Domain-Specific Modelling for Human-Computer Interaction

Simon Van Mierlo, Yentl Van Tendeloo, Bart Meyers, and Hans Vangheluwe

1 Introduction

Developing complex, reactive, often real-time, software-intensive systems using a traditional, code-centric approach, is not an easy feat: knowledge is required from both the problem domain (*e.g.*, power plant engineering), and computer programming. Apart from being inefficient and costly, due to the need for an additional programmer on the project, this can also result in more fundamental problems. The programmer, who implements the software, has no knowledge of the problem domain, or basic knowledge at best. The domain expert, on the other hand, has deep knowledge of the problem domain, but only a limited understanding of computer programs. This can result in communication problems, such as the programmer making false assumptions about the domain, or the domain expert to gloss over details when explaining the problem to the programmer. Furthermore, the domain experts will finally receive a software component that they don't fully understand, making it difficult for them to validate and modify if necessary. There is in effect a conceptual gap between the two domains, hindering productivity.

Model-Driven Engineering (MDE) [30] tries to bridge this gap, by shifting the level of specification from computing concepts (the "how") to conceptual models or abstractions in the problem domain (the "what"). Domain-Specific Modelling (DSM) [12] in particular makes it possible to specify these models in a Domain-Specific Modelling Language (DSML), using concepts and notations of a specific domain. The goal is to enable domain experts to develop, understand, and verify models more easily, without having to use concepts outside of their own domain. It

Simon Van Mierlo · Yentl Van Tendeloo · Bart Meyers
University of Antwerp, Belgium
e-mail: {simon.vanmierlo, yentl.vantendeloo, bart.meyers}@uantwerpen.be

Hans Vangheluwe
University of Antwerp, Belgium
McGill University, Canada
e-mail: {hv@cs.mcgill.ca}

allows the use of a custom visual syntax, which is closer to the problem domain, and therefore more intuitive. Models created in such DSMLs are used, among others, for simulation, (formal) analysis, documentation, and code synthesis for different platforms. There is, however, still a need for a language engineer to create the DSML, which includes defining its syntax, and providing the mapping between the problem domain and the solution domain.

1.1 Case Study

In this chapter, we explain the necessary steps for developing a system using the DSM approach, applied to the nuclear power plant control interface case study. We build this human-computer interaction interface incrementally throughout the chapter. The system consists of two parts:

1. The nuclear power plant, which takes actions as input (*e.g.*, "lower the control rods"), and outputs events in case of warning and error situations. Each nuclear power plant component is built according to its specification, which lists a series of requirements (*e.g.*, "the reactor can only hold 450 bar pressure for 1 minute"). These are specified in the model of each component. Components send warning messages in case their limits are almost reached, such that the user can take control and alleviate the problem. When the user is unable to bring the reactor to a stable state, however, the component sends an error message, indicating that an emergency shutdown is required to prevent a nuclear meltdown. There is a distinction between two types of components:
    a. Monitoring components, which monitor the values of their sensors and send messages to the controller depending on the current state of the component. An example is the generator, which measures the amount of electricity generated. Their state indicates the status of the sensors: *normal*, *warning*, or *error*.
    b. Executing components, which receive messages from the controller and execute the desired operation. An valve, for example, is either open or closed. Their state indicates the physical state of the component, for example *open* or *closed*.
2. The controller, which acts as the interface between the plant and the user. Users can send messages to the controller by pressing buttons. It is, however, up to the controller to pass on this request to the actual component(s), or choose to ignore it (possibly sending other messages to components, depending on the state of the reactor core). We implement a controller which has three main modes:
    a. Normal operation, where the user is unrestricted and all messages are passed.
    b. Restricted operation, where the user can only perform actions which lower the pressure in the reactor. This mode is entered when any of the components sends out a warning message. When all components are back to normal, full control is returned to the user.

c. Emergency operation, where control is taken away from the user, and the controller takes over. This mode is entered when any of the components sends out an error message. The controller will forcefully shut down the reactor and ignore further input from the user. As there is likely damage to the power plant's components, it is impossible to go back to normal operation without a full reboot.

We construct a DSML which allows to model the configuration of a nuclear power plant, and express the behaviour of each component, as well as the controller. We intend to automatically synthesize code, which behaves as specified in the model. Additionally, we use the ProMoBox approach [17] to verify that all of the desired (safety) properties are fulfilled by the modelled human-computer interface.

We use the open-source metamodelling tool AToMPM [29] ("A Tool for Multi-Paradigm Modelling") throughout this chapter, though the approach can be applied in other metamodelling tools with similar capabilities. All aspects of defining a new language are supported: creating an abstract syntax, defining custom concrete syntax, and defining semantics through the use of model transformations.

1.2 Terminology

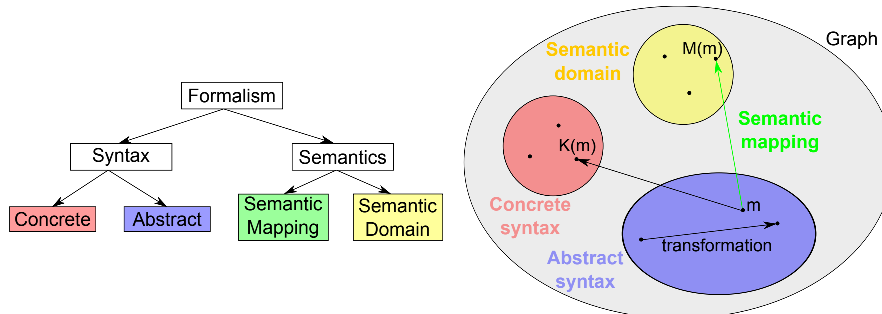

Fig. 1 Terminology.

The first step in the DSM approach when modelling in a new domain is, after a domain analysis, creating an appropriate DSML. A DSML is fully defined [13] by:

1. Its **abstract syntax**, defining the DSML constructs and their allowed combinations. This information is typically captured in a metamodel.
2. Its **concrete syntax**, specifying the visual representation of the different constructs. This visual representation can either be graphical (using icons), or textual.

3. Its **semantics**, defining the meaning of models created in the domain [10]. This encompasses both the **semantic domain** (*what* is its meaning), and the **semantic mapping** (*how* to give it meaning).

For example, $1+2$ and $(+\ 1\ 2)$ can both be seen as textual concrete syntax (*i.e.*, a visualization) for the abstract syntax concept "addition of 1 and 2" (*i.e.*, what construct it is). The semantic domain of this operation is the set of natural numbers (*i.e.*, what it evaluates to), with the semantic mapping being the execution of the operation (*i.e.*, how it is evaluated). Therefore, the semantics, or "meaning", of $1+2$ is 3.

This definition of terminology can be seen in Figure 1. Each aspect of a formalism is modelled explicitly, as well as relations between different formalisms. Throughout the remainder of this paper, we present these four aspects in detail, and present the model(s) related to the use case for each.

Aside from the language definition, properties are defined that should hold for models defined in the language and can be checked. Whereas such properties are normally expressed in property languages, such as LTL [23], we will use the ProMoBox [17] approach. With LTL, we would revert back to the original problem: the conceptual gap between the problem domain and the implementation level is reintroduced. With the ProMoBox approach, properties are modelled using a syntax similar to that of the original DSML. Thus, the modeller specifies both the requirements, or properties, and design models in a familiar notation, lifting both to the problem domain. In case a property doesn't hold, feedback is furthermore given by the system at the domain-specific level.

1.3 Outline

The remainder of this chapter is structured as follows. Section 2 presents the different aspects of syntax: both abstract and concrete. Section 3 presents an introduction to the definition of semantics, and how we apply this to our case study. Section 4 explains the need for properties at the same level as the domain-specific language, and presents the ProMoBox approach. Section 5 concludes. Throughout the chapter, we use the nuclear power plant case study to illustrate how each concept is applied in practice.

2 Syntax

A syntax defines whether elements are valid in a specified language or not. It does not, however, concern itself with what the constructs mean. With syntax only, it would be possible to specify whether a construct is valid, but it might have invalid semantics. A simple, textual example is the expression $\frac{1}{0}$. It is perfectly valid to write this, as it follows all structural rules: a fraction symbol separates two recursively

parsed expressions. However, its semantics is undefined, since it is a division by zero.

2.1 Abstract Syntax

The abstract syntax of a language specifies its constructs and their allowed combinations, and can be compared to grammars specifying parsing rules. Such definitions are captured in a metamodel, which itself is again a model of the metametamodel [14]. Most commonly, the metametamodel is similar to UML Class Diagrams, as is also the case in our case study. The metametamodel used in the examples allows to define classes, associations between classes (with incoming and outgoing multiplicities), and attributes.

While the abstract syntax reasons about the allowable constructs, it does not state anything about how they are presented to the user. In this way, it is distinct from textual grammars, as they already offer the keywords to use [13]. It merely states the concepts that are usable in the domain.

A possible abstract syntax definition for the nuclear power plant use case is shown in Figure 2. It defines the language constructs, such as a *reactor*, *pump*s, and *valve*s, which can have attributes defining its state (*e.g.*, a valve can be open or closed). It also lists the allowed associations between abstract syntax elements (*e.g.*, a generator cannot be directly connected to a steam valve).

The constructs for modelling the behaviour of the elements are also present. The abstract syntax requires all components to have exactly one behavioural definition. In this definition, the component is in a specific state which has transitions to other states. For each transition, it is possible to define cardinalities, to limit the number of outgoing links of each type. For example, the metamodel forbids a single state from having two outgoing transitions of certain types. While we don't yet give semantics to our language, we already limit the set of valid models (*i.e.*, for which semantics need to be defined). By preventing ambiguous situations in the abstract syntax, we do not need to take them into account in the semantic definition, as they represent invalid configurations.

This language definition, together with the concrete syntax definition, is used by AToMPM to generate a domain-specific syntax-directed modelling environment, which means that only valid instances can be created. For example, if the abstract syntax model specifies that there can only be a single SCRAM transition, then drawing a second one will give an error. This maximally constrains the modeller and ensures the models are (syntactically) correct by construction.

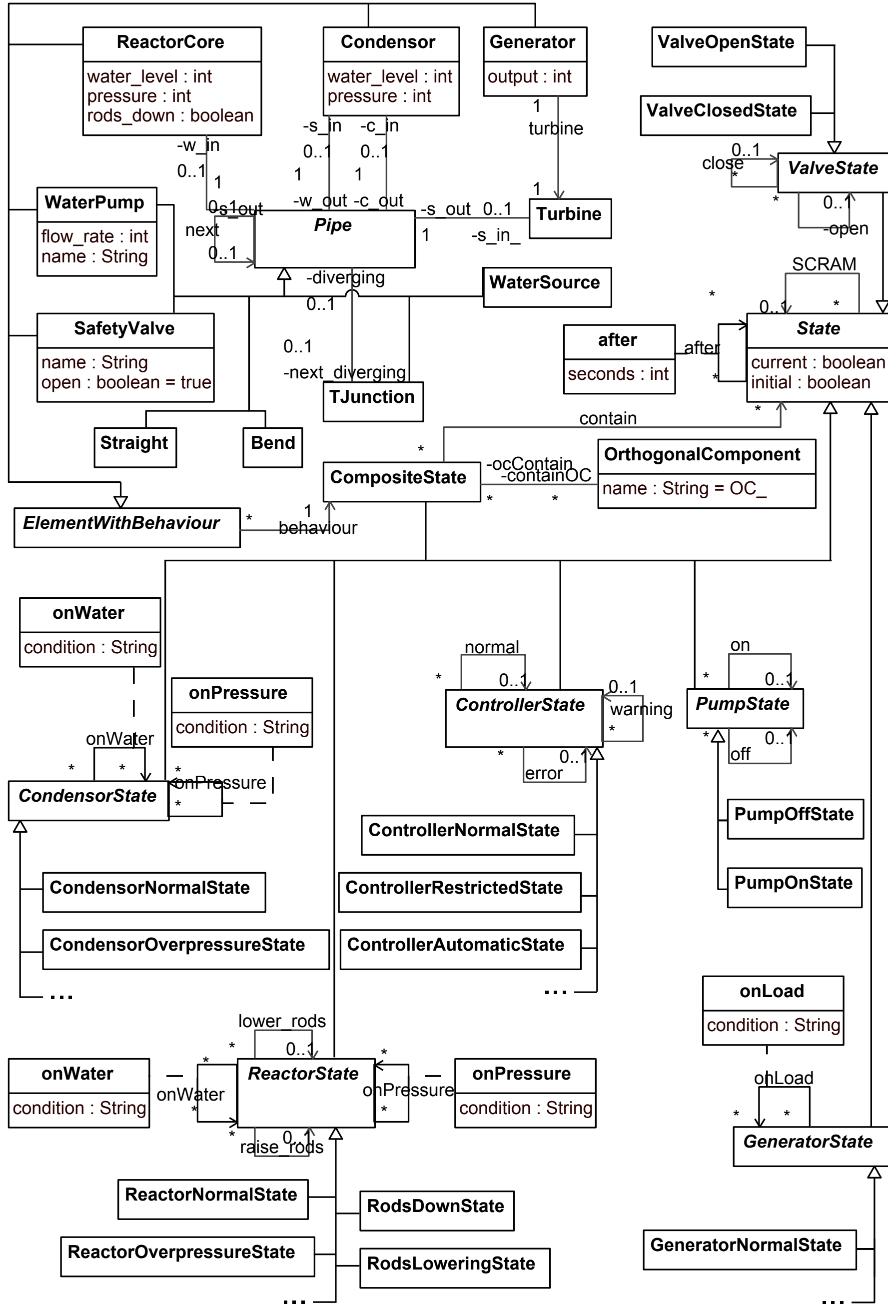

Fig. 2 Abstract syntax definition for the nuclear power plant domain (some subclasses omitted).

2.2 Concrete Syntax

The concrete syntax of a model specifies how elements from the abstract syntax are visually represented. The relation between abstract and concrete syntax elements is also modelled: each representable abstract syntax concept has exactly one concrete syntax construct, and vice versa. As such, the mapping between abstract and concrete syntax needs to be a bijective function. This does not, however, limit the number of distinct concrete syntax definitions, as long as each combination of concrete and abstract syntax has a bijective mapping. The definition of the concrete syntax is a determining factor in the usability of the DSML [1].

Multiple types of concrete syntaxes exist, though the main distinction is between **textual** and **graphical** languages. Both have their advantages and disadvantages: textual languages are more similar to programming languages, making it easier for programmers to start using the DSML. On the other hand, visual languages can represent the problem domain better, due to the use of standardized symbols, despite them being generally less efficient [22]. An overview of different types of graphical languages is given in [4]. Different tools have different options for concrete syntax, depending on the expected target audience of the tool. For example, standard parsers always use a textual language, as their target audience consists of computer programmers who specify a system in (textual) code.

While the possibilities with textual languages are rather restricted, graphical languages have an almost infinite number of possibilities. In [20], several "rules" are identified for handling this large number of possibilities. As indicated beforehand, a single model can have different concrete syntax representations, so it is possible for one to be textual, and another to be graphical.

An excerpt of a possible visual concrete syntax definition for the nuclear power plant use case is shown in Figure 3. Each of the constructs presented in the concrete syntax model corresponds to the abstract syntax element with the same name. Every construct receives a visual representation that is similar to the one defined in the case study. In case of standardized icons or symbols, it would be trivial to define a new concrete syntax model. Furthermore, a specific concrete syntax was created for the definition of the states. Each state is a graphical representation of the state of the physical component, making it easier for users to determine what happens. We chose to attach the graphical representation given in the case study to represent the *normal* functioning of the reactor core; this results in an identical representation of the 'normal reactor state' and the 'rods down state'.

Now that we have a fully defined syntax for our model, we create an instance of the use case in AToMPM, of which a screenshot is shown in Figure 4. A domain-specific modelling environment, generated from the language definition, is loaded into AToMPM, as displayed by the icon toolbar at the top, below AToMPM's general purpose toolbars. This example instance shows a model very similar to the one in the case study. Each component additionally has a specification of its dynamic behaviour. This behaviour definition specifies when to send out messages, using the state of the underlying system, as well as timeouts. The controller is also constructed as per our presented case study.

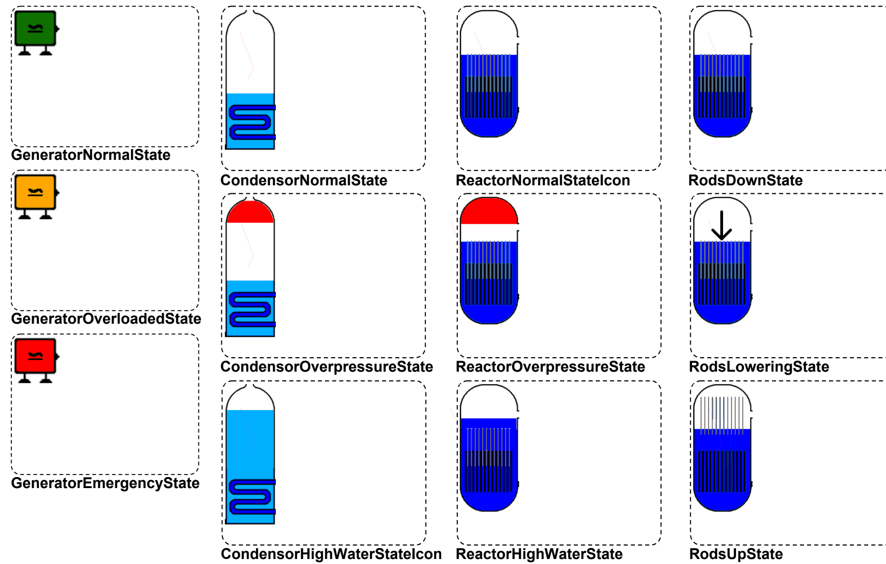

Fig. 3 Concrete syntax definition for the nuclear power plant domain (excerpt).

While the meaning of this model might be intuitively clear, this model does not yet have any meaning, as there is no semantics defined. Defining the semantics of this model is the topic of the next section.

3 Semantics

Since the syntax only defines what a valid model looks like, we need to give a meaning to the models. Even though models might be syntactically valid, their meaning might be useless or even invalid.

It is possible for humans to come up with intuitive semantics for the visual notations used (*e.g.*, an arrow between two states means that the state changes from the source to the destination if a certain condition is satisfied). There is, however, a need to make the semantics explicit for two main reasons:

1. Computers cannot use intuition, and therefore there needs to be some operation defined to convey the meaning to the machine level.
2. Intuition might only take us that far, and can cause some subtle differences in border cases. Having semantics explicitly defined makes different interpretations impossible, as there will always be a “reference implementation”.

Semantics consists of two parts: the domain it maps to, and the mapping itself. Both will be covered in the next subsections.

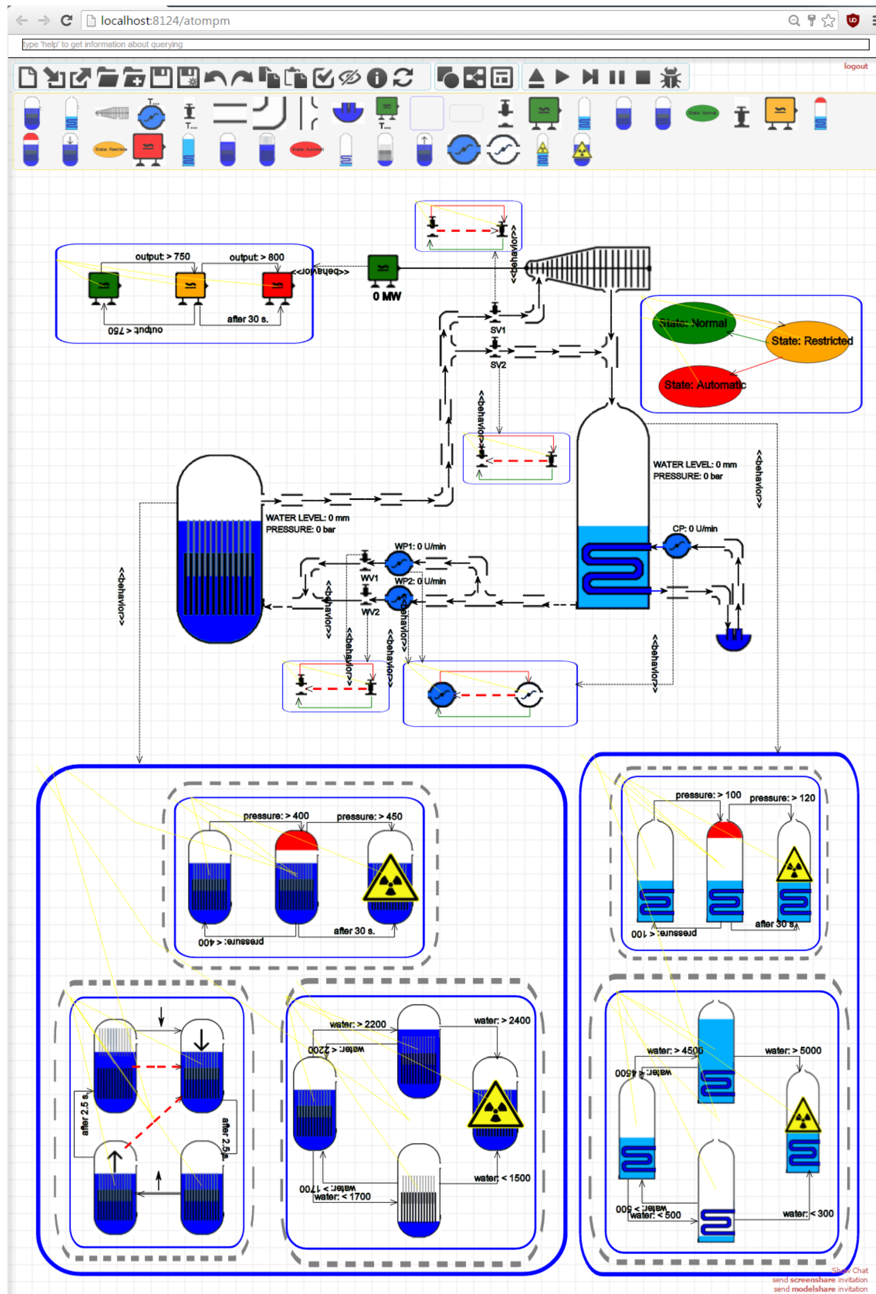

Fig. 4 Screenshot of AToMPM with an example nuclear power plant instance.

3.1 Semantic Domain

The semantic domain is the target of the semantic mapping. As such, the semantic mapping will map every valid language instance to a (not necessarily unique) instance of the semantic domain. Many semantic domains exist, as basically every language with semantics of its own can act as a semantic domain. The choice of semantic domain depends on which properties need to be conserved. For example, DEVS [31] can be used for simulation, Petri nets [21] for verification, Statecharts [8] for code synthesis, and Causal Block Diagrams [3] for continuous systems using differential equations. A single model might even have different semantic domains, each targeted at a specific goal.

For our case study, we use Statecharts as the semantic domain, as we are interested in the timed, reactive, autonomous behaviour of the system, as well as code synthesis. Statecharts were introduced by David Harel [8] as an extension of state machines and state diagrams with hierarchy, orthogonality, and broadcast communication. It is used for the specification and design of complex discrete-event systems, and is popular for the modelling of reactive systems, such as graphical user interfaces. The Statecharts language consists of the following elements:

- *states*, either basic, orthogonal, or hierarchical;
- *transitions* between states, either event-based or time-based;
- *actions*, executed when a state is entered and/or exited;
- *guards* on transitions, modelling conditions that need to be satisfied in order for the transition to "fire";
- *history* states, a memory element that allows the state of the Statechart to be restored.

Figure 5 presents a Statecharts model which is equivalent to the model in our DSML, at least with respect to the properties we are interested in. Parts of the structure can be recognized, though information was added to make the model compliant to the Statecharts formalism. Additions include the sending and receiving of events, and the expansion of forwarding states such as in the controller. The semantic mapping also merged the different behavioural parts into a single Statechart. There are two types of events in the resulting Statecharts model: discrete events coming from the operator (*e.g.*, *lower control rods*), and events coming from the environment, corresponding to sensor readings (*e.g.*, *water level* in reactor). These discrete values are changed into boundary crossing checks, which cannot be easily modelled using Statecharts. Instead, to model these, a more suitable formalism should be chosen, such as Causal-Block Diagrams [3] (CBDs). These CBD models then need to be embedded into the Statecharts model. This requires to connect both formalisms semantically, such that, for example, a signal value in the CBD model translates to an event in the Statecharts model [2]. This is out of scope for this paper, and does not influence the properties we are interested in. We assume the sensor readings are updated correctly and communicated to our Statecharts model.

As is usually the case, the DSML instance is more compact and intuitive, compared to the resulting Statechart instance. The Statecharts language itself also needs to have its semantics defined, as done in [9].

In the following subsection, we define one semantic mapping that maps onto the Statecharts language (called "translational semantics") and one mapping that maps the language onto itself (called "operational semantics").

3.2 Semantic Mapping

While many categories of semantic mapping exist, as presented in [32], we only focus on the two main categories relevant to our case study:

1. **Translational semantics**, where the semantic mapping translates the model from one formalism to another, while maintaining an equivalent model with respect to the properties under study. The target formalism has semantics (again, either translational or operational), meaning that the semantics is "transferred" to the original model.
2. **Operational semantics**, where the semantic mapping effectively executes, or simulates, the model being mapped. Operational semantics can be implemented with an external simulator, or through model transformations that simulate the model by modifying its state. The advantage of in-place model transformations is that semantics are also defined completely in the problem domain, making it suitable for use by domain experts. For our case study, this means implementing a simulator using model transformations.

Due to the possibly many semantic mappings, their chaining can be explicitly modelled in a Formalism Transformation Graph + Process Model (FTG+PM) [16]. An FTG+PM can encompass both automatic transformations (*i.e.*, through model transformations) and manual transformations (*i.e.*, through a user creating a related model).

The semantic mapping, which translates between a source and target model, is commonly expressed using model transformations, which are often called the heart and soul of Model-Driven Engineering [25]. A model transformation is defined using a set of transformation rules, and a schedule.

A rule consists of a Left-Hand Side (LHS) pattern (transformation pre-condition), Right-Hand Side (RHS) pattern (transformation post-condition), and possible Negative Application Condition (NAC) patterns (patterns which, if found, stop rule application). The rule is applicable on a graph (the host graph), if each element in the LHS can be matched in the model, without being able to match any of the NAC patterns. When applying a rule, the elements matched by the LHS are replaced by elements of the RHS in the host graph. Elements of the LHS that don't appear in the RHS are removed, and elements of the RHS that don't appear in the LHS are created. Elements can be labelled in order to correctly link elements from the LHS and RHS.

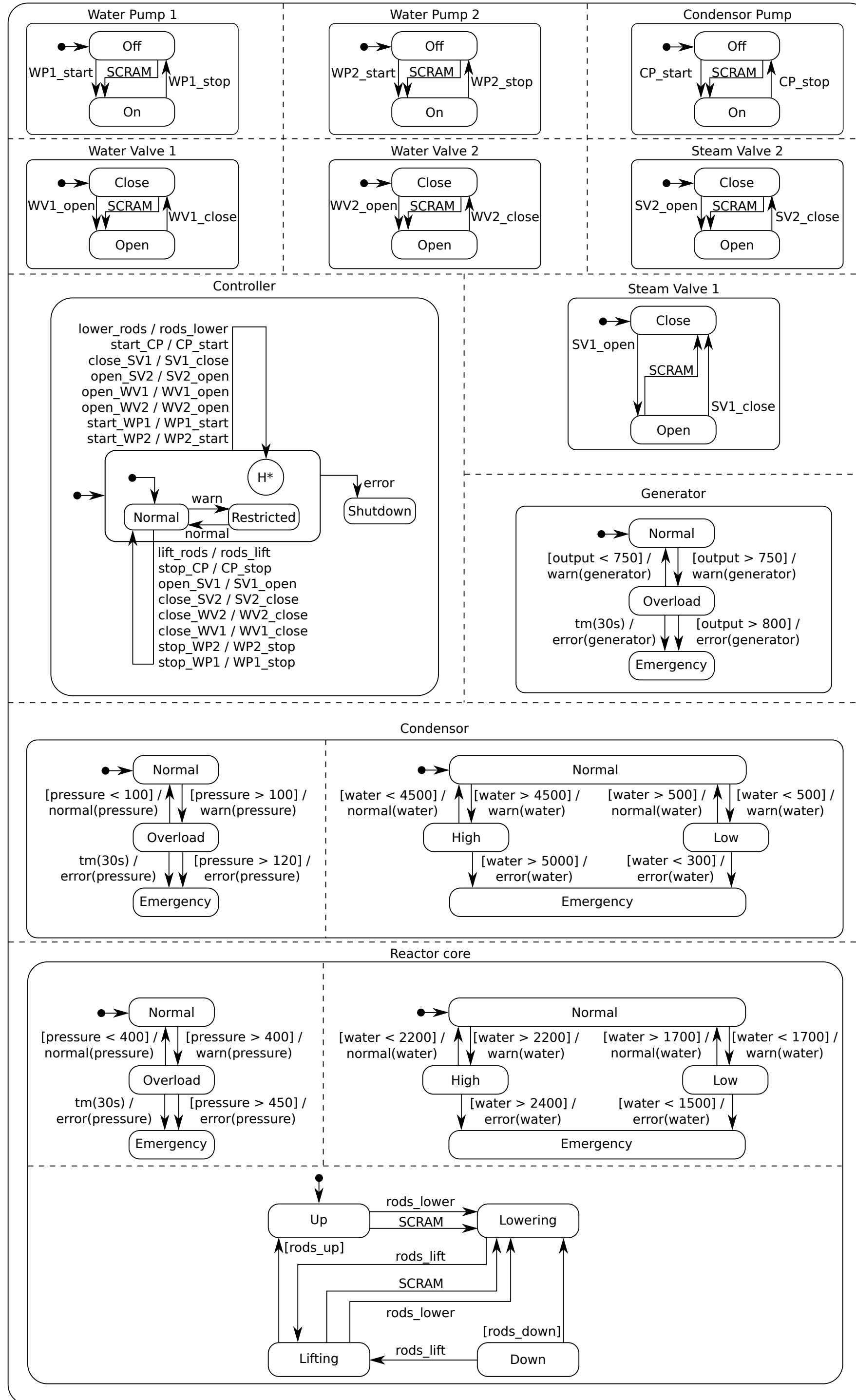

Fig. 5 Statechart equivalent to the behaviour shown in Figure 4.

A schedule determines the order in which transformation rules are applied. For our purpose, we use MoTiF [28], which defines a set of basic building blocks for transformation rule scheduling. We limit ourself to three types of rules: the *ARule* (apply a rule once), the *FRule* (apply a rule for all matches simultaneously), and the *CRule* (for nesting transformation schedules).

In the following subsections, we define both the operational and translational semantics of our modelling language.

3.2.1 Translational Semantics

With translational semantics, the source model is translated to a target model, expressed in a different formalism, which has its own semantic definition. The (partial) semantics of the source model then correspond to the semantics of the target model. As the rule uses both concepts of the problem domain and the target domain (Statecharts in our case), the modeller should be familiar with both domains. The Statecharts language, however, is still much more intuitive and, in the case of a real-time, reactive, autonomous system, more appropriate to use than the average programming language.

The schedule of our transformation is shown in Figure 6, where we see that each component is translated in turn. Each of these blocks are composite rules, meaning that they invoke other schedules. One of these schedules is shown in Figure 7, where the valves are translated. The blocks in the schedule are connected using arrows to denote the flow of control: green arrows are followed when the rule executed successfully, while red arrows are followed when an error occurred. Three pseudo-states denote the start, the succesful termination, and the unsuccesful termination of the transformation. Our schedule consists of a series of FRules, which translate all different valve states to the corresponding Statecharts states. After these are mapped, the transitions between them are also translated, as shown in the example rule in Figure 8. In this rule, we look up the Statecharts states generated from the domain-specific states, and create a link between them if there was one in the domain-specific language. For each kind of link, there needs to be such a rule. In this case, we map the SCRAM message to a Statecharts transition which takes a specific kind of event. Note that we do not remove the original model, ensuring traceability.

The generated Statechart can be coupled to a user interface, to allow user interaction. Figure 9 presents an example interface, created with Python TkInter. To the left, the user is presented with a colour-coded overview of the system. The states of operational elements (valves, for example) are marked in grey, and elements that can emit a warning or error message are marked in either green, orange, or red. On top, an overview of the current state of the system is shown, indicating what level of responsiveness the user can expect. At the bottom is a SCRAM button, which, when pressed, will simply emit the “SCRAM” event to the Statecharts model. The visual representation at the right labels all parts, and provides more in-depth information about each component. Elements can be clicked to interact with them. For example,

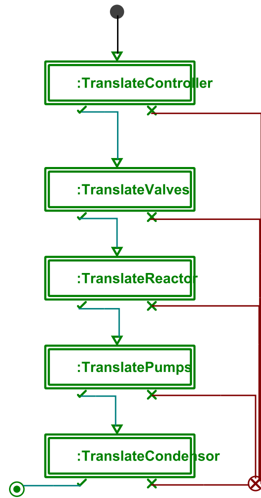

Fig. 6 Top-level transformation schedule.

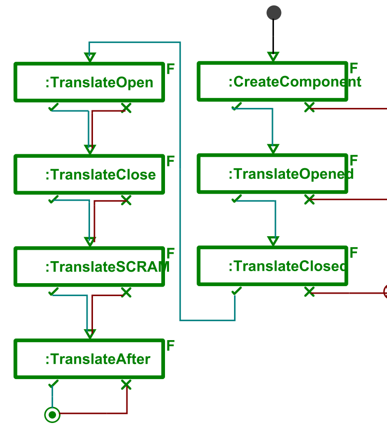

Fig. 7 Transformation schedule for valves.

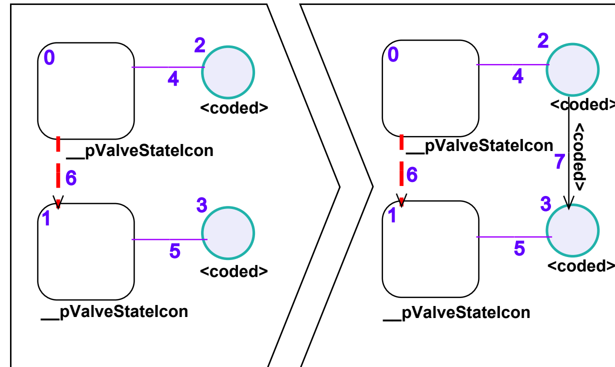

Fig. 8 Example transformation rule for translational semantics.

a pump can be enabled or disabled by clicking on it in the figure. As previously mentioned, this interface is only a minimal example to illustrate the applicability of our approach.

After creating this interface, it needs to be coupled to the Statecharts model. This is done through the use of events: the interface uses the interface of the model to raise and catch events to and from the Statecharts model. For example, pressing the "SCRAM" button raises the "SCRAM" event. Similarly, when the interface catches the event "warning high core pressure", it visualizes this change by changing the color of the pressure reading. As such, the interface doesn't implement any autonomous behaviour, but relies fully on the Statecharts model. Different interfaces can be coupled, as long as they adhere to the interface of the model.

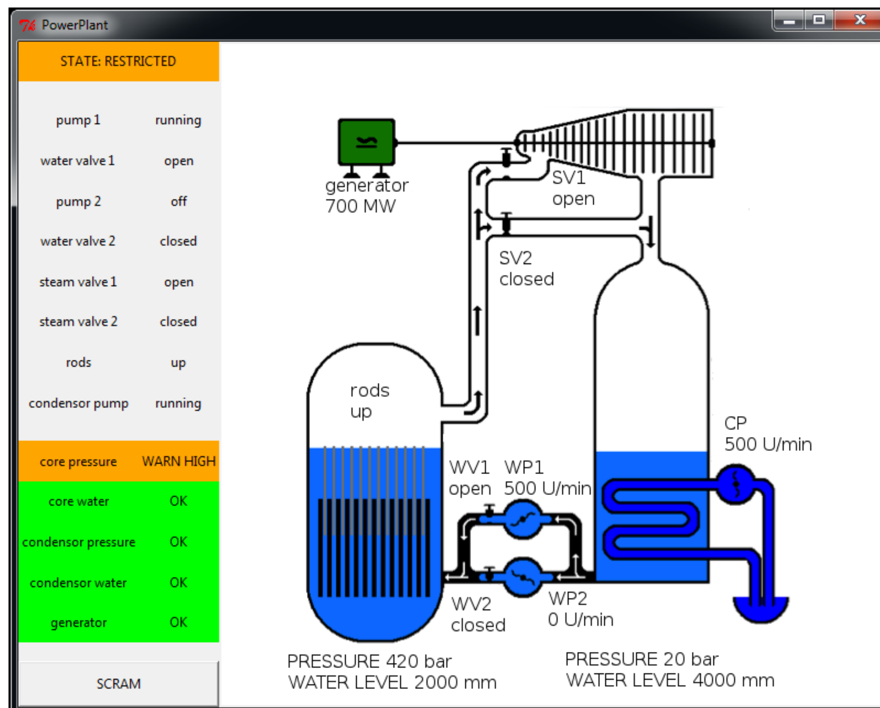

Fig. 9 Example interface in Python TkInter.

3.2.2 Operational Semantics

A formalism is operationalized by defining a simulator for that formalism. This simulator can be modelled as a model transformation that executes the model by continuously updating its state (effectively defining a “stepping” function). The next state of the model is computed from the current state, the information captured in the model (such as state transitions and conditions), and the current state of the environment. Contrary to translational semantics, the source model of operational semantics is often augmented with runtime information. This requires the creation of both a “design formalism” and a “runtime formalism”. In our case study, for example, the runtime formalism is equivalent to the design formalism augmented with information on the current state and the simulated time, as well as a list of inputs from the environment.

An example rule is shown in Figure 10. The rule changes the current state by following the “onPressure” transition. The left hand side of the rule matches the current state, the value of the sensor, and the destination of the transition. It is only applicable if the condition on the transition (*e.g.*, > 450) is satisfied (by comparing it to the value of the sensor reading). We use abstract states for both source and target of the transition, as we do not want to limit the application of the rule to a specific

combination of states: the rule should be applicable for all pairs of reactor states that have an “onPressure” transition. The right hand side then changes the current state to the target of the transition.

The schedule has the form of a “simulation loop”, but is otherwise similar to the one for translational semantics and is therefore not shown here.

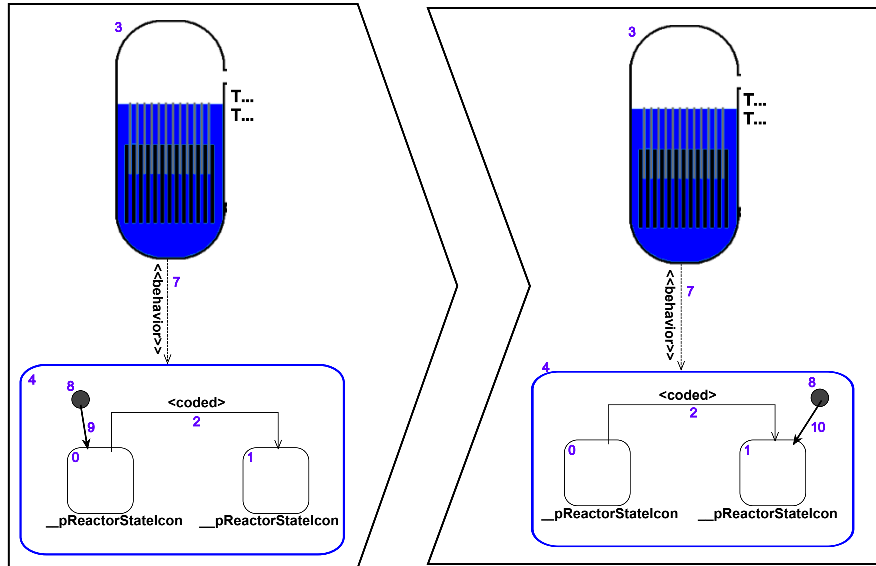

Fig. 10 Example transformation rule for operational semantics.

4 Verification of Properties

DSM mainly focuses on the design of software systems, but can also greatly help in formal verification of these systems. Requirements can be made explicit in the form of properties; questions one can ask of the system, and for which the answer is either “yes” or “no”. Depending on the nature of the verification problem, these properties can be checked using model checking, symbolic execution, constraint satisfaction, solving systems of equations, etc.

The general verification process is shown in Figure 11, 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 to be able to correct the formal model, after which the verification process can be restarted.

In the context of DSM, model-to-model transformations can be used to transform the DSML instance to a formal model, on which model checking can be ap-

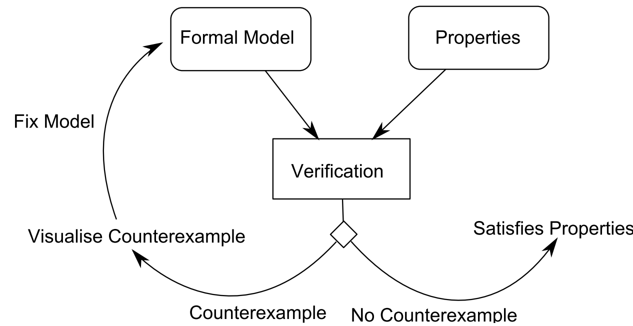

Fig. 11 Verification of formally specified systems

plied [24]. In this case the modeller needs to specify and check the properties directly on the formal model, often in a notation as LTL [23], and needs to transform the verification results back to the DSM level. Having to work with such intricate notations contradicts the philosophy of DSM, where models should be specified in terms of the problem domain.

One solution is to create a DSML for property specification in the same manner as the DSML for designing systems. An example is TimeLine [26]: a visual DSML for specifying temporal constraints for event-based systems. Developing and maintaining yet another DSML comes at a great cost, again contradicting the DSM philosophy of fast language engineering. Possible counterexamples also need to be visualised somehow, possibly requiring yet another DSML.

In this section we apply the *ProMoBox* approach [17, 18, 19] which resolves the above issues to the nuclear power plant DSML. The *ProMoBox* approach for the Nuclear Power Plant Control (*NPPC*) is laid out in Figure 12. Minimal abstraction and annotation of the *NPPC'* DSML is required to generate the *ProMoBox*—a family of five DSMLs that cover all tasks of the verification process:

- The *design language* $NPPC_D$ allows DSM engineers to design the static structure of the system, similarly to traditional DSM approaches as described earlier in this chapter.
- The *runtime language* $NPPC_R$ enables modellers to define a state of the system, such as an initial state as input of a simulation, or a particular “snapshot” or state during runtime.
- The *input language* $NPPC_I$ lets the DSM engineer model the behaviour of the system environment, for example by modelling an input scenario as an ordered sequence of events containing one or more input elements.
- The *output language* $NPPC_O$ is used to represent execution traces (expressed as ordered sequences of states and transitions) of a simulation or to show verification results in the form of a counterexample. Output models can also be created manually as part of an oracle for a test case.
- The *property language* $NPPC_P$ is used to express properties based on modal temporal logic, including structural logic and quantification.

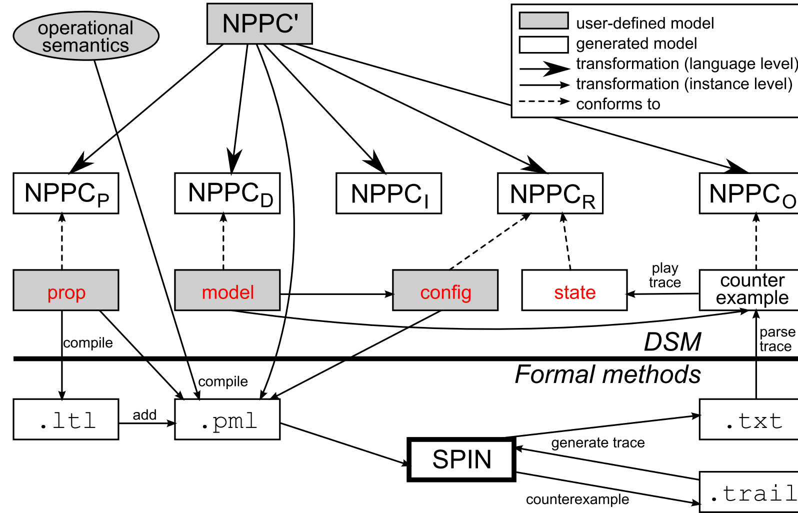

Fig. 12 The *ProMoBox* approach applied to the nuclear power plant control (*NPPC*) DSML

The bottom side of Figure 12 reflects one instance using the SPIN model checker of the verification process shown in Figure 11. The modeller uses the *ProMoBox* to model not only the nuclear power plant system, but also temporal properties. The *ProMoBox* approach supports fully automatic compilation to LTL and Promela, verification using the *SPIN Model Checker* [11], and subsequent visualisation of counterexamples. In conclusion, by using the *ProMoBox* approach the user can specify or inspect all relevant models at the domain-specific level, while the development overhead is kept to a minimum.

4.1 Abstraction and Annotation Phase

Since model checking techniques will be applied, a simplification step might be necessary to reduce the combinatorial search space of the model, to avoid long verification time or extensive memory usage. This scalability problem is inherent to model checking: typical culprits for state space explosion are the number of variables and the size of variable's domains. Attributes in the metamodel must therefore be simplified by abstracting the domain of some relevant language constructs (see Figure 13).

- All attributes of type *integer* are reduced to *boolean*s or *enumeration*s. We use system information to find the enumeration values, meaning that the DSML is reduced to an even smaller domain: the enumeration values represent ranges of integer values (for example, *high*, *low*, or *normal* water levels abstract away the

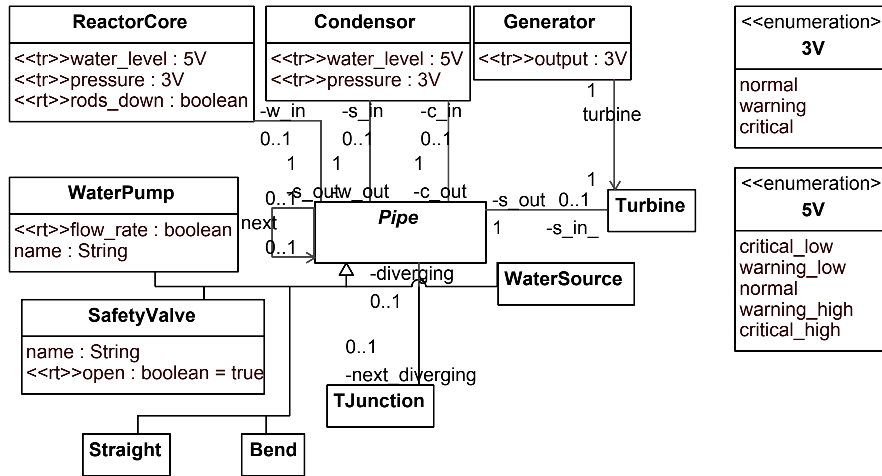

Fig. 13 The simplified and annotated metamodel (excerpt)

actual value of the water level, but can still be used for analysis, as we are only interested in what 'state' the water level is in).

- Conditions on transitions are reduced to denote which of the enumeration values yields *true* (as a set of enumeration literals).
- The attribute denoting number of seconds in the *after* attribute of the association is removed, since no concept of real time will be used in the properties.

Other simplifications to reduce the state space, such as rearranging the transformation schedule and breaking down metamodel hierarchy, are beyond the scope of this chapter. The *ProMoBox* approach includes guidelines for simplification.

Once all necessary simplifications are performed, annotations need to be added to the DSML's metamodel. Annotations add semantics to elements in the metamodel (classes, associations and attributes), denoting whether elements are *static* (do not change at runtime), *dynamic* (change at runtime, thus reflect the state of the system), or are part of the environment. To achieve this, annotations mark in which of the five languages these elements should be available. By default, elements are not annotated, meaning that they appear in the design, runtime, output, and property languages, but not in the input language.

Three annotations are provided:

- «*rt*»: runtime, annotates a dynamic concept that serves as output (*e.g.,* a state variable);
- «*ev*»: an event, annotates a dynamic concept that serves as input only (*e.g.,* a marking);
- «*tr*»: a trigger, annotates a static or dynamic concept that also serves as input (*e.g.,* a transition event).

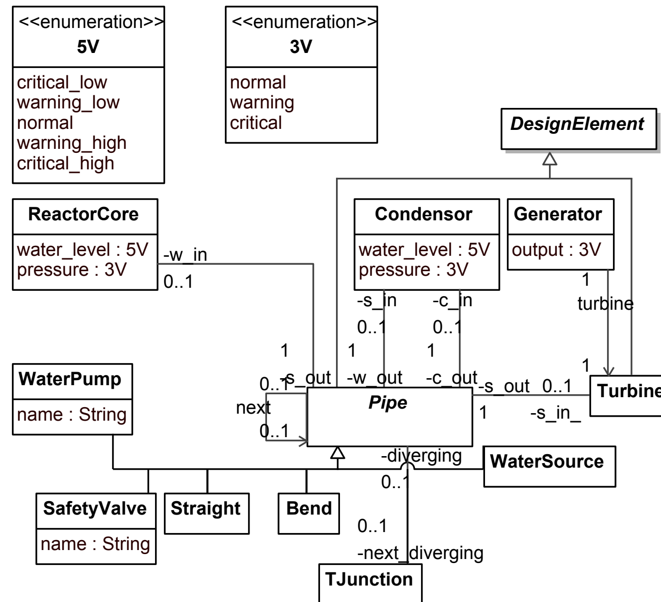

Fig. 14 The generated *NPPC* design language $NPPC_D$ (excerpt).

Other annotations can be added to the annotations model, as long as certain constraints are met (*e.g.,* availability in the design language implies availability in the runtime language).

In Figure 13 we show some attribute annotations. The attributes *flow_rate*, *open*, *rods_down* and *current* represent the observable state of the nuclear power plant and are therefore annotated with «*rt*». The attributes *water_level* and *pressure* can be changed by the environment and are therefore annotated with «*tr*». Classes, associations, and attributes such as *name* and *initial* are not annotated, meaning that they are only part of the static structure of the model.

The concrete syntax of the simplified DSML requires slight adaptation, such that the enumeration values are shown instead of the integers. The simplified and annotated metamodel, of which an excerpt is shown in Figure 13, contains sufficient information to generate the *ProMoBox* consisting of five languages.

4.2 ProMoBox *Generation Phase*

A family of five languages is generated from the annotated metamodel using a template-based approach. For every language, the elements of the annotated metamodel according to the annotations are combined with a predefined template.

For example, Figure 14 shows the metamodel of the generated *design language* $NPPC_D$. Generic template elements (shown as grey rectangles in Figure 15) are

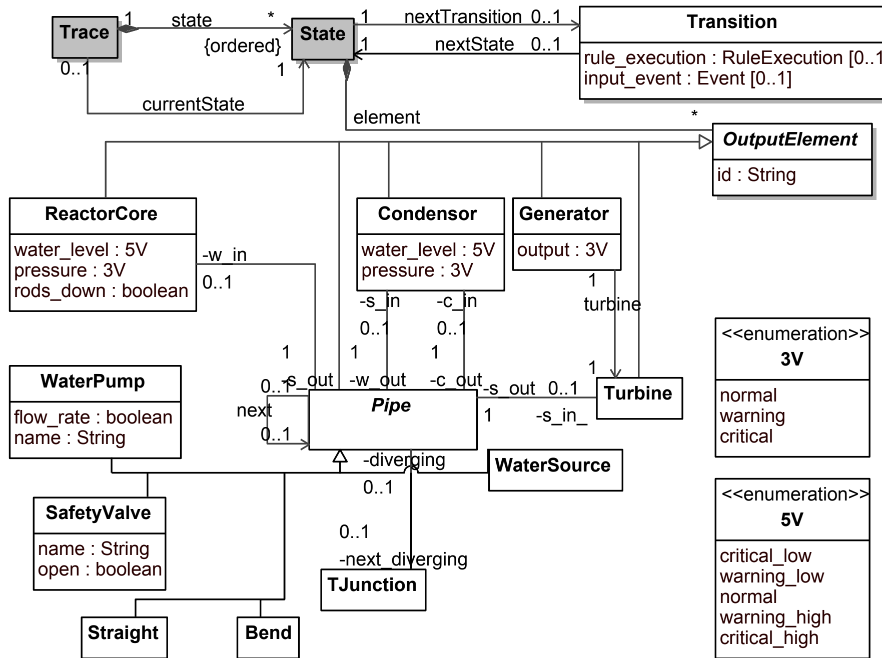

Fig. 15 The generated *NPPC* output language $NPPC_O$ (excerpt), shaded elements are part of the template.

combined with the DSL metamodel elements through the use of inheritance. The template consists of one simple class *DesignElement* with an attribute *id*, used for traceability. All DSML classes transitively inherit from this single class. Some attributes, such as *flow_rate*, *open*, *rods_down* and *current*, are left out of the design language. They represent state information that is only part of the running system, and not of the static structure of the system. Since the design language consists solely of static information, attribute values of the above attributes are not visible, nor available, in the design language's instance.

The metamodel of the *runtime language* $NPPC_R$ (not shown) is similar. It does, however, include the aforementioned attributes such that the state of the system can be represented with this language. An instance of the runtime language looks similar to an instance of the design DSML (shown in Figure 4), where the current state is marked and attribute values are shown.

Figure 15 shows the metamodel of the generated *output language* $NPPC_O$. The template represents an output *Trace* with *Transition*s between system *State*s. These *State*s contain *OutputElement*s representing a runtime state of the modelled system, such that a *Trace* represents a step-by-step execution trace. As the annotations dictate, all elements of the annotated metamodel are present in $NPPC_O$. Instances of the input ($NPPC_I$) and output ($NPPC_O$) languages (not shown) look like a sequence of runtime language instances, but can also be left implicit due to spatial constraints.

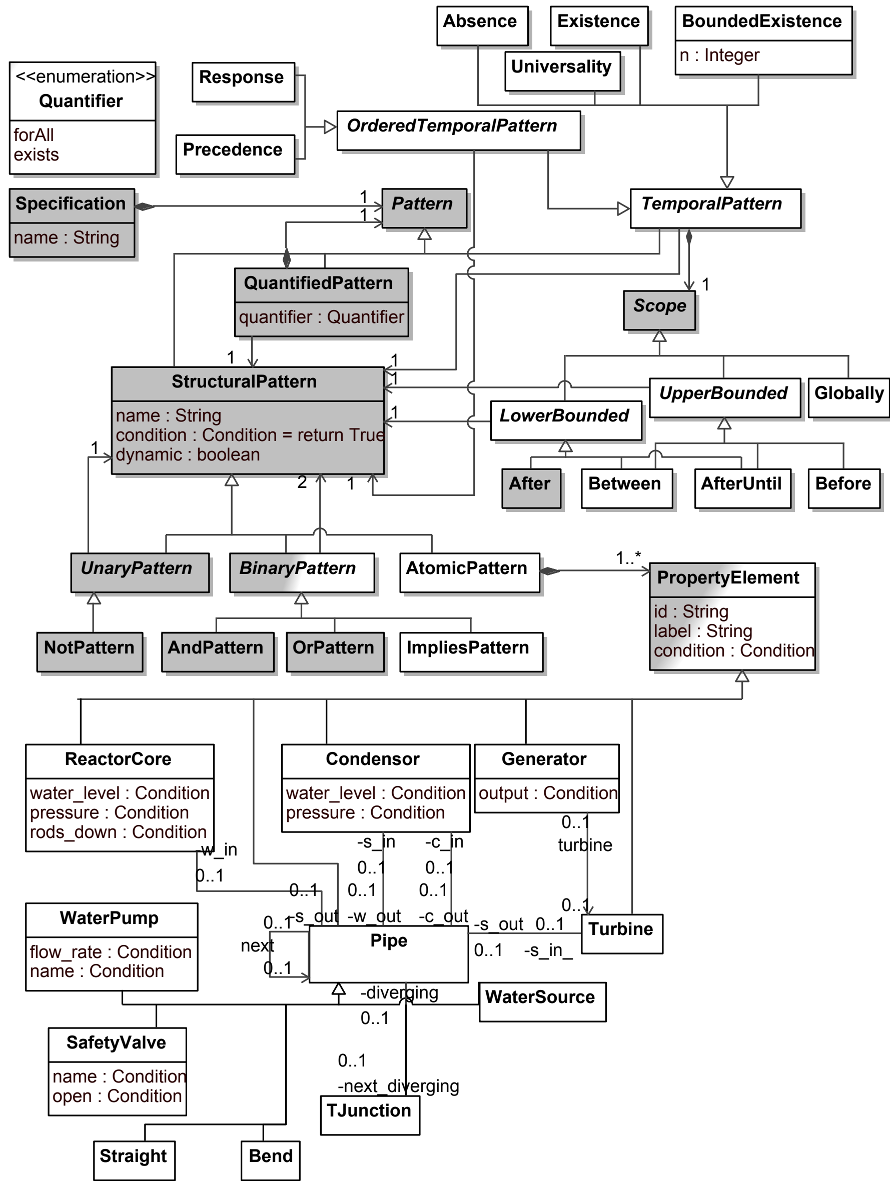

Fig. 16 The generated *NPPC* properties language *NPPC*$_P$ (excerpt), shaded elements are part of the template.

They can be “played out” step by step on the modelled system to replay the execution it represents.

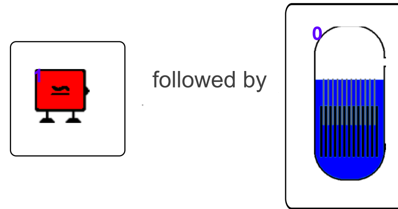

Fig. 17 Property 1: *if the generator is in the error state, the rods will eventually be lowered.*

The generated metamodel of *property language* $NPPC_P$, shown in Figure 16, allows the definition of temporal properties over the system behaviour by means of four constructs:

- quantification (∀ or ∃) allows the user to specify whether the following temporal property should apply to *all* or *at least one* match of a given pattern over the design model;
- temporal patterns (based on [5]) allow the user to choose from a number of intuitive temporal operators over occurrences of a given pattern in a state of the modelled system, such as *after the reactor is in the* low water *state, it should eventually go back to the* normal *state*. These temporal patterns can be scoped (*i.e.,* should only be checked) within certain occurrences of a given pattern;
- structural patterns (based on [6, 7]) allow the user to specify patterns over a single state of the modelled system, such as *a component is in* warning *state*. Structural patterns can be composed, but are always evaluated on a single state of the modelled system;
- instead of (a subset of) the DSML, a pattern language of the DSML (generated using the RAMification process [15, 27]) is used, such that domain-specific patterns rather than models can be specified. The effects of this are reflected in $NPPC_P$, where all attribute types are now *Condition*s (expressions that return a boolean), all abstract classes are concrete, and the lower bounds of association cardinalities is relaxed.

The resulting language has a concrete syntax similar to the DSML, and the quantification and temporal patterns (instead of LTL operators) are raised to a more intuitive level through the use of natural language.

4.3 Specifying and Checking Properties Using **ProMoBox**

Figures 17 and 18 show properties specified in the property language $NPPC_P$. These patterns reuse domain-specific language elements to allow the specification of patterns, and ultimately properties in a domain-specific syntax. Figure 17 shows a *Re-*

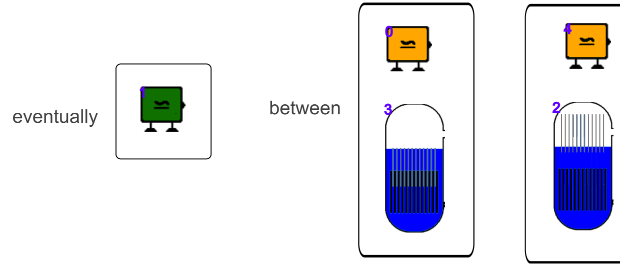

Fig. 18 Property 2: *if the generator is in the warning state, the rods can only become raised if the generator passed through the normal state*.

sponse pattern, and Figure 18 shows an *Existence* pattern, bounded by an *Upper* and *Lower* bound.

As shown in Figure 12, properties specified in the property language are compiled to LTL, and the compiler produces a Promela model [11] that includes a translation of the initialised system, the environment, and the rule-based operational semantics of the system. This translation is generic, and thus independent of the DSML. The properties are checked by the SPIN model checker, which evaluates every possible execution path of the system. In case of Figure 17, no counterexample is found, meaning that the system satisfies this property in every situation.

For the property modelled Figure 18, however, a counterexample is found which we'll discuss in more detail. The property is translated to the following LTL formula: [](Q && !R -> (!R *W* (P && !R))) where:

- P = *generator.state == normal*
- Q = *generator.state == warning && reactor.state == rods_down*
- R = *generator.state == warning && reactor.state == rods_up*
- Operator *W* is "weak until" and defined as p *W* q = ([]p) || (p U q)

The counterexample found shows it is possible for two different components to go into the warning state, after which only one of them goes back to the normal state. If that happens, the controller will go back to the normal state, as there is no counter to store how many components are in the warning state. In this normal state, it is possible to raise the rods further, without having the generator go to the normal state again, causing the property to be violated. This counterexample can be played out with SPIN to produce a textual execution trace, which is translated back to the DSM level as an instance of the output language. This execution trace can be played out on the modelled system, by visualising the states of the traces in the correct order.

The limitations of the framework are related to the mapping to Promela, as explained in [17]. In its current state, *ProMoBox* does not allow dynamic structure models. Because of the nature of Promela and the compiler's design, boundedness of the state space is ensured in the generated Promela model.

5 Conclusion

In this chapter, we motivated the use of Domain-Specific Modelling (DSM) for the development of complex, reactive, real-time, software-intensive systems. Model-Driven Engineering, and in particular DSM, closes the conceptual gap between the problem domain and solution (implementation) domain.

We presented the different aspects of a Domain-Specific Modelling Language:

1. abstract syntax to define the allowable constructs;
2. concrete syntax to define the visual representation of abstract syntax constructs;
3. semantic domain to define the domain in which the semantics is expressed;
4. semantic mapping to define the mapping to a model in the semantic domain that defines the (partial) semantics of the domain-specific model.

Each of these aspects was explained and applied to our case study: modelling the behaviour of a nuclear power plant controller and its subcomponents. The behaviour of the nuclear power plant interface is defined using both operational semantics ("simulation") and translational semantics ("mapping").

We extended our discussion to property checking, which was also lifted to the level of the problem domain. This enables domain experts to not only create models and reason about them, but also to specify properties on them. Errors, or any other kind of feedback, are also mapped back onto the problem domain, meaning that the domain expert never has to leave his domain. All aspects of modelling were thus pulled up from the solution domain, up to the problem domain, closing any conceptual gaps. Moreover, a generative approach is used, in order to limit development overhead.

Acknowledgments. This work was partly funded by a PhD fellowship from the Research Foundation - Flanders (FWO) and the Agency for Innovation by Science and Technology in Flanders (IWT). Partial support by the Flanders Make strategic research centre for the manufacturing industry is also gratefully acknowledged.

References

1. Ankica Barišić, Vasco Amaral, Miguel Goulão, and Bruno Barroca. Quality in use of domain-specific languages: A case study. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU '11, pages 65–72. ACM, 2011.
2. F. Boulanger, C. Hardebolle, C. Jacquet, and D. Marcadet. Semantic adaptation for models of computation. In *Application of Concurrency to System Design (ACSD), 2011 11th International Conference on*, pages 153–162, June 2011.
3. F. E. Cellier. *Continuous system modeling*. Springer-Verlag, 1991.
4. Gennaro Costagliola, Vincenzo Deufemia, and Giuseppe Polese. A framework for modeling and implementing visual notations with applications to software engineering. *ACM Transactions on Software Engineering Methodology*, 13(4):431–487, 2004.

5. Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Int'l Conf. Software Engineering*, pages 411–420, 1999.
6. Esther Guerra, Juan de Lara, Dimitrios S. Kolovos, and Richard F. Paige. A Visual Specification Language for Model-to-Model Transformations. In *VL/HCC*, 2010.
7. 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. *Automated Software Engineering*, 20(1):5–46, 2013.
8. David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
9. David Harel and Amnon Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering Methodology*, 5(4):293–333, 1996.
10. David Harel and Bernhard Rumpe. Meaningful modeling: What's the semantics of "semantics"? *Computer*, 37(10):64–72, 2004.
11. Gerard J. Holzmann. The Model Checker SPIN. *Transactions on Software Engineering*, 23(5):279–295, 1997.
12. Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley, 2008.
13. Anneke Kleppe. A language description is more than a metamodel. In *Fourth International Workshop on Software Language Engineering*, 2007.
14. Thomas Kühne. Matters of (Meta-)Modeling. *Software and System Modeling*, 5:369–385, 2006.
15. Thomas Kühne, Gergely Mezei, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer. Explicit transformation modeling. In *MoDELS Workshops*, volume 6002 of *Lecture Notes in Computer Science*, pages 240–255. Springer, 2009.
16. Levi Lucio, Sadaf Mustafiz, Joachim Denil, Hans Vangheluwe, and Maris Jukss. FTG+PM: An integrated framework for investigating model transformation chains. In *SDL 2013: Model-Driven Dependability Engineering*, volume 7916 of *Lecture Notes in Computer Science*, pages 182–202. Springer, 2013.
17. 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.
18. Bart Meyers and Hans Vangheluwe. A multi-paradigm modelling approach for the engineering of modelling languages. In *Proceedings of the Doctoral Symposium of the ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems*, CEUR Workshop Proceedings, pages 1–8, 2014.
19. Bart Meyers, Manuel Wimmer, Hans Vangheluwe, and Joachim Denil. Towards domain-specific property languages: The ProMoBox approach. In *Proceedings of the 2013 ACM Workshop on Domain-specific Modeling*, DSM '13, pages 39–44, New York, NY, USA, 2013. ACM.
20. Daniel Moody. The "physics" of notations: Toward a scientific basis for constructing visual notations in software engineering. *IEEE Transactions on Software Engineering*, 35(6):756–779, 2009.
21. Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.
22. Marian Petre. Why looking isn't always seeing: Readership skills and graphical programming. *Communications of the ACM*, 38(6):33–44, 1995.
23. Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.
24. Matteo Risoldi. *A methodology for the development of complex domain-specific languages*. PhD thesis, University of Geneva, 2010.

25. Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, 2003.
26. Margaret H. Smith, Gerard J. Holzmann, and Kousha Etessami. Events and constraints: A graphical editor for capturing logic requirements of programs. In *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering*, RE ’01, pages 14–22, Washington, DC, USA, 2001. IEEE Computer Society.
27. Eugene Syriani. *A Multi-Paradigm Foundation for Model Transformation Language Engineering*. PhD thesis, McGill University, Canada, 2011.
28. Eugene Syriani and Hans Vangheluwe. A modular timed graph transformation language for simulation-based design. *Software and System Modeling*, 12(2):387–414, 2013.
29. Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Simon Van Mierlo, and Hüseyin Ergin. AToMPM: A web-based modeling environment. In *Proceedings of MODELS’13 Demonstration Session*, pages 21–25, 2013.
30. Hans Vangheluwe. Foundations of modelling and simulation of complex systems. *ECEASST*, 10, 2008.
31. Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of Modeling and Simulation*. Academic Press, second edition, 2000.
32. Yingzhou Zhang and Baowen Xu. A survey of semantic description frameworks for programming languages. *SIGPLAN Notices*, 39(3):14–30, 2004.