ProMoBox: A Framework for Generating Domain-Specific Property Languages

Bart Meyers[1], Romuald Deshayes[2], Levi Lucio[3], Eugene Syriani[4], Hans Vangheluwe[1,3], and Manuel Wimmer[5]

[1] Modeling, Simulation and Design Lab (MSDL), University of Antwerp, Belgium
bart.meyers@uantwerp.be, hans.vangheluwe@uantwerp.be
[2] Institut d'Informatique, Universit de Mons, Mons, Belgium
romuald.deshayes@umons.ac.be
[3] Modeling, Simulation and Design Lab (MSDL), McGill University, Canada
levi@cs.mcgill.ca, hv@cs.mcgill.ca
[4] Software Engineering Research Group (SERG), University of Alabama, United States
esyriani@cs.ua.edu
[5] Business Informatics Group (BIG), Vienna University of Technology, Austria
wimmer@big.tuwien.ac.at

Abstract. Specifying and verifying properties of the modelled system has been mostly neglected by domain-specific modelling (DSM) approaches. At best, this is only partially supported by translating models to formal representations on which properties are specified and evaluated based on logic-based formalisms, such as linear temporal logic. This contradicts the DSM philosophy as domain experts are usually not familiar with the logics space. To overcome this shortcoming, we propose to shift property specification and verification tasks up to the domain-specific level. The *ProMoBox* framework consists of (i) generic languages for modelling properties and representing verification results, (ii) a fully automated method to specialize and integrate these generic languages to a given DSM language, and (iii) a verification backbone based model checking directly plug-able to DSM environments. In its current state, *ProMoBox* offers the designer modelling support for defining temporal properties, and for visualizing verification results, all based on a given DSM language. We report results of applying *ProMoBox* to a case study of an elevator controller.

1 Introduction

Domain-specific modelling (DSM) advocates that, providing languages that are specific to the problem space rather than to the solution space, systems are designable by domain experts while model transformations are taking care of achieving the transition to the solution space [1]. An essential activity in DSM is the specification and verification of properties to ensure the high quality of the designed systems [2]. Thus, supporting these tasks by DSM is necessary to provide a holistic DSM experience to domain engineers. However, specifying and verifying properties of systems has been mostly neglected by DSM approaches. At best, this is only partially supported by translating models to formal representations on which properties are specified and evaluated with logic-based formalisms [3], such as Linear Temporal Logic (LTL). This contradicts the DSM philosophy as domain experts are usually not familiar with temporal logic. The need

to raise the level of abstraction for specification and verification tasks is also recently raised in [4]. The authors emphasize that domain engineers should be shielded from the underlying verification technologies. In this sense, DSM includes not only the design of the system-under-study, but also the properties themselves, the representation of the run-time state of a system, the behaviour of the environment, and a visualisation of a counter-example, all at the domain-specific level. In the spirit of DSM, they should each be defined in their own domain-specific modelling language (DSML).

To overcome this shortcoming, we propose to shift property specification and verification tasks up to the DSM level, resulting in the generation and execution of *ProMoBox*. The contribution of the *ProMoBox* framework consists of (i) generic languages for modelling all artefacts that are needed for specifying and verifying temporal properties with the expressive power of LTL, (ii) a fully automated method to specialise and integrate these generic languages to a given DSML, and (iii) a verification backbone based on model checking with LTL that is directly plug-able to DSM environments.

In the following section, we introduce the running example of an elevator controller. Section 3 introduces our approach *ProMoBox* from a language engineering point of view and explains how properties are defined and verified based on a model checking backbone. Section 4 is dedicated to implementation and evaluation of *ProMoBox*. Section 5 elaborates on assumptions that are made in the current state of the approach while discussing the limitations of the approach. In Section 6 we discuss related work and conclude in Section 7.

2 Running Example

Our running example is an elevator controller modeled by a graphical DSL. This DSL enables modelling a building with floors, elevators and buttons, and defines the stepwise behaviour of this model.

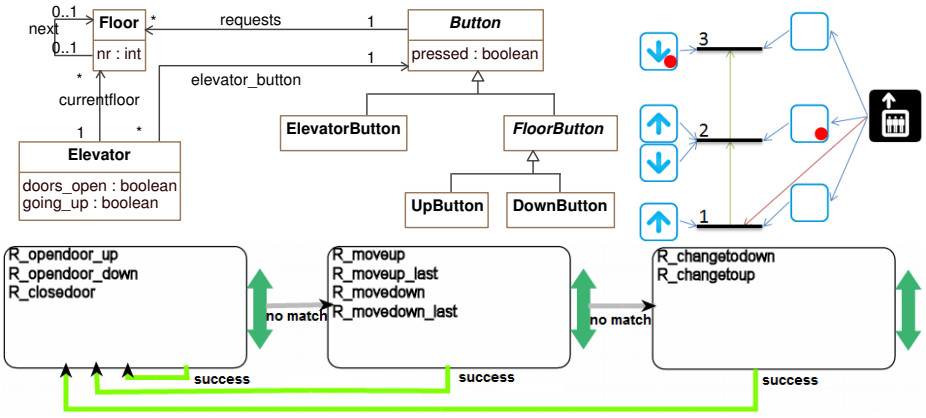

Fig. 1. The meta-model E of the Elevator DSL (top left), an instance e of the Elevator DSL representing an elevator that serves three floors (top right), and the transformation model $E_{[[.]]}$ that schedules all the operational rules (bottom).

The left of Fig. 1 shows the meta-model which we will denote by E. *Button*s can request an *Elevator* to go to a particular *Floor*. Floors are ordered by the *next* association and a derived attribute *nr* representing the *Floor* number. An *Elevator* is at exactly one *Floor*, modelled by the *currentfloor* association. An *ElevatorButton* is a button inside an *Elevator*, requesting 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 (it cannot move) and has a direction (up or down).

The top right of Fig. 1 shows an instance e with three floors, one elevator and seven buttons, depicting the concrete syntax. Pressed buttons are annotated with red dots, and are connected to the floor they request. At the top floor a button is pressed by someone who requested to go down, and inside the elevator the button to go to floor 2 has been pressed. The elevator is currently at the bottom level.

The bottom of Fig. 1 shows the transformation model $E_{[[.]]}$ of the operational semantics. The model shows how different rules are scheduled. Rounded rectangles refer to a set of rules where at most one is randomly chosen to be applied. Execution starts at the left rectangle. Grey arrows annotated with "no match" are followed when none of the rules in the source rectangle can be applied, green arrows annotated with "success" are followed when a rule was applied. Inspired from a realistic elevator controller, the rules implement how the elevator changes floors (one at a time), and opens and closes its door to serve the requests of users (modelled as pressed buttons). The elevator passes all floors that are requested on its path (which is either up or down), and opens its door when the elevator's direction corresponds to the requested direction. Pressed buttons are turned off (released) when the door opens at a requested floor and the elevator goes in that direction. When a request for a floor is made for a different floor than the elevator's current floor, the doors close and the elevator starts moving. The elevator only changes its direction when there are no more requests on its path. Note that, if the elevator is at a lower floor, it can pass by a floor where one has requested to go down without stopping, as the elevator is going in the opposite direction. The rules are not shown in Fig. 1 because of space constraint, but later in the paper, one of the rules is shown in Fig. 7.

When designing the elevator software system, we would like to verify the *Reaches-Floor* property: whenever a request for any floor is made, the elevator will eventually open its doors at the latest the second time it passes by that floor.

3 The ProMoBox

Based on preliminary ideas outlined in our previous work [5, 6], the *ProMoBox* framework consists of the following three parts.

Generic languages for modelling all artefacts that are needed for specifying and verifying properties. For a given DSML, *ProMoBox* defines a family of five sub-languages [5] that are required to modularly support property verification, covering (i) design modelling as supported by traditional DSMLs, (ii) run-time state representation, (iii) event-based input modelling (to model the behaviour of an environment), (iv) state-based output representation (to model an execution trace of the system or verification results), and (v) property specification. 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. To allow to formulate temporal properties at a high-level of abstraction, we formalise Dwyer's specification patterns [7] 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 [8, 9] that is able to produce a specialised language from a given DSML tailored to express structural patterns. The language for defining structural patterns is inspired by *PaMoMo* [10, 11], a language 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 fulfil a certain property.

A **fully automated method** to specialise and integrate these generic languages to a given DSML. We extend meta-modelling and model transformation languages with annotations, to add necessary information for every language construct and semantic step. This additional information enables the fully automatic generation of the five sub-languages and necessary transformations between the sub-languages, thus minimising the effort of the language engineer. Because of their generative definition, consistency between the languages and their models is guaranteed by construction. We use templates that describe the generic part of each language, and that are subsequently woven with the DSML. By using templates, we allow the *ProMoBox* framework to be configurable for different types of DSMLs.

A **verification backbone** based model checking directly plug-able to DSM environments. Properties in *ProMoBox* are translated to LTL and a Promela system is generated that includes a translation of the system, the environment, and the rule-based operational semantics of the system. The properties are checked by SPIN [12]. The verification results (in case of a counter-example) are translated back to the DSM level.

The *ProMoBox* approach is illustrated in Fig. 2 using the elevator example presented in Section 2. When using the *ProMoBox* approach, only the grey models in Fig. 2 need to be modelled by hand, the white models are generated. This is done in two parts: first, we define how meta-models can be annotated (E' in Fig. 2) and how the five sub-languages (*i.e.,* the design, run-time, input, output and properties languages) are generated (upper part of Fig. 2). Second, we define how mappings are generated that allow a given property to be verified on a given system, and how the results can be visualised in a domain specific way (steps 1 to 5 of Fig. 2).

3.1 The Annotated Meta-Model

The abstract syntax of the sub-languages is generated from the *annotated meta-model* that provides additional information on which parts are static (never change at run-time), dynamic (change at run-time) and which parts can be input into the system.

First, we present a formal definition of a meta-model. The complete formalisation can be found in [5]. We define Σ as the alphabet of all possible names. For simplicity, all class, association, and attribute names are globally unique and are from here on referred

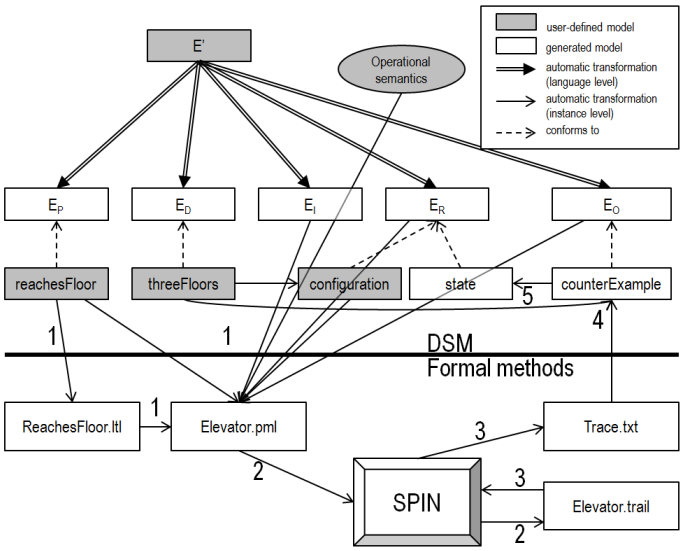

Fig. 2. Property verification with ProMoBox and SPIN.

to as the classes, associations or attributes themselves. A meta-model is defined by:

$$\mathcal{M} = (C, D, A, \alpha, \iota, P, \pi), \tag{1}$$

with

$C \subseteq \Sigma$	the set of all classes,
$D \subseteq C$	the set of all abstract classes,
$A \subseteq \Sigma$	the set of all associations,
$\alpha : A \to C \times C$	the association mapping, a total function,
$\iota : C \times C$	the set of inheritance relations,
$P \subseteq \Sigma$	the set of all attributes,
$\pi : P \to C$	the attribute mapping, a total function.

ι^+ is the set of relations of ι under transitive closure. This means that $x, y, z \in C \mid (x, y) \in \iota^+ \wedge (y, z) \in \iota^+ \implies (x, z) \in \iota^+$, and specifically, $x \in C \implies (x, x) \in \iota^+$. Note that attributes are considered to be nothing more than names. Their types are abstracted away from because of space constraints and they are not essential to explain the approach. Similarly, cardinalities of associations are not modelled in this definition.

All incoming associations defined by inheritance $in^* : C \to \mathcal{P}(A)$ are the set of incoming associations of the class or its parents:

$$in^*(c) = \left\{a \in A \mid \exists x, y \in C, \alpha(a) = (x, y) \wedge (c, y) \in \iota^+\right\}$$

All outgoing associations defined by inheritance $out^* : C \to \mathcal{P}(A)$ are the set of outgoing associations of the class or its parents:

$$out^*(c) = \left\{a \in A \mid \exists x, y \in C, \alpha(a) = (x, y) \wedge (c, x) \in \iota^+\right\}$$

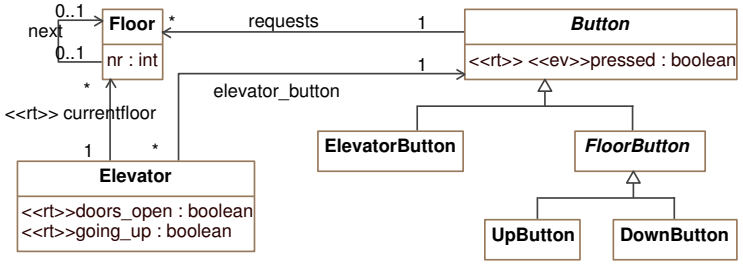

Fig. 3. The annotated metamodel E'.

All classes defined by inheritance that have an attribute $\pi^* : P \to \mathcal{P}(C)$ are the property-containing class and its subclasses:

$$\pi^*(p) = \left\{c \in C \mid \exists x \in C, \pi(p) = x \wedge (c, x) \in \iota^+\right\}$$

For example, the meta-model E in Fig. 1 can be described as:

$$\mathcal{M}_e = (C_e, D_e, A_e, \alpha_e, \iota_e, P_e, \pi_e),$$

where

$$\begin{aligned}
C_e &= \{Elevator, Floor, Button, ElevatorButton, FloorButton, \\
&\quad UpButton, DownButton\} \\
D_e &= \{Button, FloorButton\} \\
A_e &= \{currentfloor, requests, elevator_button\} \\
\alpha_e(a) &= \begin{cases} (Elevator, Floor) & \texttt{if } a = currentfloor \\ (Button, Floor) & \texttt{if } a = requests \\ (Elevator, ElevatorButton) & \texttt{if } a = elevator_button \end{cases} \\
\iota_e &= \{(ElevatorButton, Button), (FloorButton, Button), \\
&\quad (UpButton, FloorButton), (DownButton, FloorButton)\} \\
P_e &= \{nr, doors_open, going_up, pressed\} \\
\pi_e(p) &= \begin{cases} Floor & \texttt{if } p = nr \\ Elevator & \texttt{if } p \in \{doors_open, going_up\} \\ Button & \texttt{if } p = pressed \end{cases}
\end{aligned}$$

An annotated meta-model is an extension of a meta-model as defined in Formula 1:

$$\mathcal{M}' = (C, D, A, \alpha, \iota, P, \pi, S, \sigma), \tag{2}$$

where

$S \subseteq \{rt, ev\}$	the set of all supported annotations
$\sigma : C \cup A \cup P \to \mathcal{P}(S)$ $\{rt, ev, tr\}$	the annotation mapping.

All concepts (classes, associations and attributes) can be annotated with:

- *rt*: run-time, annotates a dynamic concept that serves as output (*e.g.,* a state variable);
- *ev*: an event, annotates a dynamic concept that serves as input only (*e.g.,* a marking).

More annotations are possible, but the generation of the sub-languages currently only supports those two. For example, the annotated meta-model of an elevator control

DSML, shown in Fig. 3 can be described as:

$$E' = (C_e, D_e, A_e, \alpha_e, \iota_e, P_e, \pi_e, S_e, \sigma_e), \tag{3}$$

where

$$S_e = \{rt, ev\}$$

$$\sigma_e(x) = \begin{cases} \{rt\} & \texttt{if } x \in \{doors\_open, going\_up, current\_floor\} \\ \{ev, rt\} & \texttt{if } x = pressed \end{cases}$$

In this meta-model, the language engineer specifies that *Floor*, *Elevator* and *Button*, the associations *requests* and *elevator_button* and the attribute *nr* are static as they are not annotated. A button press is an input event, and *going_up*, *doors_open*, *currentfloor* and *pressed* are dynamic.

3.2 Generation of Sub-Languages

The annotated meta-model E' includes enough detail to generate the five sub-languages as shown in Fig. 2. The generated sub-languages are each expressive enough to serve their intent. At the same time they maximally constrain the modeller so that they are maximally domain-specific. The result of this generation process with E' as input (see Fig. 3) is shown in Fig. 4. Templates are used in the generation process, shown with grey classes. These templates consist of generic language constructs, that can be instantiated to create a sub-language. The meta-models of sub-languages are generated by a function that operates on E' so that only relevant elements are used and no more annotations are present so that the result is a regular meta-model.

We formalise the approach so that the definition of the sub-languages is precise and unambiguous. For every language, there is a language mapping function $f : \mathcal{M}' \times \mathcal{M}_t \to \mathcal{M}$ that returns the sub-language meta-model $\mathcal{M}_x = (C_x, D_x, A_x, \alpha_x, \iota_x, P_x, \pi_x)$ of an annotated meta-model $\mathcal{M}' = (C, D, A, \alpha, \iota, P, \pi, S, \sigma)$ and a template $\mathcal{M}_t = (C_t, D_t, A_t, \alpha_t, \iota_t, P_t, \pi_t)$. This template $\mathcal{M}_t$ is different for every sub-language. By default, a sub-language will simply consist of $\mathcal{M}'$ without annotations, but preserving all elements. We define this default mapping function as the function $weave$: $\mathcal{M}' \times \mathcal{M} \times \Sigma \to \mathcal{M}$. The result of $weave$ is defined as follows:

$$\mathcal{M}_x = (C_x, D_x, A_x, \alpha_x, \iota_x, P_x, \pi_x),$$

where the components of $\mathcal{M}_x$ are defined by $weave(\mathcal{M}', \mathcal{M}_t, Element)$ under the condition that $Element \in C_t$: $X_x = X \cup X_t$ for $X \in \{C, D, A, \alpha, P, \pi\}$ and $\iota_x = \iota \cup \iota_t \cup \{(c, Element) \mid c \in C \wedge \nexists s \in C, (c, s) \in \iota\}$.

In case of the elevator DSML, meta-model $\mathcal{M}_x$ is the union of E' and a given template $\mathcal{M}_t$, and a all elements of E' that do not have a superclass (in this case *Floor*, *Button* and *Elevator*), become a subclass of a given *Element* class in the template $\mathcal{M}_t$.

The meta-models of all five sub-languages are defined below, and their intent is explained. We explain the approach using the elevator control DSML of which the abstract syntax is defined by E'.

The design language E_d The design language allows modellers to design systems in a general way. The static system (*i.e.,* its structure) is defined, and state or configuration information is not taken into account in this language. Its generated meta-model is

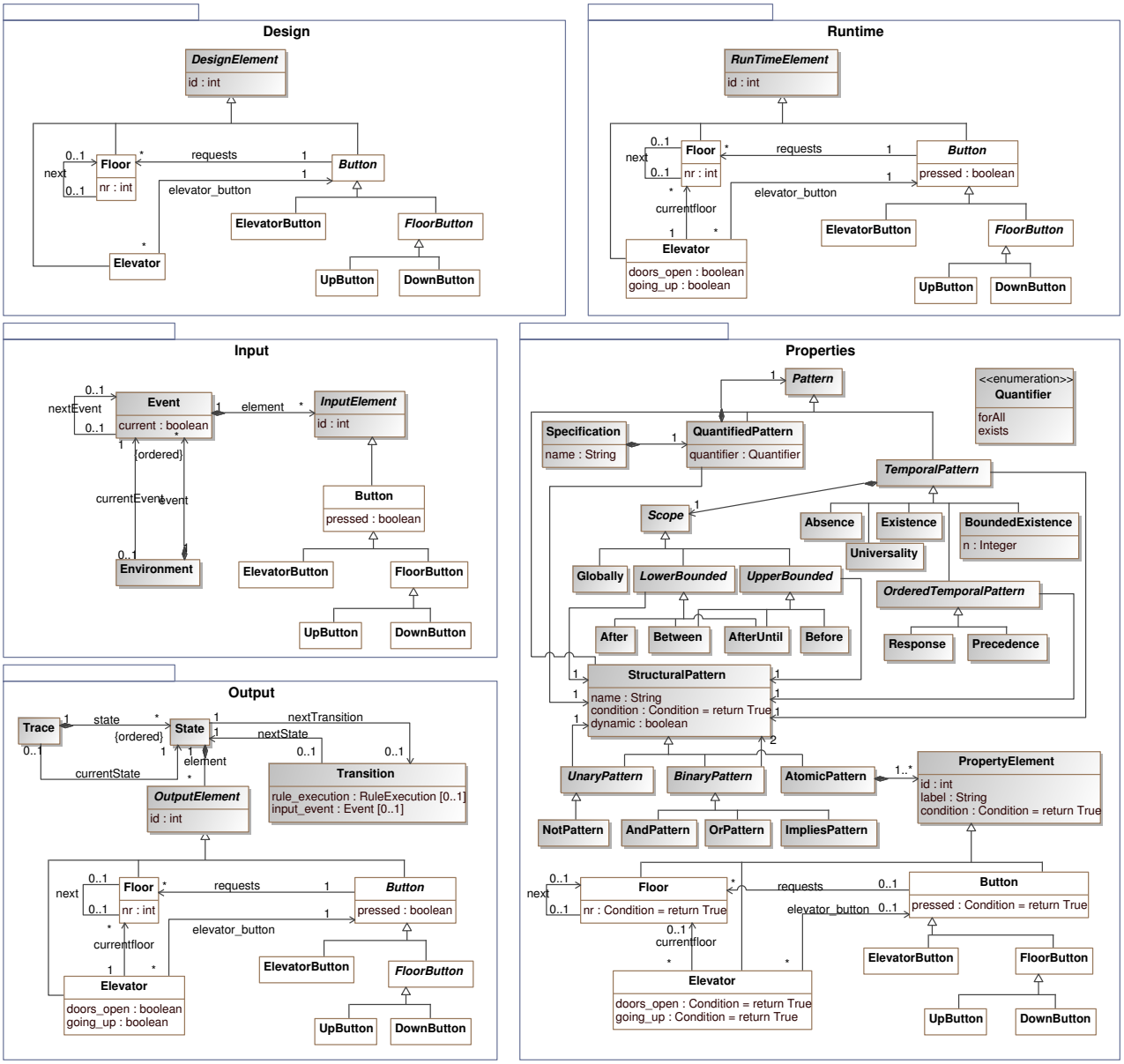

Fig. 4. The meta-models of the five sub-languages of E'.

shown in the top left of Fig. 4. In the generation process, all constructs (classes, associations and attributes) of E' annotated with *rt* and *ev* are removed. The template consists of a single *DesignElement* class with an *id* that has to be unique. This id will be used to refer to link class instances of the DSML. No dynamic constructs are available in E_d, so the modeller can only model the structure of a system (*e.g.,* how *Floor*s and *Button*s are linked), not its state.

$$\mathcal{M}_d = (C_d, D_d, A_d, \alpha_d, \iota_d, P_d, \pi_d),$$

where the components of $\mathcal{M}_d$ are defined by $design(\mathcal{M}', \mathcal{M}_t)$:

$$\begin{aligned}
C_d &= \{c \mid c \in C \cup C_t \wedge rt, ev \notin \sigma(c)\} \\
D_d &= D \cup D_t \cap C_d \\
A_d &= \{a \mid a \in A \cup A_t \wedge rt, ev \notin \sigma(a)\} \\
\alpha_d &= \alpha \cup \alpha_t \\
\iota_d &= (C_d, p_2(\iota) \cup p_2(\iota_t) \cup \{(c, DesignElement) \mid c \in C \wedge \nexists s \in C, (c, s) \in \iota\}) \\
P_d &= \{p \mid p \in P \cup P_t \wedge rt, ev \notin \sigma(p)\} \\
\pi_d &= \pi \cup \pi_t
\end{aligned}$$

where p is the projection operation and $p_i(x)$ denotes the element of x with index i.

The run-time language E_r The run-time language enables modellers to define a state of the system, *e.g.,* an initial state as input of a simulation. It can also be used to visualise a "snapshot" or state of a system, during run-time. Its generated meta-model is shown in the top right of Fig. 4. In the generation process, all constructs of E' are preserved. The template consists of a single *RunTimeElement* class with an *id*. In E_r, all information, but structure and state (*e.g., currentfloor*), is available. As all constructs of the annotated meta-model are preserved, the meta-model of a run-time language can be defined as $\mathcal{M}_r = weave(\mathcal{M}', \mathcal{M}_t, RunTimeElement)$ with $\mathcal{M}_t$ the template described above.

The input language E_i The input language lets the modeller model the environment of a system, by *e.g.,* modelling an input scenario. Its generated meta-model is shown in the middle left of Fig. 4. In the generation process, all constructs of E' that are not annotated with *ev* are removed. This means that classes that are not annotated with *ev* are removed if they do not inherit an association or attribute that is annotated with *ev*. The template models an *Environment* as an *Event* list containing *InputElements*. In E_i, a series of inputs can consist of button presses. For now, we assume that at most one button can be pressed in the same event, meaning that an event should not contain two unattached elements. If the language engineer decides that more than one or exactly one button can be pressed at the same time, he can create a variant of this template.

$$\mathcal{M}_i = (C_i, D_i, A_i, \alpha_i, \iota_i, P_i, \pi_i),$$

where the components of $\mathcal{M}_i$ are defined by $input(\mathcal{M}', \mathcal{M}_t)$:

$$\begin{aligned}
C_i = \{c \mid c \in C \wedge ((ev \in \sigma(c)) \\
&\vee (\exists p \in P, ev \in \sigma(p) \wedge c \in \pi^*(p)) \\
&\vee (\exists a \in in^*(c) \cup out^*(c), ev \in \sigma(a)))\} \cup C_t \\
D_i = D \cup D_t \cap C_i \\
A_i = \{a \mid a \in A \wedge ev \in \sigma(a)\} \cup A_t
\end{aligned}$$

$$\alpha_i = \alpha \cup \alpha_t$$
$$\iota_i = \iota \cup \iota_t \cup \{(c, InputElement) \mid c \in C \wedge \nexists s \in C, (c, s) \in \iota\}$$
$$P_i = \{p \mid p \in P \wedge ev \in \sigma(p)\} \cup P_t$$
$$\pi_i = \pi \cup \pi_t$$

The output language E_o The output language can be used to represent execution traces of a simulation. An output model is usually generated by a simulator or as a counter-example by a verification tool, but can be generated manually as well for *e.g.,* modelling an oracle for a test case. Its generated meta-model is shown in the bottom left of Fig. 4. In the generation process, all constructs of E' are preserved. The template consists of a *Trace* of *State*s and *Transition*s. This language is able to express a sequence of system states and the intermediate operations that caused the state change (a rule application in the operational semantics $E_{[[.]]}$, and/or an input event). The output of $E_{[[.]]}$, or the counter-example in verification are instances of E_o. Due to the possibly large number of elements in such an execution trace, an instance of E_o is stored more implicitly as text, and can be interpreted or "played out" by showing step-by-step an instance of the run-time language E_r. The meta-model of a output language can be defined as $\mathcal{M}_o = weave(\mathcal{M}', \mathcal{M}_t, OutputElement)$ with $\mathcal{M}_t$ the template described above.

The properties language E_p The properties language allows the user to define temporal properties, which are properties on the behaviour of systems. Its generated meta-model is shown in the bottom right of Fig. 4, is constructed from four components.

[A] The quantification of the formula by (i) *forAll* or *exists* clause(s), and (ii) corresponding structural pattern(s). The modeller can choose to model a property for all elements that match the associated structural pattern. This structural pattern is evaluated on the design model, and can thus not refer to run-time concepts. 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, if they have the same label. Quantification patterns can be nested, or can contain a temporal or structural pattern.

[B] The temporal pattern, based on Dwyer's specification patterns [7]. 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 [7]. Six patterns are supported, to express the absence, existence, bounded existence, universality response or precedence for given proposition(s). Additionally a scope can be defined: must the pattern be valid globally, or after, before, in between or after until the occurrence of

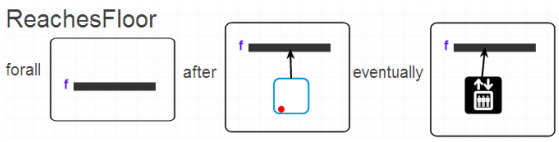

Fig. 5. The *reachesFloor* property as an instance of E_p.

given proposition(s). In total up to four proposition variables can be used in a temporal pattern, and we implement them as structural patterns, that represent patterns on the state of the system at run-time.

[C] The structural pattern, based on PaMoMo [11], for both static (when used in a quantification pattern) as well as dynamic (when used in a temporal pattern) models. Using a structural pattern, a query can be defined on a model. If the pattern is static, it returns all bound variables in found matches, and if it is dynamic it returns *true* if at least one match is found or *false* when no match is found. In our current approach, we use simple patterns (*e.g.,* the elevator is at a given floor) and an ad-hoc matching algorithm, but we intend to re-use the matching algorithm presented in [9]. Only a small part of PaMoMo's expressiveness is included in the property language, but this suffices for defining most properties. A *StructuralPattern*, and a *PropertiesElement* can hold a condition, which returns *true* by default and is in our current approach modelled as a string.

[D] The pattern elements, based on the RAM process [8,9]. The elements of a structural pattern are based on E' but need to be changed in several ways in order to allow the modeller 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. 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.

The properties pattern is composed of parts *A*, *B* and *C*, which are generic. Only component *D*, depends on E' that is subjected to the RAM process for left-hand side patterns [9]. Let us define the function $RAM : \mathcal{M}' - \mathcal{M}' \rightarrow$ that performs the RAM process for left-hand side patterns on an annotated meta-model $\mathcal{M}'$, resulting in a RAM-ified meta-model $\mathcal{M}_{RAM}$:

$$RAM(\mathcal{M}') = (C_{RAM}, \varnothing, A_{RAM}, \alpha_{RAM}, \iota_{RAM}, P_{RAM}, \pi_{RAM}, S, \sigma),$$

then the meta-model of a properties language can be defined as:

$$\mathcal{M}_p = weave(RAM(\mathcal{M}'), \mathcal{M}_t, PropertiesElement),$$

with $\mathcal{M}_t$ the template described above.

Generation of concrete syntax of the sub-languages The concrete syntax of each of the sub-languages is defined by the union of the concrete syntax of E' of which an example is shown in the top right of Fig. 1 (possibly leaving out removed concepts in case of the design and input language) and the predefined concrete syntax for the template $\mathcal{M}_t$.

An instance of the design language looks like the traditional instance of the DSML but without run-time concepts. In the case of E_d, it is impos-

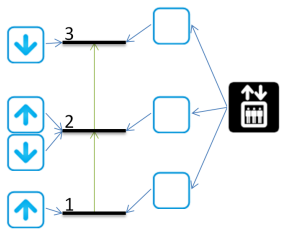

Fig. 6. The system from Fig. 1 modelled in the design language, without run-time information.

sible to model whether buttons are pressed, on which floor the elevator is, whether its doors are open and in what direction it is going. The system-under-study of Fig. 1, now modelled in E_d, is shown in Fig. 6. An example instance of the run-time language E_r looks the same as the traditional instance of the DSML, shown at the top right in Fig. 1.

An instance of the input language is not used in the context of verification by model checking. Its concrete syntax is a sequence of connected events represented as green circles containing the events visualised using the concrete syntax as shown in the top right of Fig. 1. Each step of an instance of the output language can be visualised as a run-time instance. Alternatively, it can be visualised completely at once as red circles containing the states, connected by arrows with the transition event(s) as label. An instance of the property language E_p is shown in Fig. 5. It uses a combination of text and domain-specific patterns.

3.3 Generation of Mappings for Model Checking With the SPIN Environment

Verification is automated in five steps, as depicted in Fig. 2.

Step 1: Transformation to LTL and Promela As shown in Fig. 2, a generic transformation generates the LTL formula and the Promela model by means of a model-to-text transformation. The operation results in a *.pml* file, in the example called *Elevator.pml*, that serves as input for the SPIN verification tool. The *.pml* file is generated from a number of models, and its overall structure is shown in Listing 1.1, where code snippets are referenced between $<$ and $>$. The role of each model in the compilation process is discussed below.

The design meta-model (line 3 in Listing 1.1): The design meta-model, in our case E_d is translated to a number of Promela `typedef`s. Only the three classes on top of the inheritance hierarchy become Promela types. Their instances are stored as static arrays, and instances are accessed by indexing that array. Since Promela is not an object-oriented language, inheritance and associations has to be encoded in a particular way as shown in Listing 1.2). For the types on line 1-21, inheritance is implemented by the `__subtype` attribute, that refer to any class in the design meta-model. Associations are implemented with bidirectional accessibility by `short`s, that refer to the index of the target, rather than an object. For instance, if the `currentfloor_out` of an `Elevator` is 1, its target is the `Floor` with index 1. If a target is the *null* object, its index is set to -1. The Promela `typedef`s are also influenced by the model of the initial configuration of system-under-study (modelled as a run-time instance), which is modelled as a `__System` type on line 22-26, with static arrays of 7 Buttons, 1 Elevator and 3 Floors. These numbers are extracted from the run-time instance and are predetermined, as the number of buttons, elevators and floors are static. Suppose they are not static, then a maximum number must be set because SPIN requires the state to be bounded. On line 27 the system is created, and values should be filled in (see below).

The output meta-model (line 4 in Listing 1.1): A function called `print_state` (not shown) is defined that prints the current state of the system in a predefined encoding. Only run-time concepts are printed. This, in combination with printing the input events and the applied rule (done in the rule schedule code, which is not shown), provides all the necessary information to construct an output trace.

A run-time instance (line 5-6 in Listing 1.1): After all type, variable and function declarations, the process declaration starts on line 5. Only one process is used. It starts with the initial configuration of the system (line 6, not shown in detail), by setting all values of `s` (declared in Listing 1.2 on line 27). This results in the initial state of `s`.

The operational semantics (line 7-14 in Listing 1.1): At line 7, the initial rule, in our case *opendoor_up*, is scheduled using a *go to* statement that jumps to one of the rules at line 10-13 (one of which is shown in Listing 1.5). This rule schedule is generated from the operational semantics model of Fig. 1. Upon evaluation of the rule, a boolean variable will be set that denotes whether the rule was successfully applied or failed to match. If the rule was applied, execution is continued at the environment section (line 15) and the next rule is scheduled by setting a variable according to the operational semantics model. If the rule fails, the rule schedule decides to try the next rule according to the operational semantics model. If all rules fail, code continues execution at the `SKIP_RULE` label on line 14, printing the state and subsequently continuing to the environment section. Fig. 7 shows the *movedown_last* rule of the schedule at the bottom of Fig. 1. Its Promela code resides in the overall structure of Listing 1.1 at one of the rules that are referenced at line 10-13, and is fully shown in Listing 1.5. For performance, the generated code uses a `d_step` to calculate the rule matching as an atomic step. The code generator traverses the left-hand side pattern of Fig. 7 element by element by following associations in the pattern. In Promela, the match candidates are represented by indices of `s` (line 4). The code consists of nested for loops, where match candidates are traversed checked that (1) they are not *null* (*i.e.,* the match candidate is not -1), (2) if applicable, they are not the same as a previously matched item, (3) if applicable, their dynamic type, represented by the `__subtype` attribute, is correct, and (4) if applicable, node conditions that are specified are satisfied (in case of the *movedown_last* rule the elevator should have its doors open and should go down - line 12-13, not visual in Fig. 7). When a match for the pattern is found, the right-hand side (RHS) of the rule is applied (line 26-29), which is generated from the difference between the RHS and the left-hand side of the rule. The rule is flagged successful, the state of the LTL propositions is updated (see below), and the rule is exited on line 30-32. Finally at line 43, the execution jumps back to the rule schedule, which will decide the next step.

The input meta-model (line 15-17 in Listing 1.1): At line 15, a model of the environment like Listing 1.3. It consists of an atomic block containing an if-statement. The if-statement in Promela non-deterministically chooses an option for which the guard (in this case "1") is true. This environment model thus selects a possible event that will be input for the system (lines 3-9), or none (line 10). For each event, a print statement is generated. The numbers on the left side of the dot are the node id attributes of the node as presented in Fig. 4, and can be used to denote a specific node. In this case, the *Elevator* instance has an id value of 0, the *Floor*s have id values of 1 to 3 and the *Button*s have id values between 4 and 10. Finally a jump to the `LOOP` label is generated (line 16 in Listing 1.1), so that the rule schedule can decide the next step.

A property instance (line 1-2 in Listing 1.1): The property instance, in our case *reachesFloor*, is translated to the LTL formula at line 1-3 of Listing 1.4. The LTL formula is composed by concatenating three times an Eventually pattern $\Box(!Q \vee \Diamond(Q \wedge \Diamond P))$ [7], as the property must hold for all (in this case three) floors. In Promela, it is

only allowed to specify LTL formulas without boolean expressions. Therefore, proposition variables are used in the LTL formula, and they are updated boolean expressions using when the `update_state` function is called (line 4-13). $Q0$, $Q1$ and $Q2$ represent the possible button presses at floors 0, 1 and 2, as defined by the middle pattern in Fig. 5. Note how the bound floor f is used in the boolean expressions to select the correct `s.button_` indices that match f. On line 9-11 $P0$, $P1$ and $P2$ represent the right pattern in Fig. 5, where it is checked whether the elevator is at floor f and its doors are open. The function `update_state` will need to be called every time the state of the system changes.

Step 2: Verification with SPIN Step 2 of the verification process shown in Fig. 2 is the automatic verification by SPIN on the Promela model (using the -a option). The LTL formula is checked on all possible execution traces. In this process, printing is suppressed. If the Promela model satisfies the LTL property, the verification is completed, and steps 3-5 are not followed. If the SPIN encounters a counter-example during verification, the verification process is terminated and a *.trail* file is generated, as shown in Fig. 2.

Step 3: Trace generation by SPIN In case of a counter-example, SPIN is used to perform a guided simulation using the trail on the Promela model (-t option). In this step, the print statements in the Promela model are executed, so that all relevant information about the counter-example is written to *Trace.txt*. In our example, one line in *Trace.txt* may look like: "`0.going_up=1; 0.doors_open=1; 0.currentfloor_out=2; 4.pressed=0; 5.pressed=0; 6.pressed=1; 7.pressed=1; 8.pressed=0; 9.pressed=0; 10.pressed=0;`". Other lines can show the transformation rule that is applied (*e.g.,* "`movedown_last`"), or the input that was generated by the environment model, as discussed before (*e.g.,* "`6.pressed=1`"). On the left side of each dot, the ids for model elements as presented in Fig. 4 are used to refer to the node in question. Depending on the type of the attribute/association, the value behind the equal sign is interpreted as boolean, integer or id. In case of class that can be created or deleted at run-time, all instances are printed out using newly assigned ids. For conciseness, associations are printed in one direction only.

Step 4: Transformation of the counter-example to the domain-specific level As shown in Fig. 2, the *Trace.txt* is transformed to an output model, making use of the design model to map corresponding ids. This results in an output model, that sequentially shows all the system states of the counter-example.

Step 5: Animation of the counter-example The output model can be "played" out step-by-step by visualising each state. As described in [6], one state is visualised as a run-time model, which may look like the instance model on the top right of Fig. 1.

To conclude, as shown in Fig. 2 *ProMoBox* enables the modelling and verification of properties while the user only has to provide the bare minimum of models: an annotated meta-model, the concrete syntax (implicit in Fig. 2), the operational semantics, the system he wants to verify, a configuration of the system, and the property.

4 Example and Evaluation

We implemented the *ProMoBox* framework in AToMPM [13], and the compiler that compiles models to and from Promela or text were written in Python.

```
<LTL FORMULA>
<UPDATE STATE FUNCTION DEFINITION>
<METAMODEL>
<PRINT STATE FUNCTION DEFINITION>
active proctype instance() {
  <INSTANCE>
  <SET INITIAL RULE>
  LOOP:
  <RULE SCHEDULE>
  <RULE 1>
  <RULE 2>
    ...
  <RULE N>
  SKIP_RULE: print_state();
  <ENVIRONMENT>
  goto LOOP;
}
```

Listing 1.1. The overall structure of the generated Promela model.

```
typedef Button {
  short __subtype;
  bit pressed;
  short requests_out;
  short elevator_button_in;
}
typedef Elevator {
  short __subtype;
  bit doors_open;
  bit going_up;
  short currentfloor_out;
  short elevator_button_out[3];
}
typedef Floor {
  short __subtype;
  short nr;
  short next_out;
  short next_in;
  short currentfloor_in;
  short requests_in[3];
}
typedef __System {
  Button button_[7];
  Elevator elevator_[1];
  Floor floor_[3];
}
__System s;
```

Listing 1.2. The compiled bounded meta-model.

```
atomic {
if
:: 1 -> s.button_[0].pressed=1;
        printf("4.pressed=1\n");
:: 1 -> s.button_[1].pressed=1;
        printf("5.pressed=1\n");
:: 1 -> s.button_[2].pressed=1;
        printf("6.pressed=1\n");
:: 1 -> s.button_[3].pressed=1;
        printf("7.pressed=1\n");
:: 1 -> s.button_[4].pressed=1;
        printf("8.pressed=1\n");
:: 1 -> s.button_[5].pressed=1;
        printf("9.pressed=1\n");
:: 1 -> s.button_[6].pressed=1;
        printf("10.pressed=1\n");
:: 1 -> skip;
fi;
}
```

Listing 1.3. The compiled environment model.

```
ltl reachesFloor {
  [](!Q0 || <>(Q0 && <>P0)) && [](!Q1 || <>(Q1 && <>P1)) && [](!Q2 || <>(Q2 && <>P2))
}
inline updatestate() { // called after the evaluation of a RHS
  d_step {
  Q0 = (s.button_[0].pressed == 1 || s.button_[3].pressed == 1);
  Q1 = (s.button_[1].pressed == 1 || s.button_[4].pressed == 1 || s.button_[5].pressed == 1);
  Q2 = (s.button_[2].pressed == 1 || s.button_[6].pressed == 1);
  P0 = (s.elevator_[0].currentfloor_out == 0 && s.elevator_[0].doors_open == 1);
  P1 = (s.elevator_[0].currentfloor_out == 1 && s.elevator_[0].doors_open == 1);
  P2 = (s.elevator_[0].currentfloor_out == 2 && s.elevator_[0].doors_open == 1);
  }
}
```

Listing 1.4. The compiled LTL formula.

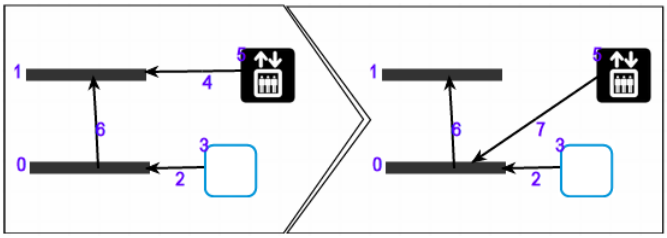

Fig. 7. The *movedown_last* rule.

```
MOVEDOWN_LAST:
MOVEDOWN_LAST_success = 0;
d_step {
short elevator5, floor1, floor0, button3, button3_candidate;
floor1 = 0;
do          // look for floor1 match
:: (MOVEDOWN_LAST_success == 0 && floor1 < 3) ->
  if          // check floor1 conditions
  :: (floor1 >= 0) ->
    elevator5 = s.floor_[floor1].currentfloor_in;
    if          // check elevator5 conditions
    :: (elevator5 >= 0 && s.elevator_[elevator5].doors_open == 0
        && s.elevator_[elevator5].going_up == 0) ->
      floor0 = s.floor_[floor1].next_in;
      if          // check floor0 conditions
      :: (floor0 >= 0 && floor0 != floor1) ->
        button3_candidate = 0;
        do          // look for button3 match
        :: (MOVEDOWN_LAST_success == 0 && button3_candidate < 3) ->
          button3 = s.floor_[floor0].requests_in[button3_candidate];
          if          // check button3 conditions
          :: (button3 >= 0 && s.button_[button3].pressed == 1) ->
            if          // global condition
            :: (s.floor_[floor0].nr < s.floor_[floor1].nr) ->
              // apply right-hand side
              s.elevator_[elevator5].currentfloor_out = -1;
              s.floor_[floor1].currentfloor_in = -1;
              s.elevator_[elevator5].currentfloor_out = floor0;
              s.floor_[floor0].currentfloor_in = elevator5;
              MOVEDOWN_LAST_success = 1; // for multi-loop break
              update_state();
              break;
            :: else -> skip; fi;
          :: else -> skip; fi;
          button3_candidate++;
        :: else -> break; od;
      :: else -> skip; fi;
    :: else -> skip; fi;
  :: else -> skip; fi;
  floor1++;
:: else -> break; od;
}
goto MOVEDOWN_LAST_schedule;
```

Listing 1.5. The compiled *movedown_last* rule.

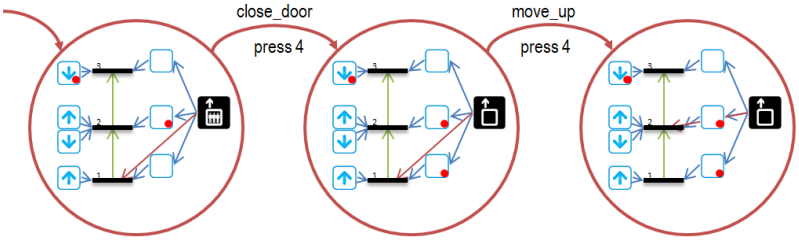

Fig. 8. The counter-example of the *staysAtSecondFloor* property.

We verified three properties on the modelled system with the configuration at the top right of Fig. 1:

- *reachesFloor*: when a button that requests the elevator to go to a certain floor is pressed, the elevator will eventually open its doors at that floor;
- *skipFloorOnce*: when a button that requests the elevator to go to a certain floor is pressed, the elevator will open its doors at that floor at the latest the second time it passes that floor;
- *staysAtSecondFloor*: when the elevator is at a certain floor, it stays at that floor. The system will not satisfy this property, and this it should yield a counter-example.

The properties are checked with SPIN [12] version 6.2.6 on a 64-bit Windows 7 SP1 PC with an Intel(R) Core(TM) i7 Q 720 CPU at 1.60 GHz 8 GB of DDR3 memory. The results are shown in Table 1. The properties *reachesFloor* and *skipFloorOnce* take more than a minute to evaluate, and use up to almost 2GB of memory as the depth of the search tree is more than 5×10^4. We can conclude that the performance of the approach in terms of time and memory consumption is acceptable but poor, as this can be considered to be a small example. Alternatively it is possible to evaluate up to a given search tree depth (using the -m option in SPIN) to obtain a fair confidence in the correctness of the modelled system.

As expected, the *staysAtSecondFloor* property yields a counter example. In that case, the verification only takes a very limited amount of time and memory. This turns out to by exemplary due to the relative simplicity of the LTL formula in comparison with the Promela system: if there is a counter-example, it is relatively quickly found. This raises the confidence of using a maximum depth for the SPIN verification.

Table 1. Verification results of the system with initial state as shown at the top right of Fig. 1.

property	counter-example	depth	# states	memory	time taken
reachesFloor	no	54422	8×10^6	1934 MB	104s
skipFloorOnce	no	54518	8×10^6	1934 MB	172s
staysAtSecondFloor	yes	255	127	0.226 MB	0.037s

5 Assumptions and Limitations

We now discuss the assumptions and current limitations of the *ProMoBox* approach.

Format of the DSL. It is assumed that we can express the abstract syntax of the DSML as a meta-model, its concrete syntax is defined graphically by icons for every abstract

syntax concept and its semantics are given by a transformation model with a rule schedule supporting control flow.

Boundedness. The rule-based nature of the operational semantics ensure a step-wise, state-based semantics. In its current state, *ProMoBox* supports DSMLs that have a notion of state. Since we apply model checking, the possible number of states must be bounded. In the example, this is assured by the limited cardinality of the run-time elements (especially the *currentfloor* association). If such boundedness is not achieved in the meta-model because of an infinite cardinality value, this value must be bounded in order to allow model checking. Such abstraction operations (including decreasing state spaces that are bounded but too large) are nonetheless key to modelling in SPIN, and are beyond the scope of this paper.

Format of the properties. The only type of properties that is currently supported is based on LTL. However, properties language also supports quantification and structural patterns, so the approach can be considered representative for a wide range of properties. Although we cannot provide any proof, we feel that the *ProMoBox* approach described in this paper can be reused for different kinds of properties by defining generic mappers to tools supporting model checking with OCL and CTL, real time properties, or properties using distributions. The target tool has to be expressive enough so that a correct structure and operational semantics can be defined, *i.e.,* all elements can be queried, variables can be stored and throughout the evaluation of the temporal formula (context-dependency), etc. The key of the approach is that it is defined on the meta-level formalisms (class diagrams, concrete syntax definitions, and rule-based transformation with scheduling), in combination with pre-defined, generic templates.

Scalability. Scalability remains the main concern however. On the one hand, model checking as a technique is a cause of scalability limitations, on the other hand generates the Promela code generator generic code, which could be optimised. A radically different solution to the problem of scalability would be not to map to a model checking approach, but instead use test case generation techniques to generate relevant test cases in the form of input models and output models (oracles). Tests are executed by using the input models as initial state, applying the operational semantics transformation, and comparing (by using model comparison, *e.g.,* the DSMDiff algorithm [14]) the resulting trace with the oracle. This illustrates how *ProMoBox* benefits from its modelling approach, because mappings to different semantic domains can be implemented. However, this research direction is not investigated for the *ProMoBox* approach.

6 Related Work

With respect to the contribution of this paper, we distinguish two 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 modelling language. Second, we discuss approaches that have a more general view on providing specification and verification support for different modelling languages.

Specific Solutions. In the last decade, a plethora of language-specific approaches have been presented to define properties and verification results for different kinds of design-oriented languages. For instance, Cimatti et al. [15] have proposed to verify component-based systems by using scenarios specified as Message Sequence Charts

(MSCs). Li et al. [16] also apply MSCs for specifying scenarios for verifying concurrent systems. The CHARMY approach [17] offers amongst other features, verification support for architectural models described in UML. Collaboration and sequence diagrams have been applied to check the behaviour of systems described in terms of state machines [18–20]. Rivera et al. [21] map the operational semantics of DSMLs to Maude, and thus, benefit from analysing 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 (see [22] for a survey). They have in common that they 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 engineers 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 propose OCL extensions, often referred to Temporal OCL (TOCL), for defining temporal properties on models [23–25]. As OCL may be combined with any modelling language, TOCL can be seen as a generic model-based property language as well. In [26, 27] the authors discuss and apply a pattern to extend modelling 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 design 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 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 presented in this paper goes directly in this direction by enabling domain engineers to use their familiar notation for defining properties and exploring the verification results.

Another approach that aims to define properties on the model level in a generic way is presented in [28]. The authors extend a language for defining structural patterns based on Story Diagrams [29] to allow for modelling temporal patterns as well. The resulting language allows to define conditionally timed scenarios stating the partial order of structural patterns. The authors argue that their language is more accessible for domain engineers, because their language allow decomposition of complex temporal properties into smaller ones by if-then-else decomposition and quantification over free variables. 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. [7] are encoded in their language, but there is no language-inherent support to explicitly apply them. In our work, we tackle these two issues in the context of DSM by reusing the notation of domain engineers for specifying properties and providing explicit language support for specification patterns.

Finally, [30] present specification patterns for describing properties over reachable states of graph grammars. These specification patterns are purely defined on graph structures (*i.e.,* nodes and edges) and thus are reusable for any modelling language. However, the authors do not discuss integration with current modelling languages to use such specification patterns for specific properties. A possible line of future work may aim to integrate such specification patterns to our generic meta-model.

7 Conclusion and Future Work

We presented the *ProMoBox* approach, in which a minimum number of models is required as input to specify and check properties with SPIN and visualise possible counter-examples, while the user is shielded from the underlying formal methods. This is made possible by using annotations on the DSML meta-model to generate five sub-languages, and by compiling models to Promela and back. The key of the approach is that all information of the DSML is explicitly modelled. We presented the approach on a state-based DSML for elevator control. The process of evaluating properties using *ProMoBox* is described in detail, including a formal description of the generation of the sub-languages, and a compiler to Promela. Our results show that *ProMoBox* is applicable for current DSMLs and the resulting specification languages are usable by domain engineers.

For future work, we intend to use *ProMoBox* in a case study for gestural interaction [31]. In this case study, we plan to do more research on the performance of model checking using *ProMoBox*. Moreover, we plan to investigate how different property languages can be supported using different templates, and how these templates can be re-used, *e.g.,* an existing template for structural properties could be re-used in the properties template that is presented in this paper. We are also interested in broadening the types of languages that are supported by ProMoBox, *e.g.,* languages that explicitly include time. We expect that this would typically result in investigating associated templates for real-time properties.

References

1. Gray, J., Tolvanen, J.P., Kelly, S., Gokhale, A., Neema, S., Sprinkle, J.: Domain-specific modeling. Handbook of Dynamic System Modeling (2007)
2. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. In: ICSE. (2007)
3. Risoldi, M.: A Methodology For The Development Of Complex Domain Specific Languages. PhD thesis, University of Geneva (2010)
4. Visser, W., Dwyer, M., Whalen, M.: The hidden models of model checking. SoSym **11** (2012) 541–555
5. Meyers, B., Deshayes, R., Lucio, L., Syriani, E., Wimmer, M., Vangheluwe, H.: The ProMoBox approach to language modelling. Technical Report SOCS-TR-2014.3, School of Computer Science, McGill University (2014)
6. Meyers, B., Wimmer, M., Vangheluwe, H.: Towards domain-specific property languages: The ProMoBox approach. In: DSM. (2013)
7. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in Property Specifications for Finite-State Verification. In: ICSE. (1999)

8. Kühne, T., Mezei, G., Syriani, E., Vangheluwe, H., Wimmer, M.: Explicit transformation modeling. In: MoDELS Workshops. (2009)
9. Syriani, E.: A Multi-Paradigm Foundation for Model Transformation Language Engineering. PhD thesis, McGill University Montreal, Canada (2011)
10. Guerra, E., de Lara, J., Kolovos, D.S., Paige, R.F.: A Visual Specification Language for Model-to-Model Transformations. In: VL/HCC. (2010)
11. Guerra, E., de Lara, J., Wimmer, M., et al.: Automated verification of model transformations based on visual contracts. ASE **20** (2013) 5–46
12. Holzmann, G.J.: The Model Checker SPIN. TSE **23** (1997) 279–295
13. Syriani, E., Vangheluwe, H., Mannadiar, R., Hansen, C., Mierlo, S.V., Ergin, H.: AToMPM: A Web-based Modeling Environment. In: MoDELS Demonstrations. (2013)
14. Lin, Y., Gray, J., Jouault, F.: DSMDiff: A Differentiation Tool for Domain-Specific Models. European Journal of Information Systems **16** (2007)
15. Cimatti, A., Mover, S., Tonetta, S.: Proving and Explaining the Unfeasibility of Message Sequence Charts for Hybrid Systems. In: FMCAD. (2011)
16. Li, X., Hu, J., Bu, L., Zhao, J., Zheng, G.: Consistency Checking of Concurrent Models for Scenario-Based Specifications. In: SDL. (2005)
17. Pelliccione, P., Inverardi, P., Muccini, H.: CHARMY: A Framework for Designing and Verifying Architectural Specifications. TSE **35** (2008) 325–346
18. Brosch, P., Egly, U., Gabmeyer, S., Kappel, G., Seidl, M., Tompits, H., Widl, M., Wimmer, M.: Towards Scenario-Based Testing of UML Diagrams. In: TAP. (2012)
19. Knapp, A., Wuttke, J.: Model checking of UML 2.0 interactions. In: MoDELS'06. (2006)
20. Schäfer, T., Knapp, A., Merz, S.: Model Checking UML State Machines and Collaborations. ENTCS **55** (2001) 357–369
21. Rivera, J.E., Guerra, E., de Lara, J., Vallecillo, A.: Analyzing Rule-Based Behavioral Semantics of Visual Modeling Languages with Maude. In: SLE. (2008)
22. Gabmeyer, S., Kaufmann, P., Seidl, M.: A classification of model checking-based verification approaches for software models. In: VOLT. (2013)
23. Ziemann, P., Gogolla, M.: OCL Extended with Temporal Logic. In: PSI. (2003)
24. Kanso, B., Taha, S.: Temporal Constraint Support for OCL. In: SLE. (2012)
25. Bill, R., Gabmeyer, S., Kaufmann, P., Seidl, M.: OCL meets CTL: Towards CTL-Extended OCL Model Checking. In: OCL Workshop. (2013)
26. Zalila, F., Crégut, X., Pantel, M.: Leveraging Formal Verification Tools for DSML Users: A Process Modeling Case Study. In: ISoLA. (2012)
27. Combemale, B., Crégut, X., Pantel, M.: A Design Pattern to Build Executable DSMLs and Associated V&V Tools. In: APSEC. (2012)
28. Klein, F., Giese, H.: Joint structural and temporal property specification using timed story scenario diagrams. In: FASE. (2007)
29. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In: TAGT. (2000)
30. da Costa Cavalheiro, S.A., Foss, L., Ribeiro, L.: Specification Patterns for Properties over Reachable States of Graph Grammars. In: SBMF. (2012)
31. Deshayes, R., Palanque, P.A., Mens, T.: A generic framework for executable gestural interaction models. In: VL/HCC. (2013)