal, Canada, 2011.

[24] W. Visser, M. Dwyer, and M. Whalen. The hidden models of model checking. *Software and Systems Modeling*, 11(4):541–555, 2012.

[25] F. Zalila, X. Crégut, and M. Pantel. Leveraging Formal Verification Tools for DSML Users: A Process Modeling Case Study. In *ISoLA*, pages 329–343, 2012.

[26] P. Ziemann and M. Gogolla. OCL Extended with Temporal Logic. In *Ershov Memorial Conference*, pages 351–357, 2003.