Proceedings of the Federated Conference on
Computer Science and Information Systems pp. 781–788

DOI: 10.15439/2015F46
ACSIS, Vol. 5

Using Domain Specific Languages to Improve the Development of a Power Control Unit

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

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

***Abstract*—To improve the design of a power control unit at Philips, two Domain Specific Languages (DSLs) have been used. The first DSL provides a concise and readable notation for the essential state transitions. It is used to generate both configuration files and analysis models. In addition, we also generate instances of a second DSL which represents test traces. This second DSL is used to generate test cases for the power control unit. The use of DSLs not only improved productivity, but also the quality of the configuration files and the test set.**

I. Introduction

THIS PAPER discusses industrial experience with the use of Domain Specific Languages (DSLs) at Philips HealthTech. We have used DSLs to develop power control units of systems for image guide therapy. These systems are used during minimally invasive medical treatments, such as the treatment of cardiology and vascular diseases. An example is the interventional X-ray system shown in Fig. 1, where X-ray images support minimally-invasive medical procedures such as placing a stent via a catheter.

Fig. 1. Interventional X-ray System

Given the long history of these systems and the frequent need for changes to support new medial procedures, it is important to keep the software architecture and its components flexible and extensible. Hence, legacy components have to be renewed to keep them prepared for the future.

An example of such a legacy component is the power control unit. It uses low-level configuration files which are hard to read and difficult to maintain. Given the increasing number of configurations and new third-party components that have to be supported, this starts to become a potential bottleneck. In addition, only a limited set of regression tests is available.

Already long ago, DSLs have been suggested as a way to raise the level of abstraction, to deal with variability, and to improve productivity and maintainability. An early overview of terminology, techniques and applications can be found in [1]. As one of the disadvantages of DSLs this paper mentions the costs of designing, implementing and maintaining a DSL. Since then, however, large improvements have been achieved in the area of language workbenches. Such tools facilitate the efficient construction of languages, editors, and transformations [2], [3]. Examples of workbenches are MetaEdit+ [4], Rascal [5], Spoofax [6], EMFText [7], and Xtext [8].

There are a number of relevant applications in the domain of embedded systems. For instance, there is an interesting laboratory experiment of the application of MetaEdit+ to heart rate monitors of Polar [9], showing 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 [10]. From this DSL, a simulation of hardware behaviour which enables software in the loop simulation has been generated. In [11], a DSL based on Xtext has been developed to generate code for real-time large-scale distributed data processing. By means of the MPS approach [12], an impressive extension of the C language has been constructed [13], [14].-

The aim of our work is to investigate whether DSLs could provide a solution to improve the maintainability and testability of our power control unit. We would like to get an answer to the following questions:

- How much time is needed to learn the tools and techniques?
- How much effort is needed to migrate the current legacy component to a component which is defined by a high-level human-usable DSL?
- Does the DSL approach support the combination with analysis techniques such as simulation tools and formal

This research was supported by the Dutch national program COMMIT and carried out as part of the Allegio project.

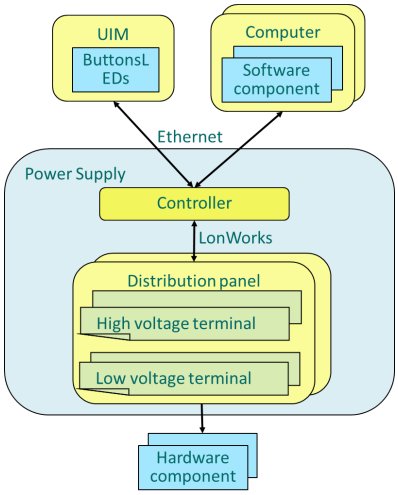

Fig. 2. Overview Power Control Unit

model checkers?
- What are the benefits of introducing these new techniques compared to the current way of working?

The paper is organized as follows. Sect. II describes our industrial case. The developed DSLs are presented in Sect. III and Sect. IV. Sect. V contains an overview of the results and an evaluation of the approach.

II. Case: Power Control Unit

The interventional X-ray systems of Philips use 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, 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 unit. This unit is installed in a technical room together with a number of cabinets which contain the required hardware components. The power control unit consists of a controller that has three interfaces, as shown in Fig. 2:

- An interface to a User Interface Module (UIM) that has On and Off buttons, and two 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 that are managed by the controller.

The controller and all distribution panels have a 16 bit microcontroller running an embedded application. They communicate with each other via LonWorks [15] 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 implements the system start-up/shut-down behaviour using a state machine with two parts:

- A high-level state machine that is part of the application running inside the controller.
- A configuration file that describes the low-level state behaviour of the power control unit.

The configuration file is used by the high-level state machine to perform the configuration-specific transitions. This high-level state machine is implemented with VisualState [16] and describes the main states and the associated LED behaviour when transitioning between these main states. These main states are:

- Off: the power control unit is not powered;
- Init: represents the start-up of the power control unit, in this state a Power On Self Test (POST) is executed;
- Standby: the system is off for the user, but power control unit is standby and some continuous power terminals are powered;
- Operational: the system is on for the user, typically all terminals provide power in this state;
- Emergency Power Off (EPO): the controller cuts off the power of the distribution panels immediately and thereby also all the terminals loose power (only the controller stays powered) - used when the user presses a red safety button;
- Stop: a terminal state which is entered when critical parts of the power control unit are detected to be faulty during the POST; in this state only the controller is powered to be able to diagnose the problem.

The low-level state machine for the power control unit defines the so-called recalls and the transitions between these recalls. Each recall denotes a required setting of the high and low voltage terminals, i.e., whether an individual terminal needs to provide power or not. These settings are described in a separate configuration file (not described in this paper).

To realize a particular recall, the controller compares the current status of the low- and high voltage terminals, which it has stored in volatile memory, with the desired status of the low and high voltage terminals. If the current status is different from the desired one, the controller starts communicating with the distribution panels to change the status. The transition from one recall to another may take a considerable amount of time, because of the inherently slow LonWorks communication. Depending on the chosen hardware components by the customer, there are two or three cabinets and it takes between 10 and 30 seconds to address all distribution panels.

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: the controller compares the current status of the low and high voltage terminals with the desired recall. If they are different the next substate is Transitioning, otherwise it is Stable, except for the first recall where it stays in Entry.

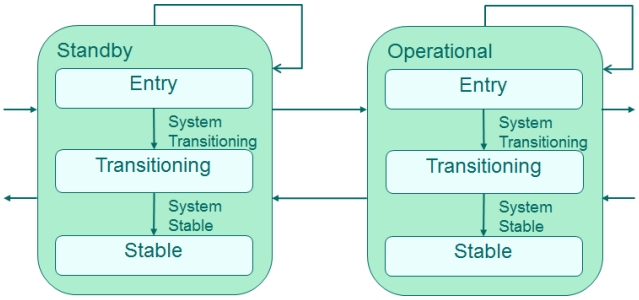

Fig. 3. Two main states and their substates

TABLE I
LINE OF A CONFIGURATION FILE FOR A LOW-LEVEL STATE MACHINE

2 2 0 00000000 00000000 112 4 2	# < OPERATIONAL > recall 2 # exit out of forced off

- Transitioning: the controller is busy changing the state of the low and high voltage terminals.
- Stable: all distribution panels have reached the desired state for the low and high voltage terminals.

Fig 3 shows part of the high level VisualState state machine with two main states and their substates. The main states and substates are fixed, whereas the number of recalls is variable and defined in the configuration file. The low-level state machine and the recalls are different for every system release. The configuration file describes for each recall and stimulus, possibly with a given guard, what the next recall is and between which main states it has to transition. This is all coded in numbers. The main states are numbered, e.g., Standby = 2 and Operational = 3 and similar for the substates: Entry = 0, Transitioning = 1, and Stable = 2. Also all stimuli and all transitions between the main states have a fixed number. The recalls have a configurable number. The guard of a transition consists of two values: the relevant values of a status register and a mask. Table I shows an example of a line in the configuration file. Everything after a # is a comment.
The first three columns of Table I describe the state or -1 if it does not care.

1) The first column is the main state which is the source of the transition (in this example, state 2 denotes Standby).
2) The second column the substate which is the source of the transition (here 2 denotes substate Stable).
3) The third column the source recall (here 0 denoting that all terminals are off).

The fourth and fifth column describe the guard.

4) The fourth column describes the bits of the status register.
5) The fifth column the mask that will be applied.

The other columns have the following meaning:

6) The sixth column, describes the stimulus number (in this example, 112 denotes pushing the on button for 3 seconds).
7) The seventh column is the number of the specified transition between two main states (it might be a self-transition).
8) 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. That is, the file is sorted on stimulus number and not on state, which hampers readability.

TABLE II
PART OF A TEST CASE

PDS:QUE9:PAR	NoErr	30000	1000	2000	On Button
PDS:SYST?	3:2:2	1000	1000	2000	System On

To test the state machine, there is an automated test tool running on a companion PC that connects to the controller of the power control unit via Ethernet. It can inject stimuli and ask the current state. A test case is a comma separated file. Table II shows two lines of a test case.

1) The first column is the network command that is send from the test tool to the power control unit (QUE injects a stimulus into the state machine and SYST asks the current state).
2) The second column is the expected response from the power control unit to the test tool ("NoErr" means that the command is successfully parsed and "3:2:2" is the current state, with main state 3 (Operational), substate 2 (Stable), and recall 2).
3) The third column is the time (in milliseconds) that the test tool waits before it sends the next command to the power control unit (in this example, the test tool waits 30 seconds between the QUE and the SYST command).
4) The fourth column is a time-out (in milliseconds) on the reply of the power control unit; within this amount of time the power control unit should send a message to indicate that it accepts the command.
5) The fifth column is a time-out (in milliseconds) on the response of the power control unit (as specified in the second column).
6) The sixth column contains comments.

A test case fails on a wrong response or on a time-out of the accept message or the response.

Every night a test suite is executed multiple times on two power control units with Jenkins [17] and the developers will find the results of the test execution in their mailbox. If test cases fail, a lot of time is spent investigating the cause and solving it in either the software of the power control unit or the test case. Test cases often fail because of timing issues in which the power control unit and the test tool are out of sync; this is almost always caused by the unreliable timing nature of LonWorks. The solution for such timing issues is an increase of the time bounds in the test cases. This results in long-lasting test cases with a lot of waiting time.

Concluding, the configuration file is hard to read, to change and to maintain, but has to be updated for every new system release. The same holds for the test cases which typically need to be updated manually for every new configuration file.

```
termstatuses = SystemInit or SystemOff or SystemFseOff or SystemOn ...
...

group = SystemFseOff and SystemEPO, recall = 0
group = SystemOff and SystemOffError, recall = 1
...
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
...
```

Fig. 4. Configuration DSL

III. Configuration DSL

To solve the problems mentioned above, we created a configuration DSL for the power control unit, using Eclipse, Xtext and Xtend [8]. This technology was chosen because the second author was familiar with it and the availability of a manual [18].

We explain the configuration DSL based on an instance fragment of the language, as shown in Fig. 4. Since the main states and their substates are always the same, there is no need to define them explicitly. The main purpose is to define the recalls and their transitions. To improve readability, the first part of the DSL instance defines 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 number with each group. The third part defines the low-level state machine, where for a main state, a termstatus, and a stimulus we define the next termstatus. A transition might have a condition - indicated by the *if* keyword - on the current substate. Note that each termstatus belongs to exactly one main state, so the *next* relation implicitly defines the next main state.

The grammar for this language has been expressed in Xtext; a fragment is depicted in Fig. 5. Based on this grammar, the Xtext framework generates an editor for the language with, for instance, content assist. This makes it easy to define instances of the languages, such as the instance shown in Fig. 4.

The Xtext framework also provides suitable primitives for language validation and the generation of files from instances. In our application, the Xtend language has been used to generate a configuration file from language instances. This generator produces a line in the configuration file for every

```
PowerConfiguration:
    'termstatuses = ' termstatNames += TermStatus
        (' or ' termstatNames += TermStatus)*
    (termStatGroups += TermStatGroup)+
    (states += State)+
;
TermStatus:
    name = ID
;
TermStatGroup:
    'group = ' termstatName += [TermStatus]
        (' and ' termstatName += [TermStatus])*
    ', recall = ' recall = INT
    ;
```

Fig. 5. Grammar of the Configuration DSL

rule in the language instance. After the generator has generated all the rules, it sorts the lines on the value in the sixth column before writing it to a configuration file. A fragment of a generated configuration file is shown in Table III.

The first eight columns, till the # sign, are the same as the manually created configuration file described in the previous section. In the comment part, the generator writes the first letter of the source main state, the source substate name, the stimulus, followed by the first letter of the target main state and the target substate name.

Additionally, we have generators that yield for every language instance a set of UML state diagrams, at several levels of abstraction, using PlantUML [19]. An impression of a generated diagram is given in given in Fig. 6 (not readable for reasons of confidentiality).

TABLE III
GENERATED CONFIGURATION FILE

```
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.ShuttingDownSystemError
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
```

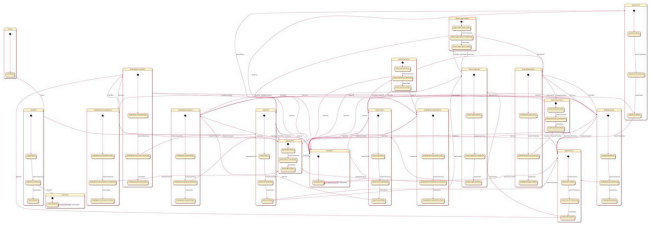

Fig. 6. State Diagram of the Configuration DSL

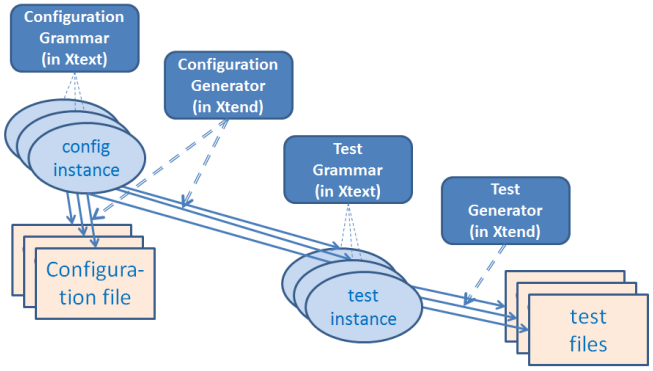

Fig. 7. Overview of the two DSLs

IV. TEST DSL

Given a formalized representation of the configuration files, we investigated the possibilities to generate test cases from this DSL. A preliminary attempt to define a generator for test cases indicated that this was possible, but became rather complex since it included strategies to define test cases as well as translations to the current test format. To separate these issues, we aimed for a more abstract representation, related to the work described in [20] concerning a tool-independent representation of test design techniques. In our context, we defined a second DSL for the definition of tests to obtain a representation of tests which is independent of the current test techniques. Fig. 7 shows the relation between these DSLs. Note that with the Xtext approach there is no separate definition of the abstract grammar with meta models.

The test DSL is explained using the instance fragment depicted in Fig. 8. Note that this instance is generated from an instance of the configuration DSL. In the first part, for each termstatus of the configuration instance three extended termstatuses are generated corresponding to the three sub-states, by adding *Entry*, *Transitioning*, and *Stable* behind the

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

Fig. 8. Test DSL

name. The string after keyword *code* matches the first three columns of the VisualState configuration file; it is obtained using the code of the main state of the termstatus, the code of the substate, and the recall number defined in the configuration instance.

The second part lists all possible transitions. This is used to generate a report about coverage of termstatuses and transitions and to generate additional tests for stimuli that should not lead to a transition. In the third part, 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. Each trace starts and ends with the same extended termstatus, which makes it possible to run a number of traces in one go. Note that this requirement makes the generation of an instance of the test DSL a bit more complicated.

The generator of the trace DSL generates a test case, as shown in Table IV, for each trace in the language instance. Every Transitioning and Stable termstatus will result in a line in the test case with a SYST command that expects the string defined after the *code* key word in the *termstatuses* part of the instance. Since Entry substates are not observable (except for the first one), they are omitted. A stimulus will result in a line in the test case with a QUE command. Note that the third column is slightly different from table II; instead of a waiting time, now also an event can be specified. Such an event is used to synchronize with the power control unit and avoids long waiting times. It makes the testing process much faster.

Additionally, we used PlantUML to generate a visualization of a test trace as a sequence diagram. An examples is depicted in Fig. 9.

TABLE IV
PART OF A GENERATED TEST CASE

PDS:SYST?	6:00:00	2500	1000	2000	SystemEPOEntry
PDS:QUE9:PAR	NoErr	2500	1000	2000	ButtonOn10sec
PDS:FWV?	3.0.0.0	T016_TRANS	1500	2000	SystemTransitioning
PDS:SYST?	3:01:02	1000	1000	2000	SystemOnTransitioning
PDS:FWV?	3.0.0.0	T017_STABLE	1500	2000	SystemStable
PDS:SYST?	3:02:02	1000	1000	2000	SystemOnStable

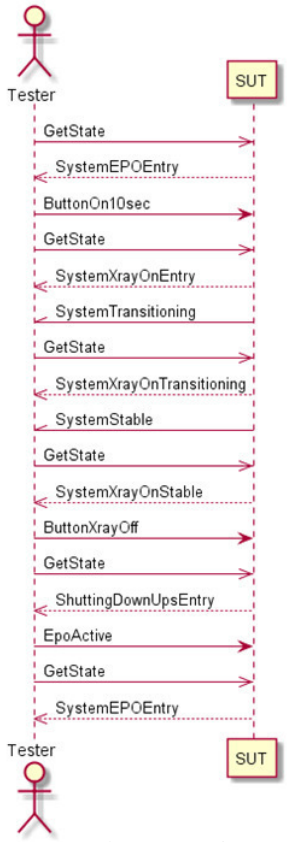

Fig. 9. Generated Sequence Diagram of a Test Case

The generator also generates a coverage file. Based on the selected termstatuses, transitions and traces, it creates a list of covered states, uncovered states, covered transitions and uncovered transitions. See Fig. 10.

Typically, we create two trace sets. The first trace set covers all transitions and is used for state and transitions coverage. A disadvantage of this set is that it also tests Emergency Power Off (EPO), which implies that only the controller of the power control remains powered; the distribution panels will lose power including the processor inside. Since this will rarely happen during normal usage, a second set is created with more realistic user scenarios where the distribution panels stay powered. Jenkins is configured to run these tests every weekend many times to test the reliability of the software running inside the power control. Outside the weekend, the full test set is run every night.

```
Covered states
 SystemFseOffEntry
 SystemFseOffTransitioning
 SystemFseOffStable
 SystemToggleTapsEntry
 SystemToggleTapsTransitioning
 SystemToggleTapsStable
 ...
Uncovered states
 PrePostEntry
 PrePostTransitioning
 PrePostStable
 ...
Covered transitions
 EpoActive
 SystemTransitioning
 SystemStable
 ButtonOn3sec
 ButtonOff10sec
 ...
Uncovered transitions
 MdsOn
 Initialized
 PdmCommissionTmo
 PostFail
 ...
```

Fig. 10. Coverage File

In the work described above, we used our own algorithm to generate test cases. As a next step we investigated the use of an existing tool to generate the test cases. We selected the SAL (Symbolic Analysis Laboratory) framework [21], [22], [23] which includes an automated test generator. The generator of the configuration DSL has been extended to generate two SAL files, corresponding to the two trace sets described above. When the test generator of SAL is supplied with a test goal - in our case the goal is to cover all transitions - it will yield traces that satisfies the goal. From this information, an instance of the test DSL is generated automatically using a small script.

V. CONCLUSION

We summarize the results in Sect. V-A. The additional generation of analysis models is described in Sect. V-B. Sect. V-C addresses the questions of Sect. I. A brief description is future work can be found in Sect. V-D.

A. Results

We started with the configuration DSL and generated a configuration file for the current system release. This generated file was successfully tested on the target hardware. Comparing the generated file with the existing one, we found a number of issues in the existing file. It contained a non-existing transition and a number of transitions were missing. As a next step, we

made a DSL instance for the next system release and generated the configuration file. The size of this new configuration file is about twice the size of the current file, which indicates the increasing complexity of our system releases.

With the generated test cases, approximately twice as much transitions are covered as the manually written tests. The 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 twice as much transition coverage.

The manually written cases were very time-dependent, with many 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 unit and immediately resumes the test case once the power control unit has reached the desired state.

B. Analysis models

In addition to the generated configuration and test files, we generated a number of analysis models. From the configuration DSL we generated models for the simulation tooling of POOSL [24] and the model checker mCRL2 [25]. The generators for mCRL2 and POOSL combine the behaviour of the high-level state machine and the low-level state behaviour described by the (generated) configuration file, into one state machine describing the complete system start-up and shutdown behaviour. Implicitly, these generators define the semantics of the DSLs. The use of multiple tools increases the confidence in the correctness of the generators.

The advantage of having a separate test DSL is that we can also generate tests or checks for the analysis models. For POOSL, we generated a tester process that communicates with the generated state machine. Every test trace results in a method that applies stimuli to the state machine process and checks whether the returned responses are as expected. The results are written to a file with a test report. For mCRL2 we generated properties to express that the expected traces occur in the generated state machine. This property can then be checked by the mCRL2 tools.

Using POOSL and mCRL2, we detected problems in the DSL instance, e.g., the simulation in POOSL revealed a missing condition. Moreover, we can detect whether a termstatus occurs in multiple main states. An alternative would be to use the validation possibilities of the Xtext/Xtend framework.

C. Evaluation

We discuss the questions listed in the introduction (Sect. I):

- *How much time is needed to learn the tools and techniques?*
  Clearly, the learning curve for new techniques depends on previous education and knowledge. The work reported here was mainly done by the first author who independently learned the DSL techniques from the manual mentioned before [18]. With a Master's in Computer Science, including course about grammars and formal techniques, the basic part of the manual requires 4 hours to install the tools and to redo the examples of the manual. This was enough to get started with the case study.
- *How much effort is needed to migrate the current legacy component to a component which is defined by a high-level human-usable DSL?*
  It took about 35 hours to create the two DSLs presented here and to integrate them with the power control unit 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.
- *Does the DSL approach support the combination with analysis techniques such as simulation tools and formal model checkers?*
  Given earlier experience with POOSL, mCRL2, and SAL, it was straightforward to write generators for these languages. For each of the three languages mentioned, this took about 5 hours. The generators to visualize the state diagram and test traces using PlantUML requires only a few hours of work.
- *What are the benefits of introducing these new techniques compared to the current way of working?*
  The complexity of our configuration files is expected to increase quickly; as already mentioned above, the file size almost doubled for a new product release. With an investment of only 35 hours, we are ready to deal with this increasing complexity. We can now create the configuration and test cases for the power control unit in a readable, easy to change and maintainable format. The tests are now 3 times faster with a double coverage. It is expected that for a new product release we need only 8 hours instead of the estimated 60 hours.

Our experience is in accordance with a recent report about the state of practice in model-driven engineering [26]. It shows that most successful applications of model-driven development use small DSLs.

D. Future Work

Future work includes an extension of the configuration DSL such that also the configuration file which contains the details of the recalls (i.e., the required setting of the high and low voltage terminals) can be generated. Similar to the low-level state machine, also this configuration file is hard to read, to change, and to maintain.

As a next step, we intend to remove the Visual State framework, and generate the full state machine directly in the C programming language. The generators for POOSL,

mCRL2, and SAL are a good indication that this will be feasible.

ACKNOWLEDGMENT

We thank the anonymous reviewers for a number of useful suggestions for improvement.

REFERENCES

[1] A. van Deursen, P. Klint, and J. Visser, "Domain-specific languages: An annotated bibliography," *SIGPLAN Notices*, vol. 35, no. 6, pp. 26–36, 2000. doi: http://dx.doi.org/10.1145/352029.352035

[2] M. Fowler, *Domain Specific Languages*. Addison-Wesley Professional, 2010.

[3] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. L. Kats, E. Visser, and G. Wachsmuth, *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.

[4] J. Tolvanen, R. Pohjonen, and S. Kelly, "Advanced tooling for domain-specific modeling: MetaEdit+," in *The 7th OOPSLA Workshop on Domain-Specific Modeling*, 2007.

[5] P. Klint, T. van der Storm, and J. Vinju, "EASY meta-programming with Rascal," in *Generative and Transformational Techniques in Software Engineering III*, ser. Lecture Notes in Computer Science. Springer, 2011, vol. 6491, pp. 222–289.

[6] L. Kats and E. Visser, "The Spoofax language workbench. rules for declarative specification of languages and IDEs," in *The 25th Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, 2010. doi: http://dx.doi.org/10.1145/1869459.1869497 pp. 444–463.

[7] Software Technology Group, TU Dresden, "EMFText," http://www.emftext.org/, 2011, version 1.4.0.

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

[9] J. Kärnä, J.-P. Tolvanen, and S. Kelly, "Evaluating the use of domain-specific modeling in practice," in *The 9th OOPSLA workshop on Domain-Specific Modeling*, 2009.

[10] 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)*, 2012, pp. 318–327.

[11] K. Chandrasekaran, S. Santurkar, and A. Arora, "Stormgen - a domain specific language to create ad-hoc storm topologies," in *Proceedings of the 2014 Federated Conference on Computer Science and Information Systems*, ser. Annals of Computer Science and Information Systems, M. P. M. Ganzha, L. Maciaszek, Ed., vol. 2. IEEE, 2014. doi: http://dx.doi.org/10.15439/2014F278 pp. 1621–1628.

[12] "Meta programming system (MPS)," http://jetbrains.com/mps, 2015.

[13] M. Voelter, D. Ratiu, B. Schaetz, and B. Kolb, "Mbeddr: An extensible C-based programming language and IDE for embedded systems," in *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH '12)*. ACM, 2012. doi: http://dx.doi.org/10.1145/2384716.2384767 pp. 121–140.

[14] M. Voelter, "Generic tools, specific languages," Ph.D. dissertation, Delft University of Technology, 2014.

[15] "LonWorks," http://www.echelon.com/technology/lonworks/, 2015.

[16] "VisualState," http://www.iar.com/Products/IAR-visualSTATE/, 2015.

[17] "Jenkins," http://jenkins-ci.org/, 2015.

[18] A. Mooij and J. Hooman, "Creating a domain specific language (dsl) with Xtext," http://www.cs.ru.nl/J.Hooman/DSL/, 2015.

[19] "PlantUML," http://plantuml.sourceforge.net/, 2015.

[20] M.-F. Wendland, "Abstractions on test design techniques," in *Proceedings of the 2014 Federated Conference on Computer Science and Information Systems*, ser. Annals of Computer Science and Information Systems, M. P. M. Ganzha, L. Maciaszek, Ed., vol. 2. IEEE, 2014. doi: http://dx.doi.org/10.15439/2014F316 pp. 1575–1584.

[21] N. Shankar, "Symbolic analysis of transition systems," in *Abstract State Machines: Theory and Applications (ASM 2000)*, ser. Lecture Notes in Computer Science, no. 1912. Springer, 2000, pp. 287–302.

[22] ——, "Combining theorem proving and model checking through symbolic analysis," in *CONCUR'00: Concurrency Theory*, ser. Lecture Notes in Computer Science, no. 1877. Springer, 2000. doi: http://dx.doi.org/10.1007/3-540-44618-4_1 pp. 1–16.

[23] G. Hamon, L. de Moura, and J. Rushby, "Automated test generation with SAL," SRI International, CSL Technical Note, January 2005.

[24] 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 *Proceedings of MEMOCODE'07*. IEEE, 2007. doi: http://dx.doi.org/10.1109/MEMCOD.2007.371231 pp. 139–148.

[25] S. Cranen, J. Groote, J. Keiren, F. Stappers, E. de Vink, W. Wesselink, and T. Willemse, "An overview of the mCRL2 toolset and its recent advances," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2013. doi: http://dx.doi.org/10.1007/978-3-642-36742-7_15 pp. 199–213.

[26] J. Whittle, J. Hutchinson, and M. Rouncefiled, "The state of practice in model-driven engineering," in *IEEE Software*. IEEE, 2014. doi: http://dx.doi.org/10.1109/MS.2013.65 pp. 79–85.