Industrial Application of Domain Specific Languages Combined with Formal Techniques

Mathijs Schuts
Philips, Best
The Netherlands
mathijs.schuts@philips.com

Jozef Hooman
TNO Eindhoven & Radboud University Nijmegen
The Netherlands
jozef.hooman@tno.nl

ABSTRACT

Two Domain Specific Languages (DSLs) have been developed to improve the development of a power control component of interventional X-ray systems of Philips. Configuration files and test cases are generated from instances of these DSLs. To increase the confidence in these instances and the generators, formal models have been generated to analyse DSL instances and to cross-check the results of the generators. A DSL instance serves as a single source from which the implementation and the formal analysis models are generated. In this way, it is easy to maintain the formal support in case of changes and for new product releases. We report about our experiences with this approach in a real development project at Philips.

CCS Concepts

•**Software and its engineering** → **Domain specific languages;** *Embedded software;* State based definitions;

Keywords

Domain Specific Language; Formal methods; Simulation; Test generation

1. INTRODUCTION

We present our experiences with the use of Domain Specific Languages (DSLs) and formal techniques in a real development project. The project concerns medical systems for image guided therapy of Philips. The business goal of the development project is to improve the maintainability and extendibility of the power control component of these systems. The power control component is responsible for executing power control scenarios, such as start-up, shutdown and power failure.

The power control component consists of a generic part that needs to be configured for every release and every different hardware configuration to obtain the desired behaviour. In the existing situation, the configuration files are difficult to maintain and defining extensions is time-consuming and error-prone. Given the increasing system complexity of the product family, this will likely create problems in future releases.

In addition to the use of DSLs, we would like to investigate whether formal techniques could contribute in a structural way to the development of a high quality power control component. This means that the focus is not on a single system but on support for a product family. It should be fairly easy to re-use the formal techniques for new product releases and configurations. In this context, the focus is on a lightweight approach [6, 8]. Since the application of formal techniques is not the main aim of the project, the costs and effort of the use of formal methods should be very low. There is no budget for tools or training.

While re-engineering the configuration of the power control component using a DSL, there was a clear need to test the component, i.e., the newly generated configuration files together with the general software framework and the hardware. Since the existing test set was not completely satisfactory, we also developed a DSL to describe test cases. We used formal techniques to generate instances of the test DSL from the configuration DSL automatically. From the test DSL we could easily generate the low-level test scripts.

A global overview of our approach is depicted in Figure 1. A DSL is used to define the behaviour of a component. From this description an implementation and a formal model are generated. The formal model is used to validate the correctness of the described behaviour and to generate test sequences. The test sequences are described as an instance of a second DSL which generates test scripts for a certain test framework.

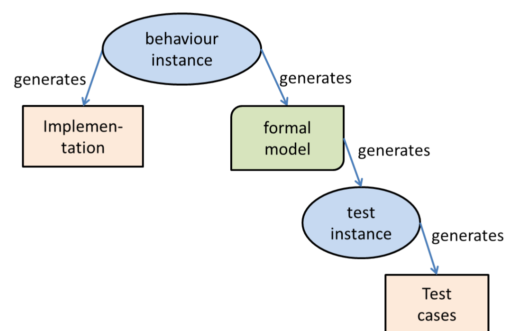

Figure 1: Overview of the approach

An advantage of the DSL approach is that there is a single source (an instance of the DSL) from which both formal models and implementation files are generated. For a new product release or system configuration, only the DSL instance has to be adapted or extended, after which all artefacts are generated automatically.

The central role of a DSL instance implies that its correctness is very important. Hence we generate models for different tools to

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
RWDSL '16, March 12 2016, Barcelona, Spain
© 2016 ACM. ISBN 978-1-4503-4051-9/16/03. . . $15.00
DOI: http://dx.doi.org/10.1145/2889420.2889421

simulate and verify DSL instances. A weak point is that the semantics of the DSL is implicitly defined by the generators and it is not obvious that all generators implement the same semantics. We use several techniques to increase the confidence in the generators. For instance, we verify that system logs, which capture real system usage, are allowed traces of the formal models. Moreover, the test DSL is used to generate tests for these formal models.

The choice of the used tools was very pragmatic. It was based on what was already known by the Philips engineer who did most of the work and the (free) availability of tools. To construct DSLs we used the Xtext plugin of Eclipse [2], supported by a manual [11]. To be able to simulate the behaviour of the power control component based on a DSL instance, we use a translation to POOSL [19], a formal language which is supported by an Eclipse IDE for editing, simulation and debugging. Formal verification has been done by means of SAL [18], because it also includes convenient support for test generation from a formal model.

Related to our approach are applications of DSLs in the domain of embedded systems. For instance, an interesting laboratory experiment of the application of MetaEdit+ to heart rate monitors of Polar [9] shows a large increase in productivity. Xtext has been used to define a DSL which models the hardware configuration of the complex lithography machines of ASML [13]. This DSL supports software-in-the-loop simulation by generating a model of hardware behaviour from DSL instances. In [4], a DSL based on Xtext has been developed to generate code for real-time large-scale distributed data processing.

An early experiment to combine DSLs and formal methods has been described in [3]. In that paper, the correctness of instances of a DSL for process scheduling is verified using the B method [1]. To increase the use of formal methods in industry, [7] propose 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 [12].

This paper is organized as follows. In Section 2 we introduce the power control component as far as needed to illustrate our approach. The DSLs to describe the behaviour and the test cases are described in Sections 3 and 4 respectively. Section 5 explains the techniques we used to increase the confidence in the formal models and the generators. Concluding remarks can be found in Section 6.

2. POWER CONTROL COMPONENT

The project described here is part of the development of a system for image guided therapy that uses X-ray to generate images which support the surgeon during minimally invasive medical procedures. An example configuration is depicted in Figure 2.

Figure 2: Interventional X-ray system

This system has a distributed architecture with a large number of hardware and software components. The system is highly configurable, i.e., customers can select a particular combination of X-ray stands, patient table, monitors, image processing capabilities, etc. Powering the hardware components, and starting up and shutting down the software components are the responsibility of the power control component. This component is installed in a technical room together with a number of cabinets which contain the supporting hardware components.

The power control component consists of a controller that has three interfaces, as shown in Figure 3:

- An interface to a User Interface Module (UIM) that has On and Off buttons, and LEDs for user feedback.
- An interface with software components running on computers.
- An interface with power distribution panels that are placed in the cabinets to power the hardware components installed within the same cabinet. Each power distribution panel has a number of individually switchable High and Low Voltage Terminals (HVT/LVT) that are managed by the controller.

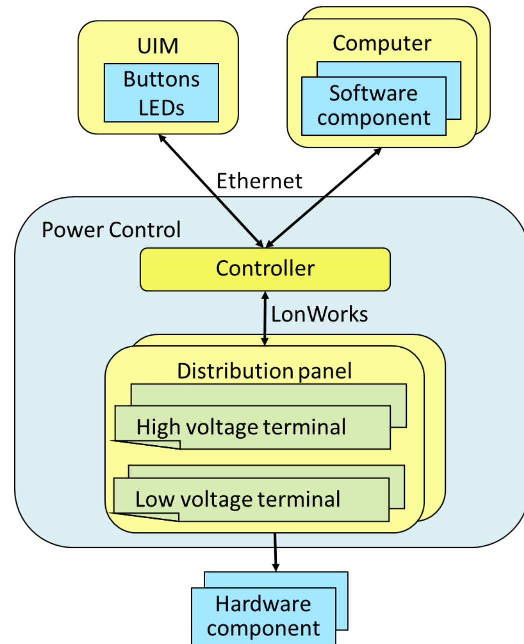

Figure 3: Overview power control component

The controller and all distribution panels have a 16 bit microcontroller running an embedded application and they communicate with each other via LonWorks [10] using a master-slave topology. LonWorks creates a communication channel superimposed on the power line with which the controller powers the distribution panels.

The controller is configured by two files: the *recalls configuration file* and the *scenarios configuration file*. A *recall* defines the state of all terminals that power the other components of the system. The recalls configuration file describes a number of possible recalls that can be used in the scenarios. A fragment is shown in Table 1, defining recall TermStandby. Everything behind a # sign is a comment.

```
# TermStandby
<RECALL 1>
<TAP>
00 7 1 0.0 0.0 0.0 # Controller_PowerBus, status = On
00 8 0 0.0 0.0 0.0 # Controller_PulsePowerBus, status = Off
04 0 1 0.0 0.0 16.0 # M_Cab_HVT1, status = On
04 1 0 0.0 0.0 16.0 # M_Cab_HVT2, status = Off
...
04 1 1 1 # M_Cab_LVT5V1, status = Off
04 2 1 1 # M_Cab_LVT12V1, status = Off
04 5 1 0 # M_Cab_LVTGbl, status = On
```

Table 1: Fragment of the recalls configuration file

The first column of the numbers describes the location of a terminal, e.g., 00 for the Controller and 04 for the M-Cabinet. The second column codes the number of the terminal. For the HVTs the third column codes if the terminal should provide power (1) or not (0). Likewise, for the LVTs the fourth column codes if the terminal should provide power (0) or not (1), but observe that the numbers are inverted. We do not explain the other columns, which are different for HVTs and LVTs.

The scenarios configuration file describes the scenarios in terms of a state machine. Transitions between recalls are not atomic, that is, during such a transition a stimulus might lead to another required recall. To represent the state of these transitions, each main state consists of three substates: Entry, Transitioning, and Stable. A line of the scenarios configuration file is shown in Table 2.

```
2 2 0 00000000 00000000 112 4 2    # recall 2 exit out of forced off
```

Table 2: Line of the scenarios configuration file

We explain the main columns:

- The first column is the main state which is the source of the transition (in this example, state 2 denotes Standby).
- The second column the substate which is the source of the transition (here 2 denotes substate Stable).
- The third column the source recall (here 0 denoting that all terminals are off).
- The sixth column, describes the stimulus number (in this example, 112 denotes pushing the on button for at least 3 seconds).
- The seventh column is the number of the specified transition between two main states (it might be a self-transition).
- The eighth column describes the required recall (recall 2 in this example). By default, the substate will be Entry.

For performance reasons, this file is sorted on the sixth column. So on stimulus number and not per state, which hampers readability.

3. DEFINING COMPONENT BEHAVIOUR

The configuration files are hard to read, to change and to maintain, but they have to be updated for every new product release. To reduce the risk of making errors, we developed a Domain Specific Language (DSL) to express the essential concepts of the configuration files in a natural and readable way. To improve the confidence in the correct behaviour of the configuration, generators are also added to create simulation models (POOSL), described in Section 3.1, and verification models (SAL), described in Section 3.2.

An overview is given in Figure 4. An instance of the DSL corresponds to a product release. It leads to one scenarios configuration file and multiple recalls configuration files corresponding to the different system configurations. A customer can choose a particular system configuration which results in different hardware components (and their associated behaviour).

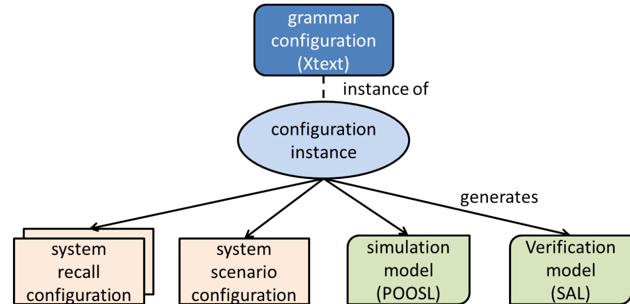

Figure 4: Overview configuration DSL transformations

The grammar for the DSL is expressed in Xtext[1]. The Xtend[2] language is very suitable to define generators, because it contains convenient constructs to refer to elements of the grammar and to define transformations. We have used Xtend to generate configuration files and analysis models from language instances.

The controller contains a high-level state machine, which describes the main states (Off, Standby, Operation, Emergency Power Off, and Stop) and their behaviour. This high-level state machine is always the same. Hence there is no need to define it explicitly in the DSL. The DSL describes a low-level state machine, defining the recalls and their transitions. Figure 5 depicts a fragment of the first part of an instance.

To improve readability, the DSL instance starts with defining meaningful names for the required status of the terminals, here called *termstatus*. Since several termstatuses might correspond to the same required settings of the terminals, the second part of the DSL groups the termstatuses and associates a recall (i.e., a state of the voltage terminals) with each group. The third part defines a state machine, where for a main state, a termstatus, and a stimulus we define the next termstatus. A transition might have a condition on the current substate, indicated by keyword *if*. Note that each termstatus belongs to exactly one main state, so the *next* relation implicitly defines the next main state.

Figure 6 shows the last part of the configuration DSL which describes the recalls. As mentioned before, for every system configuration (i.e., choice of hardware components) there is a separate recalls configuration file. Hence, the DSL specifies several configurations and a number of recalls for each configuration. For convenience, we allow the definition of a default state for all terminals which can be overridden for particular HVTs or LVTs. Moreover, for a recall configuration (e.g., `setup_derived1` in Figure 6) the setting of another configuration (e.g., `setup`) can be used and only the differences have to be specified.

3.1 POOSL

To simulate the specified behaviour of the power control component, we use the Parallel Object Oriented Specification Language

[1]eclipse.org/Xtext/

[2]eclipse.org/xtend/

```
termstatuses = SystemInit or SystemOff or
               SystemFseOff or SystemOn …
…
group = SystemFseOff and SystemEPO   recallID = AllOff
group = SystemOff and SystemOffError recall = TermStandby
group = SystemOn and SystemOnError   recallID = AllOn
…
state Init
    termstatus SystemInit
         if Transitioning stim PostFail
                               next termstatus SystemStop
                          stim Initialized
                               next termstatus SystemOff
state Standby
    termstatus SystemFseOff
                          stim EpoActive
                                next termstatus SystemEPO
         if Stable        stim ButtonOn3sec
                               next termstatus SystemOn
    termstatus SystemOff
         if Stable        stim ButtonOn3sec
                               next termstatus SystemToggleTaps
                          stim ButtonOff10sec
                               next termstatus SystemFseOff
                          stim EpoActive
                               next termstatus SystemEPO
    termstatus ShuttingDownSystem
         if Transitioning stim ShutdownTimedOut
                               next termstatus SystemOff
                          stim ShutdownCompleted
                               next termstatus SystemOff
                          stim EpoActive
                               next termstatus SystemEPO
…
```

Figure 5: First part of an instance of the configuration DSL

(POOSL) [19]. POOSL is a modelling language for systems that include both software and digital hardware. The language allows the definition of concurrent processes which communicate by synchronous message passing along ports.

The formal semantics of POOSL has been defined by means of a probabilistic structural operational semantics for the process layer and a probabilistic denotational semantics for the data layer [20]. This semantics has been implemented in a high-speed simulation engine called Rotalumis. On top of this engine a modern Eclipse IDE has been developed which is freely available[3]. More information about the use of POOSL in the context of the power control component can be found in [16].

Using the Xtext/Xtend framework we implemented a generator which delivers a POOSL model for every DSL instance. Using sockets, such a POOSL model is connected to a Graphical User Interface (GUI), see Figure 7, which allows the injection of events and the inspection of the system state, such as the settings of the HVTs and LVTs, during simulation. The simulation is used to validate and align the system behaviour with internal stakeholders. In this way, we detected problems in the DSL instance, e.g., the simulation in POOSL revealed a missing condition.

3.2 SAL

To verify the intended behaviour of the power control component, we generated a SAL model [5, 17] to enable formal model checking. Table 3 shows a fragment of the SAL model. It starts with defining the State and Stim types, which are used to define the input and output of the main module. Local variables are defined to represent the status of the LVTs and HVTs (e.g. M_Cab_HVT1)

[3] poosl.esi.nl

```
...
config name = setup
    recall TermStandby
        Default for recall      status Off
            Controller
                PowerBus        status On
            M_Cab
                HVT1            status On
                HVT4            status On
                LVT24V2a3       status On
                LVTGbl          status On
...

config name = setup_derived1
    recall TermStandby
        Use config setup
            M_Cab
                LVT24V2a3       status Off
    recall TermToggle
        Use config setup
    recall TermShutDown
        Use config setup
```

Figure 6: Last part of an instance of the configuration DSL

which can provide power (TRUE) or not (FALSE). Next the initial state and initial values of the boolean variables are specified. Subsequently, the transitions are defined. The ELSE statement at the end ensures input enabledness, i.e., in any state always all inputs can be received.

```
SALModel: CONTEXT =
BEGIN
 State : TYPE = {SystemOnStable, SystemPartlyOnStable, ...};
 Stim : TYPE = {ButtonOn3sec, ButtonOn10sec, ButtonOff, ...};
main : MODULE =
 BEGIN
 INPUT stim : Stim
 OUTPUT state : State
 LOCAL Controller_PowerBus, M_Cab_HVT1, ... : BOOLEAN
 INITIALIZATION state = SystemOffStable;
                Controller_PowerBus = TRUE;
                Controller_PulsePowerBus = FALSE; ...
 TRANSITION
 [ state = SystemOffStable AND stim = ButtonOn3sec
    -- > state' = SystemToggleTapsStable;
                Controller_PowerBus' = TRUE;
                Controller_PulsePowerBus' = TRUE; ...
 [] state = SystemOffStable AND stim = ButtonPartlyOn3sec
    -- > state' = SystemPartlyOnStable;
                Controller_PowerBus' = TRUE;
                Controller_PulsePowerBus' = FALSE; ...
 [] ELSE -- > %  implicitly: state'= state
 ]
 END;
% Properties
END
```

Table 3: Fragment of a SAL model

Table 4 list a part of the properties we verified on the SAL model. We mainly checked invariants, using the G (Globally) operator of Linear Temporal Logic (LTL). We briefly explain the theorems:

th1 The first property checks if LVTs and HVTs that belong to the same behavioural group always have the same state.

th2 The second property checks if a terminal provides power in

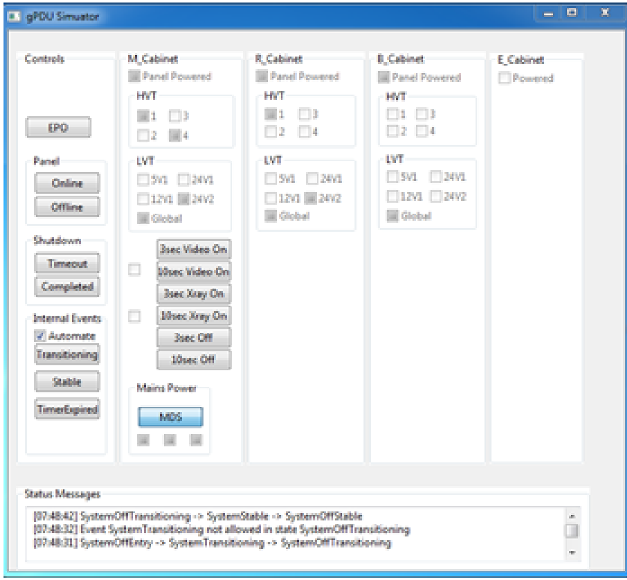

Figure 7: GUI of the simulation

the SystemOffStable state, then it also provides power in all future steps when it is in the SystemPartlyOnStable state. Similarly, with SystemPartlyOnStable and SystemOnStable exchanged.

th3 The third property checks if the LVT24V2a3 terminal in the M-Cabinet provides power in all states except for the SystemToggleTapsStable state.

th4 For a LVT to provide power it needs to be set to the on state as well as some preconditions also need to be met. Based on the hardware design, the preconditions for a LVT to provide power is that LVTGbl is on and that HVT1 provides power. The property specifies that if a LVT needs to provide power, the LVT global switch is on, and if a LVT global switch needs to provide power, the corresponding HVT1 is on.

When we checked the last property, the SAL model-checker reports a counter-example: in the SystemToggleTapsStable state the LVT24V2a3 in the R-Cabinet should provide power, but the HVT1 does not provide power in the SystemToggleTapsStable state, which is clearly wrong. We changed the DSL instance, which automatically leads to the generation of a new SAL model. Checking the new model did not reveal any errors.

3.3 Generation of Configuration Files

Having simulated and verified the DSL instance, we finally generate the configuration files. A fragment of the generated scenarios configuration file is shown in Table 5. Since the state machine described by the DSL is ordered on the states, the generator has to transform this to a list that is sorted on the stimulus number. We have added useful comments to facilitate reviewing and debugging of the generator. Similarly, for all system configurations, a recalls configuration file has been generated.

4. TESTING THE COMPONENT

The power control component has a test set to validate its behaviour, i.e., to check that the combination of the general software framework, the configuration files, and the hardware conforms to

```
th1: THEOREM main |–
G(Controller_PowerBus = M_Cab_HVT1 = M_Cab_HVT2 =
              R_Cab_HVT4 = B_Cab_HVT3) AND
...
G(M_Cab_HVT1 = M_Cab_HVT2);

th2: THEOREM main |–
G((state = SystemOffStable AND M_Cab_HVT4) =>
          G(state = SystemPartlyOnStable => M_Cab_HVT4))
AND
G((state = SystemPartlyOnStable AND M_Cab_HVT4) =>
          G(state = SystemOnStable => M_Cab_HVT4))
AND
...

th3: THEOREM main |–
FORALL (i: State): ((NOT(i = SystemToggleTapsStable)) =>
          M_Cab_LVT24V2a3);

th4: THEOREM main |–
G(M_Cab_LVT5V1 => M_Cab_LVTGbl) AND
G(M_Cab_LVTGbl => M_Cab_HVT1) AND
...
G(R_Cab_LVT5V1 => R_Cab_LVTGbl) AND
G(R_Cab_LVTGbl => R_Cab_HVT1) AND
```

Table 4: SAL property

```
6 0 0 00000000 00000000 109 19 2
   # E.SystemEPO -> BUTTON_ON10SEC -> O.SystemOn
2 2 1 00000000 00000000 112 4 3
   # S.SystemOff -> BUTTON_ON3SEC -> O.SystemToggleTaps
2 2 0 00000000 00000000 112 4 2
   # S.SystemFseOff -> BUTTON_ON3SEC -> O.SystemOn
2 1 1 00000000 00000000 115 6 0
   # S.SystemOff -> BUTTON_OFF10SEC -> S.SystemFseOff
5 1 2 00000000 00000000 117 21 5
   # O.SystemOnError -> BUTTON_OFF -> S.ShuttingDownError
3 2 2 00000000 00000000 117 5 5
   # O.SystemOn -> BUTTON_OFF -> S.ShuttingDownSystem
3 2 3 00000000 00000000 134 7 2
   # O.SystemToggleTaps -> TIMER_EXPIRED -> O.SystemOn
```

Table 5: Fragment of the generated scenarios configuration file

the requirements. Clearly, this is needed for every new product release, but it is also important to rerun all tests after maintenance, e.g., to solve issues found in the field or to upgrade hardware.

We describe the existing test cases in Section 4.1. A DSL to improve maintainability of the test cases is presented in Section 4.2. Our approach to generating test cases using SAL is explained in Section 4.3.

4.1 Test Cases

For testing the power control component a dedicated test tool has been created which reads a comma separated variable (CSV) file, executes the test case and writes a report with the test results. One test case describes a walk through the state space. All test cases start and end in the same state which makes it possible to execute them consecutively. Table 6 shows the basic format of test cases, omitting irrelevant details.

Each line describes the command that needs to be sent from the test tool to the power control component, the expected response of the power control component, and either the time the test tool needs

PDS:SYST?	6:00:00	2500	SystemEPOEntry
PDS:QUE109:PAR-1	No Error	2500	ButtonOn10sec
PDS:FWV?	3.0.0.0	T016_TRANS	SystemTransitioning
PDS:SYST?	3:01:02	1000	SystemOnTransitioning
PDS:FWV?	3.0.0.0	T017_STABLE	SystemStable
PDS:SYST?	3:02:02	1000	SystemOnStable

Table 6: Fragment of a test case

to wait before sending the next command or the event the test tool needs to receive before continuing with the next command. The last column is a comment.

In the existing situation, a test case consists of about 20 lines. For testing a single product release about 30 test cases have been constructed manually. The test cases are difficult to read and to change.

4.2 Test DSL

To describe test cases for the power control component in a maintainable way, a test DSL has been created. A fragment of a test DSL instance is depicted in Figure 8. A test DSL instance consists of the following three segments:

- In the first segment, for each termstatus of the configuration instance three extended termstatuses are generated corresponding to the three substates, by adding *Entry*, *Transitioning*, and *Stable* behind the name. The string after keyword *code* matches the first three columns of the configuration file.
- The second segment lists all possible transitions.
- In the third segment, one or more trace sets are defined. Each trace set has one or more traces. A trace consists of an initial extended termstatus, and a number of pairs consisting of a stimulus and a next extended termstatus.

```
termstatuses
termstatus SystemFseOffEntry code "2:0:0"
termstatus SystemFseOffTransitioning code "2:1:0"
termstatus SystemFseOffStable code "2:2:0"
…
transitions
transition EpoActive from termstatus SystemFseOffEntry
                     to termstatus SystemEPOEntry
transition EpoActive from termstatus SystemFseOffTransitioning
                      to termstatus SystemEPOEntry
transition ButtonOn3sec from termstatus SystemFseOffStable
                        to termstatus SystemOnEntry
…
tracesets
traceset SystemEPOEntry
trace SystemEPOEntry -> ButtonOn10sec -> SystemOnEntry ->
SystemTransitioning -> SystemOnTransitioning -> SystemStable ->
…
SystemEPOEntry
…
```

Figure 8: Instance test DSL

From an instance of the language, the generator produces a file containing a test case for every trace such as in Table 6. For the timing values in tests, the generator uses a fixed minimal waiting time. The test tooling adds a random value, where the range of these random values is a configuration parameter of the test tool.

4.3 Automated Test Case Generation

Creating test cases with the test DSL is an improvement compared to manually writing the test cases in the format the test tool takes as input. However, it is still a lot of work to construct the test cases and to guarantee a certain level of coverage. Hence, we have experimented with the automatic generation of test cases.

Since the configuration DSL describes the state behaviour and instances have been validated using POOSL and SAL, these instances form the basis for our test generation. For this purpose, the SAL test generator is used, where the SAL model described in Table 3 is extended. Auxiliary variables (e.g., t0, t1, ..) are added to every transition and updated (e.g., t0' = TRUE) when the transition is taken. These auxiliary variables are initially FALSE and only updated when the transition is taken. Table 7 depicts a fragment of an updated SAL model.

```
...
 LOCAL t0, t1, t2, t3, t4, ...,
 INITIALIZATION state = SystemOffStable;
                t0 = t1 = ... = FALSE  ...
 TRANSITION
 [ state = SystemOffStable AND stim = ButtonOn3sec
      --> state' = SystemToggleTapsStable;
           t0' = TRUE; Controller_PowerBus' = TRUE; ...
 [] state = SystemOffStable AND stim = ButtonVideoOn3sec
      --> state' = SystemVideoOnStable;
           t1' = TRUE; Controller_PowerBus' = TRUE; ...
 [] ELSE -->
 ]
 END;
% Properties
END
```

Table 7: Fragment of a SAL model

The SAL test generator is fed with three files:

- the SAL model described in Table 7 which is generated automatically from the configuration DSL;
- a file describing the test goal (all auxiliary variables t0, t1, ... recording taken transitions need to be TRUE) - this file is also generated automatically from the configuration DSL; and,
- a file that defines which information should be visible in the output and how the output should be formatted.

Based on this input, the SAL test generator yield test traces until the goal is satisfied. From the output of the SAL test generator, an instance of the test DSL is generated automatically using a small script.

With the generated test cases, approximately twice as much transitions are covered as with the manually written test cases. Theses manually written test cases only made transitions from the Stable substates. The generated test cases also make transitions from the Entry and Transitioning substates, which leads to approximately twice as much transition coverage.

The manually written cases were very time-dependent, with long waiting times. They could still fail by a slow response of hardware, which required some analysis and typically 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.

5. CHECK MODELS AND GENERATORS

To increase the confidence in the correctness of the models and the transformations, we have used several cross checks and logging information of the system in use. The power control component writes its state and stimulus events to a log file during its routine usage. To validate our approach, we have used log files made by system testers and users in the field. We implemented a small script that can be used to translate this logging information into a trace which is formatted according to the test DSL. To use these test traces for the validation of the POOSL and SAL models, we have constructed additional transformations from the test DSL to POOSL and SAL. An overview of the transformation is depicted in Figure 9.

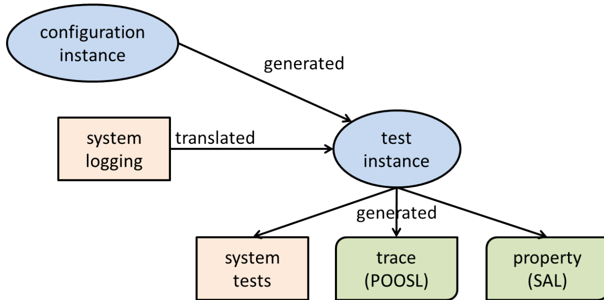

Figure 9: Overview test DSL transformations

To validate the POOSL models and generators, we generate from an instance of the test DSL a separate POOSL process which tests the POOSL model generated from the configuration DSL (see Section 3.1). This POOSL test process replaces the GUI and runs the test trace by providing the stimuli of the trace and comparing the output of the model with the output specified in the trace. The results are written to a test report.

For SAL, the test DSL is extended with a generator which delivers an LTL formula describing a test trace in SAL. Table 8 provides an example of such a formula, where X is the next operator referring to the next step. Next the SAL model checker can be used to check the existence of the test trace in the model.

```
th11: THEOREM main |-
G((state = SystemOffStable AND stim = ButtonPartlyOn3sec) =>
   X(state = SystemPartlyOnStable)) AND
G((state = SystemPartlyOnStable AND stim = ButtonOff) =>
   X(state = ShuttingDownPartlySystemTransitioning)) AND
G((state = ShuttingDownPartlySystemTransitioning AND
        stim = ShutdownCompleted) =>
   X(state = SystemOffStable));
```

Table 8: Fragment of a trace expressed in LTL

Finally, Figure 10 provides an overview of the role of SAL in our approach. It is used to verify properties of instances of the configuration DSL and to generate test traces, i.e., instances of the test DSL. Moreover, the SAL model is validated using system logs.

6. CONCLUDING REMARKS

By means of the configuration DSL we could quickly generate the configuration files for the current product release. They were tested successfully on the target hardware. Comparing the generated files with the existing ones, we found a number of issues in the existing files, such as an erroneous transition and a number of missing transitions.

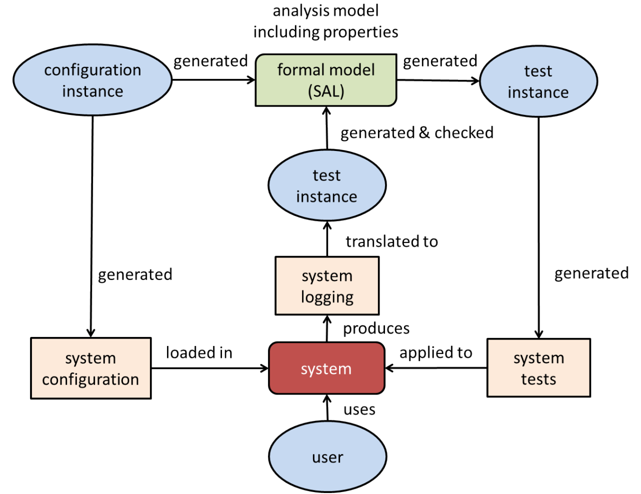

Figure 10: SAL overview

It took about 35 hours to create the two DSLs presented here and to integrate them with the power control component and the test tool. This step was sufficient to demonstrate the usefulness of the approach to management. In later increments, we added the generators for the analysis models and the use of SAL for test generation. Since the adaptation of grammars and generators is relatively easy and fast, the approach supports an incremental way-of-working. The Eclipse/Xtext framework is quite mature and provides many basic features such as syntax highlighting, auto completion, and content assist.

Given earlier experience with POOSL and SAL, it was straightforward to write generators for these languages. For each of the two languages mentioned, this took about 5 hours. These formal models were especially useful when we created a DSL instance for the next product release. By means of the POOSL and SAL tooling, we could extensively simulate and verify the new instance and at the end produce high quality configuration files.

For the new product release, the size of the files almost doubled which indicates that the complexity of our configuration files is expected to increase quickly. With a very small investment, we are ready to deal with this increasing complexity. We can now create the configuration and test cases for the power control component in a readable, easy to change and maintainable format. The new product release required only 8 hours instead of the estimated 60 hours.

SAL was very useful in generating an improved test set with twice as much transition coverage. However, model checking with SAL is quite time consuming. For instance, creating the SAL properties took 20 hours. The main reason it took this long was because model checking of some properties took at least 30 minutes or run out of memory on an 8 core i7-3720QM CPU @ 2.60 GHz with 16 GB RAM. Also the logic to express properties is limited to LTL (or a corresponding subset of CTL). So we were not able to check certain desirable properties about the existence of a path to certain states.

In future work, we intend to experiment with other model checkers, although this might require more effort to generate test cases.

An approach to generate test sequences using a model checker has been described in [15]. After some more experience with other checkers we also plan to include a more readable property language in the DSL, since the specification of properties in a form of temporal logic is not easy for industrial engineers. Moreover, we would like to investigate how to translate error traces to the DSL level, e.g. making it more readable using e.g. Plant UML [14]. In the project described here, we have already used PlantUML to visualize the state machine and test traces.

The successful combination of DSLs and formal methods is currently applied to other components of the image guided therapy systems of Philips. Although the DSL is different and C++ code is generated instead of configuration files, the approach is the same. From the DSL instances also simulation models, formal models and test cases are generated. Cross checks between the models are used to increase the confidence in the generators.

7. ACKNOWLEDGMENTS

The research of the second author was supported by the Dutch national program COMMIT and carried out as part of the Allegio project. We thank the anonymous reviewers for their useful comments.

8. REFERENCES

[1] J.-R. Abrial. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] L. Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd, 2013.

[3] J.-P. Bodeveix, M. Filali, J. Lawall, and G. Muller. Formal methods meet domain specific languages. In *Integrated Formal Methods*, volume 3771 of *Lecture Notes in Computer Science*, pages 187–206. Springer, 2005.

[4] K. Chandrasekaran, S. Santurkar, and A. Arora. Stormgen - a domain specific language to create ad-hoc storm topologies. In M. P. M. Ganzha, L. Maciaszek, editor, *Proceedings of the 2014 Federated Conference on Computer Science and Information Systems*, volume 2 of *Annals of Computer Science and Information Systems*, pages 1621–1628. IEEE, 2014.

[5] G. Hamon, L. de Moura, and J. Rushby. Automated Test Generation with SAL. CSL Technical Note, SRI International, January 2005.

[6] C. Heitmeyer. On the need for practical formal methods. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1486 of *Lecture Notes in Computer Science*, pages 18–26. Springer, 1998.

[7] P. James and M. Roggenbach. Encapsulating formal methods within domain specific languages: A solution for verifying railway scheme plans. *The Computing Research Repository*, abs/1403.3034, 2014.

[8] C. B. Jones, D. Jackson, and J. Wing. Formal methods light. *Computer*, 29(4):20–22, 1996.

[9] J. Kärnä, J.-P. Tolvanen, and S. Kelly. Evaluating the Use of Domain-Specific Modeling in Practice. In *Proceedings of the 9th OOPSLA workshop on Domain-Specific Modeling*, 2009.

[10] LonWorks. www.echelon.com/technology/lonworks, 2015.

[11] A. Mooij and J. Hooman. Creating a Domain Specific Language (DSL) with Xtext. http://www.cs.ru.nl/ hooman/DSL/, 2015.

[12] P. D. Mosses, editor. *CASL Reference Manual*, volume 2960 of *Lecture Notes in Computer Science*. Springer, 2004.

[13] I. Nagy, L. Cleophas, M. van den Brand, L. Engelen, L. Raulea, and E. Mithun. VPDS: A DSL for Software in the Loop Simulations Covering Material Flow. In *17th Int. Conf. on Engineering of Complex Computer Systems (ICECCS)*, pages 318–327, 2012.

[14] PlantUML. PlantUML in a nutshell. plantuml.com, 2015.

[15] S. Rayadurgam and M. Heimdahl. Test-sequence generation from formal requirement models. In *IEEE Int. Symp. onHigh Assurance Systems Engineering*, pages 23–31. 2001.

[16] M. Schuts and J. Hooman. Formal modelling in the concept phase of product development. In *Proc. Conf. on Software Engineering Research & Practice (SERP 2015)*, pages 3–9. WORLDCOMP'15, CSREA Press, USA, 2015.

[17] N. Shankar. Combining theorem proving and model checking through symbolic analysis. In *CONCUR'00: Concurrency Theory*, volume 1877 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2000.

[18] N. Shankar. Symbolic analysis of transition systems. In *Abstract State Machines: Theory and Applications (ASM 2000)*, volume 1912 of *Lecture Notes in Computer Science*, pages 287–302. Springer, 2000.

[19] B. D. Theelen, O. Florescu, M. Geilen, J. Huang, P. van der Putten, and J. Voeten. Software/Hardware Engineering with the Parallel Object-Oriented Specification Language. In *Proc. MEMOCODE'07*, pages 139–148. IEEE, 2007.

[20] L. van Bokhoven. Constructive tool design for formal languages; from semantics to executing models. Phd thesis, Eindhoven University of Technology, the Netherlands, 2004.