A Framework for Temporal Verification Support in Domain-Specific Modelling

Bart Meyers, Hans Vangheluwe, Joachim Denil, Rick Salay

Abstract—In Domain-Specific Modelling (DSM) the general goal is to provide Domain-Specific Modelling Languages (DSMLs) for domain users to model systems using concepts and notations they are familiar with, in their problem domain. Verifying whether a model satisfies a set of requirements is considered to be an important challenge in DSM, but is nevertheless mostly neglected. We present a solution in the form of *ProMoBox*, a framework that integrates the definition and verification of temporal properties in discrete-time behavioural DSMLs, whose semantics can be described as a schedule of graph rewrite rules. Thanks to the expressiveness of graph rewriting, this covers a very large class of problems. With *ProMoBox*, the domain user models not only the system with a DSML, but also its properties, input model, run-time state and output trace. A DSML is thus comprised of five sublanguages, which share domain-specific syntax, and are generated from a single metamodel. Generic transformations to and from a verification backbone ensure that both the language engineer and the domain user are shielded from underlying notations and techniques. We explicitly model the *ProMoBox* framework's process in the paper. Furthermore, we evaluate *ProMoBox* to assert that it supports the specification and verification of properties in a highly flexible and automated way.

- *B. Meyers is with Flanders Make vzw, Leuven, Belgium.*
  E-mail: bart.meyers@flandersmake.be
- *H. Vangheluwe and J. Denil are with the Department of Mathematics and Computer Science, University of Antwerp, Antwerp, Belgium.*
  E-mail: firstname.lastname@uantwerpen.be
- *H. Vangheluwe is with the School of Computer Science, McGill University, Montréal, Canada.*
  E-mail: hv@cs.mcgill.ca
- *R. Salay is with the Department of Computer Science, University of Toronto, Toronto, Canada.*
  E-mail: rsalay@cs.toronto.edu

1 Introduction

In *Domain-Specific Modelling* (DSM) [28] the general goal is to provide means for domain users to model systems in their problem domain. Techniques such as metamodelling and model transformation enable *language engineers* to create *Domain-Specific Modelling Languages* (DSMLs) in collaboration with domain experts. These DSMLs can be used by *domain users*. Current DSM techniques allow domain users to model at the domain level and simulate, optimise, and transform the model to other formalisms, synthesise code, generate documentation, etc.

Verifying whether a model satisfies its requirements is an important challenge in DSM [26], but is nevertheless mostly neglected by current DSM approaches. Verification has been achieved by translating models to formal representations. Logic-based formulas in formalisms such as Linear Temporal Logic (LTL) [68] and Computation Tree Logic (CTL) [24] are used to represent the temporal properties that need to be verified [72]. These temporal properties can be verified using *e.g.,* model checking techniques [14]. Currently, domain users need to have a profound knowledge of some logic to express properties. This violates the principles of DSM. Like design models, the level of abstraction for specification and verification tasks needs to be lifted to the domain level, as domain users should not be exposed to underlying technologies. Consequently, there is a consensus that in DSM, it is better to use a DSML as a property language instead of LTL or another temporal logic [8], [38], [69], [79], [86], [87], [89]. More precisely, DSM should not only address modelling the design of a system, but also its properties, its environment, its run-time state, and its execution traces, which should all be modelled at the domain level, in their own DSML. We take this to be a background assumption in our work.

In accordance with these DSM principles, various dedicated property DSMLs and tools have been developed. For example, a visual formalism called TimeLine, developed at Bell Labs, allows users to specify temporal constraints, which are automatically translated to LTL [79]. Such approaches result in very suitable tools and languages, but developing this tools comes with a high development effort. Moreover, in the context of DSM, a DSML is highly prone to change during the development cycle [77], thus additional effort is required to keep verification tools synchronised with the DSML.

Some modelling languages can be used with minor modifications for defining system designs as well as properties. The Mathworks' Simulink® language for specifying the behaviour of dynamic systems as block diagrams [51] can be reused for specifying properties, by simply adding an "assertion block" [50]. Petri nets [70], used to specify the behaviour of concurrent systems, can also be used to express temporal properties to which execution traces conform [69], *i.e.,* the execution trace of a system that is modelled as a Petri net. This provides an elegant solution, as the language, and possibly its semantics, can be re-used. This is not possible for all design DSMLs.

User-friendly patterns have been devised for temporal logic. Dwyer *et al.* [21] have defined specification patterns in terms of LTL and other temporal logics, claiming that their patterns are easier to use, and cover most of the specification they encountered (*i.e.,* over 90% [22]). Over time, given its use in theory and practice, the Specification Patterns have become a strong foundation to build on. The work has been very well cited, and is continuously been used over the years in high-quality work (*e.g.,* [3]).

Other approaches offer visual and generic modelling languages for specifying properties [40], but do not offer a domain-specific syntax, which is essential in DSM. Although these efforts allow users to specify properties in languages more appropriate than temporal logic, *no general DSM approaches exist today that provide dedicated support for verification as part of the DSML engineering process*.

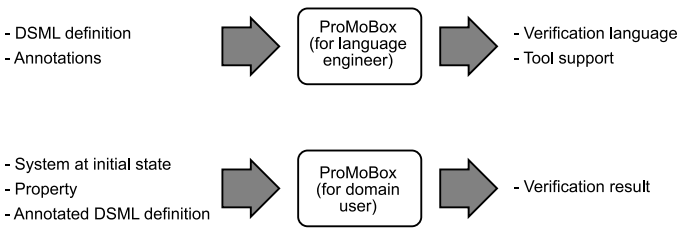

Figure 1. Overview of the *ProMoBox* approach.

Problem statement. In this paper, we address the problem of how to provide a general approach for specifying temporal properties at the domain level for behavioural DSMLs.

Contributions. The contribution of this paper is twofold. Firstly, aimed at language engineers, we have developed a novel semi-automated method and tooling for generating a domain specific property language for a DSML directly from the DSML definition. Secondly, aimed at domain users, we have developed an automated mapping to a verification backbone, which allows the domain user to verify properties and inspect counterexamples at the domain level.

This paper presents the *ProMoBox* framework, implementing these two contributions as follows. Firstly, the language engineer makes use of *ProMoBox* as shown in the upper part of Figure 1. During the language engineering phase, the framework includes a fully automated method to automatically generate verification support for a given design DSML, if additional *annotations* are provided. This includes a domain-specific verification language, and tool support including peripheral domain-specific *sublanguages* (for modelling environment, static design design, run-time state and execution traces of the system) to enable the specification and verification of these properties. Secondly, the domain user can use this mapping to verify properties at the domain level, as shown in the lower part of Figure 1. *ProMoBox* provides a fully automated mapping to a suitable verification backbone for model checking (temporal) properties. It takes as input a system modelled in the DSML, a property modelled in the newly generated verification language, and the DSML definition with annotations, to fully automatically produce a verification result (*i.e.,* an output trace in case of a counterexample). All inputs and outputs of both *ProMoBox* processes are lifted to the domain level, so that users (domain users and language engineers alike) are shielded from the underlying temporal logic and formal models. Flexibility and automation are key in *ProMoBox*. *ProMoBox* supports definition and verification of temporal properties for any discrete-time behavioural DSML, for which the semantics can be described as a schedule of graph rewrite rules.

ProMoBox is implemented using the modelling tool *AToMPM* [83]. This paper introduces the *ProMoBox* framework using the elevator controller running example presented in Section 2.3. In fact, the running example is inspired by an elevator controller model used to illustrate model checking [54]. This paper shows a mapping to the *Spin* model checker [35] and the specification patterns by Dwyer *et al.* [21] as verification backbone in detail. Furthermore, we evaluate the effort, correctness, model checking

performance, expressiveness, flexibility and limitations of using the *ProMoBox* framework.

The paper sets the stage in Section 2, where the necessary background is given on the subjects of DSM, running example, process modelling with the Formalism Transformation Graph and Process Model, temporal logic and model checking. An overview of the *ProMoBox* approach is given in Section 3. In Section 4, the *ProMoBox*'s sublanguages are presented, and it is explained how they are generated. Section 5 discusses the mapping to a verification backbone. Section 6 shows how we applied *ProMoBox* to our Elevator DSML, and demonstrates how properties are checked and how a counterexample can be produced. In Section 7, the *ProMoBox* framework is evaluated. Limitations of the approach are presented in Section 8. Section 9 discusses related work, and Section 10 concludes the paper and gives some directions for future work.

2 Background

This section presents the prerequisites to discuss the *ProMoBox* framework.

2.1 Modelling Language Engineering

The basis for this paper can be found in Domain-Specific Modelling (DSM) [38]. DSM is part of model-driven engineering (MDE). It requires modelling systems using domain concepts, rather than concepts in the solution domain (*i.e.,* the computing domain). Consequently, systems are modelled at a higher level of abstraction, and often code is generated from these high-level models. This means that the development process is split into two tasks: (i) the creation of a Domain-Specific Modelling Language (DSML) known as *engineering a language*, by the *language engineer*, in consultation with domain experts, and (ii) modelling the system using this DSML by a problem domain (but not solution domain) user, referred to as a *domain user*. Once the DSML is created, systems in the domain can be modelled in the DSML. Since the challenges addressed in this paper are an addition to language engineering, this section will mainly discuss the language engineering phase (as opposed to the model engineering phase) of the development process.

The three main aspects of a DSML, or modelling language in general, are its *abstract syntax* (describing the internal structure of a model, as a typed abstract syntax graph), the *concrete syntax* (describing how a model is represented, *e.g.,* in 2D vector graphics or in textual form) and its *semantics* (describing what a model means) [63].

In this paper, we assume that the abstract syntax is described by a metamodel [44] in the form of a class diagram [2] or a similar formalism, including additional static semantics specified as constraints. Static semantics can be described in a constraint language such as the Object Constraint Language (OCL) [65]. Instance models of the DSML are said to *conform to* or *typed by* the metamodel. In [44] this relation is referred to as a *linguistic instance of*, and throughout this paper, *instance (of)* will refer to this kind of instantiation. Conformance usually means that there is a morphism between the instance model and the metamodel, both typed, attributed, directed graphs.

The abstract syntax is represented by at least one concrete syntax, either textually or graphically (or a combination of both). Mappings between the abstract syntax and its concrete syntax(es) are called *rendering functions*, and their inverses are called *parsing functions*. These functions are used by model editors to render (changes of) the model's abstract syntax, and to parse user's concrete syntax edits to abstract syntax. In terms of graphical concrete syntax, we use a connection-based syntax [18] in this paper. This means that concrete syntax models consist of nodes and icons (one kind for each metamodel class) that are connected by edges (for each metamodel association).

Semantics can be described transformationally (also called "denotationally") or operationally [63]. Transformational semantics provide a mapping of a model in a given modelling language onto a model in a different modelling language for which a semantics is available. Operational semantics capture explicitly how a model can be executed, which we also call *simulating* the model, which is effectively mapping a model onto a trace. In this paper, semantics are always formalised as transformations, that can transform (an) input model(s) to (an) output model(s), instances of the same or different languages. Since models are represented as graphs, a popular way to specify transformations is by means of graph transformation rules.

2.2 Transformation Models

Transformations can be explicitly specified as transformation models [7], in a language that combines generic transformation concepts such as rules and a rule schedule, and concepts specific to the languages it transforms. A process called RAMification to generate such a rule-based domain-specific transformation language for given input and output DSMLs is presented by Kühne *et al.* [46].

A *rule schedule* of a transformation language is generic, and allows the modelling of how different rules are scheduled. *Rules* consist of a *left-hand side* (LHS) containing a pattern representing a condition, and a *right-hand side* (RHS) containing a pattern representing an action (elements can be created, removed or updated). LHS and RHS are generic language constructs that can contain elements, each displayed as a different shape of container. The contained elements form graph patterns that reuse concrete syntax taken from the input and trace language. When a rule is evaluated, a match for the LHS is searched for in the input model. If a match is found, the RHS is *applied* to the input model, thus changing it. If the rule *fails* to match, the input model is left untouched. Depending on the outcome (failure or application), the next rule according to the rule schedule is evaluated. Rule schedules may be implicit. An example of implicit rule schedules is that the next applied rule is chosen (pseudo-)randomly from the set of all matching rules.

The name RAMification for the process of generating a domain-specific transformation language is an abbreviation of its three stages to generate a pattern language (for LHS) and action language (for RHS) from the DSML:

1) *Relaxation*: the metamodel is relaxed by removing constraints in order to allow the creation of partial models, to uniquely identify elements across the different parts (LHS/RHS) of a rule;
2) *Augmentation*: each element in the metamodel is augmented with a label, and with an option whether subtypes should be potential match candidates. Other augmentations can be used to influence the matching process [46] but are not relevant for this paper;
3) *Modification*: for (attributes of) elements of the LHS, conditions over values are specified because the LHS pattern essentially models a condition or constraint that must be

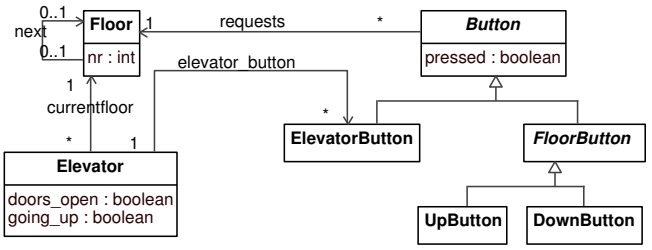

Figure 2. The metamodel of the Elevator controller DSML that serves as the running example throughout this paper.

met. Similarly, for elements of the RHS, assignments of values are specified.

Since the result of the RAMification transformation is a language definition with metamodel and concrete syntax model, the language engineer can adapt the concrete syntax model to make rules more appealing. The three stages of the RAMification process are explained more in detail using the running example in Section 2.3.

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

2.3 Running Example

The running example that is used throughout this paper is a DSML for elevator controllers. Our DSML enables modelling a building with floors, elevators and buttons. Additionally, it defines the step-wise (discrete-time) behaviour of an elevator system, such as moving up or down to a different floor, closing or opening elevator doors, or pressing buttons. The example is inspired by an elevator controller model used to illustrate model checking in [54]. A similar "Elevator Control System" is presented by Strobl and Wisspeintner [80] where it is used to demonstrate AutoFocus, a tool for modelling embedded systems [36]. Although embedded systems modelling is not the focus of this paper, the work illustrates the complexity of elevator controllers.

Figure 2 shows the metamodel of the Elevator language which we will call *E*. Elevators move between Floors responding to Button press requests. A Button requests exactly one Floor. Floors are ordered by the next association and a derived attribute nr representing the Floor number. At any time, an Elevator is at exactly one Floor, modelled by the currentfloor association. An ElevatorButton is a button inside an Elevator, allowing a passenger to request going to a certain Floor. At every Floor, there can be an UpButton to request to go up and a DownButton to request to go down. An Elevator can have its doors open (in that case it cannot move) and has a direction (up or down).

A concrete syntax model for the Elevator language in the form of icons and arrows is shown in Figure 3. For every element of *E*, there is exactly one associated Icon or Link. An icon/arrow inside a dashed box with a class or association name beneath it associates that concrete visual syntax with the corresponding abstract syntax. An icon consists of graphical elements, and text elements. This model defines the appearance of instance elements and how these appearances can change (*i.e.,* text content of text elements, or colours or transparency of graphical elements) in concert with their associated abstract syntax, by implementing specific rendering and parsing functions. In this way, as shown in the code fragment connected to all ButtonIcons, a button instance icon has a white fill which will becomes shaded if the associated button instance's attribute value for pressed is *true*, *i.e.,* when the button is lit. If the elevator is going up (going_up is *true*), then only the upward arrow is visible. If going_up is *false*, only the downward arrow is visible. Whether or not the doors are open is also visualised.

Figure 4 shows an instance model with three floors, one elevator and seven buttons, that uses the concrete syntax. As defined in the concrete syntax model, pressed buttons are shaded, and they are connected to the floor they request. On the middle floor, a button is pressed by someone requesting to go down, and inside the elevator the button to go to the top floor has been pressed. The elevator is currently at the bottom floor. Its doors are closed and its current direction is down. Note how all elements, including the links, conform to the metamodel of Figure 2. For simplicity, we limit ourselves to models with only one elevator. A system with multiple elevators is considerably more complex and would not contribute significantly to explaining the approaches presented in this paper.

The approach in this paper requires the definition of operational semantics. Figure 5 shows the transformation model $E_{[[.]]}$ specifying the operational semantics, that uses the instance model as input. This model is a *rule schedule*, and determines how different rules are scheduled. *Rules* consists of a *left-hand side* (LHS) containing a model pattern, and a *right-hand side* (RHS) containing an action. When a rule is evaluated, a match for the LHS pattern is searched for in the input model. If a match is found, the RHS is applied to the input model, thus changing it in-place. In the case that the rule was *applied*, an outgoing *success* link in the schedule (depicted as a black arrow) is followed. If no match is found, the input model is unchanged and an outgoing *notApplicable* link (depicted as a grey arrow) is followed. Execution starts at openDoor_up, since it has the isStart flag set (depicted as a bold line of the rule).

Inspired by a real elevator controller, the following rules implement how the elevator changes floors (one at a time), and opens and closes its door to honour the requests of users (modelled as pressed buttons). Three rules implementing this behaviour are shown in Figure 6. When a request for a floor is made for a different floor than the elevator's current floor, the doors close so that the elevator can start moving. This is modelled in the closeDoor rule shown in Figure 6. The LHS shows the pattern denoting an elevator with open door on a floor. Its direction is greyed out as it is not relevant to match this pattern, *i.e.,* it will match any direction. The pattern also shows a lit button (any subtype of Button, hence the icon of the abstract Button class is shown) on another floor, corresponding to the condition described above. The number labels on the top left of each pattern element serve as the relationship between LHS and RHS. Elements in the RHS with the same label, are the same elements. The action in the RHS here denotes that none of the matched elements are changed, except for the

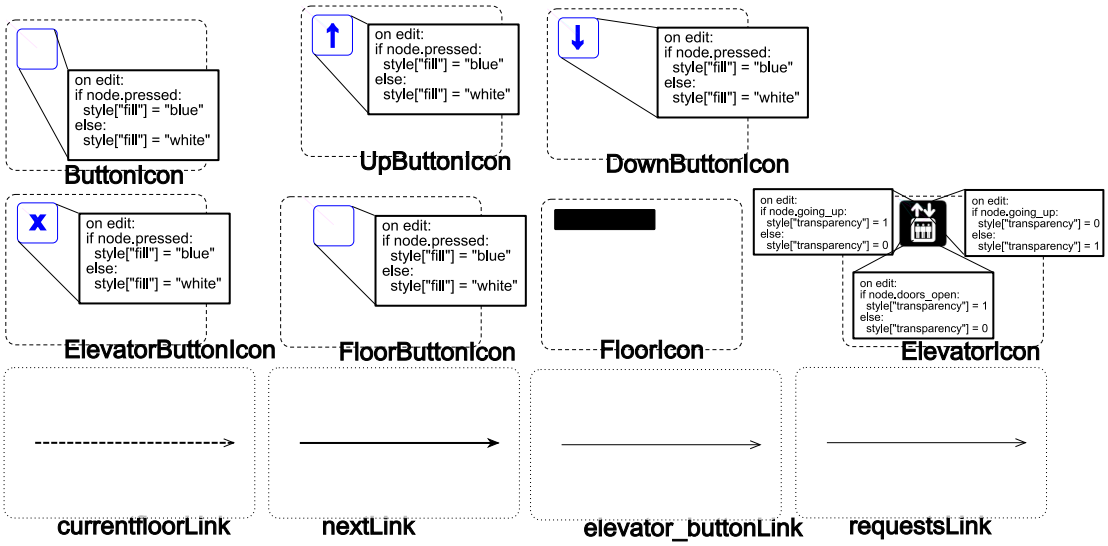

Figure 3. The concrete syntax model of the Elevator controller DSML that serves as the running example throughout this paper.

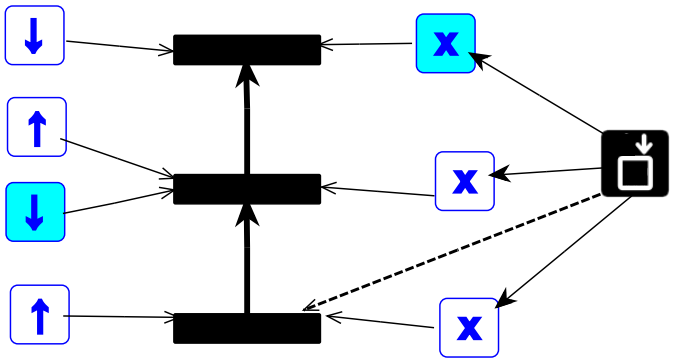

Figure 4. An Elevator model, instance of the Elevator DSML.

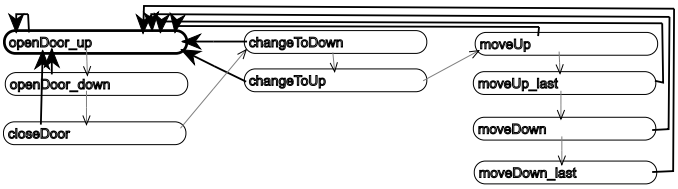

Figure 5. Rule schedule of the operational semantics of the Elevator example.

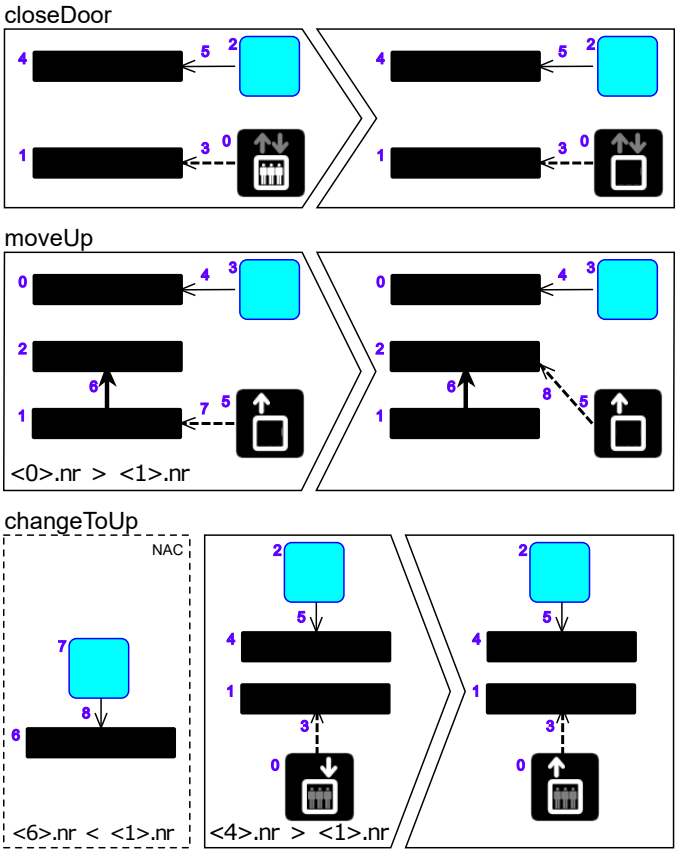

Figure 6. Three rules taken from the operational semantics of the Elevator example.

doors_open attribute value of the elevator. Further rules in the transformation denote that 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 direction requested on that floor. The case where the elevator is moving up (*i.e.,* changes its currentfloor link) is shown in the moveUp rule. Note that the LHS contains not only a pattern, but also a constraint, stating that the requested floor (with label 0) must have a higher floor number than the current floor (with label 1). Related rules (not depicted) are moveDown (the dual of moveUp), moveUp_last (where the lit button is on the next floor), and moveDown_last (the dual of moveUp_last). Pressed buttons unlight when the door opens at a requested floor and the elevator goes in that direction in the openDoor_up rule (in the case the elevator is going up) and its dual, the openDoor_down rule. The elevator only changes its direction if there are no more requests on its path. This can be seen in the changeToUp rule, which contains a negative application condition (NAC) (visualised as a dashed box) in addition to its LHS. A NAC is evaluated the same way as a LHS, but the rule will *not* be applied if a match is found. Hence, the rule will be applied if a button is pressed at a higher floor (modelled in the LHS), but

not if a button on a lower floor is lit at the same time (modelled in the NAC), *i.e.,* in its current path. Note that, if the elevator is at a lower floor and is going up, it can pass by a floor where one has requested to go down without stopping, as the elevator is going in the opposite direction, and vice versa.

The first rule that is applicable to the instance model of Figure 4 is the changeToUp rule. The LHS Elevator pattern element is bound to the only elevator in the instance model, and the currentfloor with label 3 and the Floor with label 1 are bound to the currentfloor link to the bottom Floor. Additionally, pattern elements with label 2, 4 and 5 can be bound to the middle Floor and lit DownButton with request link in between. An alternative match for pattern elements with label 2, 4 and 5 can be found as the top Floor and lit ElevatorButton with request link in between. No match for the NAC can be found since the elevator is at the bottom floor and going down, thus no buttons are in the elevator's path. Consequently, the rule matches and the RHS can be applied, in which case only pattern element 0 that is bound to the node in the instance graph changes an attribute value (*i.e.,* in the elevator, going_up is set to *true*).

If no rule is applicable, the transformation terminates. This means that the operational semantics implement the behaviour in response to the initially pressed buttons. Unlike in a realistic scenario, buttons can not be pressed during execution. One could incorporate this behaviour in $E_{[[.]]}$, by adding a rule in which a button is pressed. This however mixes input to the system, with its reactive behaviour, and can therefore be considered bad practice. A better solution would be to separately model input to the system in a dedicated input language.

Note that if the rules are scheduled differently, *i.e.,* if the move rules are scheduled before the change direction rules, the NACs in the change direction become superfluous. We use this $E_{[[.]]}$ in the running example, to be able to showcase support for NACs where necessary.

As can be seen from Figure 6 the modelling language for rules is composed from some generic language constructs for LHS, RHS and NAC, each displayed as a different shape of container. They include a constraint or action, and domain-specific language constructs that borrow syntax from the DSML. As stated before, this language can be generated using a transformation that takes the DSML metamodel and concrete syntax model as input and the metamodel and concrete syntax model of the transformation language as output, called the RAMification transformation [46]:

1) *Relaxation*: the metamodel is relaxed in order to allow for patterns which are partial models: constraints on lower multiplicities in the metamodel are removed, abstract classes are made concrete so they can be instantiated (see also the buttons used in rules of Figure 6), and global constraints are removed. For example, Buttons and FloorButtons become instantiable, and a Button does not need to be connected to a Floor in a pattern (but does in a model conform to the Elevator metamodel);
2) *Augmentation*: each class and association in the metamodel is augmented with a label attribute for pattern binding (its concrete syntax is a label in the top left corner of the icon or next to the link), and each class with subclasses is augmented with a Boolean attribute denoting whether subtypes should be matched. Other augmentations can be used to influence the matching process [46] but are not relevant in the context of this paper;

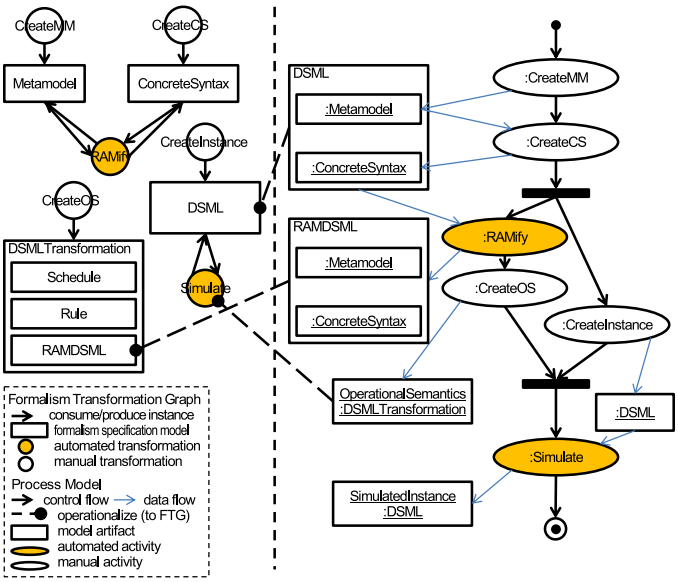

Figure 7. A FTG (left) and PM (right) of the DSM process.

3) *Modification*: for elements of the LHS, the type of each attribute is changed to the type Condition because the LHS pattern essentially models a condition or constraint that must be met. For example, a lit button in a LHS has pressed == True as value of its pressed attribute. Similarly for elements of the RHS the type is changed to Action, which allows assigning new values to attributes. For example, a lit button can be turned off in the LHS by setting pressed = False as value of its pressed attribute.

Since the result of the RAMification transformation is a language definition with metamodel and concrete syntax model, the language engineer can adapt the concrete syntax model to make rules more appealing. An example of this is shading the elevator direction or doors grey in the icon to denote that there is no constraint on the respective attribute values.

Throughout the paper, the adjective *traditional* will mean *according to the state of the art as described in Section 2.3*. A more detailed description of domain-specific modelling can be found in [49], as well as an introduction of the DSM tool *AToMPM*, which will be used in this paper.

2.4 Formalism Transformation Graph and Process Model (FTG+PM)

It has become clear that the DSM process involves different models (*e.g.,* domain-specific models, metamodels, transformation models, etc.) in various modelling formalisms (DSMLs, class diagrams, transformation languages, etc.) that each serve a specific purpose in the DSM process. DSM approaches use multiple modelling languages and deal with their consistency and interaction. This means that the MDE process for development and/or execution of the modelled system becomes more complex as well, and is highly customised.

The Formalism Transformation Graph and Process Model (FTG+PM) [48] captures the MDE process by explicitly modelling it. The FTG+PM consists of the Formalism Transformation Graph (FTG) and its complement, the Process Model (PM), and models all kinds of activities in the MDE lifecycle such as requirements development, domain-specific design, verification, simulation,

analysis, calibration, deployment, code generation, and execution. The FTG describes in an explicit and precise way formalisms, and their relationships, as transformations between models in these formalisms. The PM specifies an MDE process using these formalisms and transformations, and can be used as a basis for process enactment. Figure 7 shows a FTG+PM of the DSM process.

The FTG depicts (domain-specific) formalisms as labelled rectangles. Transformations (in the broad sense of the word) between models in those formalisms are depicted as labelled small circles. Formalisms can be connected to transformations by arrows that depict formalisms as in- and output of these transformations: executing a transformation takes instances of incoming formalisms as input, and creates instances of outgoing formalisms as output.

The PM is specified in the UML Activity Diagram 2.0 language [66]. The ovals (actions) in the Activity Diagram correspond to executions of the transformations declared in the FTG. Labelled rectangles (data objects) in the PM correspond to models that are consumed or produced by actions, and are instances of the formalisms in the FTG with the same label. Thin arrows in the PM indicate data flow, while thick arrows indicate control flow. Similar to the models, the arrows must also have corresponding arrows in the FTG, meaning that their input and output nodes must correspond. We also use Activity Diagrams control flow constructs for a PM such as joins and forks, represented by horizontal bars, and decisions, represented by diamonds. (Executions of) transformations can be automatic (shaded) or manual (white). The meaning of the FTG+PM is presented in depth in [48].

We use the FTG+PM in the context of DSM, to describe both the language engineering phase and the system modelling phase, while the FTG+PM was up to now only used for the latter [64]. Since languages are created in the language engineering phase and are later used in the system modelling phase, they need to appear on both the PM and FTG side. To this end, we extend the FTG+PM language with the operationalise relationship. Using the operationalise relationship, a metamodel and concrete syntax model can be *put into operation* as a language, meaning models that conform to the language can be created. A tool can automate this as compilation or interpretation of the metamodel and concrete syntax model, possibly by generating a (syntax-directed) language-specific editing environment. Note the box around metamodel and concrete syntax model to denote that they are operationalised together to form a language (semantics is not yet included in the box). Similarly, a transformation model can be put into operation by compiling or interpreting it, meaning it becomes executable, often to give semantics to a language. In the process model of Figure 7 all activities from the top up to the CreateOS activity are performed by the language engineer, and the remaining two activities are performed by the domain user.

More in detail, the language engineer first creates a metamodel in the CreateMM activity. Then he creates a concrete syntax model in the CreateCS activity as well as a mapping specifying the bidirectional relationship between both (note that this is not visible in Figure 7), which results in a DSML definition. Then, he/she RAMifies the DSML in the automatic RAMify transformation. The resulting RAMDSML is part of the DSMLTransformation language. This is the language of the OperationalSemantics transformation model, which is subsequently created in the CreateOS activity. This concludes the creation of a DSML. Next, the domain user models a system in the CreateInstance activity. He/she can simulate this model, by executing the operational semantics in the Simulate activity, resulting in a new DSML instance.

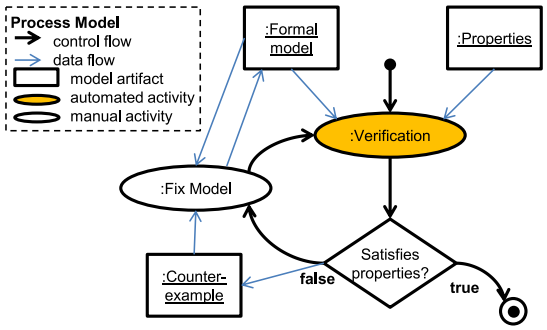

Figure 8. Verification of properties.

Note that relationships between artifacts might become visible when following links transitively. For example, in the PM, one might expect that in the PM the OperationalSemantics model should be input to the Simulate activity. The OperationalSemantics model is however not literally an input of Simulate. Instead, the Simulate transformation in the FTG is an operationalisation of the OperationalSemantics model. Thus, the Simulate activity in the PM is implicitly related to the OperationalSemantics model.

2.5 Temporal Properties

Properties allow us to encode questions we want to ask the modelled system, with an answer that is yes or no (*i.e.,* property satisfied or not). Examples are: (a) *Are there algebraic loops in my block diagram?*, or (b) *If I press a button in an elevator, will the elevator eventually reach the corresponding floor?* Useful properties are extracted from requirements, and it can be verified whether a model of a system *satisfies* these properties. The general verification process is shown in the Process Model (PM) of Figure 8, where a formal model and properties are fed into a verification engine. If no counterexample is found, the system satisfies the property. If a counterexample is found, it needs to be visualised so the user may correct the formal model, after which the verification process can be restarted.

We distinguish two kinds of properties: structural properties and temporal properties. *Structural properties* are evaluated statically on the structure of a model, *e.g.,* on the abstract syntax graph. Structural properties can be expressed in *e.g.,* a constraint language like the Object Constraint Language (OCL) [65]. Example (a) can be expressed in OCL. *Temporal properties* are defined in terms of time, *i.e.,* on paths of program execution, or execution *traces*.

One such way to express temporal properties is using Linear Temporal Logic (LTL) [68], as done in the *ProMoBox* approach. LTL is built up from propositions (*e.g.,* ψ and ϕ), on which the following temporal operators can be applied (yielding expressions that are also propositions): $\Box\psi$ (globally ψ), $\Diamond\psi$ (finally ψ), $\bigcirc\psi$ (next ψ), and $\psi\,\mathcal{U}\,\phi$ (ψ until ϕ). The logic operators $\vee$, $\wedge$, $\neg$, $\rightarrow$ and $\leftrightarrow$ can be used in formulas, and operator precedence can be enforced by brackets. Throughout this paper we will use $\psi\,\mathcal{W}\,\phi$ ("weak until" – where ϕ is not required to occur) which can be defined as $(\psi\mathcal{U}\phi) \vee \Box\psi$. Knowing the syntax of LTL is not mandatory for understanding this paper, but the interested reader can find a complete summary of the syntax and semantics of LTL in [23].

Another temporal logic is Computation Tree Logic (CTL) [24], that reasons about execution tree branches rather than single traces. Its syntax is similar to LTL, including the use of propositions, logic operators and brackets. It differs however in the

temporal operators used: $AG\psi$, $EG\psi$, $AF\psi$, $EF\psi$, $AX\psi$, $EX\psi$, $A[\psi\ \mathcal{U}\ \phi]$, $E[\psi\ \mathcal{U}\ \phi]$ where the A in the operator means "for all paths", and E means "there exists at least one path". The G (globally), F (finally), X (next), and U (until) correspond to the LTL operators. Similarly to LTL, a "weak until" operator is defined as $A[\psi\ \mathcal{W}\ \phi] = \neg E[\neg\phi U(\neg\psi \wedge \neg\phi)]$ and $E[\psi\ \mathcal{U}\ \phi] = \neg A[\neg\phi W(\neg\psi \wedge \neg\phi)]$

The logic operators $\vee$, $\wedge$, $\neg$ and $\rightarrow$ can be used in LTL and CTL formulas.

Using LTL or CTL, temporal properties such as those in example (b) can be expressed. It is an example of a liveness property (something must eventually happen). The other class of temporal properties is the class of safety properties (something should not happen). There is a slight difference in verifying liveness and safety properties, as in order to find a counterexample in case of a liveness property, it must be shown that the proposition that must eventually happen is not present in traces with possibly infinite length.

2.6 Model Checking

Verifying whether a model satisfies a temporal property can be done in several ways. Some techniques for verification include manually inspecting the model, using testing techniques, symbolic execution, model checking, etc. In the *ProMoBox* approach we focus on model checking. Model checking is an automated approach, where it is determined whether a model satisfies a property by exhaustively searching for a counterexample, which takes the form of an execution trace. If no such counterexample is found, it is certain that the model satisfies the property.

Advantages of model checking include that it is a fully automated method, and that the outcome is reliable (unlike for *i.e.,* testing techniques). It has some major drawbacks however: the search space has to be finite, and large search spaces are infeasible to analyse. As the verification time as well as the required memory easily suffers from combinatorial explosion, the approach only allows a limited number of variables in the modelled system. This means that although model checking is in itself fully automated, often a manual abstraction step is required to make the technique applicable to a model [4]. Throughout this paper, two tools for model checking will be used.

Spin [35] is a tool that allows users to write system descriptions in a textual programming language called *Promela*, encode properties in LTL, and verify whether the system description satisfies these properties. The default verification algorithm visits every possible state of the system description by building the state space explicitly. This is a directed graph with state vectors as nodes and statement executions as transitions. By traversing this complete state space a conclusive answer to the satisfaction of an LTL formula can be produced. *Spin*'s main application is the verification of concurrent systems. Although very powerful, writing system descriptions in *Promela* and LTL formulas requires a background in programming and logic, skills that domain users do not necessarily have. In our approach, we intend to leverage the power of *Spin*, without having to expose its technicalities (*i.e.,* its languages and interface) to domain users.

Groove [71] is a tool for specifying rule-based transformations on systems described as typed graphs. A type graph can be defined (much like a metamodel), and graphs can conform to the type graph. Similar to our view on model transformation, transformations consist of rules that, if matched, manipulate the graph. Rules are scheduled using an imperative language. *Groove* excels in its feature of verification of temporal logic properties written in LTL and CTL, which is typically not available in a domain-specific modelling tool. *Groove* is used in this paper to evaluate the impact of replacing the verification backbone (*i.e., Spin*).

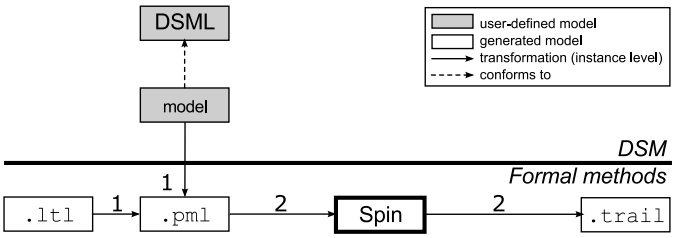

Figure 9. Verification in Domain-Specific Modelling.

3 Overview of the *ProMoBox* Approach

The state of the art in verification for DSM is illustrated in Figure 9. The diagram is divided into the domain layer where models are domain-specific, and the lower layer where artifacts are represented as logic formulas or code whose creation requires a significant theoretical background. A DSML has been developed, and systems can be modelled that conform to the DSML. Transformations can be defined for simulating or translating this model, for generating documentation or platform-specific code. Note that these activities are not shown in Figure 9, as it exclusively focuses on the verification activity. Step 1 of Figure 9 consists of creating the formal model for verification. A transformation automatically compiles any model that conforms to the DSML to a formal representation (in this case *Promela*, a textual `pml` file), bridging the DSM layer and formal methods layer (step 1).

A first problems arises, as this transformation is created by the language engineer, who is not necessarily and expert in verification methods. The second problem is related to the domain user. While the DSML and model are usable by non-technical domain users, the necessary LTL-formulas (`ltl` files in the lower "formal methods" layer) for verification are not easily usable by domain users. In step 2 the formal model is executed in *Spin*, performing the actual model checking. In case of a counterexample, a `trail`-file representing the output trace is generated by *Spin*. This presents an additional problem: understanding the trail also requires a similar theoretical background. In conclusion, the problem with the current state of the art in verification is that the language engineer and domain user still have to work in the formal methods layer.

The goal of *ProMoBox* is to fully lift the user experience (both for language engineer and domain user) which was hitherto limited to the construction of system design models (and did not include requirements or property models), to the DSM layer, with a minimal increase in user effort. This section presents the use cases of *ProMoBox*, and gives an overview of the *ProMoBox* framework.

3.1 Use cases of *ProMoBox*

As *ProMoBox* builds on the principles of DSM, it consists of a language engineering phase (the upper part of Figure 1) and a system modelling phase (the lower part of Figure 1). Consequently, the primary use case of *ProMoBox* consists of these two phases, explained in detail as follows: (1) *generate a verification language and support for a DSML*, to allow multiple different applications of

(2) *modelling a property and verifying whether the system satisfies this property*. The former is performed by the language engineer and domain experts, the latter is performed by the domain user. Both are lifted to the domain level, thus hiding underlying formal methods and tools. This use case that includes these two phases will be the main thread throughout the paper, and will be referred to as *UC1*.

Other, less common use cases may occur, that nevertheless need support:

- *UC2*: verification support needs to be created for an already existing DSML with *ProMoBox*. This is different from *UC1*, because in some cases, the DSML needs to be adapted to ensure that it is compliant with *ProMoBox*. This use case will be addressed in Section 4.4;
- *UC3*: the DSML needs to be changed after the *ProMoBox* is applied. This may include the DSML's abstract syntax, concrete syntax or semantics. This use case will be addressed in Section 4.5;
- *UC4*: the *ProMoBox* framework itself needs to be adapted to support new or changing technology. This use case will be addressed throughout Section 4 and Section 5.

Throughout the paper, we will discuss how these use cases are supported by *ProMoBox*, and we attribute certain design choices to these use cases. These use cases are discussed in detail in the evaluation of Section 7.5.

3.2 Outline of the *ProMoBox* Approach

The *ProMoBox* framework presented in this paper builds on our earlier work [20], [57], [60]. *ProMoBox* stands for "Properties and (design) Models developed (Boxed) in concert". The language engineering support of *ProMoBox* consists of the following three parts.

The first part is the definition of five sublanguages. According to DSM principles everything should be modelled using the most appropriate formalisms. Therefore, *ProMoBox* replaces the traditional DSML with five sublanguages (each DSMLs) for modelling all artifacts that are needed to specify and verify properties [57]. The five sublanguages are the following:

- A *design language* for design modelling as supported by traditional DSMLs. With this language, the static structure (*i.e.,* language concepts that do not change at run-time) of the system is modelled. In case of Elevator, this includes all Elevator concepts, without the currentfloor association, nor the doors_open, going_up and pressed attributes;
- A *run-time language* for run-time state representation. The design language always includes all elements of the design language, plus dynamic state information that can change at run-time. Run-time instances are always associated with a design instance with the same static structure. One design instance possibly has multiple run-time instances corresponding with it, representing all possible states of the model. Note that in traditional DSM, the DSML often includes run-time concepts, meaning that no distinction is made between static structure and dynamic state. The running example was also presented including dynamic state information in Section 2.3. In fact, the metamodel shown in Figure 2 is the same as Elevator's run-time language;
- An *input language* to model event-based input (to model the environment in which the system operates). This language represents a stream of input events. In case of Elevator, this language only consists of a sequence of button presses;
- A *trace language* for state-based output representation (to model an execution trace of the system or verification counterexample). An execution trace is a sequence of run-time states connected with transitions that represent execution steps (*i.e.,* operational semantics' rule executions). The trace language can be used to represent execution traces of a simulation. A trace model is usually generated by a simulator or as a counterexample by a verification tool. It can be generated manually as well for *e.g.,* modelling an oracle for a test case. In the Elevator case, this is a sequence (in terms of operational semantics steps) of run-time states with references to *e.g.,* move_up, open_doors, etc. (see Figure 5).
- A *property language* for property specification (to model temporal or structural properties). The properties language allows the user to define temporal properties, which are properties on the behaviour of systems. Properties are represented by temporal and structural operators over propositions. These propositions are patterns that can be matched or not matched (resulting in *true* or *false*) on a run-time state or static structure of a system. Since properties reason about state and structure, all language constructs of the design language and run-time language are included in the property language. In the Elevator case, a property can express that whenever a button is pressed, the elevator should eventually reach the corresponding floor.

The second part is the generation of these five sublanguages. As the traditional DSML is replaced by five languages (*i.e.,* DSMLs), it would be time consuming to keep these intimately related sublanguages presented above consistent. Therefore, a fully automated method generates these sublanguages from a single DSML specification, keeping the five sublanguages consistent by construction. If necessary (*i.e.,* in case of *UC2*), simplifications are made in the DSML's metamodel, to address the scalability issues of model checking. We extend metamodelling and model transformation languages with annotations, to add necessary information for every language construct and to introduce a conceptual simulation step. This additional information enables the fully automatic generation of the five sublanguages and necessary transformations between the sublanguages, thus minimising the effort of the language engineer. This way, not only building the five sublanguages requires less effort, but also maintaining consistency in case of DSML changes. This is because only the annotated DSML needs to be edited, after which the sublanguages can be regenerated. This addresses the use case *UC3*, which deals with evolution of the DSML. Each of the sublanguages is built from a DSML-independent template, and domain-specific language concepts. Although the templates are predefined, the resulting modularity allows extensive modification of the templates, as desired by *UC4*. By using templates and a generative approach, the *ProMoBox* framework becomes configurable for various DSMLs.

The third part is the mapping to and from a verification backbone. A verification backbone based on the *Spin* model checker [35] is directly plugable to DSM environments. Properties in *ProMoBox* are translated to LTL and a *Promela* model is generated that includes a translation of the domain-specific system

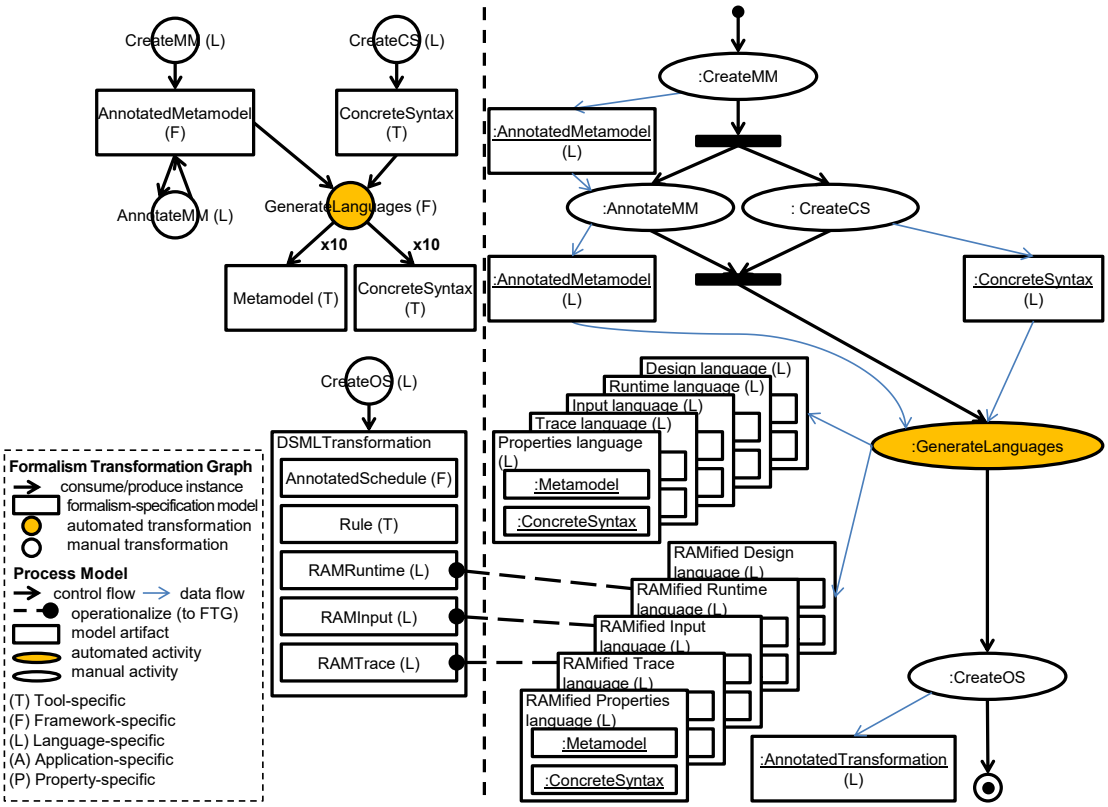

Figure 10. FTG+PM of language engineering with *ProMoBox*.

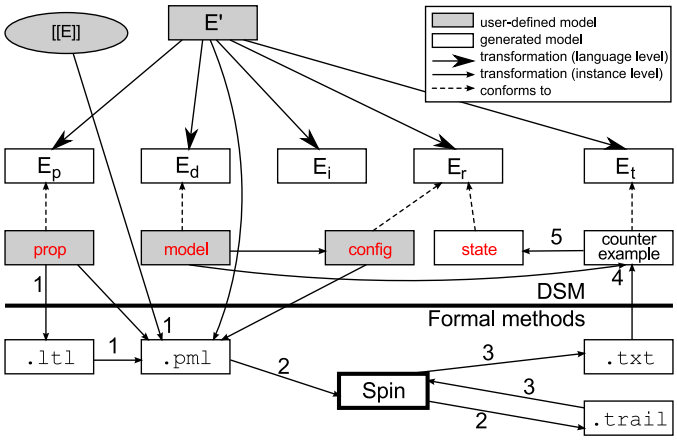

Figure 11. Property verification with *ProMoBox* and *Spin*.

specification, its initial state, the environment, and the rule-based operational semantics of the system. The verification results (in case of a counterexample) are translated back to the domain level. We argue that verification tools other than *Spin* can be used in the *ProMoBox* framework, as required by *UC4*. This use case is evaluated in Section 7.

The Elevator *ProMoBox* is illustrated in Figure 11. When using the *ProMoBox* approach, only the grey models in Figure 11 need to be modelled by hand, the white models are generated. The five sublanguages E_d, E_r, E_i, E_t and E_p are generated from an annotated metamodel E'. A property model prop, a design model model and a run-time model config can be modelled by the domain user in these sublanguages. Note that only a minimal number of models needs to be created by hand thanks to the generative character of *ProMoBox*. As shown in steps 1-5, a property can be verified automatically by transforming these models to a *Promela* model and LTL formula, execute the *Spin* tool and in case of a counterexample, transform it to an instance of the trace language. These steps will be explained in detail further in the paper. It is important to note that all manually created models are at the DSM layer, meaning that both language engineers and domain users do not need to take a look "under the hood". This is exactly the intent of DSM.

The next section explains language engineering with *ProMoBox* in detail, in the standard case of *UC1*. As the *ProMoBox* approach as presented in Figure 11 involves a significant number of modelling artifacts and transformation steps, the FTG+PM of Figure 10 serves as a process-oriented view on *ProMoBox* and will serve as a guide throughout this section. To put all artifacts in the FTG+PM (*i.e.,* languages, models and transformations) in perspective, they are marked with a specification level:

- (T) tool-specific: an artifact marked (T) (*e.g.,* a language for metamodelling) is defined by the DSM tool, and implemented by a DSM tool builder. As *ProMoBox* builds on DSM, it is a prerequisite for *ProMoBox*;
- (F) framework-specific: an artifact marked (F) (*e.g.,* a transformation generating the five sublanguages from an annotated DSML) is defined by the *ProMoBox* approach. One who aims to implement the *ProMoBox* framework will have to define this artifact, but once defined it can be used for any DSML *ProMoBox*;
- (L) language-specific: an artifact marked (L) (*e.g.,* annotating a metamodel) is defined by a language engineer when defining a DSML using *ProMoBox*;

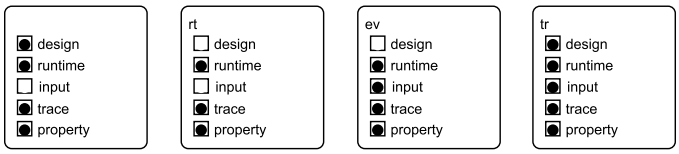

Figure 12. The annotations model of the *ProMoBox* framework.

- (A) application-specific: an artifact marked (A) (*e.g.,* the model shown in Figure 4) is defined by a domain user who instantiates the DSML to model a system. Since this section discusses the language engineering phase, application-specific artifacts will only appear in Section 6 discussing the system modelling phase;
- (P) property-specific: an artifact marked (P) (*e.g.,* a model of a property or its counterexample) is defined by a domain user who models a property. Similar to application-specific artifacts, property-specific artifacts will only appear in Section 6.

The Process Model on the right side of the FTG+PM shown in Figure 10 starts with two manual traditional DSM tasks of creating a metamodel in the CreateMM activity as well as specifying a concrete syntax model (CreateCS). This results in an :AnnotatedMetamodel instance (rather than a Metamodel instance, yet still without annotations) and a :ConcreteSyntax instance. Then, the specific activities of the *ProMoBox*, further explained in the remainder of this section, are outlined. In Section 4.1, we define how metamodels can be annotated (the AnnotateMM activity, resulting in E' in Figure 11) to define the relationship with the five sublanguages. In Section 4.2, we discuss the sublanguages in detail and how they are generated from the annotated metamodel (the AnnotateMM activity). In Section 4.3, we present how to use these languages to create the annotated operational semantics which fine-tunes the behavioural semantics of the DSML *ProMoBox* (the CreateOS activity). In Section 4.4 we divert from the main FTG+PM track to give insight in how to migrate DSMLs, as created following the traditional process shown in Figure 7, to *ProMoBox*.

4 Language Engineering with *ProMoBox*

This section presents the architecture of the *ProMoBox* framework. It discusses the different languages and models of the framework, and it is explained how the framework can be applied to a DSML. Section 4.1 to 4.3 explains language engineering with *ProMoBox* in detail, in the standard case of *UC1*. In Section 4.4, *UC2* is addressed, and 4.5 discusses *UC3*. We present a customisable approach, to cater *UC4*. The Elevator case is used as a running example.

4.1 Defining a *ProMoBox*

In parallel with the :CreateCS activity, metamodel elements need to be annotated manually in the :AnnotateMM activity in Figure 10.

The metamodel elements (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 and output (*e.g.,* a button press);

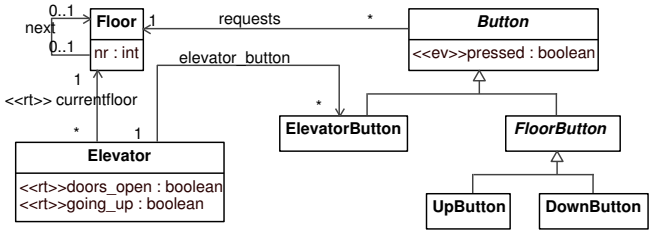

Figure 13. The annoted metamodel of the Elevator example.

- *tr*: a trigger, annotates a static concept that also serves as input and output (*e.g.,* a key stroke – not available in the Elevator case).

These annotations are explicitly modelled as *annotation types* in the annotations model of Figure 14 which is shown in Figure 12. The annotations model is an instance of the AnnotationTypes DSML. Technically, each annotation type represents *sublanguage membership*, visualised by checked boxes in Figure 12. An annotation on an metamodel element denotes in which sublanguage the element becomes available (*i.e.,* included in the sublanguage's metamodel). For example, if an association is annotated with *rt*, it will be included in the run-time metamodel, the trace metamodel and the property metamodel, but not in the design metamodel and the input metamodel. We say for such an element, that it is (for example) *part of* the run-time metamodel. In case of annotations on an attribute that require the inclusion of the attribute in a sublanguage, their containing class is included as well, even if it should not be included by itself. This way, attributes will always have a containing class. Associations are treated similarly; their source and target classes will be included if the association is included. Furthermore, subclasses inherit annotations from their superclasses.

The annotations model of Figure 12 can be extended with more annotation types, and will be automatically incorporated by the *ProMoBox* framework. A possible example could be an annotation that denotes inclusion in the output metamodel only, as it is a derived value (e.g., the total amount of Buttons pressed) of the systems state. An annotation type must adhere to the following rules:

- an annotation type should result in at least one inclusion in a sublanguage's metamodel;
- if an annotation type denotes inclusion in the design metamodel, then it must denote inclusion in the run-time metamodel as well;
- if an annotation type denotes inclusion in the input metamodel, then it must denote inclusion in the run-time metamodel as well;
- if an annotation type denotes inclusion in the run-time metamodel, then it must denote inclusion in the property metamodel as well.

The first annotation type of Figure 12 defines the case where no annotation is applied to a metamodel element. In that case, the concept is not part of the input language. These rules are incorporated in the static semantics of the AnnotationTypes DSML.

If multiple annotations are applied to an element, the union of all sublanguage memberships is taken. This union has to adhere to the above rules.

As shown in Figure 13, in the case of Elevator and its metamodel of Figure 2, all language concepts are static, except for the currentfloor association, and the doors_open, going_up and pressed attributes. The values of currentfloor, doors_open and going_up can change at run-time when the operational semantics transformation Figure 5 is applied. Therefore, the *rt* annotation is used, meaning that they will be part of the run-time, trace and property metamodel. The pressed attribute is annotated with *ev*, and can thus also serve as input. In fact, it will be the only metamodel element that appears in the input language.

As shown in Figure 10, from an annotated metamodel and a concrete syntax model, the fully automatic GenerateLanguages transformation generates a metamodel and a concrete syntax model for each of the five sublanguages, and for the RAMified sublanguages (so that the sublanguages can be used in transformations), resulting in a total of 10 languages.

4.2 Generating the *ProMoBox* Languages

As shown in Figure 11, the five *ProMoBox* sublanguages are generated from an annotated metamodel and a concrete syntax model using a template-based approach. The fully automatic GenerateLanguages transformation of Figure 10 of one language and its RAMified counterpart is depicted in Figure 14.

4.2.1 Design, Run-time, Input and Trace Languages

The design, run-time, input and trace languages are generated in a similar fashion. As for the abstract syntax, the metamodel is first filtered producing an ordinary metamodel in the FilterMM transformation. In this transformation, all language elements that are according to the annotations model not part of the language to be generated are removed, taking into account that annotations on attributes or associations are inherited by the related classes. From the elements that remain, remaining annotations are removed, so that the result is an ordinary metamodel.

Then, depending on the language to be generated, a predetermined metamodel template is added to the metamodel in the MergeMM transformation. All templates have an Element class, with an attribute id, to which an inheritance relationship is created from all DSML-classes of the metamodel. This id will be used to link elements between different models. The result is the metamodel of the sublanguage.

Figure 15 to 18 show the generated metamodels of the Elevator DSML. The template elements are shaded. The template of the design language and the trace language consists of only one abstract *element*-class. The remainder of the metamodel are the DSML-specific elements, if they were annotated to be part of the language. This way, the dynamic elements currentfloor, doors_open, going_up and pressed do not appear in the design language E_d, but do appear in the run-time language E_r. In the input language, the template includes 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 (the empty Event indicates that there is no input at that time). 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 (see also Section 7.5.1). This would be a manifestation of *UC4*. The template of the trace language E_t 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 (rule_executions, rule applications in the operational semantics $E_{[[.]]}$, and/or an input events). Instances of E_t are often generated. The output of $E_{[[.]]}$, or the counterexample in verifications are instances of E_t.

The design instance of the elevator system with three floors is shown in Figure 19, from which dynamic concepts are excluded. Figure 4 is a valid instance of E_r as it includes dynamic concepts, and can take the role as run-time instance. Instances of the input language are out of scope of this paper, because the model checking approach in this paper implies that all possible input should be taken into account. The interested reader can find an example of an instance of an input language in [20]. Due to the possibly large number of elements in such an execution trace, an instance of E_t is stored in textual form, and can be interpreted or "played out" by showing step-by-step an instance of the run-time language E_r. This is shown further in this paper, in Figure 28. The interested reader can find an explicit instance of a trace language in [57].

The concrete syntax model of each of these languages is generated in a similar way. As depicted in Figure 14, the original concrete syntax model is filtered in the :FilterCS transformation. All icons and links of classes and associations that are not part of the to be generated metamodel are removed. Additionally, all concrete syntax elements such as text elements that contain references to attributes that are not part of the to be generated concrete syntax model are removed as well. Then, the template is added in the :MergeCS transformation, adding icons and links of respective template classes and associations. This results in a concrete syntax model, with a complete mapping to the abstract syntax model.

The result of the generation process for one sublanguage is an ordinary metamodel and concrete syntax model, thus fully compliant with the DSM tool.

4.2.2 Property Language

The property language E_p deserves special attention as it is the pivotal language of the *ProMoBox*, and its metamodel is generated in a slightly different way to the four other sublanguages, as shown in Figure 14 due to the decision node.

After filtering the metamodel according to annotations, an additional transformation (RAMification) is executed that produces a pattern language, suited for transformation languages [46], [82] as explained in Section 2.3. This language can however also be used to express structural patterns for properties as the same principle of pattern matching is re-used in this context. At the bottom of Figure 20 the RAMified DSML elements are shown. All attribute types are now conditions, abstract classes are now concrete classes, and lower bounds of multiplicities are relaxed to 0.

Next, the template for property languages is added to the metamodel, resulting in the metamodel of Figure 20. PropertyElement, the superclass of all DSML classes, includes a general condition, a label for inter-pattern matching similarly to [46], [82] and an id for inter-model traceability similar to the four other sublanguages. The template consists of a property Specification, which can be composed of the following four language parts:

- *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 a given structural pattern. This structural pattern is evaluated on the design model, and can thus not refer to run-time concepts, because the match set must be static. Consequently, the property must be satisfied for all, or

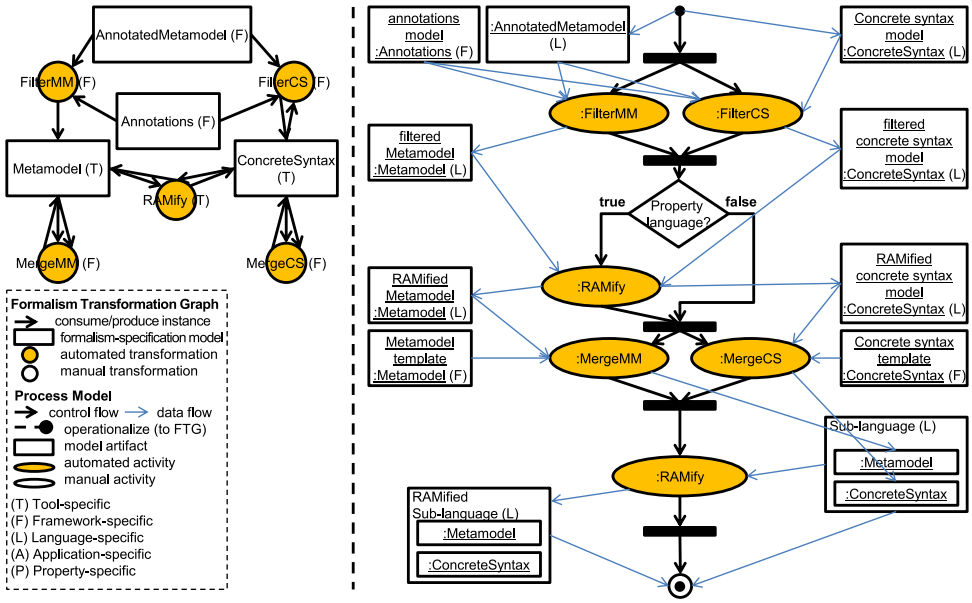

Figure 14. FTG (left) and PM (right) of generating one of the five sublanguages.

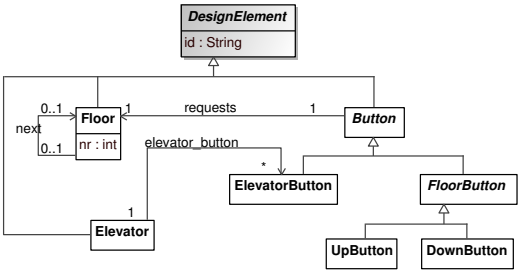

Figure 15. Metamodel of the Elevator design language E_d.

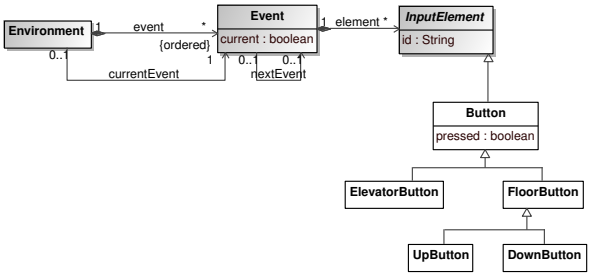

Figure 17. Metamodel of the Elevator input language E_i.

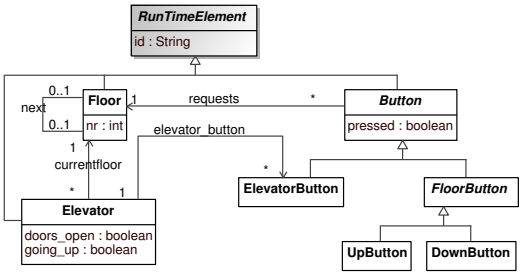

Figure 16. Metamodel of the Elevator run-time language E_r.

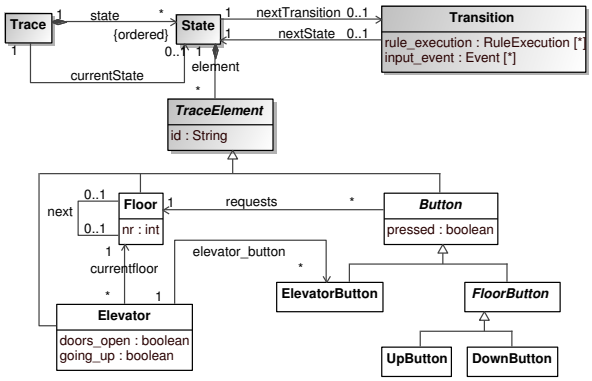

Figure 18. Metamodel of the Elevator trace language E_t.

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.

- *The LTL operators*, so that *ProMoBox* can match the expressive power of LTL. One exception is LTL's next operator, which is not included in *ProMoBox* as its semantics remain unclear in this context (see Section 8.3 for a discussion). The supported LTL operators include all other operators as defined in Section 2.5: temporal operators $\Box\psi$, $\Diamond\psi$, and $\psi\ \mathcal{U}\ \phi$, and logic operators $\vee$, $\wedge$, $\neg$, $\rightarrow$ and $\leftrightarrow$. The operands ψ and ϕ are structural properties of the system, modelled as structural patterns (explained below). Rather than using these LTL operators, the user is encouraged to use temporal patterns, as introduced below.
- *The temporal pattern*, based on the specification patterns by Dwyer *et al.* [21]. The available temporal patterns are listed in Table 1. The temporal pattern allows the user to specify a pattern over a given scope, *e.g.,* "the

property	*meaning*	*type*	*scope*	*LTL formula*
Absence	ψ is false	Safety	Globally	$\Box(\neg\psi)$
			Before ρ	$\Diamond\rho \rightarrow (\neg\psi\, \mathcal{U}\, \rho)$
			After σ	$\Box(\sigma \rightarrow \Box(\neg\psi))$
			Between σ and ρ	$\Box((\sigma \wedge \neg\rho \wedge \Diamond\rho) \rightarrow (\neg\psi\, \mathcal{U}\, \rho))$
			After σ until ρ	$\Box(\sigma \wedge \neg\rho \rightarrow (\neg\psi\, \mathcal{W}\, \rho))$
Existence	ψ becomes true	Liveness	Globally	$\Diamond(\psi)$
			Before ρ	$\neg\rho\, \mathcal{W}\, (\psi \wedge \neg\rho)$
			After σ	$\Box(\neg\sigma \vee \Diamond(\sigma \wedge \Diamond\psi))$
			Between σ and ρ	$\Box(\sigma \wedge \neg\rho \rightarrow (\neg\rho\, \mathcal{W}\, (\psi \wedge \neg\rho)))$
			After σ until ρ	$\Box(\sigma \wedge \neg\rho \rightarrow (\neg\rho\, \mathcal{U}\, (\psi \wedge \neg\rho)))$
Bounded existence (2)	ψ becomes true at most 2 times	Safety	Globally	$(\neg\psi\, \mathcal{W}\, (\psi\, \mathcal{W}\, (\neg\psi\, \mathcal{W}\, (\psi\, \mathcal{W}\, \Box\neg\psi))))$
			Before ρ	$\Diamond\rho \rightarrow ((\neg\psi\wedge\neg\rho)\, \mathcal{U}\, (\rho\vee((\psi\wedge\neg\rho)\, \mathcal{U}\, (\rho\vee((\neg\psi\wedge\neg\rho)\, \mathcal{U}\, (\rho\vee((\psi \wedge \neg\rho)\, \mathcal{U}\, (\rho \vee (\neg\psi\, \mathcal{U}\, \rho))))))))$
			After σ	$\Diamond\sigma \rightarrow (\neg\sigma\, \mathcal{U}\, (\sigma\wedge(\neg\psi\, \mathcal{W}\, (\psi\, \mathcal{W}\, (\neg\psi\, \mathcal{W}\, (\psi\, \mathcal{W}\, \Box\neg\psi))))))$
			Between σ and ρ	$\Box((\sigma\wedge\Diamond\rho) \rightarrow ((\neg\psi\wedge\neg\rho)\, \mathcal{U}\, (\rho\vee((\psi\wedge\neg\rho)\, \mathcal{U}\, (\rho\vee((\neg\psi\wedge\neg\rho)\, \mathcal{U}\, (\rho\vee((\psi \wedge \neg\rho)\, \mathcal{U}\, (\rho \vee (\neg\psi\, \mathcal{U}\, \rho))))))))))$
			After σ until ρ	$\Box(\sigma \rightarrow ((\neg\psi\wedge\neg\rho)\, \mathcal{U}\, (\rho\vee((\psi\wedge\neg\rho)\mathcal{U}(\rho\vee((\neg\psi\wedge\neg\rho)\mathcal{U}\, (\rho\vee((\psi \wedge \neg\rho)\, \mathcal{U}\, (\rho \vee (\neg\psi\, \mathcal{W}\, \rho) \vee \Box\psi)))))))))$
Universality	ψ is true	Safety	Globally	$\Box(\psi)$
			Before ρ	$\Diamond\rho \rightarrow (\psi\, \mathcal{U}\, \rho)$
			After σ	$\Box(\sigma \rightarrow \Box(\psi))$
			Between σ and ρ	$\Box((\sigma \wedge \neg\rho \wedge \Diamond\rho) \rightarrow (\psi\, \mathcal{U}\, \rho))$
			After σ until ρ	$\Box(\sigma \wedge \neg\rho \rightarrow (\psi\, \mathcal{W}\, \rho))$
Precedence	ϕ precedes ψ	Safety	Globally	$\neg\psi\, \mathcal{W}\, \phi$
			Before ρ	$\Diamond\rho \rightarrow (\neg\psi\, \mathcal{U}\, (\phi \vee \rho))$
			After σ	$\Box\neg\sigma \vee \Diamond(\sigma \wedge (\neg\psi\, \mathcal{W}\, \phi))$
			Between σ and ρ	$\Box((\sigma \wedge \neg\rho \wedge \Diamond\rho) \rightarrow (\neg\psi\, \mathcal{U}\, (\phi \vee \rho)))$
			After σ until ρ	$\Box(\sigma \wedge \neg\rho \rightarrow (\neg\psi\, \mathcal{W}\, (\phi \vee \rho)))$
Response	ϕ responds to ψ	Liveness	Globally	$\Box(\psi \rightarrow \Diamond\phi)$
			Before ρ	$\Diamond\rho \rightarrow (\psi \rightarrow (\neg\rho\, \mathcal{U}\, (\phi \wedge \neg\rho)))\, \mathcal{U}\, \rho$
			After σ	$\Box(\sigma \rightarrow \Box(\psi \rightarrow \Diamond\phi))$
			Between σ and ρ	$\Box((\sigma \wedge \neg\rho \wedge \Diamond\rho) \rightarrow (\psi \rightarrow (\neg\rho\, \mathcal{U}\, (\phi \wedge \neg\rho)))\, \mathcal{U}\, \rho)$
			After σ until ρ	$\Box(\sigma \wedge \neg\rho \rightarrow ((\psi \rightarrow (\neg\rho\, \mathcal{U}\, (\phi \wedge \neg\rho)))\, \mathcal{W}\, \rho))$

Table 1
List of temporal properties (adapted from [21]).

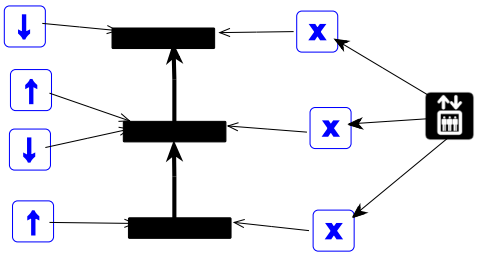

Figure 19. An instance of E_d modelling the design of Figure 4.

absence of an occurrence of ψ, after the occurrence of σ", or "an occurrence of ψ is responded to by an occurrence of ϕ, between occurrences of σ and ρ" (with proposition variables ψ, ϕ, σ and ρ). Over 90% of the properties that were investigated by Dwyer *et al.* can be expressed in this simple framework [22]. Given its use in theory and practice, the specification patterns have become a strong foundation to build on, making it an excellent baseline for our work. In our work, six patterns are supported, to express the absence, existence, bounded existence, universality response or precedence for given proposition(s). Note that our approach allows more patterns to be specified, as explained in Section 5.5. Additionally, a scope can be defined specifying whether the pattern is applicable globally, or after, before, in between or after until the occurrence of given proposition(s). The patterns and scopes are shown as template classes in Figure 20, which is meaningful as Dwyer *et al.* identified numerous variants of their patterns, which are used depending on the domain [22]. 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. Table 1 shows each pattern and its type: *safety* or *liveness* property. A safety property is a property where something *should not* happen, and a liveness property is a property where something *should* happen eventually. Technically, the property is a safety property when its translation to Büchi automata has no non-accepting states with a path to an accepting state [1]. Safety and liveness properties are verified in a different way, as liveness properties could be violated when the system can enter a loop that excludes the proposition that should happen. The LTL formula is shown for every temporal pattern. Some of them can be confusing to come up with manually, *e.g.,* existence of ψ before ρ, in which the case where ρ never occurs should be incorporated in the formula. Other LTL formulas are long and contain a high parenthesis depth, such as the bounded existence patterns. Note that these formulas do not yet include quantification and structural patterns, so they can easily end up being more complex.

- *The structural pattern*, based on PaMoMo [30], [31], for both design (when used in a quantification pattern) as well as run-time (when used in a temporal pattern) models. Only

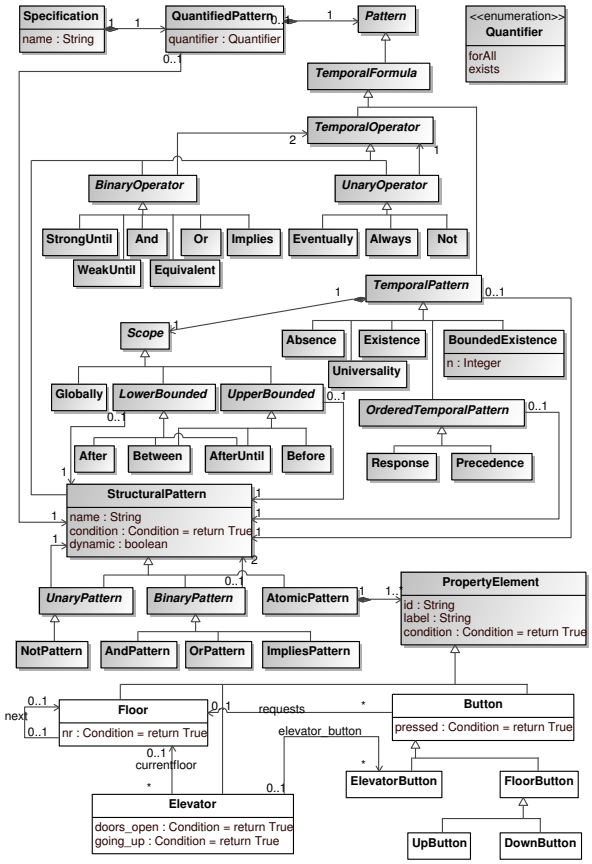

Figure 20. Metamodel of the Elevator Property Language E_p.

a small part of PaMoMo's expressiveness is included in the property language, but this suffices for defining most properties. The basic pattern is an AtomicPattern, which contains the DSML elements. Just like PropertiesElement, a StructuralPattern, can hold a condition, which returns *true* by default. Note that two types of structural patterns exist: static patterns that define patterns on a design model and dynamic patterns that define patterns on a run-time model. The former are used in quantifier patterns, the latter are used in temporal pattern propositions. The distinction is made using the dynamic attribute. This could have been modelled explicitly by adding StaticStructuralPattern and DynamicStructuralPattern as subclasses of StructuralPattern, each referring to their respective DSML elements. Since this would require duplicating the structural pattern elements and DSML elements, a constraint in metamodel of E_p enforces the use of correct patterns instead.

Apart from being RAMified, the concrete syntax model is generated in a similar way as the other sublanguages. The concrete syntax template for property languages consists of a combination of natural language and containers denoting visual, DSML patterns. In light of *UC4*, the language engineer could choose to change the concrete syntax according to his or her preference to *e.g.,* icons denoting the temporal patterns, by changing the concrete syntax model. This does not affect the abstract syntax model however, as explained in Section 2.1.

Figure 21 shows a property for the Elevator DSML. The visual, domain-specific syntax is very similar to the concrete syntax of the

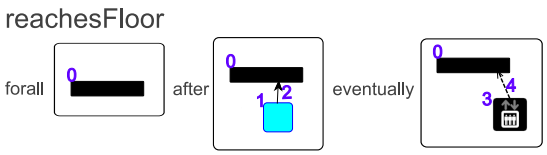

Figure 21. The reachesFloor instance of E_p: for any floor, after a button is pressed, the elevator will eventually open its doors on that floor.

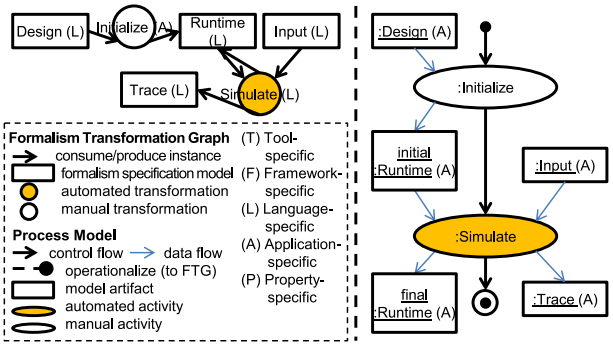

Figure 22. FTG+PM of operational semantics usage with *ProMoBox*.

rules in Figure 5. It is combined with a textual syntax to define temporal operators. A quantification pattern is used to denote that this property must be valid for all matches of the pattern, *i.e.,* for all floors. Its structural pattern (visualised by a container) binds a Floor of the design model of Figure 19 to the pattern element with label 0. On the right an existence pattern (the Elevator will reach the previously bound Floor) with lower temporal boundary (after a Button is pressed that requests the previously bound Floor) is shown.

4.3 Operational Semantics Annotation

ProMoBox makes input and output explicit in its sublanguages. Whereas the traditional Simulate transformation of Figure 7 has a DSML instance as input (*i.e.,* the model in its initial state), and produces a DSML instance (*i.e.,* the model in its final state), the Simulate transformation in *ProMoBox* takes a run-time model (initial state, generated or manually created from the design model) and an input model (input events) as input, and generates a run-time model (the final state) and a trace model (simulation trace) as shown in Figure 22. The transformation rules are restricted in that they can only change run-time elements of the model, and can only use design elements for matching. This restriction guides the language engineer in creating a correct model of the operational semantics. Moreover, since information about input and output needs to be incorporated in the semantics of the language, the operational semantics of Figure 5 need to be annotated. More specifically, the operational semantics have to define *when* (*i.e.,* at what point in the rule schedule) a new input event is applied to the system, and a new state is added to the execution output trace. Different semantics are possible.

Perhaps the most straightforward semantics would be to read an event from the input environment and add the state to the output trace, following each successful rule execution of the operational semantics. This however implies a direct correlation between the rules of the operational semantics and a conceptual "time step" (*i.e.,* an abstraction of a significant period of time) of the semantics, while this is not necessarily so. In the Elevator example, it makes

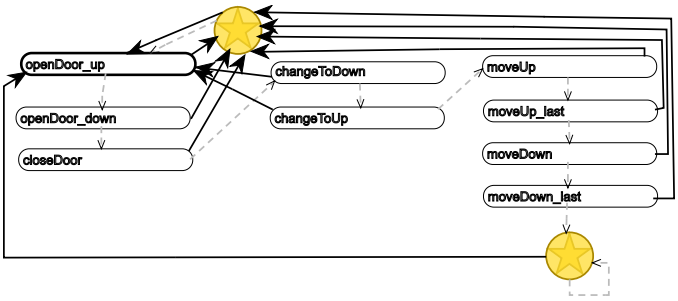

Figure 23. Rule annotated schedule of the operational semantics of the Elevator example.

sense to consider moving up or opening the doors to be interpreted as a conceptual time step, but maybe not changing the direction, as this happens on a much smaller time-scale than the elevator movement. Time-scale abstraction is hence appropriate [62]. Also, a more complex conceptual step might need more than one rule to express it, causing semantically inconsistent states in between. For example, the rule that opens the doors also immediately unlights the corresponding button. Suppose this would be modelled in two rules, then the language engineer might want to avoid adding the state to the trace that is in between those two actions.

To be able to distinguish between rules we introduce annotations for rule schedules, that allow determining the points when input is read and output is written. The CreateOS activity of Figure 10 includes the traditional modelling of the operational semantics (mentioned in Section 2.3, and out of scope of this paper) and an annotation of the rule schedule. Two new language constructs are used in Figure 23: the input star and the Boolean attribute conceptualStep that allows full or dashed transitions.

When the schedule arrives in an input star, the next input event is read, and the currentEvent link (see Figure 17) is pointed to the next input event, if available. The input star thus models the interleaving between input language and run-time language. This cooperation between heterogeneous models is modelled using principles of semantic adaptation, where data (in this case button presses), time (in this case the conceptual time steps) and control (in this case determined by the rule schedule) of one formalism is adapted to the other [9], [55]. The input star can be translated to a generated rule implementing this. The rule links all elements in the input event by id to elements of the run-time model and propagates changes to the run-time model. The corresponding element in the run-time model can always be found because all elements of the input model are also present in the run-time model. Note that using metamodel annotations on classes would require the input language to be changed in order to be of full use. Because dynamically created class instances would not be addressable by id, the events should consist of patterns to indicate what part of the model receives input (*i.e.,* what button is lit). To that end, the DSML elements in the input language would have to be RAMified similarly to the property language, and structural pattern elements might have to be added. We choose not to do this to keep the input language simple, and because there are no additional limitations to our approach. After executing the input star, the successful link is modelled if the read event was not empty, or the notApplicable link if the event was empty.

Transitions in an annotated rule schedule can include appending the current state to the output trace (full line), or not (dashed line). By default, notApplicable links are dashed and successful links are full. Appending to the output trace can be implemented as a rule Append that can be generated from the annotated metamodel. A transition with a full line can be translated to a traditional transition by replacing it with a transition of the same type to Append (a new reference to Append for each transformed transition), followed by a notApplicable and successful link to the target. The translation from an annotated transformation model to a traditional transformation model is defined as a higher order transformation, that serves as the transformational semantics of the annotated transformation.

In the example of Figure 23, output is written after every successful rule application, except after reading input. Input is read after every successful rule application, except for the rules that change the direction. Note that the bottom input star keeps reading input events until something has changed (*i.e.,* a non-empty event was applied).

4.4 Migrating to *ProMoBox* Compliance

Throughout this section, we have assumed that the language engineer can start modelling a language from scratch. In this section, we will focus on *UC2*: how to simplify an available, traditionally modelled DSML and how to migrate this DSML to make it compatible with the *ProMoBox* approach.

4.4.1 Simplification

In some cases, the metamodel needs to be simplified manually to address scalability issues of model checking. We investigated this simplification step by applying the *ProMoBox* approach to a DSML for gestural interaction [20]. Together with annotating the metamodel and operational semantics model, this is the only manual task that needs to be performed in comparison with traditional DSML language design. As explained in Section 2.6, it is common practice to improve efficiency in model checking via abstracting the model (*i.e.,* by reducing the search tree by abstracting the model) [4]. To this end, we limit annotated metamodels with the following constraints:

- **attribute abstraction**: strings and floating point variables cannot be used as dynamic variables, although they can be used as static variables. Such variables need to be simplified to Booleans or integers, to decrease the domain size of the variable. Composite types, such as lists or maps, are not supported. The user may choose to model a number of attributes that represent an array, but this is not recommended as the number of variables is best kept as small as possible;
- **integer limitation**: integer values are restricted to a maximum of $[0..255]$ (byte) or $[-2^{15} - 1..2^{15} - 1]$ (short), depending on the user's choice when compiling the model;
- **association limitation**: associations can be instantiated a maximum number of times, depending on the user's choice when compiling the model;
- **dynamic class limitation**: similarly, class instantiation (*i.e.,* classes must be part of the design metamodel) can be instantiated a maximum number of times.

Note that these constraints make the number of distinct states of the DSML (*i.e.,* instances of its run-time language) finite. This is essential in order to be able to analyse the entire state space, as done by explicit-state model checkers such as *Spin*.

In addition to adhering to these constraints, following some guidelines has a direct impact on the performance:

- **avoid dynamic classes**: it is not recommended in *ProMoBox* to include dynamic instantiation of classes in the DSML as it severely reduces efficiency by increasing the state space as *Promela* does not support dynamic instantiation;
- **limit variables**: it is good practice to limit the number of variables, especially integer attributes and dynamic associations (*i.e.,* that are not part of the design metamodel);
- **limit inheritance**: the number of inheritance links should be limited if possible, as inheritance needs to be emulated in *Promela*. Flattening the inheritance hierarchy reduces the execution time, because for pattern instances, less candidates have to be evaluated;
- **limit input language**: since model checking will take any possible scenario into account, the execution time is heavily dependent on what input it can get. Consequently, the number of distinct inputs scenarios should be limited by making abstractions that decrease the complexity of the input language (*i.e.,* limit the number of elements that are part of the input metamodel). When possible, it is recommended to translate integers in the input language to enumerations.

The operational semantics transformation can be simplified in a similar way. There are no constraints in the design of the operational semantics, but some guidelines should be taken into account:

- **determinism**: it is recommended to avoid nondeterminism in the transformation's rule schedule, as this can drastically increase the size of the state space;
- **number of pattern matches**: the number of matches that a pattern is expected to have should be kept to a minimum as this increases the state space width. In particular, abstract classes in a pattern should be used with caution.

These simplifications only affect the DSML formalism. Note that the eventual model checking performance will also be heavily affected by the size of the system model and the complexity of the property.

4.4.2 Migration of Existing Models

A traditional metamodel can be automatically converted to an annotated metamodel. The metamodel language (*i.e.,* Class Diagrams) is a subset of *ProMoBox*'s metamodelling language Annotated Class Diagrams language, which only adds the possibility to annotate metamodel elements with different predefined annotations. We can analyse the migration impact of changes by interpreting switching from Class Diagrams to Annotated Class Diagrams as a language evolution [58]. In this case, only additive, non-breaking changes are made. This means that the conformance of the Class Diagram instances (*i.e.,* the actual metamodels), is not *broken* after migrating them to Annotated Class Diagrams, only there are no annotations used. Depending on the tool that is used, it might however be necessary to change the namespace of the types used in the metamodel to explicitly denote that the metamodel is an annotated metamodel. This can be done using a simple transformation that traverses all metamodel elements and changes the type. Once the metamodel is migrated, the regular *ProMoBox* process can continue from AnnotateMM activity in Figure 10.

A traditional operational semantics model can be automatically converted to an annotated transformation model as well. Similarly, this is an additive, non-breaking change in the transformation language, where the rule schedule is migrated to an annotated rule schedule, and the patterns in rules are migrated to the run-time language. The migrated rule schedule will not yet include input stars, and will have the default output behaviour. The migrated rules do not change (except for namespace changes if necessary), and can be used in *ProMoBox* as is.

An existing concrete syntax model does not need to be migrated as traditional concrete syntax modelling is still used in the *ProMoBox* approach.

As a consequence of the migration of a traditional metamodel, existing instances should be converted as well. The traditional metamodel might include static, dynamic and input concepts, which are all included in the run-time language according to the annotation type rules. This means that to migrate instances to the run-time language, yet again, no explicit migration is necessary. If however the instance has to be migrated to the design language or input language, some language constructs might become unavailable. These metamodel changes can be calculated in retrospect by analysing the difference between the traditional metamodel and the design or input metamodel. The resulting metamodel changes are eliminate metaclass, and eliminate metaproperty (which covers both associations and attributes) [58]. These are subtractive, breaking and resolvable changes, so a corresponding migration transformation can be generated. The transformation will remove all instances of removed meta-elements, thus again establishing conformance with the target language (*i.e.,* design or input language).

As a conclusion, all existing DSM artifacts can be automatically migrated to *ProMoBox*, so that the additional required effort for the language engineer is limited to annotating the metamodel and transformation schedule.

4.5 Evolution of the DSML in *ProMoBox*

The DSML, notably :AnnotatedMetamodel, :ConcreteSyntax and :AnnotatedTransformation in Figure 10, may evolve, causing inconsistencies among other modelling artifacts. As the *ProMoBox* approach is generative, this generative process can be redone in case of evolution, consequently solving inconsistencies between sublanguages. Since :AnnotatedMetamodel is created at the very start of the *ProMoBox* process, all subsequent activities have to be redone. There are however shortcuts possible:

- if only :ConcreteSyntax evolves, :AnnotatedMetamodel and the metamodel of the five sublanguages can remain untouched. Only the concrete syntax models have to be regenerated (the automated :FilterCS, :MergeCS and :RAMify activities) in Figure 14. In *AToMPM*, instance models like in Figure 27 and operational semantics transformation models that use the concrete syntax do not need to change. We validated this by changing the button icons (compared to [57]), which was achieved in a matter of minutes;
- if only :AnnotatedTransformation evolves, only the compilation to the verification backbone and the verification itself have to be redone, similar to changing :Property. No models need to be adapted. We validated this by optimising our operational semantics as explained in Section 7.3, by reordering rules and removing the NACs, which was also achieved within minutes;
- in case of an evolution of :AnnotatedMetamodel, co-evolution may happen for (some of the) sublanguages

#	Annotated metamodel change operation	Type	Sublang. ch.	Sublanguage instance migration operation
1	Generalise property	Non-breaking	1	None
2	Add class	Non-breaking	2	None
3	Add non-obligatory property	Non-breaking	3	None
4	Make class concrete	Non-breaking	4	None
5	Extract abstract superclass	Non-breaking	5	None
6	Extract superclass	Non-breaking	6	None
7	Flatten abstract hierarchy	Non-breaking	7	None
8	Push property from abstract class	Non-breaking	8	None
9	Pull property to abstract class	Non-breaking	9	None
10	Eliminate class	Breaking and resolvable	10	Eliminate instances
11	Eliminate property	Breaking and resolvable	11	Eliminate instances
12	Make class abstract	Breaking and resolvable	12	Eliminate instances
13	Extract class	Breaking and resolvable	13	Extract properties and add instances
14	Inline class	Breaking and resolvable	14, 3, 20, 10	Inline properties and remove instances
15	Flatten hierarchy	Breaking and resolvable	15	Eliminate superclass instances
16	Push property	Breaking and resolvable	16	Eliminate properties from superclass instances
17	Rename class	Breaking and resolvable	17	Change instances
18	Rename property	Breaking and resolvable	18	Change instances
19	Change class annotation	Breaking and resolvable	2, 10	Eliminate instances
20	Add obligatory property	Breaking and unresolvable	20	Add default instances
21	Pull property	Breaking and unresolvable	21	Add default properties for superclass instances
22	Restrict property	Breaking and unresolvable	22	Remove instance if non-compliant
23	Change property annotation	Breaking and unresolvable	3, 20, 11	Add default instances *or* Eliminate instances

Table 2
Catalogue of change operations adapted from [58], [12] and [34], including change operations (operation 19 and 23) for changes in metamodel annotations, and with the possible resulting change operations on the sublanguages (Sublang. ch.).

and their instances, as shown in Table 2. In the change catalogue, *property* applies to both association and attribute. It might very well be possible that changes to the annotated metamodel are not applied to one or more of the five sublanguages. For example, if an association annotated with *rt* is added, the design metamodel and input metamodel will remain the same. Consequently, their instance models do not need to be migrated either.

In case of a change in a sublanguage, it needs to be regenerated. Additionally, the respective co-evolution scenario described Table 2 must be followed. It lists all possible change operations on the annotation metamodel, together with its type (*i.e.,* non-breaking, breaking and resolvable, or breaking and unresolvable). The next column indicates what the possible change operation may be for each of the sublanguages. For most of the change operations, this may be either no change (if the affected elements are not in the sublanguage, as described above), or the same change. For example, suppose an *Eliminate property* operation on the annotated metamodel, of an attribute annotated with *rt*. This change does not change the design and input metamodel because the attribute is not included in these sublanguages, but results in the same *Eliminate property* operation on the run-time, trace and property metamodel.

As can be seen from Table 2, a special operation is *Inline class*, that moves a class' properties to a class to which it is associated with a one-to-one mapping, and removes the class and the association. In case the to be deleted class was not part of the sublanguage but the receiving class is, this results in a *Add non-obligatory property* or *Add obligatory property* operation on the sublanguage. The other way around, in case the to be deleted class was part of the sublanguage but the receiving class is not, this results in a *Eliminate class* operation on the sublanguages.

Because annotations were added to the metamodelling language in the *ProMoBox* framework, two additional change operations are added, namely *Change class annotation* and *Change property annotation*. As indicated in Table 2, *Change class annotation* may result in *Add class* and *Eliminate class* operations on sublanguages, and *Change property annotation* may result in *Add non-obligatory property*, *Add obligatory property* or *Eliminate property*.

In any case, the resulting changes of the sublanguages are fully compliant with [58], and co-evolution rules can be applied to its instances. The migration operation is shown in the rightmost column of Table 2. In case of breaking and unresolvable changes, a default migration operation is suggested, but often a customised migration operation needs to be developed, or even manual migration needs to be employed.

If a migrated instance is input to the Compile2Pml activity, the full verification process as described above has to be restarted. Incremental techniques might be used to only partly regenerate or recompile, but since the impact of the automated activities can be neglected, this is out of scope of this paper.

In [20] we went a step further by applying *ProMoBox* to a DSML for the gestural interaction domain, including a simplification step as described in Section 4.4.1. This process only took slightly longer than a traditional DSM process. This confirms the discussion of Section 7.1.

The use of generative techniques for generating sublanguages possibly introduces problems in case of evolution of a DSML, as stated in use case *UC3*. Indeed, instances of any of the sublanguages may become invalid, and need to be migrated to the new DSML version. Such problems are present in a regular DSML context as well, and in this section we showed how to apply resolutions

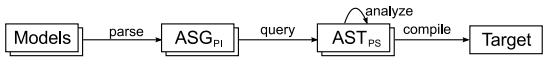

Figure 24. The *ProMoBox Promela* compilation process.

for DSML evolution to the *ProMoBox* approach. Each part of a DSML can evolve, namely the abstract syntax, the concrete syntax, and the operational semantics. The impact of the latter two are completely automated in the *ProMoBox* framework. Using the 23 change operations, we can conclude that the evolution of the DSML, although it entails five sublanguages, can be treated as a regular language evolution problem. A solution to the language evolution problem is given in, amongst other works, [58], and is therefore out of scope for this paper.

Another type of inconsistency that may occur is that changing a design or run-time model may invalidate the property model. However, this change does not render the property model syntactically incorrect. Structural patterns that were intended to match, may simply never match. In this framework, we do not provide dedicated support for handling the fact that changing a design model may invalidate a property model. Nonetheless, some auxiliary language constructs can be used to address such consistency issues. Firstly, quantification patterns may be evaluated at design time. Secondly, the introduced id attribute refers to model elements in the design model, and can be used for consistency checking. Nevertheless, because property models are first-class modelling artifacts, we consider the modelling of correct property models the responsibility of the domain user.

5 Mapping to a Verification Backbone

As shown in Figure 11, a generic transformation generates a *Promela* model (a `pml` file) containing an LTL formula by means of a model-to-text transformation. This section explains this part in detail. The *Promela* model is compiled from the following models:

- the annotated metamodel, that is used to generate the *Promela* run-time metamodel and environment;
- an initial state in the form of a run-time model;
- a property model;
- an operational semantics model;
- the annotations model (not shown);
- the LTL compilation rules, specifying how temporal patterns are translated to LTL formulas (not shown), which is added for flexibility (see Section 5.5).

A dedicated *Promela* model is generated that is optimised for a specific modelled system, initial state and property. The compilation process follows the steps shown in Figure 24. First, the models are parsed following the traditional concrete syntax parsing, resulting in a platform independent abstract syntax graph ASG_{PI}. Then, the abstract syntax is queried taking the *Promela* target language into account to obtain a platform specific abstract syntax tree AST_{PS}. The distinction between ASG_{PI} and AST_{PS} is made to allow easier replacement of the verification backbone, in case of *UC4*. The AST_{PS} is used to perform semantic analysis that organises the tree to generate optimised *Promela* code. Finally, the AST_{PS} is compiled to *Promela* including an LTL formula. The compilation results in a `pml` file that serves as input for the *Spin* verification tool.

Promela is a language specifically designed for explicit state model checking, meaning that during model checking the state space, *i.e.,* a graph of states of the system, is formed. This means that in designing a suitable *Promela* model, certain restrictions have to be taken into account:

- the *state space* (*i.e.,* number of different states and transitions) needs to be as small as possible. This means that nondeterminism has to be limited to constrain the breadth of the state space and that the number of atomic statements has to be limited to constrain the depth of the state space. This influences the verification time and memory consumption;
- the *state vector* has a static size and needs to be as small as possible, so that the individual nodes in the graph are as small as possible. This mainly influences the memory consumption of the verification.

```
1  <METAMODEL>
2  hidden __System s;
3  __RuntimeSystem r;
4  <ENVIRONMENT>
5  <LTL FORMULA>
6  active proctype instance() {
7    <INSTANCE>
8    <SET INITIAL RULE>
9    do ::
10     atomic {
11       <RULE SCHEDULE>
12       <RULE 1>
13       <RULE 2>
14       ...
15       <RULE N>
16       <PRINT STATE>
17       <UPDATE STATE>
18     }
19   od;
20 }
```

Listing 1. The overall structure of the generated *Promela* model.

The overall structure of the `pml` file is shown in Listing 1, where code snippets are referenced between angle brackets, which are explained in detail below. The *Promela* model is structured as follows, and the role of each model in the compilation process is discussed in the remainder of this subsection:

- line 1: code for the metamodel (see Section 5.1) which implements the definition of types;
- line 2-3: declaration of the system variables for the static `s` and dynamic part `r` of the system, which are typed by the metamodel. The `hidden` keyword denotes that `s`, the static part of the system, is not part of the state vector;
- line 4: code for the input language (see Section 5.3) which is implemented as a function that nondeterministically executes an input event;
- line 5: an LTL formula implementing the temporal pattern (see Section 5.5);
- line 6: start of the main process (stops at line 20);
- line 7: code for the initial state (see Section 5.2) initialising the system variable by assigning values to fields of `r` and `s`;
- line 8-15: code for operational semantics (see Section 5.4), including a `do`-loop, *i.e.,* the simulation loop, that applies a rule at each indivisible atomic iteration (ends at line 19). This includes a rule schedule, which branches (by using `goto`s) to code for rules;
- line 16: code for the trace language (see Section 5.6), which prints out the current state of the system in a predefined format, which can be written to a text file;

```
1  mtype = {movedown_last_, opendoor_up_, changetoup_, changetodown_,
       opendoor_down_, movedown_, moveup_last_, closedoor_, environmentstep2_,
       environmentstep1_, moveup_, ElevatorButtonType, ElevatorType, FloorType,
        UpButtonType, DownButtonType}
2  typedef Button {
3    short __subtype;
4    short requests_out; // Floor out
5    short elevator_button_in; // Elevator in
6  }
7  typedef RuntimeButton {
8    bit pressed;
9  }
10 typedef Elevator {
11   short __subtype;
12   short elevator_button_out[3]; // ElevatorButton out
13 }
14 typedef RuntimeElevator {
15   bit doors_open;
16   bit going_up;
17   short currentfloor_out; // Floor out
18 }
19 typedef Floor {
20   short __subtype;
21   short nr;
22   short next_out; // Floor out
23   short next_in; // Floor in
24   short requests_in[3]; // Button in
25   short currentfloor_in; // Elevator in
26 }
27 typedef __System {
28   Button button_[7];
29   Elevator elevator_[1];
30   Floor floor_[3];
31 }
32 typedef __RuntimeSystem {
33   RuntimeButton button_[7];
34   RuntimeElevator elevator_[1];
35 }
```

Listing 2. The compiled bounded meta-model.

```
1  d_step {
2    // ElevatorButton $atompmId:1
3    s.button_[0].__subtype = ElevatorButtonType;
4    r.button_[0].pressed = true;
5    s.button_[0].requests_out = 0;
6    s.button_[0].elevator_button_in = 0;
7    // ElevatorButton $atompmId:2
8    s.button_[1].__subtype = ElevatorButtonType;
9    r.button_[1].pressed = false;
10   s.button_[1].requests_out = 1;
11   s.button_[1].elevator_button_in = 0;
12   (...)
13
14   // Elevator $atompmId:0
15   s.elevator_[0].__subtype = ElevatorType;
16   r.elevator_[0].doors_open = true;
17   r.elevator_[0].going_up = false;
18   s.elevator_[0].elevator_button_out[0] = 2;
19   s.elevator_[0].elevator_button_out[1] = 0;
20   s.elevator_[0].elevator_button_out[2] = 1;
21   r.elevator_[0].currentfloor_out = 2;
22
23   // Floor $atompmId:4
24   s.floor_[0].__subtype = FloorType;
25   s.floor_[0].nr = 2;
26   s.floor_[0].requests_in[0] = 3;
27   s.floor_[0].requests_in[1] = 0;
28   s.floor_[0].requests_in[2] = -1;
29   s.floor_[0].next_in = 1;
30   s.floor_[0].next_out = -1;
31   s.floor_[0].currentfloor_in = -1;
32   (...)
33 }
```

Listing 3. The compiled design model and initial state (abbreviated).

- line 17: code for evaluating the property propositions (*i.e.,* structural patterns) which will be executed after every state change (*i.e.,* rule application) (see Section 5.5).

5.1 Compilation of the Metamodel

A metamodel is created in the form of `typedef` statements as shown in Listing 2, that allow the declaration of statically structured types. First (line 1), an `mtype` declaration is given, which introduces symbolic names for concrete metamodel types and rules (the latter will be used in Section 5.4). Only the three classes on top of the inheritance hierarchy become *Promela* types (line 2-26). The instances of these types are stored as static arrays, and instances are accessed by indexing that array. Attributes are converted to `typedef` fields (*e.g.,* line 8), with corresponding types. Since *Promela* is not an object-oriented language, inheritance and associations have to be encoded, as shown in Listing 2. Inheritance is implemented by the `__subtype` attribute (*e.g.,* line 3), that refers to any class in the run-time or design metamodel. Associations are implemented with bidirectional navigability by fields of type `short`, that refer to the index of the target, rather than to an object (*e.g.,* line 4). For instance, if the `currentfloor_out` (line 17) of an `Elevator` is 1, its target is the `Floor` with index 1. If no link exists, its index is set to -1. Two kinds of types are created: static types (`Button`, `Elevator`, `Floor` and `__System`) that model the design language, and dynamic types (`RuntimeButton`, `RuntimeElevator` and `__RuntimeSystem`) that model the additional elements of the run-time language. This distinction is made so that the state vector only contains a minimum of state information.

The model of the initial state of the modelled system (modelled as a run-time instance) is taken into account in three ways to ensure that the state space of the *Promela* model is bounded. First, a `__System` type is declared (line 27-31) with static arrays of 7 Buttons, 1 Elevator and 3 Floors, and a `__RuntimeSystem` type (line 32-35) with 7 Buttons and 1 Elevator, referring to the multiplicities of the design model. These two types represent the static and dynamic part of the modelled system, and both are instantiated once as shown in Listing 1 on line 2-3. Second, maximum array sizes for fields implementing associations are chosen according to the multiplicities of the design model. If the number of possible instances of a dynamic association is unbounded, a maximum number is set to ensure boundedness of the *Promela* model. Third, only one end of each dynamic association needs to be stored in the dynamic state vector, and the one with the lowest multiplicity of the design model is chosen in order to limit the state vector size. Both ends are stored to improve navigability but will constantly be kept synchronised, so only one end has to be part of the state vector. In this example, the `currentfloor_out` field (line 17) has a multiplicity of maximum 1 as there is only one Elevator in the design model, while the `currentfloor_in` field (line 25) has a multiplicity of maximum 3 as there are three Floors, so the former is stored in the state vector.

5.2 Compilation of the Model and Initial State

The first statements of the one and only active process started on line 6 in Listing 1 set the values of the initial state of the modelled system as shown partially in Listing 3. The values for each model element are set, creating attribute values (*e.g.,* line 4), type values (*e.g.,* line 3) and association links (*e.g.,* line 5). In comments the corresponding type and element id from the modelling tool are given as documentation. This results in the static `__System` instance `s` and the initial state of the dynamic `__RuntimeSystem` instance `r`. The element order of array fields in `__System` and `__RuntimeSystem` is by definition the same, meaning

```
1  inline environment() {
2    if
3      :: true -> skip
4      :: r.button_[0].pressed == 0 ->
5        d_step { success = true; r.button_[0].pressed = true; }
6      :: r.button_[1].pressed == 0 ->
7        d_step { success = true; r.button_[1].pressed = true; }
8      :: r.button_[2].pressed == 0 ->
9        d_step { success = true; r.button_[2].pressed = true; }
10     :: r.button_[3].pressed == 0 ->
11       d_step { success = true; r.button_[3].pressed = true; }
12     :: r.button_[4].pressed == 0 ->
13       d_step { success = true; r.button_[4].pressed = true; }
14     :: r.button_[5].pressed == 0 ->
15       d_step { success = true; r.button_[5].pressed = true; }
16     :: r.button_[6].pressed == 0 ->
17       d_step { success = true; r.button_[6].pressed = true; }
18   fi
19 }
```

Listing 4. The compiled environment model.

that *e.g.,* `s.button_[2]` refers to the same Button instance as `r.button_[2]`. The initialisation code is in a `d_step` block, making it an indivisible, deterministic block of code.

5.3 Compilation of the Input Language

Listing 4 shows a macro containing a model of one execution step of the environment that implements the input language. It represents passing through an input star during simulation. It consists of an if-statement that can set the `pressed`-value of any Button to *true* (or 1) if it was not yet *true*. *Promela* evaluates the if-statement by evaluating all of its conditions, then chooses randomly one option that evaluates to *true*, and executes the corresponding body. The code of an environment step that is shown above ensures that at most one, but also no (see line 3) button can be pressed. Other semantics can be chosen which would result in a variation of this macro, such as: exactly one option has to be chosen, more than one but unique options can be chosen, more than one and the same options can be chosen, etc. Also, in our current implementation, a Boolean input is implemented that can only be turned on, not off, by the environment. The environment strongly influences the size of the state space, as this macro is typically executed a multitude of times and might result in a significant number of state space branches. Therefore, the possible options (in this case maximally 8) should be kept to a minimum by wisely choosing the above described environment variant. This often requires an abstraction. The success variable is necessary to denote whether a rule was applied and reflects whether the success or notApplicable link exiting an input star should be followed in the rule schedule. The print statement will be used to create a textual report in the case of a counterexample (see Section 5.6).

5.4 Compilation of the Operational Semantics

Listing 5 shows the partial rule schedule of Figure 23 in *Promela*. This rule schedule is part of the simulation loop shown in Listing 1. In line 1-14, control flow is directed to the correct rule. The initial rule is set earlier in the code, right before the start of the simulation loop, by assigning the `nextrule` variable. From line 16 onward, the schedule is modelled which is activated after evaluating a rule. After evaluation, the rule was either successful (*e.g.,* line 18) or not applicable (*e.g.,* line 22), which is modelled by the `success` variable that is set in the rule code. According to this, the new value of `nextrule` is determined, and depending on whether output must be added to the output trace control flow is directed to `OUTPUT`

```
1  if
2    :: (nextrule == movedown_last_) -> goto MOVEDOWN_LAST;
3    :: (nextrule == movedown_) -> goto MOVEDOWN;
4    :: (nextrule == moveup_last_) -> goto MOVEUP_LAST;
5    :: (nextrule == opendoor_up_) -> goto OPENDOOR_UP;
6    :: (nextrule == closedoor_) -> goto CLOSEDOOR;
7    :: (nextrule == changetoup_) -> goto CHANGETOUP;
8    :: (nextrule == changetodown_) -> goto CHANGETODOWN;
9    :: (nextrule == initialize_) -> goto INITIALIZE;
10   :: (nextrule == environmentstep2_) -> goto ENVIRONMENTSTEP2;
11   :: (nextrule == environmentstep1_) -> goto ENVIRONMENTSTEP1;
12   :: (nextrule == opendoor_down_) -> goto OPENDOOR_DOWN;
13   :: (nextrule == moveup_) -> goto MOVEUP;
14 fi;
15
16 MOVEDOWN_LAST_schedule:
17 if
18   :: (success == true) -> // when successful
19   rule = movedown_last_;
20   nextrule = environmentstep1_;
21   goto OUTPUT;
22   :: else -> // when not applicable
23   nextrule = changetodown_;
24   goto UPDATESTATE;
25 fi;
26 OPENDOOR_UP_schedule:
27 if
28   :: (success == true) -> // when successful
29   rule = opendoor_up_;
30   nextrule = environmentstep1_;
31   goto OUTPUT;
32   :: else -> // when not applicable
33   nextrule = opendoor_down_;
34   goto UPDATESTATE;
35 fi;
36 (...)
37 ENVIRONMENTSTEP1_schedule:
38 if
39   :: (success == true) -> // when successful
40   nextrule = opendoor_up_;
41   goto UPDATESTATE;
42   :: else -> // when not applicable
43   nextrule = opendoor_up_;
44   goto UPDATESTATE;
45 fi;
46
47 ENVIRONMENTSTEP2:
48 success = false;
49 environment();
50 goto ENVIRONMENTSTEP2_schedule;
51
52 ENVIRONMENTSTEP1:
53 success = false;
54 environment();
55 goto ENVIRONMENTSTEP1_schedule;
```

Listing 5. The compiled rule schedule (abbreviated) and environment steps.

(see Listing 1 line 16) or directly to `UPDATESTATE` (see Listing 1 line 18), after which a new iteration of the simulation loop starts. Note that a new iteration starts after every evaluation of a rule, successful or not.

The input stars are modelled as rules on line 47-55 and as such, can be triggered by line 1-14. The `success` variable is set to false, and can be set to *true* in the function call to the environment macro of Listing 4. Like any rule, control flow is directed to the right `schedule` block at the end of the rule.

The compiled changeToUp rule is shown in Listing 6. Just like the input star, it starts by setting the `success` variable to false and ends with a `goto` statement. A rule consists of an LHS (the pattern that must be matched), optional NACs (given an LHS match, patterns that should not match) and an RHS (the effect of the pattern).

For each of these three parts, its pattern elements are compiled one by one. The order in which elements are compiled is determined by a sorting algorithm, that sorts according to a score for each

```
1  CHANGETOUP:
2  success = false;
3  e0 = 0; // this is the only Elevator so index must be 0
4  f1 = r.elevator_[e0].currentfloor_out; // direct access
5  b2_indexes[0]=0; b2_indexes[1]=1; b2_indexes[2]=2; b2_indexes[3]=3; b2_indexes
       [4]=4; b2_indexes[5]=5; b2_indexes[6]=6;
6  b2_max = 7;
7  do
8  :: (success == false && b2_max > 0) ->
9    if
10   :: ((e0 >= 0) && (s.elevator_[e0].__subtype == ElevatorType)
11   && (r.elevator_[e0].going_up == false)
12   && (f1 >= 0)
13   && (s.floor_[f1].__subtype == FloorType)
14   && (b2_max > 0) && (b2_indexes[0] >= 0)
15   && (s.button_[b2_indexes[0]].__subtype == ElevatorButtonType
16       || s.button_[b2_indexes[0]].__subtype == DownButtonType
17       || s.button_[b2_indexes[0]].__subtype == UpButtonType)
18   && (r.button_[b2_indexes[0]].pressed == true)
19   && (s.button_[b2_indexes[0]].requests_out >= 0)
20   && (s.button_[b2_indexes[0]].requests_out != f1)
21   && (s.floor_[s.button_[b2_indexes[0]].requests_out].__subtype == FloorType)
22   && (s.floor_[s.button_[b2_indexes[0]].requests_out].nr>s.floor_[f1].nr))
23   -> b2 = b2_indexes[0]
24   (...)
25   :: else -> break
26   fi;
27   f4 = s.button_[b2].requests_out; // direct access
28   // NAC_
29   NAC__success = false;
30   b7_indexes[0]=0; b7_indexes[1]=1; b7_indexes[2]=2; b7_indexes[3]=3;
       b7_indexes[4]=4; b7_indexes[5]=5; b7_indexes[6]=6;
31   b7_max = 7;
32   do
33   :: (NAC__success == false && b7_max > 0) ->
34     if
35     :: ((b7_max > 0) && (b7_indexes[0] >= 0)
36     && (s.button_[b7_indexes[0]].__subtype == ElevatorButtonType
37         || s.button_[b7_indexes[0]].__subtype == DownButtonType
38         || s.button_[b7_indexes[0]].__subtype == UpButtonType)
39     && (r.button_[b7_indexes[0]].pressed == true)
40     && (s.button_[b7_indexes[0]].requests_out >= 0)
41     && (s.button_[b7_indexes[0]].requests_out != f1)
42     && (s.floor_[s.button_[b7_indexes[0]].requests_out].__subtype
43         == FloorType)
44     && (s.floor_[s.button_[b7_indexes[0]].requests_out].nr
45         < s.floor_[f1].nr))
46     -> b7 = b7_indexes[0]
47     (...)
48     :: else -> break
49     fi;
50     f6 = s.button_[b7].requests_out; // direct access
51     // NAC_matched
52     NAC__success = true;
53     break;
54   :: else -> break
55   od;
56   if
57   :: (NAC__success == false) ->
58     r.elevator_[e0].going_up = true;
59     success = true;
60     break;
61   :: else ->
62     b2_max--; temp = b2_indexes[b2_max]; b2_indexes[b2_max] = b2_indexes[
         b2_index]; b2_indexes[b2_index] = temp;
63   fi;
64 :: else -> break
65 od;
66 goto CHANGETOUP_schedule;
```

Listing 6. The compiled changeToUp rule (abbreviated).

element, representing the expected probability to find a match. This compilation order is illustrated in Figure 25, where the score of each element, followed by the order of compilation between brackets, is shown at the right side of each element on Figure 25. The score given to each pattern element is determined by the following formula: $10 \cdot nrOfAttributes - 2 \cdot multiplicity - subclassMatching$, with $nrOfAttributes$, the number of attributes with a constraint, $multiplicity$, the number of elements of that type, and $subclassMatching$ whether or not (value 1

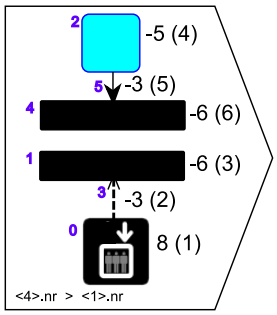

Figure 25. The order of compilation of the LHS of the changeToUp rule.

or 0) this element can be matched with candidates of a subtype (*e.g.,* an UpButton can match a Button pattern element). Pattern elements with high scores are more constrained and thus less likely to match. For efficient pattern matching, it is desirable to know as soon as possible if the pattern will not match, and it is therefore preferable to try to match "hard" elements first. A different sorting algorithm can be plugged in easily. From a compiled element, the code generator traverses the pattern by following links from the element. A similar pluggable sorting algorithm is implemented for links to determine which of the pattern element links should be matched first.

In *Promela*, a match candidate is represented by an array index stored in a variable and the variable name is composed of the first letter of its type, followed by the label of the pattern element (*e.g.,* `e0` on line 3 represents a match candidate for the Elevator with label 0 in the changeToUp rule in Figure 5). In principle, the code for matching the elements and thus finding the right candidate consists of nested blocks in order of element traversal. Depending on the element one out of two kinds of code blocks is generated:

- a *do block*, if there are multiple candidates, *e.g.,* for matchingButtons. The do block iterates over all candidates until one is found that satisfies the element constraints. According to the pattern matching semantics, a random candidate has to be found. This is achieved by storing all valid candidates in a temporary `_indexes` array (line 5, 30). The first `_max` (line 6, 31) elements in the array are the candidates that are not evaluated yet. Since at the start no elements are evaluated yet, `_max` is equal to the length of `_indexes`. If a candidate does not satisfy the pattern element constraints, its `_indexes` entry is swapped and `_max` is decreased so that the candidate is not in the first `_max` elements of `_indexes` anymore (line 62). In each iteration, a candidate is chosen by an if statement that chooses a random candidate for which the additional conditions are met (line 9-26, 34-49). Each option has the same condition, but a different candidate is used from the `_indexes` array, *e.g.,* `b2_indexes[0]` is used for the first candidate as the Button with label 2, `b2_indexes[1]` as the second candidate (not shown), etc. For brevity, only one option is shown per if statement. If a candidate satisfies the condition, the candidate variable is set (line 23, 46). If no valid candidate can be found, control breaks out of the do loop (line 25, 48).
- an *if block*, if there is only one candidate, *e.g.,* for matching a Floor pattern element that is reached via the `requests_out` link from a Button. No `_indexes` variable

has to be used, because only the condition of the single candidate has to be checked.

In practice however, the nested structure is folded where possible into a smaller number of tests, by combining conditions of multiple pattern elements. This way the cyclomatic complexity, and consequently the state space, is decreased significantly. For the changeToUp rule, the LHS can be folded into a single do loop, and the NAC is folded into a single do loop as well.

The conditions that are used to check whether a candidate satisfies the constraints of a pattern element (line 10-22, 35-45) are the following:

1) they are not *null* (*i.e.,* the match candidate is not -1) and there are still possible candidates (line 10, 12, 14, 19, 35, 40);
2) if applicable, they are not the same as a previously matched item (line 20, 41),
3) if applicable, their dynamic type, represented by the `__subtype` attribute, is correct (line 10, 13, 15-17, 21, 36-38, 42-43) and,
4) if applicable, element conditions that are specified are satisfied (line 11, 18, 22, 39, 44-45).

The order of evaluating the candidates as shown in Figure 25 can be recognised in the order of expressions in the conditions.

If a match is found for the LHS (line 7-26, 64-65), and no match is found for a NAC (line 32-55), the right-hand side (RHS) of the rule is applied (line 58), which is generated from the difference between the RHS and the left-hand side of the rule. The rule is flagged successful and is exited (line 59-60). Finally the execution jumps back to the rule schedule, which will decide the next step (line 66).

5.5 Compilation of the Property Instance

The property instance is translated to an LTL formula (line 5 in Listing 1) and *Promela* code. *Promela* code is necessary to evaluate the formula's propositions which are modelled as structural patterns (line 17 in Listing 1). The compilation to an LTL formula is done according to the formulas of Table 1.

Additional to these formulas, Dwyer *et al.* identified more variants in [22], *e.g.,* whether scopes are open or closed on the left and right (by default scopes are closed on the left and open on the right), constrained response, chained patterns, number of links in chains, maximum number in bounded existence, etc. The number of possible combinations is enormous. In order to potentially support all possible combinations of variants, we introduced a dedicated mechanism to add variants. Rather than hard-coding the translation of a predefined set of variants in the compiler, we now allow the user to specify model-to-text rules that intuitively map given template extensions to LTL. Consequently, the language engineer can not only introduce new variants of the pattern system, but can also create new temporal properties in the template for property languages and specify their semantics in terms of LTL. The mechanism works at the domain-independent level, so once a new variant is specified, it is available for any DSML.

An example of such an LTL compilation rule is shown in Figure 26, where it is specified how a bounded existence with upper limit of 2 is translated to LTL. In the LHS of the rule, an instance the pattern is shown, with two generic structural patterns, called P and Q. In the RHS of the rule the corresponding LTL formula is specified, and P and Q are used as propositions. The compiler can use this rule as a template, and replace P and Q with actual propositions during the compilation process.

Additionally, quantification patterns are statically evaluated on the design model and for each match an LTL formula is instantiated, joined by logical *and* in case of *for all* quantification, and logical *or* in case of *exists* quantification. For example, the reachesFloor property of Figure 21 for the design model of Figure 19 can be compiled to the formula $\Box(\neg P0 \vee \Diamond(P0 \wedge \Diamond Q0)) \wedge \Box(\neg P1 \vee \Diamond(P1 \wedge \Diamond Q1)) \wedge \Box(\neg P2 \vee \Diamond(P2 \wedge \Diamond Q2))$. The two proposition patterns of Figure 21 (*i.e.,* the two rightmost structural patterns), are thus translated to six propositions, as both are expanded to each of the three floors. The proposition $P0$ represents "a button is pressed at the first floor", $Q0$ represents "the elevator has opened its doors at the first floor", etc. Note how the resulting formula does not only depend on the property, but also on the modelled system.

The propositions of the property must be evaluated after every rule application of the operational semantics. This is done in the `UPDATE STATE` block (line 17 in Listing 1), which is shown in Listing 7. The evaluation of propositions `P0` (line 5-16) and `Q0` line (17-24) are shown, and `P1`, `P2`, `Q1`, and `Q2` are not shown but are similar, modulo the different value of `f0` (line 4). Since the code represents matching a pattern, it is similar to the code implementing a rule. The main difference however is that we are only interested in *whether* a match can be found, while it is not important *which* match is found. Therefore, the choice of candidates can be deterministic (line 3), allowing for more optimal and simpler code that only traverses all candidates (line 6-16). If a match is found, the proposition variable becomes 1 (line 12), else it remains 0.

5.6 Compilation and Parsing of the Counterexample

If the transition that is followed in the rule schedule dictates that output must be written, the schedule directs to the `OUTPUT` label in the `PRINT STATE` block (line 16 in Listing 1) which is shown in Listing 8. The last executed rule and state of the system are printed out using the id of the corresponding model element. This information can be used to trace back the value to the run-time model, and allows for playing out a trace. The `printf` statements are only executed during simulation in *Spin*, not during model checking.

After the *Promela* model is generated (step 1 in Figure 11), *Spin* uses model checking to find a counterexample of the property (step 2 in Figure 11). If a counterexample is found, a *trail* file is generated, which can be simulated in *Spin*, to execute the print state code of Listing 8 (step 3 in Figure 11). The resulting text is structured according to the grammar shown in Listing 9 and represents a trace, *i.e.,* the succession of states. The first line of the text contains the LTL formula (corresponding to Table 1), the subsequent lines are generated during simulation in *Spin*, and can be:

- a system step: rules of the operational semantics, created by executing the `printf` statements of Listing 8;
- an environment step: similar to a system step, but representing the occurrence of an input (*i.e.,* line 14 and 15 of Listing 8);
- a property step: the progression of the LTL formula as so-called never claim (*i.e.,* Büchi automaton), which is generated by *Spin* [35]. This can be either a regular progression to a subformula of the LTL formula, or the indication that an acceptance cycle is reached.

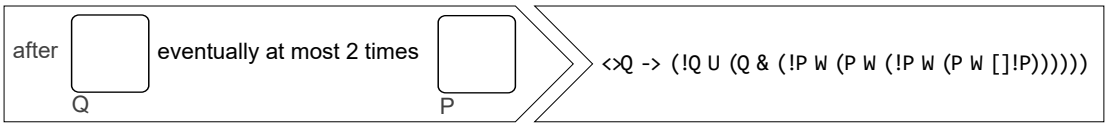

Figure 26. The LTL compilation rule for a bounded existence pattern with after scope and $n = 2$.

```
UPDATESTATE:
skip;
d_step {
  f0 = 0; // statically set
  P0 = 0;
  do
  :: ((P0 == 0) && (b1 < 3) && (s.floor_[f0].requests_in[b1] >= 0)
     && (s.button_[s.floor_[f0].requests_in[b1]].__subtype ==
      ElevatorButtonType
        || s.button_[s.floor_[f0].requests_in[b1]].__subtype ==
      DownButtonType
        || s.button_[s.floor_[f0].requests_in[b1]].__subtype == UpButtonType)
     && (r.button_[s.floor_[f0].requests_in[b1]].pressed == 1)) ->
    P0 = 1;
    break;
  :: (b1 >= 3) -> break;
  :: else -> b1++;
  od;
  Q0 = 0;
  e3 = 0; // via id field
  if
  :: ((r.elevator_[e3].doors_open == 1)
     && (f0 == r.elevator_[e3].currentfloor_out)) ->
    Q0 = 1;
  :: else -> skip;
  fi;
  (...)
}
```

Listing 7. The compiled proposition patterns of the property.

```
OUTPUT:
skip;
d_step {
  printf("%d: ", timestep);
  if
  :: (rule == movedown_last_) -> printf("MOVEDOWN_LAST ");
  :: (rule == movedown_) -> printf("MOVEDOWN ");
  :: (rule == moveup_last_) -> printf("MOVEUP_LAST ");
  :: (rule == opendoor_up_) -> printf("OPENDOOR_UP ");
  :: (rule == closedoor_) -> printf("CLOSEDOOR ");
  :: (rule == changetoup_) -> printf("CHANGETOUP ");
  :: (rule == changetodown_) -> printf("CHANGETODOWN ");
  :: (rule == initialize_) -> printf("INITIALIZE ");
  :: (rule == environmentstep2_) -> printf("ENVIRONMENTSTEP2 ");
  :: (rule == environmentstep1_) -> printf("ENVIRONMENTSTEP1 ");
  :: (rule == opendoor_down_) -> printf("OPENDOOR_DOWN ");
  :: (rule == moveup_) -> printf("MOVEUP ");
  :: else -> skip;
  fi;
  printf("$atompmId:1.pressed=%d;", r.button_[0].pressed);
  printf("$atompmId:2.pressed=%d;", r.button_[1].pressed);
  printf("$atompmId:3.pressed=%d;", r.button_[2].pressed);
  printf("$atompmId:7.pressed=%d;", r.button_[3].pressed);
  printf("$atompmId:8.pressed=%d;", r.button_[4].pressed);
  printf("$atompmId:9.pressed=%d;", r.button_[5].pressed);
  printf("$atompmId:10.pressed=%d;", r.button_[6].pressed);
  printf("$atompmId:0.doors_open=%d;", r.elevator_[0].doors_open);
  printf("$atompmId:0.going_up=%d;", r.elevator_[0].going_up);
  if
  :: (r.elevator_[0].currentfloor_out == 0) ->
    printf("$atompmId:0.currentfloor_out=$atompmId:4;");
  :: (r.elevator_[0].currentfloor_out == 1) ->
    printf("$atompmId:0.currentfloor_out=$atompmId:5;");
  :: (r.elevator_[0].currentfloor_out == 2) ->
    printf("$atompmId:0.currentfloor_out=$atompmId:6;");
  fi;
  printf("\n");
  timestep = timestep + 1;
}
```

Listing 8. The compiled code for printing the current state.

The trace ends with a line stating how many internal steps are executed by *Spin*. The printed text is directed to a text file, which can be interpreted by a generic parser that implements the grammar of Listing 9. From this trace and the design model, a trace model can be generated (step 4 in Figure 11). This is done by, for each state in the trace, mapping the state to the design model, thus forming a trace language State. The Transitions in between states are collected from the `rule_application` non-terminal.

The trace model can be played out on the run-time model, so that the counterexample is visualised step by step (step 5 in Figure 11). Because the trace model usually has this purpose, it is parsed implicitly to memory instead of constructing a E_t instance.

6 The Elevator *ProMoBox* in Action

We implemented the *ProMoBox* framework in *AToMPM* [83], and the compiler for models to *Promela* and the parser for text to models were written in Python, using *AToMPM* bindings. Figure 27 shows the FTG+PM of the full process of using *ProMoBox* for modelling and verification, including the modelling of the system and property. In accordance with the two DSM phases explained in Section 2.1, this modelling process is typically carried out by the domain user and starts where the process of the language engineer of Figure 10 stops.

The user starts by modelling a system in the design language. The design model can be automatically converted to a run-time model using the generic ToRuntime transformation. This transformation is similar to the migration described in Section 4.4.2, meaning that it only consists of a namespace switch, if necessary. Linguistically, no model elements are changed. In this case however, an initial state still needs to be modelled in the SetInitialState activity. Obligatory run-time attributes (such as the pressed, going_up and doors_open attributes) and run-time associations with a minimum cardinality greater than 0 (such as the currentfloor association) are required in Elevator and Button instances in the run-time language. Consequently, the result of the transformation is a model that is conform to a "relaxed" run-time language (much like the intermediate metamodel described in [59]), *i.e.,* the run-time language modulo constraints on the minimum cardinality of run-time associations. In this relaxed run-time language, the initial state can be modelled in the SetInitialState activity so that the resulting model is a valid run-time model. In case of the elevator example, the currentfloor link should be set and initial values for pressed attributes, going_up and doors_open should be given. The result of the initialisation step in our example is Figure 4. Instead of modelling by hand, default values can be used, if available. For this particular model however, it is not relevant which initial state is chosen, as all states turn out to be reachable from one another. In parallel with modelling the system, a property is modelled in the ModelProperty activity, which results for this example in the property model shown in Figure 21.

The remainder of the process model in Figure 27 explains the

```
output_trace     = ltl "\n" { (system_step | environment_step | property_step)
      "\n"} spinend "\n"
ltl              = "ltl" property_name ":" ltl_string
property_name    = ID
system_step      = step_nr ":" rule_application system_state
step_nr          = NUMBER
rule_application = ID
system_state     = {variable "=" value ";"}
variable         = "$atompmId:" atompmid "." feature
value            = NUMBER
environment_step = "INPUT" system_state
property_step    = regular_step | cycle
regular_step     = "Never claim moves to line" NUMBER "[" ltl_string "]"
cycle            = "<<<<<START OF CYCLE>>>>>"
spinend          = "spin:trail ends after" NUMBER "steps"

ltl_string       = operand | "(" ltl_string ")" | ltl_string binop ltl_string
      | unop ltl_string
operand          = "true" | "false" | proposition
binop            = "U" | "W" | "&&" | "||" | "->"
unop             = "!" | "[]" | "<>"
proposition      = ID

ID               = ("a".."z"|"A".."Z"|"_") {"a".."z"|"A".."Z"|"0".."9"|"_"}
NUMBER           = ("0".."9") | ("1".."9")("0".."9")*
```

Listing 9. The grammar in EBNF of the text file generated by running the *Promela* model with a *trail* file in *Spin* (after step 3 in Figure 11).

five steps of Figure 11 in detail. The property model, the run-time model, and the existing annotated metamodel and operational semantics are translated to a *Promela* model as explained in Section 5 in the Compile2Pml activity. This *Promela* model, containing an LTL formula, is fed to the *Spin* model checker in the VerifyWithSPIN activity. Depending of the type of the property (liveness or safety, as indicated in Table 1), *ProMoBox* automatically instructs *Spin* whether to look for acceptance cycles or not. The report of the verification is stored as a text file. In the HasCounterExample? transformation, the report is automatically analysed to conclude whether a counterexample is found. If none is found, the model checking process finishes and it is concluded that the system satisfies the property. In case of a counterexample, *Spin* produces a `trail` file, containing a counterexample scenario. Subsequently, *Spin* is executed to perform a guided simulation in the PrintTrace transformation, following the scenario described in the `trail` file. In guided simulation, the generated print statements (see Section 5.6) are executed, which results in a textual execution trace. As explained in Section 5.6, the resulting output stream is directed to a text file which is a sequence of system states and transitions, and which follows the format of Listing 9. This text file can be automatically transformed to a trace model by a generic parser with the TransformTrace transformation. Note however that for performance reasons, the result of this transformation is an implicit trace model, as actual trace models tend to be rather large – in the order of magnitude of the number of elements in the design model times the number of states in the trace trace. In order to inspect the counterexample at the level of the DSML, the trace model can be automatically loaded in *AToMPM* as shown in Figure 28, showing the initial state and a toolbar (shown below *AToMPM*'s main toolbar) to step through the counterexample. As shown in this toolbar, a counterexample can be loaded, it can be fully played out, one step can be taken, and a full play-out can be paused or stopped, which is done in the manual PlayTrace activity.

When applying this process to the running example, (maybe unexpectedly) a counterexample for the reachesFloor property is found. When playing out the counterexample as shown in Figure 28, it can be seen that the system might get in an infinite loop consisting of three steps: (1) a button is pressed on the floor where the closed elevator currently is, (2) the elevator opens its doors and unlights the button, and (3) the elevator closes its doors. These three steps can be repeated infinitely many times (which is shown when playing out the counterexample), even if buttons on other floors are pressed, thus resulting in an acceptance cycle representing this fairness problem. Indeed, this way, the requested floor will never be reached, as the system is flooded with other requests.

We checked the property with *Spin* [35] version 6.4.4 on a 64-bit Windows 7 SP1 PC with an Intel(R) Core(TM) i7 Q 720 CPU at 1.60 GHz (up to 2.80 GHz) with 8 GB of 1600 MHz DDR3 memory. As is typical for *ProMoBox* in case of a counterexample [20], [57], [60], the execution time is short. In this case, *Spin* takes less than 1 millisecond to find a counterexample, and uses 5 KB of memory.

Another property is shown in Figure 29 stating that if the elevator doors are closed, they will eventually open again. The system satisfies this property, so this no counterexample is produced for this property. Model checking with *Spin* results in a traversal of the entire state space, and takes 48 milliseconds, using 409 KB of memory.

7 Evaluation of *ProMoBox*

In this section, the *ProMoBox* approach is evaluated, by answering the following research questions:

- *RQ1*: Is the language engineering effort of using *ProMoBox* improved compared to manual methods? The hypothesis is that using *ProMoBox* requires less effort than using existing methods.
- *RQ2*: Does *ProMoBox* improve the quality (*i.e.,* decrease of errors) when using the verification support compared to manual methods? The hypothesis is that *ProMoBox* decreases the number of errors when using verification support.
- *RQ3*: Is there a model checking performance cost in using *ProMoBox* compared to manual methods? The hypothesis is that using *ProMoBox* does not introduce an additional performance cost compared to manual methods.
- *RQ4*: How does the expressiveness of the resulting verification language in *ProMoBox* compare to the specification patterns by Dwyer *et al.*? The hypothesis is that *ProMoBox* is at least as expressive as the specification patterns, excluding the semantically unclear "next" operation (see also Section 8.3).
- *RQ5*: What is the customisability of the *ProMoBox* framework? The hypothesis is that the effort to customise the *ProMoBox* framework is acceptable.

In case of an experiment, the experimental setup is described in detail. When necessary, factors that may jeopardise the validity of our results are discussed. These threats to validity are classified as follows [76], and we briefly repeat their definition:

- *Construct validity*: This aspect of validity reflect to what extent the operational measures that are studied really represent what the researcher have in mind and what is investigated according to the research questions.
- *Internal validity*: This aspect of validity is of concern when causal relations are examined. When the researcher is investigating whether one factor affects an investigated factor there is a risk that the investigated factor is also affected by a third factor. If the researcher is not aware of the third factor and/or does not know to what extent

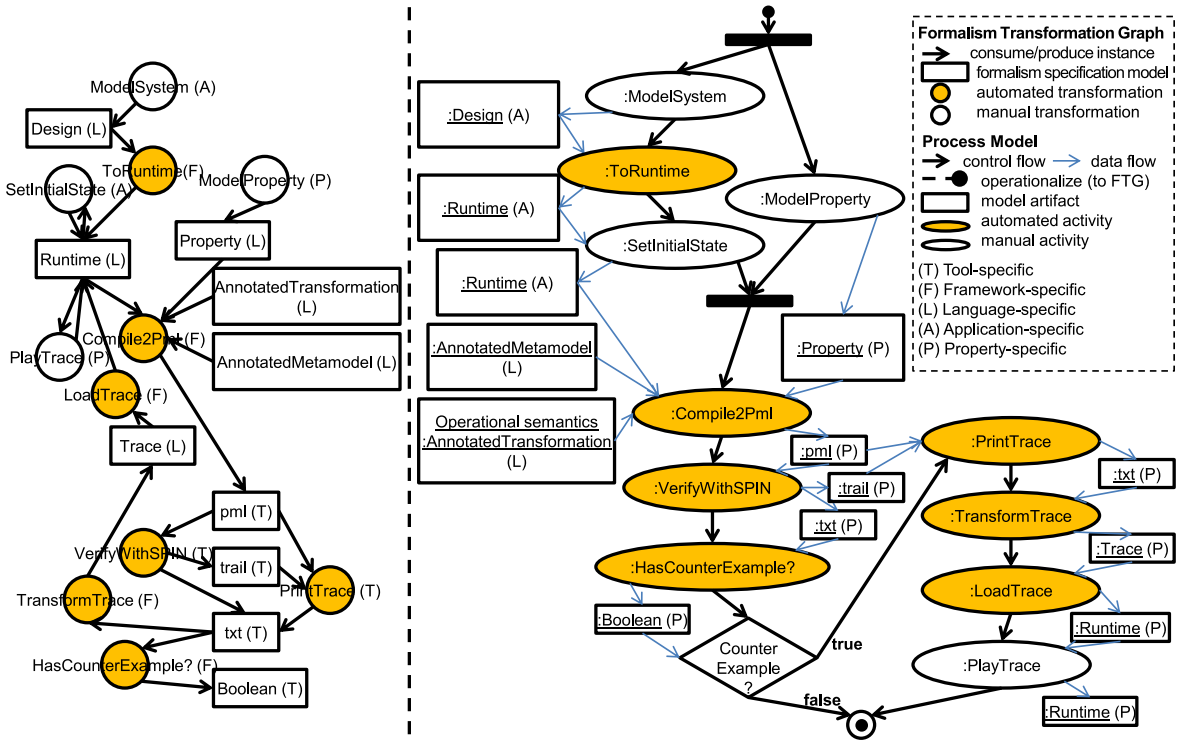

Figure 27. FTG (left) and PM (right) of the domain user's activities using *ProMoBox*.

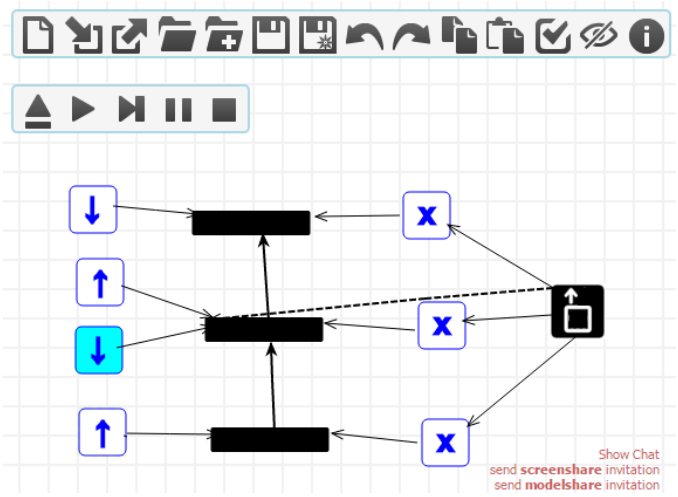

Figure 28. Using the toolbar for playing out the counterexample trace in *AToMPM*.

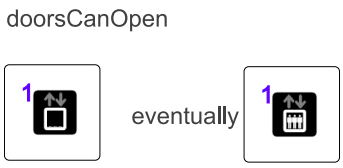

Figure 29. The doorsCanOpen property.

it affects the investigated factor, there is a threat to the internal validity.

- *External validity*: This aspect of validity is concerned with to what extent it is possible to generalise the findings, and to what extent the findings are of interest to other cases outside the investigated case. During analysis of external validity, the researcher tries to analyse to what extent the findings are of relevance for other cases.
- *Reliability*: This aspect is concerned with to what extent the data and the analysis are dependent on the specific researchers or tools. Hypothetically, if another researcher later on conducted the same study, the result should be the same.

7.1 *RQ1* Modelling Effort

This section presents an evaluation to answer the research question *RQ1*: Is the language engineering effort of using *ProMoBox* improved compared to manual methods? We use a logic argument and an experiment to answer *RQ1*.

We will evaluate the amount of manual work for the language engineer. Note that we do not compare effort for the domain user, as we assume that, according to DSM principles, a DSML should be created by the language engineer to lift a domain user task to the domain level. The domain user then only has to limit him- or herself to the bare necessities of the task at hand, using the most suitable language. We will however compare to "incomplete" DSM solutions for verification, to properly assess the effort needed when using *ProMoBox*.

In the literature, we distinguish four approaches related to verification support for DSMLs (see also Section 9):

- ***Approach 0***: no support for properties is available, thus, it is impossible for the domain user to verify properties. This approach serves as the baseline in our comparison, as our approach builds on traditional DSM, where no support for verification is available;
- ***Approach 1***: no DSML for properties is available. Instead, properties are directly modelled in logic, but there is a

	Approach 0	*Approach 1*	*Approach 2*	*Approach 3*	*ProMoBox*
DSML modelling	man.	man.	man.	man.	man.
DSML annotation	n/a	n/a	n/a	n/a	man.
System model mapping to verification backbone	no	man.	man.	man.	aut.
Property language modelling	no	no	man.	man.	gen.
Property mapping to verification backbone	no	no	man.	man.	aut.
Counterexample parsing	no	no	no	man.	aut.

Table 3
Comparison of the degree of automation between existing methods and *ProMoBox*.

mapping to a formal language (*e.g.,* [72], following the architecture shown in Figure 9);

- ***Approach 2***: a DSML for properties is created including a mapping to a verification backbone, but no counterexample parsing is supported (*e.g.,* [40]);
- ***Approach 3***: a DSML for properties is created including a mapping to and counterexample parsing from a verification backbone (*e.g.,* [79]). This is in fact the only approach that offers the same functionality as *ProMoBox*.

The comparison is based on the different tasks that need to be performed by the language engineer, and are extracted from the activities of the FTG+PM models in this paper. Naturally, the language-specific activities (L) have to be taken into account, but also a number of framework-specific activities (F), as they represent language-specific activities for property verification in the absence of *ProMoBox*. The other (tool-specific, application-specific, property-specific) activities are not part of the language engineering phase. We identified the following tasks that reflect FTG+PM activities as follows:

- DSML modelling: represents the traditional DSML modelling activities (*i.e.,* CreateMM, CreateCS and CreateOS in Figure 7 and Figure 10);
- DSML annotation: represents the annotation activity (*i.e.,* AnnotateMM, and partly CreateOS in Figure 7);
- System model mapping to verification backbone: represents the part of the Compile2Pml in Figure 27 activity dealing with mapping the system model;
- Property language modelling: represents the part of the GenerateLanguages activity in Figure 10 (thus replacing one instance of Figure 14) dealing with the creation of a property language;
- Property mapping to verification backbone: represents the part of the Compile2Pml in Figure 27 activity dealing with mapping the property model;
- Counterexample parsing: represents the activities for generating the counterexample and transforming it to the domain-specific level (*i.e.,* the PrintTrace,TransformTrace and LoadTrace in Figure 27).

According to the existing methods we encountered, the framework-specific activities in the FTG+PM models that are not mentioned above are not required for providing verification support.

The comparison between existing methods with *ProMoBox* is shown in Table 3. Rows represent tasks as explained above, and columns represent the DSM approaches. The approaches differ in whether they support, either manually or automatically, the given six language engineering activities. Note that the annotation activity is not applicable (entry "n/a") for existing methods. If the activity is not supported (entry "no"), this lack of support puts an additional burden on the domain user, because the activity is not lifted to the domain level. If the activity is marked "man.", it requires a manual effort from the language engineer resulting in a DSM solution for the domain user. A "gen." marking means that a generative process enables the execution of the activity and "aut." refers to an automated process.

The table shows that language engineering with *ProMoBox* requires the additional effort of annotating the DSML compared to existing methods. In particular, compared to a traditional DSML without support for verification (Approach 0), this is the only additional task. Apart from that, the more advanced the existing method, the more manual activities are required. In order to obtain the same support as *ProMoBox* provides (*i.e.,* Approach 3), five tasks need to be performed manually by the language engineer, compared to automated tasks when using the *ProMoBox* approach.

7.1.1 Experimental Setup

To gain more detailed insight into how much effort each task represents, we provide a quantitative analysis of two case studies. In this experiment, we wish to obtain a quantified measure of effort. We deliberately did not measure effort in terms of time spent, in order to avoid a bias introduced by the skills of the language engineer subjected to the test.

The popular COCOMO approach [39] gives an estimate of effort based on the estimated size (*i.e.,* lines of code) of a project, by defining a linear relationship between effort and size. In [81], it is shown that the COCOMO model can be transposed to model-driven engineering. Considering this, we will use the size of the model as the determining factor for effort. In this experiment, approaches are partly modelled, but may also contain Python code for mappings to and from the verification backbone. Therefore, we make a distinction between model size and code size. The *model size* metric is the sum of the number of model elements (nodes and edges), plus the number of connections between model elements, plus the number of attributes of model elements. The *code size* metric is the number of lines of code.

The COCOMO model only provides a rough estimate of effort. Nevertheless, creating a more complex model may require more modelling effort than creating an equally large, but less complex model. Therefore, we will compare the complexity of different approaches to complement the size metrics. As suggested in [32], McGabe's cyclomatic complexity metric [52] can be applied to DSM. Similar to the size metrics, cyclomatic complexity is determined separately for models and code. Cyclomatic complexity for models is defined as $E - N + P$, where E is the number of connections between model elements (*i.e.,* nodes and edges), N is the number of model elements, and P is the number of connected components in the model. Note that the number of attributes is not included in the model, as this would equally increase E and N, thus resulting in the same metric score. As total cyclomatic complexity of a program is the weighted average of the cyclomatic complexity of every unit, the *model complexity* metric is a weighted average of the cyclomatic complexity of every model. The *code complexity* metric is a weighted average of all code created in a case study. The cyclomatic complexity of Python code is measured with the *radon 1.4.2* Python package.

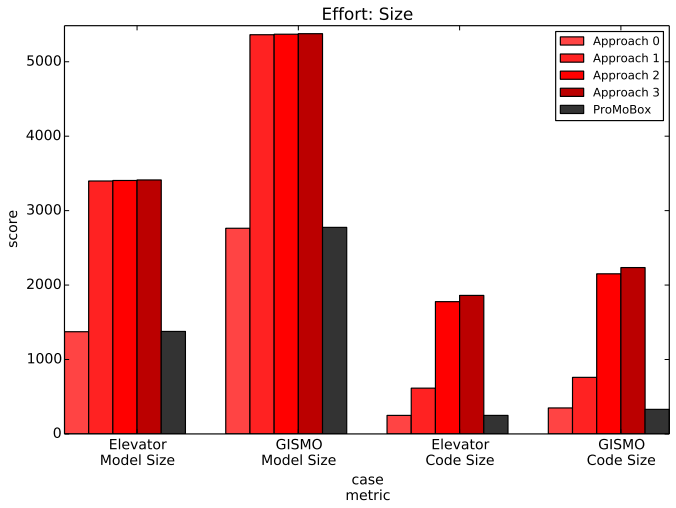

Figure 30. Total language engineering effort comparison of traditional DSM approaches and the *ProMoBox* approach in terms of size (lower scores are better).

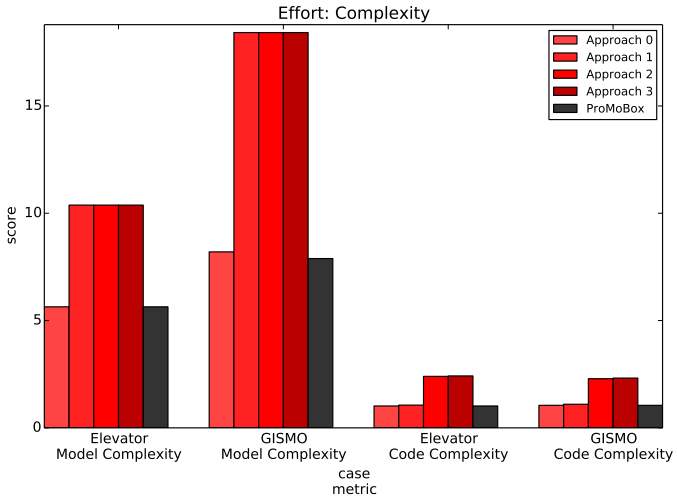

Figure 31. Total language engineering effort comparison of traditional DSM approaches and the *ProMoBox* approach in terms of complexity (lower scores are better).

We analyse two case studies, namely the Elevator and GISMO [20], and obtain metrics for size and complexity. The case studies reflecting the existing methods deliberately result in semantically equal models and transformations as the respective models and transformations in the *ProMoBox* approach. Semantically preserving differences may be present, but the domain user must be presented with syntactically the same language and mappings.

7.1.2 Findings

The results of the analysis of the size metrics are shown in Figure 30. When comparing Approach 0 and the *ProMoBox* approach, the model size scores are highly comparable and code size scores are the same. This indicates that the additional effort for annotating the DSML is minimal compared to the total effort. Additionally, it can be observed that in both case studies and for both metrics, *ProMoBox* performs considerably better than existing methods for verification (Approach 1-3), because of *ProMoBox*'s automation.

The complexity metrics are shown in Figure 31. The cyclomatic complexity when using *ProMoBox* is comparable, if not lower, compared to when using existing methods.

We conclude that the language engineering effort is significantly less when using the *ProMoBox* approach, compared to the effort when using existing approaches that include verification support. Moreover, using *ProMoBox* requires only a relatively small amount of additional work (*i.e.,* the annotation of the DSML definition) compared to traditional DSM which does not include verification support.

7.1.3 Threats to Validity

Construct validity. The metrics size and complexity may not be a suitable representation of effort. We experimented with different metrics:

- time metrics that take Approach 0 as baseline. The time it takes to annotate a DSML definition takes in our experience significantly less time than the time it takes to finish Approach 1 to 3;
- model metrics that count instances of classes plus instances of associations as elements;
- model metrics that do not take into account attributes, meaning that no code at all is taken into account;
- model metrics that do not include measuring concrete syntax, because the concrete syntax models consist of many elements and many attributes to define icons. This means that the language engineer can choose how detailed these icons can be, which nevertheless greatly influences the size metric;
- metrics based on computational complexity, determining the time complexity (*i.e.,* big $\mathcal{O}$ notation) of the mental effort required for each activity. However, such an analysis is in our opinion not sufficiently backed up.

Each of these metrics resulted in the same conclusion: *ProMoBox* requires less effort than existing DSM methods for verification. Consequently, we believe that other metrics will yield similar results, because starting from the Approach 0 baseline, *ProMoBox* only requires the additional effort of annotation (which turns out to be minimal), while existing methods require significant additional modelling and/or coding.

Internal validity. The experiment may be influenced by the fact that semantically similar models and transformations were created using existing approaches. When manually creating verification support, more fine-tuned design choices can be made. Once again, since the additional effort for annotation is relatively very small compared to manually modelling verification support, we are convinced that this will not influence the conclusion.

External validity. The experiment is conducted on two cases only. Their results are very similar, while the cases are sufficiently different, stemming from an entirely different domain. Notably, GISMO [20] has a significantly larger metamodel compared to Elevator, and heavily uses subclassing. Moreover, the number of run-time elements are very limited in GISMO, while its input language is relatively large. For other case studies, we do not expect results that would change our conclusion.

Using other modelling tools for Approach 1-3 may influence the results. In particular, template-based code generation languages like EGL [75] can be used for generating *Promela* code. Again, we do not expect that this would change the outcome of this

evaluation, because of the additional effort required for annotation in the *ProMoBox* approach is very small.

For this comparison, we only consider approaches that are applicable to DSMLs in general, thus excluding modelling languages that are expressive enough to support property specification (*e.g.,* [51]). Such languages may include verification support. If not, only a translation to and from a verification backbone is needed, for both the part of the language for describing properties and the system. That case can be compared to Approach 3: the difference is that system modelling and property modelling are bundled into a single language, but the total effort remains similar. We do not take into account these specific DSMLs in this comparison, as *ProMoBox* focuses on a large class of languages that usually do not have this advantage.

Reliability. The first author of this paper first implemented *ProMoBox*, and then conducted large parts of this experiment. In the GISMO case, the experiment was conducted by a domain expert with expertise in language engineering (*i.e.,* the first author of [20]), except for the mappings to and from Promela, which were implemented by the first author of this paper. The involvement of the first author of this paper may have influenced how models and transformations were implemented using existing methods. The author has attempted to be as objective as possible, and relied for example on an existing *Promela* model for the Elevator case, and on the state pattern for the GISMO case.

7.2 *RQ2* Correctness

This section presents an evaluation to answer the research question *RQ2*: Does *ProMoBox* improve the quality (*i.e.,* decrease of errors) when using the verification support compared to manual methods? We conducted a qualitative study [78] that investigated the ability of domain users to write correct properties using verification support generated by the *ProMoBox* approach. The study allowed us to gain insight in usability of the *ProMoBox* approach.

7.2.1 Experimental Setup

The study involved candidates using the *ProMoBox* approach, followed by an open interview where we assess the experience of the user. We let six candidates write and verify properties for a DSML model representing a Wristwatch Controller[1] using the approaches below:

- *ProMoBox*: A *ProMoBox* was generated from the simplified and annotated DSML definition as presented in Section 4. Users were instructed to specify properties using the generated property language, and had to check these properties and inspect counterexamples that could be played out on a run-time model. This was all done in *AToMPM*, as presented in Section 6;
- LTL, Spin and its GUI *JSpin*: The Wristwatch Controller was translated to Promela. Through the GUI *JSpin*, users were instructed to write LTL properties for the system, check with Spin whether the wristwatch controller satisfies this property, and inspect textual counterexample traces that were generated from print statements. Since the system is represented as a DSML model, this is according to Approach 1 from Section 7.1, which we consider the current standard for verification support in DSM.

1. The Wristwatch Controller DSML is an extension of an exercise that has been developed for a course on modelling and simulation.

The Wristwatch Controller DSML and model were more complex than the illustrative Elevator example in this paper: the Wristwatch Controller metamodel contains 29 elements, and the model used in the case study has over 100 elements. The DSML contains a subset of Statecharts [33], including orthogonal regions, composite states, history, actions and raising events to model a controller, and additional constructs representing an environment consisting of buttons, external events, observable variables and function implementations. The instance model with documentation annotations, as it was presented to the candidates, is shown in Figure 32.

The candidates for the study were selected to represent the target audience for the *ProMoBox* approach, namely domain experts that use DSML models in their domain, but who are not necessarily familiar with verification methods. The candidates were representative domain users, as all of them were acquainted with Statecharts. Moreover, the candidates were familiar with DSM, and *AToMPM*. The candidates did not need to have experience with LTL and Spin, but most of the candidates had at least some experience.

The experiment was carried out as follows:

1) 10-minute introduction about the goal of the experiment;
2) 1.5-hour tutorial on LTL, Spin and *JSpin*;
3) 1-hour tutorial on *ProMoBox*;
4) 20 exercises in randomised order per candidate, for 10 of which the candidate had to define a property using *ProMoBox*, and for 10 using LTL. Then, using the respective framework, the candidate was instructed to verify whether the system satisfies the property, and interpret the counterexample, if any. At the start of the exercises, the candidates were given the model expressed in the DSML and in Promela. Each exercise involved a requirement written in English. Half of the candidates were instructed to solve the first ten exercises with LTL, and the last ten with *ProMoBox*, and the other half of the candidates were instructed to do the opposite, to counter possible bias. Candidates were instructed to skip the exercise if a solution was not found within five minutes. An instructor was present throughout the experiment to assist with technical questions. For each question, candidates filled out a form asking whether they were able to formulate an answer to the question, whether they felt that the verification result conformed to what they expected, and whether they could understand the counterexample (in case of a counterexample);
5) to be able to get an insight about usability, the candidates filled out a form to calculate the system usability scale [10] for *JSpin* and *ProMoBox*. This consists of 10 statements on the topic of usability, for which a score from 1 to 5 must be given depending on the extent to which the candidate agrees to the statement;
6) a one-on-one 15-minute interview was conducted during which the candidates could openly discuss their experience. Each candidate was asked to give an open answer to the following questions:
   - How was your experience with *JSpin*/*ProMoBox*?
   - Did you feel you were able to write correct properties with *JSpin*/*ProMoBox*?
   - What did you think of hiding the concepts of liveness and safety in *ProMoBox*?

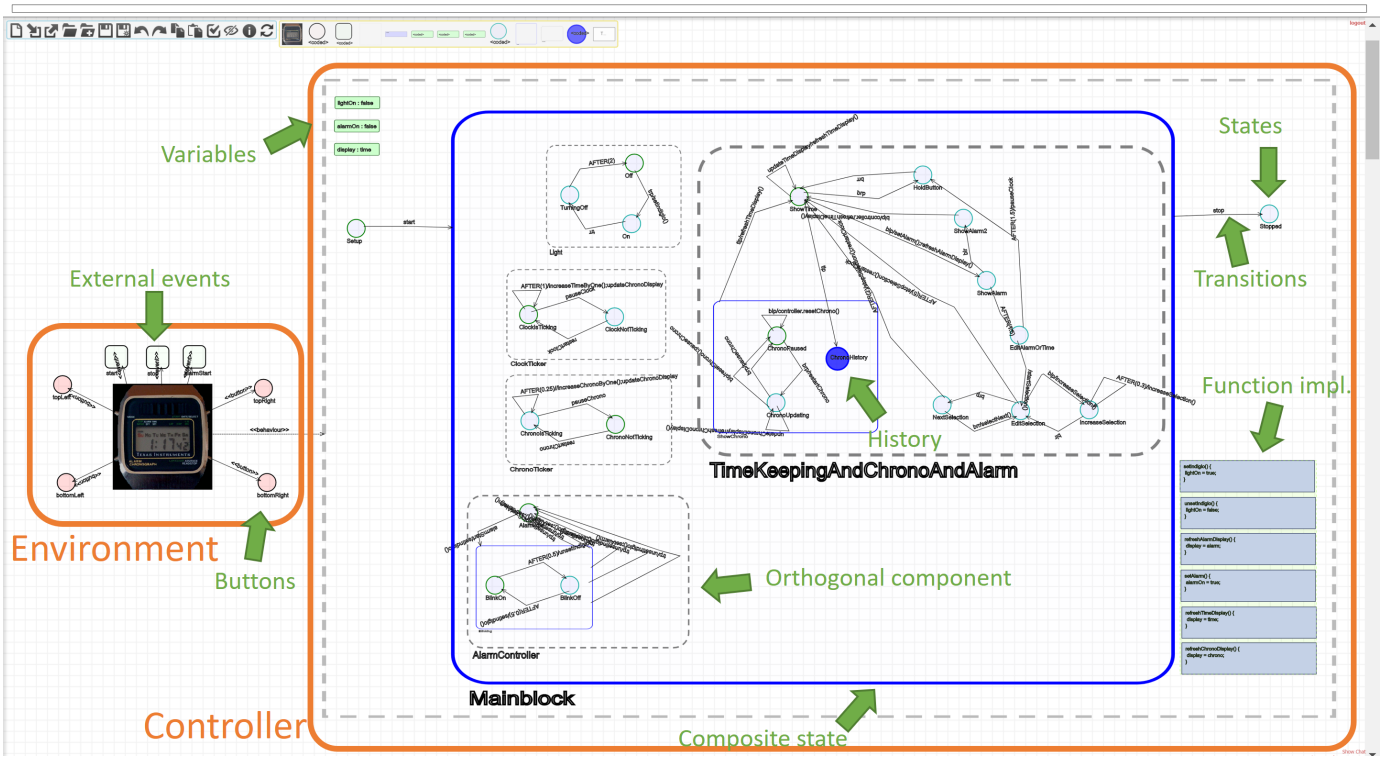

Figure 32. The Wristwatch Controller DSML instance in *AToMPM*, as given to the candidates.

- What is your proficiency prior to the experiment with LTL/Spin/Statecharts/DSM?
- Do you have remarks about the experiment?
- What are possible improvements of *ProMoBox*?
- In the interest of writing correct properties, which would you prefer: *JSpin* or *ProMoBox*?
- In general, which would you prefer: *JSpin* or *ProMoBox*?

In addition to interviewing the candidates, we inspected and scored their solutions. However, as the limited number of candidates in the experiment is insufficient to acquire statistical relevance for the quantified outcome, we refrain from any statistics in our analysis of these scores. Instead, we use these scores to support the candidates' claims.

7.2.2 Findings

Five out of six candidates claimed to have limited or no proficiency in LTL and Spin (one candidate had adequate experience), and all candidates claimed to have good proficiency in Statecharts and DSM. This confirms that the candidates are representative target users of the *ProMoBox* approach, as they are confident in their domain and DSM but are not experts in verification of temporal properties.

During the interview, all candidates claimed to prefer *ProMoBox* over *JSpin*, both in the interest of writing correct properties, and in general. Candidates claimed they felt they were often not able to write correct properties with *JSpin* given their training, and that this was improved a lot with *ProMoBox*.

A long learning curve was a general remark when using *JSpin*. Candidates claimed it was hard to interpret counterexamples from the textual trace of generated *Promela* model, and figure out variable names for propositions from the *Promela* model. The experience with *JSpin* was frustrating for many candidates. Some candidates thought the textual syntax allowed for quick editing of LTL formulas. The candidate that had adequate experience with LTL felt differently about *JSpin*, and felt that the experience was "ok". Nevertheless, the candidate had significantly more correct answers using *ProMoBox* than using *JSpin*, conforming to the general trend among all candidates.

When discussing *ProMoBox*, the candidates indicated that the learning phase is significantly shorter, due to the familiar DSML syntax when writing properties and interpreting counterexamples. This was backed up by the candidate scores: all candidates scored at least as good for their final 5 *ProMoBox* exercises, compared to their first 5 *ProMoBox* exercises. For *JSpin* exercises, scores were similar for the first 5 and last 5 exercises. The use of patterns was also lauded, as well as the ability to write correct structural patterns. Candidates suggested improvements for debugging support, as error messages now appear in a console.

We analysed the correctness scores of the exercises' solutions. All candidates scored better when using *ProMoBox*, compared to *JSpin*. Table 4 shows per exercise how many times a successful answer was provided out of total number of answers in LTL (third column) and for *ProMoBox* (fifth column). For example, 2/3 means that that particular exercise was carried out by 3 candidates, of which 2 times the candidate found a correct solution. Note that the combined total number of answers per exercise is 6, as each exercise needs to be solved by every candidate using either *JSpin* or *ProMoBox*. For only one out of twenty exercises (highlighted in bold), a higher percentage of candidates had a correct answer in LTL compared to *ProMoBox*.

Similarly, from the 10 exercises in each approach (*ProMoBox* or *JSpin*), all candidates was able to express the same number or more properties using *ProMoBox*. Table 4 shows per exercise how

	JSpin		*ProMoBox*	
Exercise	Expressed	Correct	Expressed	Correct
1	2/3	2/3	3/3	2/3
2	2/3	**1/3**	3/3	**0/3**
3	1/2	0/2	4/4	1/4
4	1/2	0/2	4/4	3/4
5	3/3	2/3	3/3	3/3
6	5/5	1/5	1/1	1/1
7	5/5	0/5	1/1	0/1
8	1/1	0/1	5/5	3/5
9	2/3	0/3	3/3	0/3
10	3/3	1/3	3/3	1/3
11	3/4	0/4	2/2	1/2
12	3/4	1/4	2/2	1/2
13	2/3	0/3	3/3	2/3
14	2/2	1/2	4/4	3/4
15	3/4	0/4	2/2	1/2
16	2/2	0/2	4/4	3/4
17	2/2	0/2	4/4	4/4
18	**3/4**	0/4	**1/2**	1/2
19	**3/3**	0/3	**2/3**	1/3
20	0/2	0/2	4/4	1/4
Total	48/60	9/60	58/60	32/60

Table 4
Aggregate results of the study, per question.

many times a property could be expressed out of total number of answers in LTL (second column) and for *ProMoBox* (fourth column). For only two out of twenty exercises (in bold), a higher percentage of candidates was able to express a property in LTL compared to *ProMoBox*.

Interestingly, in both of the exercises that were expressed more easily in LTL, all LTL formulas were incorrect. Moreover, the total number of exercises that could be expressed in *JSpin* (48/60) is significantly different from the total number of correct properties (9/60) compared to *ProMoBox* (58/60 compared to 32/60). This corresponds to the candidates' claim that they have difficulties trusting the LTL formulas they write. When inspecting the errors in the LTL formulas, typical errors are:

- forgetting a temporal operator (globally/finally) before a proposition;
- disregarding a special case, *e.g.,* the case where some condition never occurs;
- using an incorrect structural pattern.

The results of applying the system usability scale were in favour of the *ProMoBox* approach. The *JSpin* approach scored on average 33.3%, which is deemed "unacceptable", while *ProMoBox* scored on average 68.5%, which is deemed "acceptable", with adjective rating "ok" [10]. The results for *ProMoBox* can be split up as follows: four candidates gave a score indicating an adjective rating of "ok" (50%-73%), and two candidates that gave a score indicating an adjective rating of "good" (73%-85%).

We conclude from the study that the candidates felt they were able to write properties in *ProMoBox* more correctly than in *JSpin*. This is backed up by the candidates' scores. Our interpretation is that for the domain user, correctness is improved because all models he/she creates, edits or reads, are at the domain level. The domain user thus reaps the DSM benefits of lowering the chances of error because of the used generative techniques [38], compared to traditional methods where models need to be created or inspected at the formal methods layer.

7.2.3 Threats to Validity

Construct. We did not explicitly provide the candidates with specification patterns and their LTL representation for the *JSpin* exercises. However, at least one candidate looked up the patterns and used them. Nevertheless, the score of the candidate was comparable to the other candidates' scores. While we believe that employing specification patterns improves the correctness of properties, we believe that the most important factor is that the domain user can specify and verify properties at the domain level.

External Validity. As discussed above, the number of candidates is low, as we selected candidates based on their knowledge in DSM and *AToMPM*. Because of the low population, we refrain from using statistics on these scores. Instead, to draw conclusions, we focus on the qualitative aspect of this study, *i.e.,* the interview, and use the scores to illustrate the candidates' conclusions.

The score on the system usability scale is only applicable to the target audience for the *ProMoBox* approach. In other words, this does not mean that the usability of *JSpin* is unacceptable for LTL experts. Instead, the score of *JSpin* illustrates the motivation of our work, *i.e.,* that verification tools are not suitable for domain users that are not LTL experts.

The Wristwatch Controller DSML is used as a single example of DSML, which may introduce bias. Nevertheless, we think the Wristwatch Controller DSML is a suitable case study, for three reasons. Firstly, the candidates are familiar with the DSML and the model, which allows them to act in the experiment as domain experts. Secondly, it is a variant of a model that has been used and tested extensively for several years in a master course on modelling. Thirdly, the Wristwatch Controller is very suitable to devise plausible, intricate temporal properties.

Reliability. We intentionally selected candidates that are familiar with our research tool *AToMPM*, so that there is no bias in the study that is caused by unfamiliarity of *AToMPM*. However, as *AToMPM* is an academic prototype and has a limited user base, this means that the candidates already had limited prior knowledge of *ProMoBox*, through previous presentations on the topic. Nevertheless, the candidates never used the *ProMoBox* approach before. Moreover, the experiment required the candidates to follow training on both approaches, meaning that prior knowledge was required before starting the exercises. We believe that the reliability of the experiment is higher when candidates are selected that know *AToMPM*, than that they have to get acquainted with *AToMPM* during the experiment.

7.3 *RQ3* Model Checking Performance

This section presents an evaluation to answer the research question *RQ3*: Is there a model checking performance cost in using *ProMoBox* compared to manual methods? We use a logic argument, backed up by an experiment to answer *RQ3*.

7.3.1 Analysis of the Generated Promela Model

When considering performance, we are interested in the execution time and memory consumption of model checking in *Spin* for the generated *Promela* model, compared to the manually created *Promela* model encoding the same system. We consider this with the metrics verification time (in seconds), memory usage (in KB), number of states and transitions in the *Spin* state space, and state vector in *Spin*. When analysing the compilation of Section 5, the following may be argued:

- The state vector is minimised according to the annotated metamodel. The compilation is implemented in a way that the variable `__RuntimeSystem` is the minimal and only variable represented in the state vector (see Section 5.1). All other variables are `hidden`, *i.e.,* not part of the state. This assumes that the correct annotations are set in the annotated metamodel.
- The number of states and transitions depends on (1) what information is represented in the state vector, (2), the LTL formula that is verified and (3) the algorithm in the *Promela* model.
  The LTL formula is generated according to the verification patterns by Dwyer *et al.* [21], and is therefore minimally represented as shown in Table 1. Quantification patterns are translated to multiple LTL formulas. A manually created LTL formula can therefore not be less complex than an LTL formula which is generated from a property model.
  The number of states and transitions is determined by the number of times control structure (*i.e.,* conditions and loops) are traversed when executing the algorithm. The algorithm mostly depends on the code generated from the operational semantics. Other code snippets have significantly less impact, as they are evaluated only once in the beginning of the execution (*i.e.,* declaration of the metamodel and initialisation of the model on line 1-3 and 7-8 of Listing 1, Listing 2, and Listing 3), or only include maximally one `d_step` per iteration (*i.e.,* (a) the compiled environment model on line 4 of Listing 1, Listing 4, (b) the compiled code for printing the current state on line 16 of Listing 1, Listing 8, and (c) the compiled proposition patterns on line 17 of Listing 1, Listing 7). Consequently, the operational semantics in *Promela* have the highest impact on the model checking time.
  We argue that the operational semantics in *Promela* is in the same time complexity as the manually created semantics. The compiled rule schedule is a one-to-one translation of the rule schedule. Using this rule schedule, similar control structures as in *Promela* (*i.e.,* conditions and loops) can be modelled. Therefore, the rule schedule will be in the same time complexity.
  The evaluation of the rules' LHS heavily influences the algorithm of the *Promela* model as the underlying subgraph matching algorithm is computationally expensive. One could correctly argue that, when manually modelling the similar problem in *Promela*, one would not make use of such an expensive subgraph matching algorithm, but would rather use simple conditional statements. However, we encode the LHS in such a way that conditions are folded into a minimally nested control structure (see Listing 6). We argue that this reflects the minimal control structure, as one would model manually in *Promela*. Note that conditions tend to become quite long, but this does not affect the number of states and transitions.
  Consequently, because of the optimisations in *ProMoBox*, the generated algorithm is in the same time complexity as a manual algorithm.
- The memory usage of *Spin* is the number of states multiplied by the state vector size, and is thus a derived metric. Hence, it is in the same time complexity as a manual algorithm.
- The verification time depends on the number of states/transitions and the time per transition. This time per transition is influenced by the number of statements/expressions per deterministic step of *Promela* code. This has, compared to the number of states/transitions, a minor influence on overall verification time, because of *Spin*'s optimisations such as lazy evaluation. Consequently, the verification time of a generated *Promela* model should be comparable to a manual *Promela* model.

Overall, we conclude that compared to manual modelling in *Promela*, *ProMoBox*'s mapping to *Promela* does not introduce additional complexity in the code that changes its time complexity, meaning that model checking performance is comparable.

7.3.2 Experimental Setup

We reinforce the above claims by an experiment. In this experiment, we compare the model checking performance between *Promela* models that are generated by *ProMoBox* for the Elevator DSML and a manually constructed model. The generated code for the operational semantics is the most complex, and influences performance most. Therefore, we are especially interested in whether the *Promela* model of the operational semantics is indeed in the same time complexity as a manual *Promela* model (as reasoned above), depending on the size of the run-time model. We argue that the Elevator case is a good candidate for this comparison, as it extensively uses the features of a DSML specification, such as inheritance, associations, attributes, different multiplicities, different annotations, LHSs, RHSs, NACs, condition and action code, non-trivial property propositions, multiple input stars, different attribute types, non-empty environment and trace language, and patterns with multiple match candidates.

We perform this experiment only for the Elevator DSML, because for that DSML we can compare our results to a manually created model that is a variant of a similar model presented in Merz and Navet's "on the verification of real-time systems" [54]. The model presented in the book is simpler than our case study. Consequently we adapted it so that it behaves the same as our rule-based variant. Since it is derived from a model presented by experts in the domain of verification, this model is representative for a model built by an average *Promela* user. The model is shown in Appendix A and corresponds to the running example, which has three floors and seven buttons. In this case however, no buttons are pressed in the initial state, but a simple environment is modelled where buttons can be pressed. We use a meaningless LTL formula $\Box(True)$ that forces a full state space traversal so that we can focus on the operational semantics.

The *ProMoBox Promela* model used as running example throughout this paper, has some slight differences compared to the compiled Elevator case of Section 2.3:

- we use the more optimal rule schedule where the change direction rules are evaluated before the move rules, so that the NAC in the change direction rules can be removed as mentioned in Section 2.3;
- similar to the manual *Promela* model, no buttons are pressed in the initial state;
- we use the same LTL formula $\Box(True)$, which ensures that the entire state space will be traversed.

Variants of the model in Appendix A and of the *ProMoBox* model are used that have different numbers of floors and buttons. For every number of floors, we use at least as many buttons as

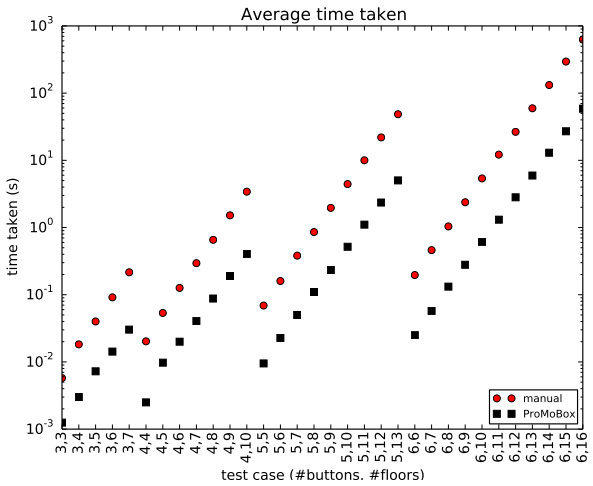

Figure 33. Comparison of the state space traversal time.

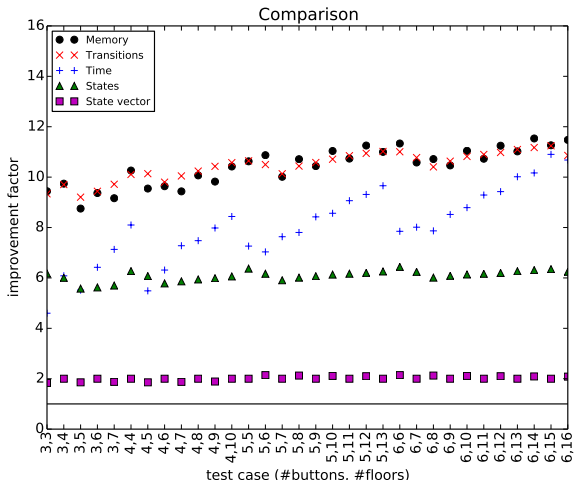

Figure 35. Improvement factor of the traversal time and memory usage.

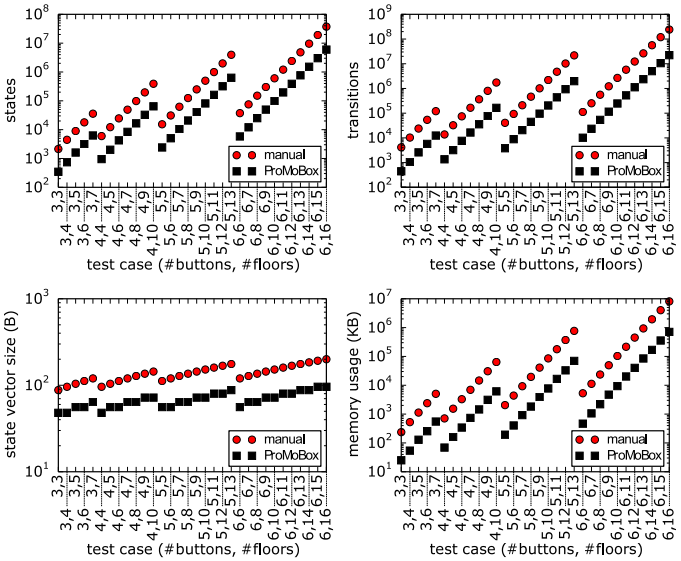

Figure 34. Comparison of the state space traversal memory usage.

the number of floors, so that every floor can be requested, and at most as many buttons as needed to have one elevator button, up button and down button for each floor, with no down button for the ground floor and no up button for the top floor. In the variants, buttons are equally distributed over floors. We cover all variants up to the number of floors that causes *Spin* to run out of memory.

We used *Spin* [35] version 6.4.4 on a 64-bit Windows 7 SP1 PC with an Intel(R) Core(TM) i7 Q 720 CPU at 1.60 GHz (up to 2.80 GHz) with 8 GB of 1600 MHz DDR3 memory. To optimally address the research question, we only inspect model checking time and memory consumption as reported by *Spin*, and do not take into account compilation time by *ProMoBox*. This is because model checking is the first and foremost limiting factor in usability, as explained in Section 2.6.

7.3.3 Findings

The results are shown in Figure 33 and Figure 34. Both plots use a logarithmic scale on the vertical axis. All test models are shown on the horizontal axis, indexed by two numbers: the number of floors and the number of buttons. We verified each *Promela* model 50 times, and excluded the 5 highest and 5 lowest outliers. The averages of the execution times without outliers is shown in Figure 33. Figure 34 shows memory-related resources. The number of states and number of transitions reflect the size of the state space. The state vector size is shown and the memory usage for the state space is shown (excluding memory used for optimisations, which is determined by *Spin* run-time parameters). Note how the required resources grow exponentially with the size of the environment (*i.e.,* the number of buttons) as discussed in Section 4.4.1. This behaviour can be seen in both the generated, and the manual *Promela* models. In this experiment, the generated *Promela* models need fewer resources for all test cases.

Figure 35 shows the improvement factor of all resources for every test case. The plot indicates no decline in improvement for larger models. Interestingly, the improvement (especially in time but also in memory) increases for the more complex cases that have a larger number of buttons.

The results indicate that the generated *ProMoBox Promela* models are in the same time complexity and space complexity as the manually created models. The generated *ProMoBox Promela* models perform better in this experiment. This supports the argumentation above, and indicates that *ProMoBox* comes with no additional cost in model checking performance.

7.3.4 Threats to Validity

Internal validity. We are quite confident that the compiler is correct, as the Elevator case covers most linguistic features as explained above. Nevertheless, we cannot prove that the compiler contains no errors. To address this, we have extensively tested the compiler, resulting in branch coverage of over 90%.

Construct validity. The manually created *Promela* model may not be representative for an average *Promela* user and may influence the model checking results of the manual *Promela* model. We used the model from a book [54] explaining how to properly use *Promela*, where model checking performance is also taken into account. We continued on this model to make minor adaptations on the manual model, to make it comparable to models in our DSML. Other factors may play however, like slightly different semantics, which may influence modelling decisions of a *Promela* user.

Reliability. As we have performed the experiment on a Windows machine, results concerning execution time may be

influence by OS behaviour. To address this, we performed every experiment 50 times, and excluded outliers.

External validity. In the experiment, we only investigated a single DSML, on the one hand because we have a manual model for comparison for the Elevator case. On the other hand, we feel that further evaluation on the performance of generated *Promela* models diverges from the contribution of this paper for two reasons. First, the mapping to *Promela* and LTL is only one example of a possible verification backbone for *ProMoBox*. In this respect, a mapping to *Groove* and CTL is shown in Section 7.5.1. Second, optimising the generated *Promela* models requires profound knowledge of *Promela* and *Spin* rather than of the DSML and is thus beyond the scope of this DSM-centred contribution. Nevertheless, we argue that the Elevator case is representative and covers most of *ProMoBox*'s features, as explained in Section 7.3.2.

Furthermore, we have not experimented with different properties, as we believe that sufficient proof of an optimal mapping from temporal patterns to LTL is given in [21]. Moreover, the techniques used in the manual *Promela* model may not be generalisable to other domains. For example, the behaviour in another domain may not be optimally representable using a rule-based approach as in the *ProMoBox* approach, but may be very suitable for representation as a *Promela* model. Nevertheless, DSMLs with rule-based semantics represent a significant class of DSMLs. Finally, we did not include compilation time by *ProMoBox* in our evaluation, as we reason that model checking time will generally be the limiting factor.

Despite these threats to validity, we feel that the experiment illustrates the logic argument of this section.

7.4 *RQ4* Expressiveness

This section presents an evaluation to answer the research question *RQ4*: How does the expressiveness of the resulting verification language in *ProMoBox* compare to the specification patterns by Dwyer *et al.*? When introducing a new language, it is important to assess its expressiveness. To this end, we evaluate the expressiveness of any generated properties DSML by evaluating the template of the properties language. This is done by evaluating whether all of the specification patterns by Dwyer *et al.* can be expressed using the *ProMoBox* approach.

The specification patterns by Dwyer *et al.* can be defined in terms of LTL [21]. To be able to technically able to express all specification patterns excluding the semantically ambiguous "next" operation (see also Section 8.3), we included a functionally complete set of LTL operators (except "next") in *ProMoBox*.

Nevertheless, it is the intent of the approach to allow the user to specify properties using temporal patterns. In the remainder of this section, we will discuss which (variants of) specification patterns by Dwyer *et al.* are supported in the template of the properties language as it is presented in this paper.

The template of the properties language supports six patterns: universality, existence, absence, bounded existence, response and precedence. These can be combined with five scopes: globally, before, after, between, and after until. Nevertheless, next to these patterns, Dwyer *et al.* identified several variations of these patterns in [22]:

- chained response and precedence. For example, a 2-3 chained response means that the occurrence of two stimuli in the presented order must result in the occurrence of three responses in the presented order. Any multiplicities can be applied, and a regular response/precedence can be considered a 1-1 chained response/precedence;
- upper bound of the bounded existence pattern;
- nested specification patterns. In [22] an example is given for CTL, namely "infinitely often P", translated to AG(AF(P)), as a universal pattern instantiated with parameter AF(P). This substitution is only valid for some scopes or patterns, and no further analysis is made in [22];
- scope boundaries. Scopes are defined as closed on the left and open on the right. Variations can be defined that close the left end of the scope and/or open the right;
- variants of chain patterns. Chain patterns can use absence instead of existence between chains;
- constrained response/precedence: absence between stimulus and response; analogue for precedence;
- next response. The response must be immediately next. We will not take this variant into account, because the semantics of "immediately next" are not clear, as explained in Section 8.3.

The number of possible combinations of these patterns is enormous: open vs. closed, constrained vs. not constrained, chained vs. not chained, number of links in chains, etc. Implementing all possible variants is infeasible, and was never the intention of the pattern system in [21]. For instance, the website by the same authors dedicated to the specifications patterns[2] lists the main patterns and only some of the variants. According to Dwyer, the intention of the specification patterns is that users would customise patterns depending on their domain. Therefore, we include customisation support in our approach, by allowing users to:

1) specify new patterns by changing the template for generating properties languages,
2) specify the translation of newly defined patterns by using LTL compilation rule system in Section 5.5.

This way, all variations above can be implemented, and the correct mapping to LTL can be specified. To have a representative starting point, we implemented constrained response.

In summary, the main patterns are implemented in *ProMoBox*, support for implementation of variants is provided by *ProMoBox*, and, if desired, LTL formulas can be used.

7.5 *RQ5* Customisability

This section presents an evaluation to answer the research question *RQ5*: What is the customisability of the *ProMoBox* framework? We use an experiment to answer *RQ5*. Two customisation scenarios can be distinguished. An experiment is conducted for each of these scenarios (representing *UC4*).

Customisation of the property language *i.e.,* the metamodel template for properties. We do not consider the customisation of templates of other sublanguages, as this sublanguages exist to support the property sublanguage. However, customisation of other templates results in a similar scenario.

Customisation of the verification backbone. A change in the verification backbone may result in a change of all mappings to, and from, the verification backbone.

Given that the framework is intended for long term use by an organisation, we would expect that acceptable customisation effort for a trained customiser should be on the order of a few person days of effort rather than tens or hundreds of person days of effort.

2. http://patterns.projects.cs.ksu.edu/

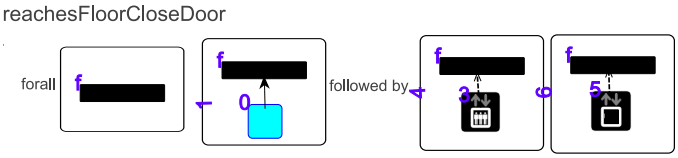

Figure 36. The variant of the reachesFloor property: after a button is pressed, the elevator will eventually open its doors at the corresponding floor, and then close them again.

7.5.1 Experimental Setup

For both experiments, changes were made to the current implementation of *ProMoBox* in *AToMPM*. We are interested in the effort for customisation. The metric for customisability we use in this experiment is amount of development time (in hours) spent applying the customisation task to *ProMoBox*. The changes were made by the first author of this paper. The developer was undisturbed during the experiment.

Customisation of the property language. The first experiment involves a customisation of the property language, by changing the metamodel template for properties in Figure 14 (as shown in Figure 20). As indicated by the (F) mark of the template, this is in fact a change of the *ProMoBox* framework itself, instead of a change of its input models. A consequence of the customisation is that the concrete syntax template might have to be updated and that the property language has to be regenerated. Also, the co-evolution rules apply to its instances, possibly requiring the full verification process as described above to be redone. In this case, since the template has changed, the Compile2Pml activity needs to be adapted as well, so that the changed language is adequately compiled.

The experiment consists of adding additional patterns called "chained response" and "chained precedence" [22] to the property language template. These are similar to "response" and "precedence" but have more than one cause and/or effect that need to happen in a prescribed order.

Customisation of the verification backbone. In the second experiment, the entire verification backbone is replaced. We replaced the mapping to *Promela* and LTL by a mapping to *Groove* and CTL. Since models in *Groove* are graphs and transformations are rule-based just like in our view on DSM, the mapping is more straightforward than to *Promela*. The downside is that *Groove* is not as expressive as *Promela* (notably the lack of NAC patterns and a limited scheduling language) and the performance of model checking is significantly lower. Regarding the mapping to CTL, similar to the LTL formulas presented in Table 1, Dwyer *et al.* [21] also provide a mapping to CTL for every temporal pattern.

7.5.2 Findings

Customisation of the property language. Implementing the additional patterns in the template and adapting the compiler took less than two hours, and no existing property models had to be changed, since the customisation can be classified as an additive, non-breaking change.

Using the new patterns we were able to extend the reachesFloor property by adding a second restriction that the doors should close again after they were opened. This new property is shown in Figure 36.

The customisation of the following models has a similar impact:

- the impact of changing the template of another sublanguage is similar. In particular, changing the trace language template differs slightly as it requires the parser Transform-Trace to be changed instead of the compiler Compile2Pml;
- the impact of changing the concrete syntax of a template is fully automated. It only requires the regeneration of the concrete syntax of the sublanguages;
- the impact of changing the annotations model of Figure 12 is fully automated with respect to the language engineer. It requires all sublanguages to be regenerated. For the sublanguage instances co-evolution rules apply, so depending on the change the instance might have to be migrated and the verification process might have to be redone.

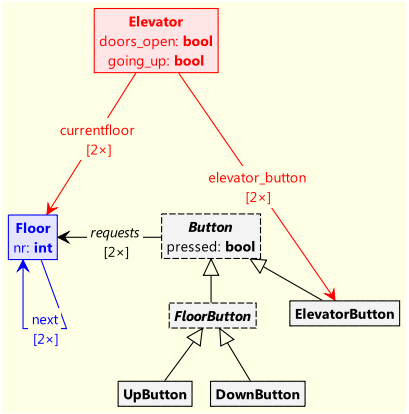

Figure 37. The *Groove* type graph generated from the metamodel.

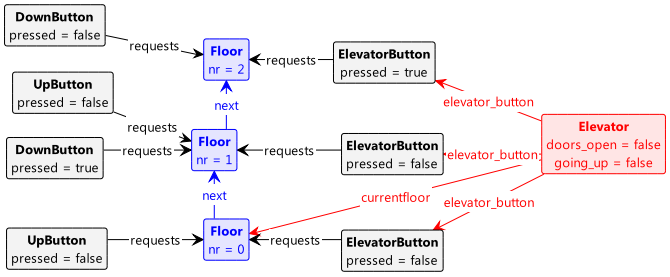

Figure 38. The *Groove* instance graph generated from the run-time model.

This experiment indicates that the effort for customising the property language is acceptable.

Customisation of the verification backbone. The mapping to *Groove* took 32 hours during four days to implement, including getting familiar with *Groove*.

A similar compilation strategy is followed as shown in Figure 24, where AST_{PS} is now *Groove*-specific, and the target is divided in multiple different models implementing *Groove* type graphs, instances, rules, environment, property propositions and traces (as XML-files), and a rule schedule and CTL formula (in textual syntax):

- The run-time metamodel of Figure 16 is translated to a *Groove* type graph as shown in Figure 37. Colours are added for clarity, and the multiplicities are encoded in the edges (not visible).
- The *Groove* equivalent of the instance depicted in Figure 4 is shown in Figure 38. A generic concrete syntax is used, with rectangles, arrows with labels, and expressions.

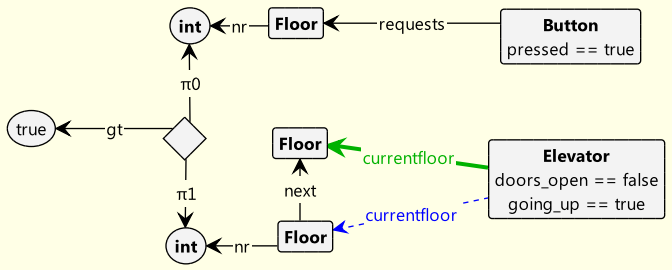

Figure 39. The *Groove* rule for MoveUp.

```
1  while(true) {
2    try {
3      openDoor_up_UpButton | openDoor_up_ElevatorButton;
4      Environment | nothing;
5    } else {
6      try {
7        openDoor_down_DownButton | openDoor_down_ElevatorButton;
8        Environment | nothing;
9      } else {
10       try {
11         closeDoor;
12         Environment | nothing;
13       } else {
14         try {
15           moveUp;
16           Environment | nothing;
17         } else {
18           try {
19             moveUp_last;
20             Environment | nothing;
21           } else {
22             try {
23               moveDown;
24               Environment | nothing;
25             } else {
26               try {
27                 moveDown_last;
28                 Environment | nothing;
29               } else {
30                 try {
31                   changeToDown;
32                 } else {
33                   try {
34                     changeToUp;
35                   } else {
36                     Environment;
37                   }
38                 }
39               }
40             }
41           }
42         }
43       }
44     }
45   }
46 }
```

Listing 10. The rule schedule in *Groove*.

- The translation of the MoveUp rule (see Figure 5) is shown in Figure 39. The pattern includes the LHS and RHS, where all pattern conditions represent the LHS, and all statements, creations and deletions represent the RHS and are applied if a match is found. In this case, the RHS consists of deleting a currentfloor link (dashed) and creating a new one on the floor above (bold). This notation allows for concise rules. The left part of the rule visualises a simple comparison operation representing the LHS condition.
- The complete rule schedule is shown in Listing 10. This schedule is generated from the more optimised schedule that first evaluates the move rules, followed by the change direction rules. The main control structure of the schedule is an infinite while loop, representing the simulation loop. The rules are tried in the correct order. If successful, the environment is evaluated if necessary, and a new iteration of the main while loop is started. If unsuccessful, the next rule is tried. Note that, since *Groove* does not support conditions on pattern element types, the openDoor_up and openDoor_down rules have to be split into two rules: one matching a directional button and one matching an elevator button.
- The environment distilled from the annotated metamodel is shown on the left of Figure 40, and presses a button that is not yet pressed. Additional to this rule, an empty rule nothing is created to give *Groove* the option to not perform an environment step.
- According to the specification patterns by Dwyer *et al.* [21] and taking quantification into account as presented in Section 5.5, the property is translated to the CTL formula:
  `A[!Q0 W (Q0 & AF(P0))] & A[!Q1 W (Q1 & AF(P1))] & A[!Q2 W (Q2 & AF(P2))].`
- The propositions are translated to *conditions* in *Groove*, which are rules that do not have a side effect. *Groove* automatically evaluates after every state change whether or not a match for these conditions can be found. The condition `P0` is shown on the right in Figure 40.
- In case a property yields a counterexample, it is translated back to the domain-specific level. In the case of *Groove*, a counterexample is shown as a highlighted trace in the (partial) state space. The sequence of states can be saved, and each state has the same format as the *Groove* equivalent of a run-time model (*e.g.,* as shown in Figure 38). Consequently, these states are parsed and can be translated to an output model or can be played out in *AToMPM*.

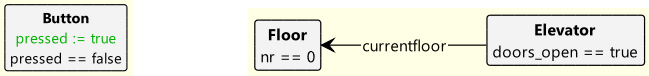

Figure 40. The *Groove* environment generated from the annotated metamodel (left) and the *Groove* proposition rule for $P0$, the elevator opens its doors at the ground floor (right).

This experiment is an indication that the effort to customise the verification backbone is acceptable.

7.5.3 Threats to Validity

Construct validity. The experimental results only include development time. However, time may have to be allocated to testing. As the amount of time for testing highly depends on the desired confidence the developer wants to reach, this is not included in this experiment. Furthermore, we used no other metric than time spent to measure flexibility. Also, other metrics may be deemed relevant, such as metrics that measure maintainability of the resulting system.

Reliability. The experiment was carried out by the first author of the paper, who is also the developer of *ProMoBox*. This highly influences the resulting time spent. Nevertheless, we reason that this benefits the experiment, as the experiment is not influenced by a lack of knowledge for the involved tools and languages (*e.g., AToMPM*, Python, verification patterns). Time spent on getting acquainted with these tools and languages would bias the results, as we do not wish to evaluate the user friendliness of

the involved technologies. In this sense, using a developer who knows the involved technologies improves the internal validity of the experiment. Note that time was spent during the experiment for getting acquainted with *Groove*.

External validity. We only conducted two experiments for assessing customisability. In certain cases, extensions may be more difficult. For instance, we only added temporal patterns in the template of the property language. One might wish to add *e.g.,* real time to the property language. We consider such changes too intricate to be called a customisation, because they would require not only the property language to change, but also the verification backbone, the input and trace language, and even the transformation language for operational semantics. However, this is interesting for future work. Furthermore, no other sublanguage templates were adapted, because we reason that the sublanguages have a supporting role towards the property language. However, a very intricate change may cause these templates to be changed as well. Again, we consider such changes out of scope of this evaluation. The time it takes to change the verification backbone similarly depends on the degree of difference between the *ProMoBox* framework (which relies on typed, attributed, directed graphs and rule-based semantics) and the verification backbone. In case of *Groove*, this cognitive distance is low, especially compared to *Promela*. Nevertheless, while the time spent on customisations can vary, we feel that the experiments illustrate that the *ProMoBox* framework can indeed be customised well to support other variations of properties or verification backbones.

8 Scope and Limitations of the Approach

In this section we discuss the assumptions and limitations of the approach.

8.1 Format of the DSML

It is assumed that we can express the abstract syntax of the DSML as a metamodel in the form of a class diagram. The concrete syntax is defined graphically by icons for every abstract syntax concept. The semantics is given by a transformation model with a rule schedule supporting control flow and graph transformation rules. Under these conditions, an operational *ProMoBox* can be generated from any DSML.

8.2 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 explicit state model checking, the number of possible states must be bounded. This is guaranteed by limiting the multiplicity of the run-time elements. If such boundedness is not achieved in the metamodel because of an infinite multiplicity value, this value must be bounded (possibly through abstraction) in order to allow model checking. Likewise, other simplification steps might be performed to ensure boundedness, as explained in Section 4.4.1.

8.3 Format of the Properties

We currently support temporal properties with quantification and structural patterns. The properties can be mapped to LTL and CTL, so the approach can be considered representative for a wide range of properties. *ProMoBox* does not include support for the "next"-operator (*i.e.,* something must happen immediately after something else), as "immediately" can be interpreted in many ways. A promising direction is to use the principle of the conceptual time step (see Section 4.3) and define "immediately" as "in the next conceptual time step".

Because of *ProMoBox*'s customisability, we feel that the approach described in this paper can be reused for different kinds of properties by defining generic mappers to tools supporting model checking with *e.g.,* OCL, 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 updated throughout the evaluation of the temporal formula (context-dependency), etc. The key to automation of the approach is that it is defined at the meta-level (class diagrams, concrete syntax definitions, and rule-based transformation with scheduling), in combination with predefined, generic templates.

We use a combination of natural language and patterns as concrete syntax for properties. Although very expressive, simple properties can be cumbersome to model and can still be confusing, which was the very problem *ProMoBox* tried to solve. The contribution of *ProMoBox* is however to provide means to ease the specification and verification of properties. We showed that because of the template-based approach the modeller can easily change the concrete syntax, if for example, a user prefers a visual syntax for patterns. Also, syntactic sugar may be added to easily express and visualise features that occur often. For example, the pattern structure might be bypassed if it consists of a single element. In that respect, and because a domain-specific concrete syntax is used, we feel that the *ProMoBox* framework improves the understandability of the properties.

8.4 Scalability

As scalability limitations are inherent to model checking, it remains a main concern. The compiler generating *Promela* could be further optimised. On the one hand, further generic optimisations can be applied to the compiler by a cooperation between a DSM and *Promela* expert. On the other hand, the compiler might be extended to take user-defined optimisation information into account. For example, since pattern matching is the bottleneck of rule-based transformation, search plans [85] can be incorporated in the approach, to allow an optimal matching order of pattern elements.

Another way to contain the scalability issue is to extend and quantify the modelling guidelines, so that a prediction of the model checking time and memory consumption can be given. To this end, extensive empirical research is needed to quantify the relationship between model characteristics and model checking performance. This relationship is different for every verification backbone.

A radically different solution to the problem of scalability would be to not 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 trace models (oracles). A first step in this direction has been taken in [56], where the *ProMoBox* framework is extended to the generation of a domain-specific testing language, including execution semantics. This illustrates how *ProMoBox* benefits from its flexible modelling approach, because mappings to different semantic domains can be implemented. However, this research direction is not yet investigated for the *ProMoBox* approach.

9 Related Work

With respect to the contribution of this paper, we distinguish two types 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.

9.1 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.* [13] have proposed to verify component-based systems by using scenarios specified as Message Sequence Charts (MSCs). Li *et al.* [47] also apply MSCs for specifying scenarios for verifying concurrent systems. The CHARMY approach [67] 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 [11], [41], [42]. These mentioned approaches are just a few examples that aim at specifying temporal properties for models and verifying them by model checkers (see [27] for a survey). They have in common that they offer language-specific property languages. However, these approaches are not aiming to support language engineers in the task of building domain-specific property languages.

TimeLine [79] is specifically designed to address the design of temporal properties in a visual manner. Analogous to LTL formulas, these TimeLine properties can be transformed to *Promela never claim*s (*i.e.,* Büchi automata), which can be directly used for model checking by *Spin*. TimeLine was used to convert informal requirements written in English to formal requirements that are still very readable. The problem that TimeLine addresses is highly related to the problem statement of this paper, but in our approach we generalised this approach for DSM.

Compared to our previous work [57], we have put a great amount of effort in improving the compiler to Promela. Table 5 shows the results of the experiment presented in [57] compared to the results in this paper. Both experiments represent the running example of Figure 4, executed with the LTL formula $\Box(True)$. Although only one experiment could be validly compared, it illustrates the compiler's improvement. The dramatic improvement can be attributed the more optimal encoding of the state vector, and the limitation in decisions (*i.e.,* loops and conditions), which limits the number of bounds as discussed in Section 7.3.

9.2 Generic Solutions

This work is based on the specification patterns of Dwyer *et al.* [21]. The patters are evaluated in [22], and out of 555 specifications collected, 511 (92%) matched one of the patterns. The work in [17] continues the work, and introduces a textual language that includes structural patterns and quantification, much like the framework presented here. However, the presented language is limited to Java only.

In [84], Varró presents an approach in which a metamodel with operational semantics and an instance model can be transformed to a transition system. Transition systems are used as the *de facto* interchange standard of the Symbolic Analysis Laboratory [6], a framework for combining tools for formal methods. Safety properties and deadlock can be analysed. The approach also makes the distinction between static and dynamic language concepts for reducing the state space.

Rivera *et al.* map models and their operational semantics of DSMLs to rewriting logic [74], as well as metamodels [73]. Maude [15] can verify properties using rewriting logic, so operational semantics can be subjected to analyzing methods provided out-of-the-box of Maude environments such as reachability analysis and checking of temporal properties specified in LTL. The approach maps rules to rewriting logic (which is in essence rule-based), while our mapping to *Promela* supports a broader platform for rule-based semantics.

There are some approaches that aim to shift the specification and verification tasks to the model level in a more generalised manner. First of all, there are approaches that propose OCL extensions, often referred to Temporal OCL (TOCL), for defining temporal properties on models [8], [37], [89]. As OCL may be combined with any modelling language, TOCL can be seen as a generic model-based property language as well. In [16], [87], [88] the authors discuss and apply a pattern to extend modelling languages with events, traces, and further run-time 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. Contrary to our approach, a transformation for the latter has to be built by hand. Furthermore, instead of using patterns as propositions like we propose, helper functions have to be written (requiring knowledge of OCL), while temporal properties in TOCL use these helper functions. 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. Interestingly, Combemale *et al.* [16] argue that executable DSMLs require languages similar to our design, run-time, input and trace languages. However, in their approach, the different metamodels must be specified explicitly by the language engineer, rather than using a generative approach that requires minimal annotation of the metamodel.

Another approach that aims to define properties on the model level in a generic way is presented by Klein and Giese [40]. The authors extend a language for defining structural patterns based on Story Diagrams [25] 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 users, because their language allows 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 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.* [21] are encoded in their

	time taken (s)	*states*	*transitions*	*state vector size (B)*	*memory usage (MB)*
previous compiler [57]	72.2	2 235 884	20 411 565	244	554.399
current compiler	0.22	35 774	120 704	120	5.049

Table 5
The performance of the generated *Promela* model for the running example: comparison of this work with previous work [57].

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 users for specifying properties and providing explicit language support for specification patterns.

Finally, da Costa Cavalheiro *et al.* [19] 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.

None of these approaches supports visual, domain-specific syntax for all used models as in our approach. Moreover, no solutions exist that use a generative approach to define sublanguages similar to the used DSML.

10 Conclusion and Future Work

In this paper, we presented a solution in the form of *ProMoBox*, a framework that integrates the definition and verification of temporal properties in discrete-time behavioural DSMLs, whose semantics can be described as a schedule of graph rewrite rules. Thanks to the expressiveness of graph rewriting, this covers a very large class of problems. With *ProMoBox*, the domain user models not only the system with a DSML, but also its properties, input model, run-time state and output trace. The DSML is thus comprised of five sublanguages, which share domain-specific syntax. The sublanguages are generated from a single metamodel to keep them consistent and to avoid duplication, that is annotated to denote the role of each language concept. The operational semantics of the DSML is modelled as a transformation and is annotated with information about input and output. The modelled system and its properties are transformed to *Promela*, and properties are verified with *Spin*, a tool for explicit state model checking. In case a counterexample is found, its execution trace is transformed to the domain-specific level as a trace model, which can be played out. Thus, the language engineer (whilst modelling a DSML), and the domain user (whilst modelling and verifying properties) are shielded from underlying notations and techniques. The process of the *ProMoBox* framework is explicitly modelled in a Formalism Transformation Graph and Process Model. We used an elevator controller as a running example, and showed how the five sublanguages can be generated and how properties can be verified with *ProMoBox*.

Additionally, we provided a detailed evaluation of the *ProMoBox* approach by evaluating five research questions. For each research question that compares to traditional DSM methods (*i.e., RQ1* to *RQ5*), the *ProMoBox* approach performed favourably, confirming the hypotheses presented in the introduction of Section 7. A discussion of the scope and limitations of the approach as required was presented in Section 8

In conclusion, *ProMoBox* provides a solution for the specification and verification of properties in a highly flexible and automated way, according to DSM principles.

Compared to previous work [20], [57], [60], the following significant additions were introduced in this paper:

- specific research questions have been added in the introduction of Section 7;
- the background has been extensively explained in Section 2;
- the process is explained extensively using the FTG+PM models throughout the paper;
- different use cases are presented in Section 3.1 and are discussed in Section 7;
- the AnnotationTypes DSML has been added in Section 4.1, to allow the creation of new annotations;
- annotations on the operational semantics have been added in Section 4.3;
- the compiler to *Promela* has been improved significantly. The improvement is discussed in Section 7.3.
- an evaluation has been added in Section 7. In previous publications, there was no evaluation section;
- a tool for "playing out" a counterexample has been added in Section 5.6 and Section 6.

An interesting thread for future work is checking different kinds of properties, such as real-time properties. In [29], [43] the authors extend the specification patterns of Dwyer *et al.* with real-time information. These properties could be mapped to a tool for verifying real-time systems, such as UPPAAL [5], which allows the user to specify a system as a timed automaton and specify temporal properties with time bounds.

Moreover, in more complex models, the model checking approach falls short however because of its computational complexity. Therefore we aim to investigate a more scalable alternative to model checking, in the form of test case generation. We believe that the five sublanguages are highly suitable to address this. A test case can be encoded as a run-time model and an input model. A trace model (or a variation thereof representing a trace pattern rather than a trace) can serve as an oracle. We want to investigate whether the test case and oracle can be generated from a design model and a property model, thus requiring the same input as in the current *ProMoBox* approach. The test case generation strategy can then be plugged in, whether it be random selection according to a normal distribution, search-based testing [53], or a different strategy. Since testing does not guarantee correctness like model checking does, running the test case generation phase should return a coverage report [61]. This coverage report should be presentable at the DSM level as well, and could include covering every transition in the transformation schedule, every path in the transformation schedule, every element in the design model, every element of the input metamodel, etc. We aim for an equally flexible and automated approach.

ACKNOWLEDGEMENT

We wish to thank professor Matthew Dwyer for his insights in the specification patterns, and their true intention.

REFERENCES

[1] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[2] Charles Ashbacher. "the unified modeling language reference manual, second edition", by james rumbaugh. *Journal of Object Technology*, 3(10):193–195, 2004.

[3] Marco Autili, Lars Grunske, Markus Lumpe, Patrizio Pelliccione, and Antony Tang. Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar. *IEEE Trans. Software Eng.*, 41(7):620–638, 2015.

[4] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[5] Johan Bengtsson, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL - a tool suite for automatic verification of real-time systems. In Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag, editors, *Hybrid Systems III: Verification and Control, Proceedings of the DIMACS/SYCON Workshop, October 22-25, 1995, Ruttgers University, New Brunswick, NJ, USA*, volume 1066 of *Lecture Notes in Computer Science*, pages 232–243. Springer, 1995.

[6] Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, César Mu noz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, N. Shankar, Eli Singerman, and Ashish Tiwari. An overview of SAL. In C. Michael Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, Hampton, VA, jun 2000. NASA Langley Research Center.

[7] Jean Bézivin, Fabian Büttner, Martin Gogolla, Frédéric Jouault, Ivan Kurtev, and Arne Lindow. Model transformations? transformation models! In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings*, volume 4199 of *Lecture Notes in Computer Science*, pages 440–453. Springer, 2006.

[8] Robert Bill, Sebastian Gabmeyer, Petra Kaufmann, and Martina Seidl. OCL meets CTL: towards CTL-extended OCL model checking. In Jordi Cabot, Martin Gogolla, István Ráth, and Edward D. Willink, editors, *Proceedings of the MODELS 2013 OCL Workshop co-located with the 16th International ACM/IEEE Conference on Model Driven Engineering Languages and Systems (MODELS 2013), Miami, USA, September 30, 2013.*, volume 1092 of *CEUR Workshop Proceedings*, pages 13–22. CEUR-WS.org, 2013.

[9] Frédéric Boulanger, Cécile Hardebolle, Christophe Jacquet, and Dominique Marcadet. Semantic adaptation for models of computation. In Benoît Caillaud, Josep Carmona, and Kunihiko Hiraishi, editors, *11th International Conference on Application of Concurrency to System Design, ACSD 2011, Newcastle Upon Tyne, UK, 20-24 June, 2011*, pages 153–162. IEEE Computer Society, 2011.

[10] John Brooke. Sus: A retrospective. *J. Usability Studies*, 8(2):29–40, February 2013.

[11] Petra Brosch, Uwe Egly, Sebastian Gabmeyer, Gerti Kappel, Martina Seidl, Hans Tompits, Magdalena Widl, and Manuel Wimmer. Towards scenario-based testing of UML diagrams. In Achim D. Brucker and Jacques Julliand, editors, *Tests and Proofs - 6th International Conference, TAP 2012, Prague, Czech Republic, May 31 - June 1, 2012. Proceedings*, volume 7305 of *Lecture Notes in Computer Science*, pages 149–155. Springer, 2012.

[12] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. Automating co-evolution in model-driven engineering. In *12th International IEEE Enterprise Distributed Object Computing Conference, ECOC 2008, 15-19 September 2008, Munich, Germany*, pages 222–231. IEEE Computer Society, 2008.

[13] Alessandro Cimatti, Sergio Mover, and Stefano Tonetta. Proving and explaining the unfeasibility of message sequence charts for hybrid systems. In Per Bjesse and Anna Slobodová, editors, *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, pages 54–62. FMCAD Inc., 2011.

[14] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[15] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[16] Benoît Combemale, Xavier Crégut, and Marc Pantel. A design pattern to build executable dsmls and associated v&v tools. In Karl R. P. H. Leung and Pornsiri Muenchaisri, editors, *19th Asia-Pacific Software Engineering Conference, APSEC 2012, Hong Kong, China, December 4-7, 2012*, pages 282–287. IEEE, 2012.

[17] James C. Corbett, Matthew B. Dwyer, John Hatcliff, and Robby. A language framework for expressing checkable properties of dynamic software. In Klaus Havelund, John Penix, and Willem Visser, editors, *SPIN Model Checking and Software Verification, 7th International SPIN Workshop, Stanford, CA, USA, August 30 - September 1, 2000, Proceedings*, volume 1885 of *Lecture Notes in Computer Science*, pages 205–223. Springer, 2000.

[18] Gennaro Costagliola, Andrea De Lucia, Sergio Orefice, and Giuseppe Polese. A classification framework to support the design of visual languages. *J. Vis. Lang. Comput.*, 13(6):573–600, 2002.

[19] Simone André da Costa Cavalheiro, Luciana Foss, and Leila Ribeiro. Specification patterns for properties over reachable states of graph grammars. In Rohit Gheyi and David A. Naumann, editors, *Formal Methods: Foundations and Applications - 15th Brazilian Symposium, SBMF 2012, Natal, Brazil, September 23-28, 2012. Proceedings*, volume 7498 of *Lecture Notes in Computer Science*, pages 83–98. Springer, 2012.

[20] Romuald Deshayes, Bart Meyers, Tom Mens, and Hans Vangheluwe. Promobox in practice : A case study on the GISMO domain-specific modelling language. In Daniel Balasubramanian, Christophe Jacquet, Pieter Van Gorp, Sahar Kokaly, and Tamás Mészáros, editors, *Proceedings of the 8th Workshop on Multi-Paradigm Modeling co-located with the 17th International Conference on Model Driven Engineering Languages and Systems, MPM@MODELS 2014, Valencia, Spain, September 30, 2014.*, volume 1237 of *CEUR Workshop Proceedings*, pages 21–30. CEUR-WS.org, 2014.

[21] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In Mark A. Ardis and Joanne M. Atlee, editors, *Proceedings of the Second Workshop on Formal Methods in Software Practice, March 4-5, 1998, Clearwater Beach, Florida, USA*, pages 7–15. ACM, 1998.

[22] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In Barry W. Boehm, David Garlan, and Jeff Kramer, editors, *Proceedings of the 1999 International Conference on Software Engineering, ICSE' 99, Los Angeles, CA, USA, May 16-22, 1999.*, pages 411–420. ACM, 1999.

[23] E. Allen Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 995–1072. MIT Press Cambridge, 1990.

[24] E. Allen Emerson and Edmund M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comput. Program.*, 2(3):241–266, 1982.

[25] Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. Story diagrams: A new graph rewrite language based on the unified modeling language and java. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Theory and Application of Graph Transformations, 6th International Workshop, TAGT'98, Paderborn, Germany, November 16-20, 1998, Selected Papers*, volume 1764 of *Lecture Notes in Computer Science*, pages 296–309. Springer, 1998.

[26] Robert B. France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. *CoRR*, abs/1409.6620, 2014.

[27] Sebastian Gabmeyer, Petra Kaufmann, and Martina Seidl. A classification of model checking-based verification approaches for software models. In *Second Workshop on Verification Of Model Transformations (VOLT)*, 2013.

[28] Jeff Gray, Sandeep Neema, Juha-Pekka Tolvanen, Aniruddha S. Gokhale, Steven Kelly, and Jonathan Sprinkle. Domain-specific modeling. In Paul A. Fishwick, editor, *Handbook of Dynamic System Modeling*. Chapman and Hall/CRC, 2007.

[29] Volker Gruhn and Ralf Laue. Patterns for timed property specifications. *Electr. Notes Theor. Comput. Sci.*, 153(2):117–133, 2006.

[30] Esther Guerra, Juan de Lara, Dimitrios S. Kolovos, and Richard F. Paige. A visual specification language for model-to-model transformations. In Christopher D. Hundhausen, Emmanuel Pietriga, Paloma Díaz, and Mary Beth Rosson, editors, *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2010, Leganés-Madrid, Spain, 21-25 September 2010, Proceedings*, pages 119–126. IEEE Computer Society, 2010.

[31] Esther Guerra, Juan de Lara, Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, and Wieland Schwinger. Automated verification of model transformations based on visual contracts. *Autom. Softw. Eng.*, 20(1):5–46, 2013.

[32] Esther Guerra, Paloma Díaz, and Juan de Lara. Visual specification of metrics for domain specific visual languages. *Electr. Notes Theor. Comput. Sci.*, 211:99–110, 2008.

[33] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[34] Markus Herrmannsdoerfer, Sebastian Benz, and Elmar Juergens. Cope - automating coupled evolution of metamodels and models. In *23rd European Conference on Object-Oriented Programming (ECOOP)*, pages 52–76, 2009.

[35] Gerard J. Holzmann. The model checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[36] Franz Huber, Bernhard Schätz, Alexander Schmidt, and Katharina Spies. Autofocus: A tool for distributed systems specification. In Bengt Jonsson and Joachim Parrow, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems, 4th International Symposium, FTRTFT'96, Uppsala, Sweden, September 9-13, 1996, Proceedings*, volume 1135 of *Lecture Notes in Computer Science*, pages 467–470. Springer, 1996.

[37] Bilal Kanso and Safouan Taha. Temporal constraint support for OCL. In Krzysztof Czarnecki and Görel Hedin, editors, *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers*, volume 7745 of *Lecture Notes in Computer Science*, pages 83–103. Springer, 2012.

[38] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. John Wiley & Sons, March 2008.

[39] Chris F. Kemerer. An empirical validation of software cost estimation models. *Commun. ACM*, 30(5):416–429, 1987.

[40] Florian Klein and Holger Giese. Joint structural and temporal property specification using timed story scenario diagrams. In Matthew B. Dwyer and Antónia Lopes, editors, *Fundamental Approaches to Software Engineering, 10th International Conference, FASE 2007, Held as Part of the Joint European Conferences, on Theory and Practice of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4422 of *Lecture Notes in Computer Science*, pages 185–199. Springer, 2007.

[41] Alexander Knapp, Stephan Merz, and Christopher Rauh. Model checking - timed UML state machines and collaborations. In Werner Damm and Ernst-Rüdiger Olderog, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems, 7th International Symposium, FTRTFT 2002, Co-sponsored by IFIP WG 2.2, Oldenburg, Germany, September 9-12, 2002, Proceedings*, volume 2469 of *Lecture Notes in Computer Science*, pages 395–416. Springer, 2002.

[42] Alexander Knapp and Jochen Wuttke. Model checking of UML 2.0 interactions. In Kühne [45], pages 42–51.

[43] Sascha Konrad and Betty H. C. Cheng. Real-time specification patterns. In Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, editors, *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, pages 372–381. ACM, 2005.

[44] Thomas Kühne. Matters of (meta-)modeling. *Software and System Modeling*, 5(4):369–385, 2006.

[45] Thomas Kühne, editor. *Models in Software Engineering, Workshops and Symposia at MoDELS 2006, Genoa, Italy, October 1-6, 2006, Reports and Revised Selected Papers*, volume 4364 of *Lecture Notes in Computer Science*. Springer, 2007.

[46] Thomas Kühne, Gergely Mezei, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer. Explicit transformation modeling. In Sudipto Ghosh, editor, *Models in Software Engineering, Workshops and Symposia at MODELS 2009, Denver, CO, USA, October 4-9, 2009, Reports and Revised Selected Papers*, volume 6002 of *Lecture Notes in Computer Science*, pages 240–255. Springer, 2009.

[47] Xuandong Li, Jun Hu, Lei Bu, Jianhua Zhao, and Guoliang Zheng. Consistency checking of concurrent models for scenario-based specifications. In Andreas Prinz, Rick Reed, and Jeanne Reed, editors, *SDL 2005: Model Driven, 12th International SDL Forum, Grimstad, Norway, June 20-23, 2005, Proceedings*, volume 3530 of *Lecture Notes in Computer Science*, pages 298–312. Springer, 2005.

[48] Levi Lucio, Sadaf Mustafiz, Joachim Denil, Bart Meyers, and Hans Vangheluwe. The Formalism Transformation Graph as a Guide to Model Driven Engineering. Technical Report SOCS-TR2012.1, School of Computer Science, McGill University, March 2012.

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

[50] The Mathworks. Model verification using simulink control design and simulink verification blocks. http://www.mathworks.com/help/slcontrol/ug/model-verification-using-simulink-control-design-and-simulink-verification-blocks.html. Accessed: December 2015.

[51] The Mathworks. Simulink - simulation and model-based design. http://nl.mathworks.com/products/simulink/. Accessed: May 2015.

[52] Thomas J. McCabe. A complexity measure. *IEEE Trans. Software Eng.*, 2(4):308–320, 1976.

[53] Phil McMinn. Search-based software test data generation: a survey. *Softw. Test., Verif. Reliab.*, 14(2):105–156, 2004.

[54] Stephan Merz. An introduction to model checking. In Stephan Merz and Nicolas Navet, editors, *Modeling and Verification of Real-Time Systems - Formalisms and Software Tools*, pages 81–116. ISTE Publishing, 2008.

[55] Bart Meyers, Joachim Denil, Frédéric Boulanger, Cécile Hardebolle, Christophe Jacquet, and Hans Vangheluwe. A DSL for explicit semantic adaptation. In Christophe Jacquet, Daniel Balasubramanian, Edward Jones, and Tamás Mészáros, editors, *Proceedings of the 7th Workshop on Multi-Paradigm Modeling co-located with the 16th International Conference on Model Driven Engineering Languages and Systems, MPM@MoDELS 2013, Miami, Florida, September 30, 2013.*, volume 1112 of *CEUR Workshop Proceedings*, pages 47–56. CEUR-WS.org, 2013.

[56] Bart Meyers, Joachim Denil, István Dávid, and Hans Vangheluwe. Automated testing support for reactive domain-specific modelling languages. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2016, pages 181–194, New York, NY, USA, 2016. ACM.

[57] Bart Meyers, Romuald Deshayes, Levi Lucio, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer. ProMoBox: A framework for generating domain-specific property languages. In *Software Language Engineering*, volume 8706 of *Lecture Notes in Computer Science*, pages 1–20. Springer International Publishing, 2014.

[58] Bart Meyers and Hans Vangheluwe. A framework for evolution of modelling languages. *Sci. Comput. Program.*, 76(12):1223–1246, 2011.

[59] Bart Meyers, Manuel Wimmer, Antonio Cicchetti, and Jonathan Sprinkle. A generic in-place transformation-based approach to structured model co-evolution. *ECEASST*, 42, 2011.

[60] Bart Meyers, Manuel Wimmer, and Hans Vangheluwe. Towards domain-specific property languages: The ProMoBox approach. In *Proceedings of the 2013 ACM Workshop on Domain-specific Modeling*, pages 39–44. ACM New York, NY, USA, 2013.

[61] Joan C. Miller and Clifford J. Maloney. Systematic mistake analysis of digital computer programs. *Commun. ACM*, 6(2):58–63, 1963.

[62] Pieter J. Mosterman. *Hybrid Dynamic Systems: A Hybrid Bond Graph Modeling Paradigm and its Application in Diagnosis*. PhD thesis, Vanderbilt University, 1997.

[63] Pieter J. Mosterman and Hans Vangheluwe. Computer automated multi-paradigm modeling: An introduction. *Simulation*, 80(9):433–450, 2004.

[64] Sadaf Mustafiz, Joachim Denil, Levi Lucio, and Hans Vangheluwe. The FTG+PM framework for multi-paradigm modelling: an automotive case study. In Cécile Hardebolle, Eugene Syriani, Jonathan Sprinkle, and Tamás Mészáros, editors, *Proceedings of the 6th International Workshop on Multi-Paradigm Modeling, MPM@MoDELS 2012, Innsbruck, Austria, October 1-5, 2012*, pages 13–18. ACM, 2012.

[65] Object Management Group. Object constraint language version 2.4. Technical report, OMG, 2014.

[66] Object Management Group. OMG Unified Modeling Language Version 2.5. Technical report, OMG, March 2015.

[67] Patrizio Pelliccione, Paola Inverardi, and Henry Muccini. CHARMY: A framework for designing and verifying architectural specifications. *IEEE Trans. Software Eng.*, 35(3):325–346, 2009.

[68] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977.

[69] Elham Ramezani, Dirk Fahland, and Wil M. P. van der Aalst. Where did I misbehave? diagnostic information in compliance checking. In Alistair P. Barros, Avigdor Gal, and Ekkart Kindler, editors, *Business Process Management - 10th International Conference, BPM 2012, Tallinn, Estonia, September 3-6, 2012. Proceedings*, volume 7481 of *Lecture Notes in Computer Science*, pages 262–278. Springer, 2012.

[70] Wolfgang Reisig and Grzegorz Rozenberg, editors. *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, September 1996*, volume 1491 of *Lecture Notes in Computer Science*. Springer, 1998.

[71] Arend Rensink. Explicit state model checking for graph grammars. In Pierpaolo Degano, Rocco De Nicola, and José Meseguer, editors, *Concurrency, Graphs and Models, Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*, volume 5065 of *Lecture Notes in Computer Science*, pages 114–132. Springer, 2008.

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

[73] José Eduardo Rivera, Francisco Durán, and Antonio Vallecillo. Formal specification and analysis of domain specific models using maude. *Simulation*, 85(11-12):778–792, 2009.

[74] José Eduardo Rivera, Esther Guerra, Juan de Lara, and Antonio Vallecillo. Analyzing rule-based behavioral semantics of visual modeling languages with maude. In Dragan Gasevic, Ralf Lämmel, and Eric Van Wyk, editors, *Software Language Engineering, First International Conference, SLE 2008, Toulouse, France, September 29-30, 2008. Revised Selected Papers*, volume 5452 of *Lecture Notes in Computer Science*, pages 54–73. Springer, 2008.

[75] Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and Fiona Polack. The epsilon generation language. In Ina Schieferdecker and Alan Hartman, editors, *Model Driven Architecture - Foundations and Applications, 4th European Conference, ECMDA-FA 2008, Berlin, Germany, June 9-13, 2008. Proceedings*, volume 5095 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2008.

[76] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, 2009.

[77] Laurent Safa. The practice of deploying DSM Report from a Japanese appliance maker trenches. In Jeff Gray, Juha-Pekka Tolvanen, and Jonathan Sprinkle, editors, *Sixth Object-Oriented Programming, Systems, Languages and Applications Workshop on Domain-Specific Modeling*, pages 185–196. University of Jyväskylä, October 2006.

[78] Carolyn B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Trans. Software Eng.*, 25(4):557–572, 1999.

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

[80] Frank Strobl and Alexander Wisspeintner. Specification of an elevator control system – an autofocus case study. Technical Report TUM-I9906, Technische Univerität München, 1999.

[81] Sagar Sunkle and Vinay Kulkarni. Cost estimation for model-driven engineering. In Robert B. France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson, editors, *Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30-October 5, 2012. Proceedings*, volume 7590 of *Lecture Notes in Computer Science*, pages 659–675. Springer, 2012.

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

[83] Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Simon Van Mierlo, and Hüseyin Ergin. Atompm: A web-based modeling environment. In Yan Liu, Steffen Zschaler, Benoit Baudry, Sudipto Ghosh, Davide Di Ruscio, Ethan K. Jackson, and Manuel Wimmer, editors, *Joint Proceedings of MODELS'13 Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition co-located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013), Miami, USA, September 29 - October 4, 2013.*, volume 1115 of *CEUR Workshop Proceedings*, pages 21–25. CEUR-WS.org, 2013.

[84] Dániel Varró. Automated formal verification of visual modeling languages by model checking. *Software and System Modeling*, 3(2):85–113, 2004.

[85] Gergely Varró, Katalin Friedl, and Dániel Varró. Adaptive graph pattern matching for model transformations using model-sensitive search plans. *Electr. Notes Theor. Comput. Sci.*, 152:191–205, 2006.

[86] Willem Visser, Matthew B. Dwyer, and Michael W. Whalen. The hidden models of model checking. *Software and System Modeling*, 11(4):541–555, 2012.

[87] Faiez Zalila, Xavier Crégut, and Marc Pantel. Leveraging formal verification tools for DSML users: A process modeling case study. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies - 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part II*, volume 7610 of *Lecture Notes in Computer Science*, pages 329–343. Springer, 2012.

[88] Faiez Zalila, Xavier Crégut, and Marc Pantel. Formal verification integration approach for DSML. In Ana Moreira, Bernhard Schätz, Jeff Gray, Antonio Vallecillo, and Peter J. Clarke, editors, *Model-Driven Engineering Languages and Systems - 16th International Conference, MODELS 2013, Miami, FL, USA, September 29 - October 4, 2013. Proceedings*, volume 8107 of *Lecture Notes in Computer Science*, pages 336–351. Springer, 2013.

[89] Paul Ziemann and Martin Gogolla. OCL extended with temporal logic. In Manfred Broy and Alexandre V. Zamulin, editors, *Perspectives of Systems Informatics, 5th International Andrei Ershov Memorial Conference, PSI 2003, Akademgorodok, Novosibirsk, Russia, July 9-12, 2003, Revised Papers*, volume 2890 of *Lecture Notes in Computer Science*, pages 351–357. Springer, 2003.

Bart Meyers Bart Meyers is a research engineering at the strategic research centre Flanders Make vzw in Leuven (Belgium), where he conducts research on model-based systems engineering. Before, Bart Meyers was a postdoctoral researcher in the department of Mathematics and Computer Science at the University of Antwerp (Belgium). He obtained his Ph.D. in 2016 in the University of Antwerp. Since 2009, he is a member of the Antwerp Systems and Software Modelling Lab (AnSyMo) research group. His research interests are in the field of language engineering in the context of domain-specific modelling. More specifically, he investigates domain-specific property languages and, more recently, product line engineering. His research is directed towards generative approaches, supported by core model-driven engineering techniques metamodelling and model transformation. He has written papers that were accepted at the International Conference on Software Language Engineering (SLE) and were published in the Science of Computer Programming journal (SCP). In 2014 and 2015, he was the co-organiser of the summer school Domain-Specific Modelling Theory and Practice, when it was held in Antwerp (http://www.dsm-tp.org). Contact him at bart.meyers@flandersmake.be, or visit http://msdl.cs.mcgill.ca/people/bart/.

Hans Vangheluwe Hans Vangheluwe is a Professor in the Antwerp Systems and software Modelling (AnSyMo) group within the department of Mathematics and Computer Science at the University of Antwerp in Belgium, an Adjunct Professor in the School of Computer Science at McGill University, Montreal, Canada and an Adjunct Professor at the National University of Defense Technology in Changsha, China. AnSyMo is an Associated Lab of Flanders Make, the strategic research centre for the Flemish manufacturing industry. In a variety of projects, often with industrial partners, he develops and applies the model-based theory and techniques of Multi-Paradigm Modelling (MPM) in application domains as diverse as waste water treatment and automotive software. He is an Associate Editor of ACM's Transactions on Modeling and Computer Simulation (TOMACS), of the International Journal of Critical Computer-Based Systems, and of the International Journal of Adaptive, Resilient and Autonomic Systems. He is the chair of the EU COST Action Multi-Paradigm Modelling for Cyber-Physical Systems (MPM4CPS). Contact him at hv@cs.mcgill.ca, or visit http://msdl.cs.mcgill.ca/people/hv/.

Joachim Denil Joachim Denil is currently a post-doctoral researcher at the Antwerp Systems and software Modelling (AnSyMo) group in the University of Antwerp. AnSyMo is an Associated Lab of Flanders Make, the strategic research centre for the Flemish manufacturing industry. He received his Ph.D. in computer science and his B.Sc. and M.Sc. in Electronics from the university of Antwerp. He received his B.Sc. in computer Science from the Free University of Brussels. Joachim also pursued post-doctoral research at McGill University on the Canada-wide NECSIS project. His main research interest is the design of software-intensive and cyber-physical systems, in particular multi-paradigm modelling, embedded system design, simulation-based design, etc. Contact him at joachim.denil@uantwerpen.be, or visit http://msdl.cs.mcgill.ca/people/joachim/.

Rick Salay Rick Salay is a research associate in the Department of Computer Science at the University of Toronto and currently a visiting researcher at the Antwerp Systems and Software Modelling Lab at the University of Antwerp, His visit is funded by Flanders Make, the strategic research centre for the Flemish manufacturing industry. He received a B.A.Sc. and M.A.Sc. in Systems Design Engineering from University of Waterloo (1991) and a Ph.D. in Computer Science from the University of Toronto (2010). His research focus is on developing formal theories about non-formal concepts such as modeler intent and modeler uncertainty in order to provide a foundation for tool support that will help software engineering practitioners. He regularly serves on program committees for software engineering conferences and workshops. Prior to his Ph.D., he had a 15 year career in advanced software product development holding various senior software design roles, most recently as chief architect at InSystems Technologies Inc. (now Oracle). Contact him at rsalay@cs.toronto.edu, or visit http://www.cs.toronto.edu/~rsalay.