Industrial Application of Formal Models Generated from Domain Specific Languages

Jozef Hooman[1,2(✉)]

[1] Embedded Systems Innovation by TNO, Eindhoven, The Netherlands
jozef.hooman@tno.nl
[2] Radboud University, Nijmegen, The Netherlands

Abstract. Domain Specific Languages (DSLs) provide a lightweight approach to incorporate formal techniques into the industrial workflow. From DSL instances, formal models and other artefacts can be generated, such as simulation models and code. Having a single source for all artefacts improves maintenance and offers a high return on investment of the initial modelling effort. Since DSLs can be used to capture essential domain information at a high level of abstraction, this supports formal verification early in the development process. We discuss our experiences with this approach in a number of real industrial development projects.

1 Introduction

Many companies suffer from a long test and integration phase. The main reason is that a large number of problems are detected in this phase. Repairing these problems is often not easy and corrections might lead to new problems. Hence, it is difficult to manage this phase and to predict when it can be completed.

Our approach aims at detecting problems much earlier in the development process by means of various modelling techniques. Although it is well-known that it is very cost-effective to detect problems as early as possible (see, e.g., [31]), the main challenge is to realize this in an existing industrial development process with continuous pressure to meet deadlines. Therefore our approach tries to reduce the modelling effort by reusing models for multiple purposes, such as performance analysis and the generation of configuration files, tests or code.

In this paper we concentrate on the use of formal techniques to increase the confidence in the correctness of the models. In the industrial context, the use of lightweight formal methods has been advocated frequently [13,17]. These methods hide a lot of the mathematical details from the user and do not aim at generic modelling and analysis techniques. By specializing on a particular type of design pattern and a particular set of properties, more efficient and effective approaches can be developed.

As a starting point, we analyse the industrial use of the Analytical Software Design (ASD) approach [24] which is supported by commercial tooling. This approach combines a restricted tabular notation for component behaviour, formal

Supported by the Dutch national program COMMIT.

E. Ábrahám et al. (Eds.): de Boer Festschrift, LNCS 9660, pp. 277–293, 2016.
DOI: 10.1007/978-3-319-30734-3_19

verification of a limited number of properties, and code generation. The evaluation is based on a number of industrial development projects and a comparison with Uppaal [9].

Based on these experiences, we experiment with the use of Domain Specific Languages (DSLs). Recent DSL technology allows a fast definition of a dedicated language, the automatic generation of a powerful editor for this language, and a convenient mechanism to generate text (e.g., analysis models or code) from language instances. We report about experiences with DSLs in combination with various formal methods in a number of real industrial development projects.

Related Work. Related to ASD are commercial tools that combine formal methods and code generation. VDMTools[1] supports code generation from models specified using the VDM++ language [11]. The tool Atelier B[2] has been used to develop a number of safety-critical systems using the B-method [1]. The SCADE Suite[3] provides formal techniques for specification, verification and code generation. These techniques are quite generic and the correctness proofs for VDM and B models may require interactive theorem proving. ASD is much more restricted than the approaches mentioned above to achieve a high level of automation and to support compositional verification.

The use of DSLs has been proposed for more than a decade, see e.g. the overview in [30]. An early experiment to combine DSLs and formal methods has been described in [4]. In that paper, the correctness of instances of a DSL for process scheduling is verified using the B method. To increase the use of formal methods in industry, [10] proposes the encapsulation of formal methods within domain specific languages. A DSL of the railway domain is formalized by means of the algebraic specification language CASL [16]. Recent developments of the DSL technology make it feasible to apply this on a much larger industrial scale.

Industrial Context. Most of the work reported here was done at Philips Healthcare, with a focus on interventional X-Ray (iXR) systems, see Fig. 1. These systems are used for minimally invasive surgery, for instance, improving the throughput of a blood vessel by placing a stent via a catheter where the surgeon is guided by X-ray images. These techniques avoid, for instance, open heart surgery.

Overview. This paper is structured as follows. The ASD approach is evaluated in Sect. 2, which also includes a comparison to Uppaal. Based on our observations, Sect. 3 describes our approach to combine formal methods and DSLs in industry. Section 4 contains three industrial applications at Philips on components of iXR systems. Concluding remarks can be found in Sect. 5.

[1] http://www.vdmtools.jp/en/.
[2] http://www.atelierb.eu/en/.
[3] http://www.esterel-technologies.com/products/scade-suite/.

Fig. 1. Interventional X-ray system

2 Experiences with ASD

Section 2.1 contains a brief overview of the ASD method. More explanation, examples, and applications at Philips can be found in [15,21]. A summary of the experiences at Philips with ASD is given in Sect. 2.2. Section 2.3 compares ASD with Uppaal.

2.1 ASD Background

The Analytical Software Design (ASD) method [23,24] is a component-based technology that aims at enabling the application of formal methods into industrial practice. The approach has been supported by the commercial development tool ASD:Suite which embeds ASD into a software design environment. This tool was developed by the company Verum. After a re-start of this company, the approach was recently renamed to Dezyne, but our experiences concern the use of the original ASD method.

ASD Models. Models are represented in ASD by state-transition tables. The ASD approach distinguishes two types of models:

- An *interface model* specifies the external behaviour of a component without referring to any internal behaviour. This forms the formal contract of the interaction between the component and its clients.
- A *design model* implements a certain interface model and typically uses services of other components, the so-called *used components*, by referring to their interface models.

An example of ASD models is depicted in Fig. 2. It shows a part of a design model of a component (DComponent) with fragments of its interface model (IComponent) and two used interfaces (IUsedComp1 and IUsedComp2).

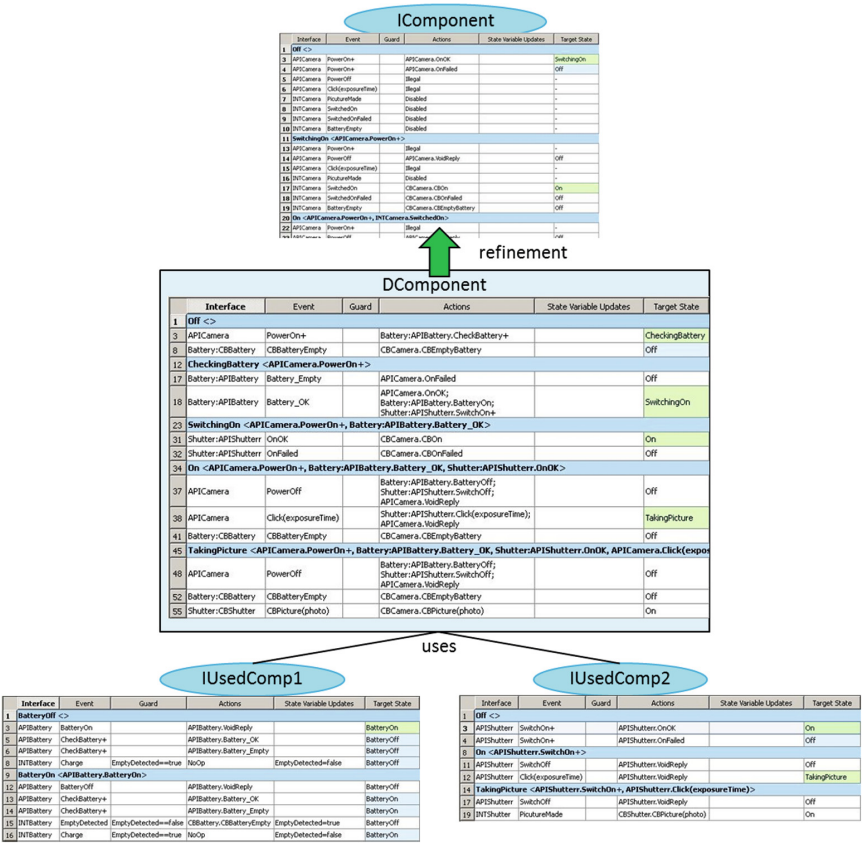


Fig. 2. Example of ASD models

Each model is a state machine, represented by a table. In every state the response to all possible input events has to be specified. It is possible to specify that an event is illegal in a certain state, i.e., that it should not occur in that state.

Design models should be deterministic; non-determinism is allowed in interface models. In ASD, communication between client and server components is asymmetric, using synchronous calls and asynchronous callbacks.

- Clients issue synchronous calls to server components, where the client is blocked until the server accepts the call and eventually returns it to the client.
- Servers can communicate with their clients by asynchronous callbacks. Callbacks are stored in a so-called callback queue (FIFO).

Another restriction is that an ASD component cannot make decisions based on the values of parameters in a received call. Only the names of calls or callbacks can influence the flow of control. Hence, ASD only supports data-independent

control components. Other components have to be implemented in another way, e.g., by manual coding.

Formal Verification. The ASD:Suite automatically translates the ASD tabular specifications into CSP models and verifies them using the formal refinement checker FDR2 [26]. (FDR is an abbreviation of Failures-Divergence Refinement.) All CSP and FDR2 details are hidden from end-users. When a property does not hold, an error trace is represented as a sequence diagram.

Only a fixed set of properties can be verified. The main checks performed by the ASD:Suite are:

- *Consistency checks*: verify whether a design model is deterministic and correctly uses the interfaces of its used components.
- *Refinement checks*: verify whether the interface model of a component is correctly refined by the design model in combination with the interface models of the used components.

Observe that the ASD approach is compositional [8,14], since the refinement checks only use the interface models of the used components. This compositional way of verification avoids the well-known state space explosion problem and enables industrial scalability, because components can be checked in isolation. It requires, however, a careful design of the system such that the design pattern of ASD is used and the components themselves are kept small (to avoid state explosion at the component level).

Code Generation. ASD:Suite supports the generation of code from design models for a number of programming languages (C, C++, C#, Java), which is important for industrial acceptance. This avoids an error prone manual translation from models to code and is expected to improve productivity. Observe that the formal model and the code are both based on the same source, see Fig. 3.

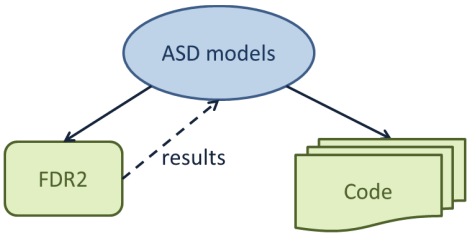


Fig. 3. The ASD approach

2.2 Industrial Application of ASD

The ASD approach has been used at Philips Healthcare in a number of different development projects. We summarize an extensive evaluation published before [22] concerning software quality and productivity:

- Since ASD only checks a limited set of properties, it may not always lead to defect-free software. However, compared to the industry standards, very few defects were found in the code generated by ASD. In general, most defects were easy to find and to fix.
- The data of the projects in which ASD was used indicate an improved productivity compared to industrial standards. This was partly due to the fact that after the modelling phase, verified code is generated automatically. It was also observed that less time was spent integrating and manually testing the generated code. Clearly, ASD prevented problems earlier in the development process. As an example, most developers were impressed by the fast detection of race conditions, because their experience is that these problems are difficult to detect by testing and usually show up very late in the development process with the conventional approach.

The limitations of the ASD method, however, might prevent large-scale introduction into the industrial workflow:

- The approach is limited to event-based control components. It is not suitable for low-level real-time controllers and data-intensive components. Designers find it difficult to decide what to do in ASD and what not.
- ASD assumes a hierarchical control architecture with synchronous method calls from top to bottom and asynchronous callbacks in the other direction. Although this gives a clear structure, it is not always easy to construct such a hierarchy, especially because the size of components and the number of callbacks should be small to allow fast model checking. Moreover, when software engineers are used to object-oriented designs this might require a paradigm shift.
- After a few changes, the state-transition tables might become large and difficult to maintain; there are hardly any structuring mechanisms, e.g., to indicate that a certain transition is common to a set of states.
- There is no systematic means to evaluate and to analyse the complexity of ASD models, e.g., to detect early that model checking might take too long or to decide that refactoring is needed after changes.
- Verification is limited; there is no possibility to express the desired input/output behaviour of a component. For instance, one would like to express that certain input calls lead to specific calls to the used components.
- There is no possibility to simulate a component or the combination of a number of components to validate that the desired behaviour has been modelled.

2.3 Comparison with Uppaal

Based on experiences with ASD, industrial users asked for other formal verification methods without the limitations of ASD. At FEI Company, where ASD is used to develop control software for electron microscopes, we experimented with additional support using Uppaal [9]. Uppaal is an integrated environment for modelling, validation and verification of systems modelled as networks of

timed automata. The Uppaal tool was chosen because of its nice and understandable user interface and the simulation possibilities, which appeared to be attractive for industrial users. The most important reason was the possibility to verify other properties than the ASD checks, which increases the range of faults that can be detected early. The timing aspects of Uppaal have not been used, but might be relevant in later studies.

Uppaal Models. Uppaal uses timed automata with synchronous communication along channels, extended with data types (bounded integers, arrays, etc.). For more details about Uppaal we refer to [2].

Uppaal has been applied to a camera safety system. This system should guarantee that an expensive and very sensitive camera is protected against a too high dose of electrons. An important part of this system is the software that keeps track of the location of the camera, the blocking of the electron beam by other components and the intensity of the electron beam. This part of the software was generated using the ASD approach.

We translated the ASD models to Uppaal. Translating the tabular representation to the automata of Uppaal is rather straightforward. The main issue was that simulation and verification in Uppaal requires a closed set of models which includes the environment of the component(s) under study. We obtained a closed system by using the interface model of an ASD design model. This interface model was translated into an Uppaal model of a client of the component by reversing the direction of sending and receiving.

Formal Verification. Most important difference with ASD is that Uppaal allows the verification of user-defined properties which have to be expressed in a version of temporal logic. In general, we concentrated on properties that had not yet been verified by the standard checks of ASD:Suite. The properties to be verified have been defined in cooperation with the software architect and the system architect. By means of this verification, two major issues were found. These issues could not be found in the ASD approach since it does not allow this type of verification.

A disadvantage of above approach is that the temporal logic expressions are not easy to read by industrial users. As an alternative, we experimented with the use of observers, similar to [5]. An observer is an additional parallel automaton which observes the communication between the other automata and enters an error location when an incorrect trace is observed. This approach was more convenient for our industrial users.

Comparison from Industrial Perspective. We observed that ASD and Uppaal are complementary in many respects. The industrial engineers appreciate the possibilities to simulate the Uppaal models. This especially concerns the joint simulation of a number of components, which was clearly missing in ASD:Suite. A disadvantage, compared to the compositional approach of ASD, is that one more easily encounters the state explosion problem, so scalability of Uppaal is limited.

The larger range of verification possibilities of Uppaal is a clear advantage, although expressing properties in temporal logic is not very convenient for industrial users. The timing properties of Uppaal have not been used in our experiment, but the possibility to express timing is seen as a valuable asset. A strong point of ASD is the generation of code from a verified model. The fact that different programming languages are supported provides a kind of platform independence which is important to enable future technology changes.

3 Domain Specific Languages and Formal Methods

The experiences with ASD and Uppaal indicate that one would like to combine the strong points of different techniques. Making transformations between models of different formalisms, such as ASD and Uppaal, would be very time consuming. Typically, subtle semantic differences make it very hard to define a correct generic transformation. Given our experiences with the use of Domain Specific Languages (DSLs) to define the behaviour of frequently changing components at a high level of abstraction [19], we have combined this with the generation of formal models.

Our approach is based on Xtext[4], an Eclipse plugin on top of the Eclipse Modelling Framework. Based on the definition of a grammar, the Xtext plugin generates a meta-model, a parser and an Eclipse-based editor for the language defined by the grammar. Moreover, it generates convenient starting points to implement validation, scoping, and the generation of text (such as code and models) using the Xtend language[5] [3].

In addition to formal verifications tools, we also use simulations based on POOSL (Parallel Object Oriented Specification Language) [27]. POOSL is a formal modelling language for systems that include both software and digital hardware. The formal semantics of POOSL has been defined in [28] by means of a probabilistic structural operational semantics for the process layer and a probabilistic denotational semantics for the data layer. The operational semantics of POOSL has been implemented in a high-speed simulation engine called Rotalumis. Recently, by means of Xtext, a modern Eclipse IDE has been developed on top of an improved Rotalumis simulation engine[6].

For the (re)design of a component we proceed along the following steps:

1 Define a grammar in Xtext to capture the essential domain concepts.
2 Define an instances of the language defined in the previous step. Discuss this instance with domain experts to obtain a first definition of the required behaviour. When needed, the grammar is adapted.
3 Implement validation rules to check well-formedness properties of language instances.

[4] http://eclipse.org/Xtext/.
[5] http://eclipse.org/xtend/.
[6] http://poosl.esi.nl.

4 Implement a generator which yields a POOSL model that can be used to simulate the intended behaviour. Often we connect the simulated POOSL model by means of a socket to a visualization (e.g. in Java) of the externally visible behaviour. Adapt language instances after feedback of industrial stakeholders.
5 Implement a generator which yields a formal model that enables formal verification. We have used various formal techniques for this step. Clearly, language instances are adapted when errors are found.
6 Implement a generator which yields code, configuration files, or tests, depending on the industrial needs.

4 Applications of Combining DSLs with Formal Methods

The approach described in the previous section has been applied to three components of the iXR system introduced in Sect. 1. We have developed a DSL for collision prevention in combination with an SMT solver (Sect. 4.1), a DSL for power control, supported by SAL (Sect. 4.2) and a DSL for pedal handling with formal verification by means of mCRL2 (Sect. 4.3).

4.1 DSL Collision Prevention and SMT Solver

The first project is related to the moving parts of an iXR system. Such a system consists of one or two so-called C-arms, each carrying an X-ray generator and a detector. During the treatment of a patient, the C-arms, the detectors, and the patient table can be moved to obtain optimal projections for the images. Safety of an iXR system includes the avoidance of collisions between these heavy physical objects and with humans, such as patient and medical staff.

The goal of the project was to re-develop the collision prevention components in order to facilitate systematic reuse of safety-critical software across product configurations and medical applications. An overview is given in [20], details about the formal approach can be found in [18].

To stay close to the requirements formulation, we have developed a DSL which is targeted at the type of rules we want to express. These rules basically specify restrictions on the speed of the moving parts when the distances between these parts are below certain thresholds or when distance sensors detect an object (e.g., patient or medical staff). A part of a DSL instance is depicted in Fig. 4.

Basic validation checks have been implemented by means of the validation mechanism of Xtext. This includes, for instance, type checking and consistency checks on the hardware configuration. Clearly such checks are limited and there is a need for more analysis before generating code and performing time-consuming system tests. For the collision prevention component, the focus was on two types of analysis: performance evaluation of the required execution times and formal verification of correctness properties. This has been achieved by defining transformations to analysis models. Additionally, code has been generated, as shown in Fig. 5. We briefly discuss these three ingredients.

```
restriction ApproachingTableTopAndBeam
    activation
        Distance(TableTop, Beam) < 35 mm + 15 cm
    effects
         UserGuidance "TableTop and Beam approaching"
          relative limit TableTop*[Rotation, Translation],
                         Beam*[Rotation, Translation],
              at (Distance(TableTop, Beam) - 35 mm) / 15 cm

restriction VeryCloseTableTopAndBeam
    activation
        Distance(TableTop, Beam) < 20 mm
    effects
         UserGuidance "TableTop and Beam very close"
          relative limit TableTop*[Rotation, Translation],
                         Beam*[Rotation, Translation],
              at 0
…
```

Fig. 4. Part of a DSL instance for collision prevention

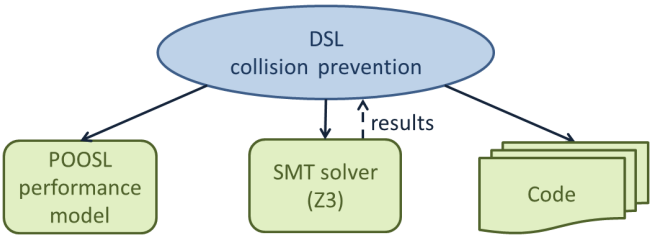


Fig. 5. DSL for collision prevention and transformations

Performance. The collision prevention component is part of a real-time control loop that executes with a certain frequency. Hence it is important that the collision prevention component can execute within the period of the real-time loop. The performance analysis of collision prevention concentrates on the computations needed to compute distances between objects [29]. It uses a generated POOSL model to perform simulations and to obtain statistics about expected execution times. The model uses performance profiles of the basic computation steps. Moreover, it has been calibrated using performance measurements on an existing component.

Formal Verification. To obtain fully automated and fast formal analysis, a generator has been written in Xtend which generates for every DSL instance an SMT (Satisfiability Modulo Theories) model. This model is analysed using the SMT-solver Z3 [7]. If the property does not hold, delta debugging is used to identify the rules in the DSL instance that contribute to the failure. This leads to warnings in the Eclipse-based editor for the DSL at appropriate places.

Formal verification addresses four types of correctness properties:

- Well-definedness of expressions; for instance, absence of division by zero.
- Speed limits are within the specified range.
- Safety of movement control; for instance, if two objects are close to each other and still approaching, then their speeds are restricted.
- Absence of deadlock; there is no position of the objects such that no further movements are possible.

To make formal verification feasible, several abstractions have been applied, e.g., concerning the acceleration characteristics of the physical objects and timing aspects. Using these abstractions, the experiments described in [18] show that fast analysis and user feedback is feasible for realistic instances of the DSL. For the correctness properties mentioned above, the applied abstractions may result in false positives. Moreover, for the deadlock check it may also result in false negatives. Nevertheless, also the deadlock check is useful as it can detect certain typical mistakes in the collision prevention rules.

Code Generation. For the generation of source code, a code generator has been developed that transforms the high-level concepts from our DSL into executable code. By means of some glue code, we have integrated the generated code within the existing system software. The result has been evaluated on the physical system, including all hardware components. This has been used to test whether a specific set of rules has been modelled correctly. Note that the analysis techniques help to find errors earlier, but they cannot detect everything. For instance, movement profiles might have some inaccuracy and heavy physical parts cannot be stopped immediately. After sufficient testing, the code generation can be used for generating production code and then it adds immediate value to the modelling efforts.

4.2 DSL Power Control and SAL

The second application concerns the power control component of an iXR system. This component is responsible for executing power control scenarios, such as start-up, shut down and power failure. During such a scenario the power control component is the master of the system and all other components follow the instructions of the power control component.

The power control component contains a generic part that needs to be configured for every release and every different hardware configuration. In the existing situation, the configuration files are difficult to maintain and to extend. Given the increasing system complexity of the product family, this will likely create problems in future releases. The business goal of the development project is to improve the maintainability and extendibility of the power control component. Additionally, there is a need to improve the existing test set which is very time consuming without having a large coverage.

As before, we define a DSL to express the essential information needed to generate the configuration files automatically from DSL instances. To increase

the confidence in the correctness of DSL instances, we also generate a few analysis models such as a POOSL model, see Fig. 6. Formal verification has been done by means of SAL [12,25], because it also includes convenient support for test generation from a formal model. Instead of generating tests directly, we generate instances of a separate DSL to express test traces. This makes it possible to generate test for various test frameworks, but also test for the POOSL simulation and checks for the SAL models. This can be used to cross check the models and increase the confidence in the generators.

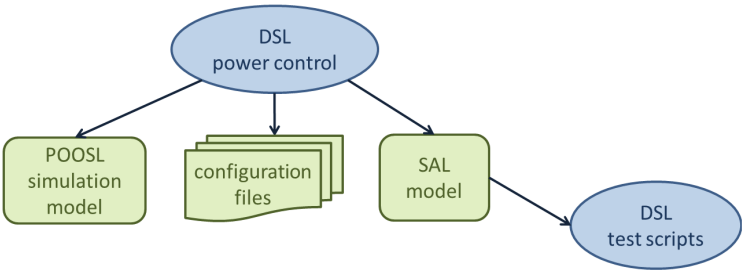


Fig. 6. DSL for power control and transformations

Simulation with POOSL. We implemented a generator which delivers a POOSL model for every DSL instance. By means of a socket, the simulation of a POOSL model is connected to a Java program which provides a Graphical User interface (GUI). This GUI contains buttons to simulate external input and to inject errors. It also shows the resulting power states of the connected devices. This simulation is very useful for a first validation of the model and to align with many parties (architects, designers, suppliers, service engineers, etc.) about the required system behaviour. In this way we detected a number of modelling mistakes and clarified a few issues concerning error handling in the power control component.

Checking Properties with SAL. To obtain exhaustive checks on DSL instances, a generator for SAL models has been implemented. A number of properties has been verified, e.g., expressing that certain groups of devices are in the same power state and that the preconditions for hardware components are fulfilled. By means of SAL, we detected a few additional errors, such as a situation where a device is not powered due to an error in the hardware precondition.

Generating Configuration Files and Tests. From the DSL we generated the configuration files. In addition we used SAL to generated a large number of test scripts. Note that these tests concern the overall system behaviour, including the generic software part, the configuration files and the hardware,

With the generated test cases, approximately twice as much transitions are covered compared to the manually written tests. These manually written tests were also were very time-dependent with many long waiting times. They could

still fail due to a slow response of hardware, which typically resulted in a further increase of the waiting times. By having all concepts described in a clear and concise way using DSLs, we could make the test cases much more efficient. Instead of waiting all the time, the test tool now synchronizes with the power control component and immediately resumes the test case once the control component has reached the desired state.

4.3 DSL Pedal Handling and mCRL2

Our approach has also been applied to the pedal handling component of an iXR system. This component deals with the selection of types of X-ray (high dose or low dose, one or two X-ray sources) and starting and stopping X-ray by means of pedals. Since the current implementation is difficult to maintain, the aim is to refactor the component and partly re-implement it. Also an improvement of the user-perceived behaviour is foreseen. The work described here concentrates on obtaining an unambiguous description of the required behaviour and a good test environment which can be used to test whether new implementations conform to the required behaviour.

In this case, we have defined a DSL to capture the requirements concerning the externally visible behaviour of the component, including the error behaviour. The component has 25 possible input events (including 9 error events) and more than 50 possible output commands. The DSL describes for each input event the resulting output, i.e., the type of X-ray and the status of the user display (e.g., whether a live image is shown, a previously captured image, or a blank screen). A fragment of a DSL instance (changed and simplified for reasons of confidentiality) is shown in Fig. 7.

For each DSL instance we automatically generate a POOSL model to simulate the requirements, an mCRL2 model to verify properties of the model, and a test model suitable for model-based testing. Figure 8(a) contains an overview.

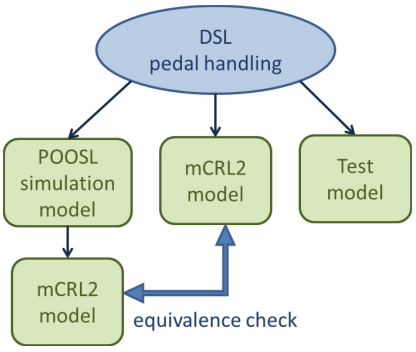
Simulation with POOSL. Similar to the previous section, the generated POOSL model has been coupled by means of a socket to a Java GUI which allows a simulation of pedal presses and the injection of errors. It shows the resulting X-ray and the display of images, as shown in Fig. 8(b). This visualization was very helpful to discuss unclear scenarios, especially in case of errors. It has been used extensively to discuss new behaviour with system architects and designers, resulting in significant changes of error handling behaviour. The simulation will be used as a reference for the implementation of the new behaviour.

Formal Verification with mCRL2. To obtain exhaustive checks on the requirements specification, we generate mCRL2 models. mCRL2 is a process algebra with extensions for data and time. The supporting toolset [6] includes the formal verification of processes with respect to properties expressed in a modal μ-calculus. To avoid that industrial users have to use this logic, we have extended the DSL with a simple language to express safety properties. This is sufficient to express important properties such as “no X-ray is generated if no pedal is pressed”. A few errors in the requirements have been detected in this way.

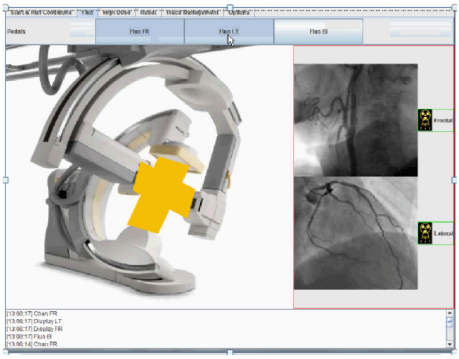
```
trigger: LowOn guard: LowReq == false
 do: LowReq := true ;
     if NOT HighReq
     then if LowOK then OutputType := Fluo ;
                        Display := Live ;
          else OutputType := Standby ; Display := Blank ;
          fi
     fi
…
trigger: HighOff guard: HighReq == true
 do: HighReq := false ;
     if LowReq
     then if LowOK then OutputType := Fluo ;
                        Display := Live ;
          else OutputType := Standby ; Display := Blank ;
          fi
     else OutputType := Standby ; Display := Prev ;
     fi

trigger: LowError guard: LowOK == true do: LowOK := false ;
…
```

Fig. 7. Fragment of a DSL instance for pedal handling



(a) DSL for Pedal Handling and Transformations



(b) Simulation with POOSL

Fig. 8. Pedal handling

Observe that the translation to mCRL2 defines a formal semantics of our DSL. Discussions on the mCRL2 generator clarified a few points about the precise meaning of our DSL, e.g., concerning the level of atomicity. To increase our confidence in the generators we have used an existing translator for a subset of POOSL (covering the generated POOSL models) to mCRL2. We showed that, for a number of DSL instances, the mCRL2 model of the direct DSL to mCRL2 translation is bisimulation equivalent to the result of the combined DSL to

POOSL and POOSL to mCRL2 translations. These three transformations have been implemented by three different persons.

Model-Based Testing. Finally, we have defined a generator which translates DSL instances into a state machine represented in the Axini Modelling Language. The TestManager of the company Axini uses this model to perform model-based testing on an implementation. We have used this technique to validate our requirements model for the existing implementation. It led to a few adaptations of our DSL instance. When the new requirements model for the enhanced pedal handling behaviour has been fixed, we automatically obtain a test environment for the new implementation.

5 Concluding Remarks

Our experience in real industrial development projects indicates that the current DSL technology allows a fast and convenient introduction of formal methods. We observe two main categories of DSLs: (1) a DSL which expresses requirements and finally leads to tests; (2) a DSL which expresses the behaviour of a design, finally leading to code. In both cases, the generation of simulation models in combination with a visualization of externally visible behaviour is very important to align with many stakeholders such as users, marketing, system architects, and engineers. Next, the generation of formal models and exhaustive verification is useful to check consistency and important domain properties. Often it is convenient to include an easy-readable definition of properties in the DSL. An advantage of the DSL approach is that it leads to models that are already at a high level of abstraction. If needed, additional abstractions can be made in the generator.

Another advantage is that a DSL instance is the source of all generated artefacts. Any change in the DSL instance automatically leads to an update of all these artefacts. This avoids the usual maintenance problem to keep formal models consistent with the frequent changes in an industrial context. In this context it is important that there are a number of industrial benefits independent of formal techniques, such as easy changeable domain knowledge, platform independence, early simulation and validation of behaviour, and automatic generation of code or tests. Then the use of formal techniques is a small investment which fits easily in the overall approach and has additional benefits.

Acknowledgments. This paper summarizes results of earlier papers and collaborations with many people from Philips (including Mathijs Schuts, Robert Huis in 't Veld, and Rob Albers), the Eindhoven University of Technology (Ammar Osaiweran, Sarmen Keshishzadeh), and TNO-ESI colleagues (Arjan Mooij, Richard Doornbos). Many thanks goes to all of them for the very pleasant collaboration. The anonymous reviewers are acknowledged for several useful comments.

References

1. Abrial, J.-R.: The B-book: Assigning Programs to Meanings. Cambridge University Press, New York (1996)
2. Behrmann, G., David, A., Larsen, K.G.: A tutorial on Uppaal. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
3. Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend. Packt Publishing Ltd., United Kingdom (2013)
4. Bodeveix, J.-P., Filali, M., Lawall, J., Muller, G.: Formal methods meet domain specific languages. In: Romijn, J.M.T., Smith, G.P., van de Pol, J. (eds.) IFM 2005. LNCS, vol. 3771, pp. 187–206. Springer, Heidelberg (2005)
5. Bozga, M., Graf, S., Ober, I., Ober, I., Sifakis, J.: The IF toolset. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 237–267. Springer, Heidelberg (2004)
6. Cranen, S., Groote, J.F., Keiren, J.J.A., Stappers, F.P.M., de Vink, E.P., Wesselink, W., Willemse, T.A.C.: An overview of the mCRL2 toolset and its recent advances. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013 (ETAPS 2013). LNCS, vol. 7795, pp. 199–213. Springer, Heidelberg (2013)
7. de Moura, L., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
8. de Roever, W.-P., de Boer, F., Hanneman, U., Hooman, J., Lakhnech, Y., Poel, M., Zwiers, J.: Concurrency Verification: Introduction to Compositional and Non-compositional Methods. Cambridge University Press, New York (2001)
9. Doornbos, R., Hooman, J., van Vlimmeren, B.: Complementary verification of embedded software using ASD and Uppaal. In: Proceedings 8th International Conference on Innovations in Information Technology (IIT 2012), pp. 60–65 (2012)
10. Eakman, G., Reubenstein, H., Hawkins, T., Jain, M., Manolios, P.: Practical formal verification of domain-specific language applications. In: Havelund, K., Holzmann, G., Joshi, R. (eds.) NFM 2015. LNCS, vol. 9058, pp. 443–449. Springer, Heidelberg (2015)
11. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs For Object-oriented Systems. Springer, London (2005)
12. Hamon, G., de Moura, L., Rushby, J.: Automated Test Generation with SAL. CSL Technical Note, SRI International, January 2005
13. Heitmeyer, C.L.: On the need for *practical* formal methods. In: Ravn, A.P., Rischel, H. (eds.) FTRTFT 1998. LNCS, vol. 1486, pp. 18–26. Springer, Heidelberg (1998)
14. Hooman, J.: Specification and Compositional Verification of Real-Time Systems. LNCS, vol. 558. Springer, Heidelberg (1991)
15. Hooman, J., Huis in 't Veld, R., Schuts, M.: Experiences with a compositional model checker in the healthcare domain. In: Liu, Z., Wassyng, A. (eds.) FHIES 2011. LNCS, vol. 7151, pp. 93–110. Springer, Heidelberg (2012)
16. James, P., Roggenbach, M.: Encapsulating formal methods within domain specific languages: A solution for verifying railway scheme plans. The Computing Research Repository, abs/1403.3034 (2014)
17. Jones, C.B., Jackson, D., Wing, J.: Formal methods light. Computer **29**(4), 20–22 (1996)
18. Keshishzadeh, S., Mooij, A.J., Mousavi, M.R.: Early fault detection in DSLs using SMT solving and automated debugging. In: Hierons, R.M., Merayo, M.G., Bravetti, M. (eds.) SEFM 2013. LNCS, vol. 8137, pp. 182–196. Springer, Heidelberg (2013)

19. Mooij, A.J., Hooman, J., Albers, R.: Gaining industrial confidence for the introduction of domain-specific languages. In: Proceedings of IEESD 2013, pp. 662–667. IEEE Computer Society (2013)
20. Mooij, A.J., Hooman, J., Albers, R.: Early fault detection using design models for collision prevention in medical equipment. In: Gibbons, J., MacCaull, W. (eds.) FHIES 2013. LNCS, vol. 8315, pp. 170–187. Springer, Heidelberg (2014)
21. Osaiweran, A., Schuts, M., Hooman, J.: Experiences with incorporating formal techniques into industrial practice. Empirical Softw. Eng. **19**(4), 1169–1194 (2014)
22. Osaiweran, A., Schuts, M., Hooman, J., Groote, J.F., van Rijnsoever, B.: Evaluating the effect of a lightweight formal technique in industry. STTT Int. J. Softw. Tools Technol. Transf. (STTT) **18**(1), 93–108 (2016)
23. Broadfoot, G.H.: ASD case notes: costs and benefits of applying formal methods to industrial control software. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 548–551. Springer, Heidelberg (2005)
24. Broadfoot, G.H., Broadfoot, P.J.: Academia and industry meet: some experiences of formal methods in practice. In: Proceedings of the Tenth Asia-Pacific Software Engineering Conference Software Engineering Conference, APSEC 2003, pp. 49–58. IEEE Computer Society (2003)
25. Shankar, N.: Combining theorem proving and model checking through symbolic analysis. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 1–16. Springer, Heidelberg (2000)
26. Formal Systems. Failures-divergences refinement (FDR) (2014)
27. Theelen, B.D., Florescu, O., Geilen, M., Huang, J., van der Putten, P.H.A., Voeten, J.: Software/Hardware engineering with the parallel object-oriented specification language. In: Proceedings of MEMOCODE 2007, pp. 139–148. IEEE (2007)
28. van Bokhoven, L.J.: Constructive tool design for formal languages; from semantics to executing models. Phd thesis, Eindhoven University of Technology, The Netherlands (2004)
29. van den Berg, F., Remke, A., Mooij, A., Haverkort, B.: Performance evaluation for collision prevention based on a domain specific language. In: Balsamo, M.S., Knottenbelt, W.J., Marin, A. (eds.) Computer Performance Engineering. LNCS, vol. 8168, pp. 276–287. Springer, Heidelberg (2013)
30. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. SIGPLAN Not. **35**(6), 26–36 (2000)
31. Westland, J.C.: The cost of errors in software development: evidence from industry. J. Syst. Softw. **62**, 1–9 (2002)