Radboud Repository

Radboud University Nijmegen

PDF hosted at the Radboud Repository of the Radboud University Nijmegen

The following full text is a publisher's version.

For additional information about this publication click this link.
http://hdl.handle.net/2066/175500

Please be advised that this information was generated on 2018-07-07 and may be subject to change.

Industrial Experiences in Applying Domain Specific Languages for System Evolution

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Radboud Universiteit Nijmegen
op gezag van de rector magnificus prof. dr. J.H.J.M. van Krieken,
volgens besluit van het college van decanen
in het openbaar te verdedigen op
donderdag 21 september 2017 om 10:30 uur precies

door
M.T.W. Schuts

Promotor:

Prof. dr. J.J.M. Hooman

Manuscriptcommissie:

Dr. ir. A.J. Mooij (TNO-ESI)
Prof. dr. F.W. Vaandrager
Prof. dr. J.J. Vinju (Technische Universiteit Eindhoven)
Prof. dr. ir. J.M.W. Visser
Prof. dr. ir. J.P.M. Voeten (Technische Universiteit Eindhoven)

Copyright © 2017 by M.T.W. Schuts

Omslag: Anneke van Sonsbeek
Druk: Gildeprint Drukkerijen

ISBN 978-94-6233-666-7

Industrial Experiences in Applying Domain Specific Languages for System Evolution

DOCTORAL THESIS

to obtain the degree of doctor
from Radboud University Nijmegen
on the authority of the Rector Magnificus prof. dr. J.H.J.M. van Krieken,
according to the decision of the Council of Deans
to be defended in public on
Thursday, September 21, 2017 at 10:30 hours

by
M.T.W. Schuts

Supervisor:

Prof. dr. J.J.M. Hooman

Doctoral Thesis Committee:

Dr. ir. A.J. Mooij (TNO-ESI)
Prof. dr. F.W. Vaandrager
Prof. dr. J.J. Vinju (Eindhoven University of Technology)
Prof. dr. ir. J.M.W. Visser
Prof. dr. ir. J.P.M. Voeten (Eindhoven University of Technology)

Copyright © 2017 by M.T.W. Schuts

Cover: Anneke van Sonsbeek
Print: Gildeprint Drukkerijen

ISBN 978-94-6233-666-7

ACKNOWLEDGEMENTS

This acknowledgement concludes a period of approximately six years in which I applied new techniques at Philips and a period of half a year in which I wrote this thesis. I would like to use the opportunity to thank a number of persons.

First and foremost, I owe many thanks to Jozef Hooman without whom it would have been impossible to write this thesis. It started with searching a topic for my Master's thesis. Jozef arranged an internship at Philips, where he was co-located for a research project of TNO-ESI, and guided me through my internship in the lunch breaks.

The collaboration that started during my Master's thesis project continued the years thereafter. I learned many new techniques and I liked to apply all those new techniques in industry. With Jozef I brainstormed on many occasions on how to apply them in the best possible way. Some of these projects were interesting enough to publish about, so that is what we did. After a few years of applying these techniques and publishing papers, there was enough material to write a thesis and Jozef became my promoter. I am very grateful for all the time Jozef Hooman spent as my mentor and promoter. Jozef helped me a lot in streamlining the storyline and guide me through the difficult process of writing this thesis.

I met Ammar Osaiweran at Philips where he was a PhD candidate when I was writing my Master's thesis. I would like to thank Ammar for the nice time we had at Philips. We published a number of papers together and I like the time we spent at a conference in Tallinn, Estonia. I do not think I would have written this thesis if Ammar did not occasionally asked if I wanted to write a PhD thesis.

I would like to thank my two department managers at Philips: Ad Jurriens and Paul Tielemans. Ad for offering me a job at Philips and giving me the space to apply new techniques. Paul for promoting me to lead software designer which gave me more freedom to apply the techniques in the components for which I was responsible. Paul even suggested new problems where I could apply the techniques. In addition, I would like to thank Dirk-Jan Swagerman for stimulating the use of new techniques within the software departments in general and his support of the ComMA framework in particular.

The work presented in this thesis all took place at Philips. I worked in different

projects and teams. I would like to thank all the team members for the pleasant time and their support. Next a list of all the projects that contributed to this thesis in chronological order.

I would like to thank the team members of the project in which I created the Power Control Service (PCS): Fahmy Ali, John Bekx, Fehim Begtasevic, Maruschka de Bruijn-Maessen, Michel van Geffen, Robert Huis in ‘t Veld, Arjen Kaiser, Ron van Kesteren, Arjan van Lankveld, Dave Mollet, Leendert Nelemans, Maarten van de Sande, and Arjan Versluys.

I am grateful to the team members of the two Power Distribution Unit (PDU) projects: Tolgay Akkaya, Jaap van Alphen, Maria Berruezo, Laura Calvino, Jelle Haandrikman, Gijs Hobo, Edwin Hoogeveen, Ron Jansen, Rob de Jong, Pratik Khedkar, John Kennis, Rob Kleihorst, Ben Nijhuis, Ivo Pullens, Herman Roebbers, Andre Schurer, Wim Swinkels, Peter Timmermans, Marcel van der Veeken, Stephan Verhelst, Martin van Vliet Paul Wingelaar, and Edwin van Woudenberg.

This thesis is based on several papers, which are joint work with a number of co-authors which are gratefully acknowledged: Jan Friso Groote, Ivan Kurtev, Bart van Rijnsoever, Frits Vaandrager, and Jacco Wesselius.

I would like to thank the members of the doctoral thesis committee: Arjan Mooij, Frits Vaandrager, Jurgen Vinju, Joost Visser, and Jeroen Voeten for their suggestions for improving an earlier version of the manuscript.

Last but not least I would like to thank my family and friends for their support.

May 2017

CONTENTS

1 Introduction **1**
- 1.1 Context . . . 1
- 1.2 Problem Statement . . . 2
- 1.3 Goal and Evaluation Criteria . . . 2
- 1.4 Approach . . . 3
  - 1.4.1 Domain Specific Languages . . . 4
  - 1.4.2 Overview Applied Techniques . . . 4
  - 1.4.3 General Aspects of Industrial Cases . . . 6
- 1.5 Industrial Context . . . 7
- 1.6 Thesis Outline . . . 9

2 Related Work **13**

3 Language for Creating New Components **19**
- 3.1 Motivation for Applying ASD . . . 19
- 3.2 Fundamentals of ASD . . . 20
  - 3.2.1 ASD Interface Models . . . 21
  - 3.2.2 ASD Design Models and Model Checking . . . 25
- 3.3 Integrating ASD in Industrial Workflow . . . 29
  - 3.3.1 The TDD Approach . . . 31
  - 3.3.2 The ASD Approach . . . 31
- 3.4 Context of the PCS . . . 32
- 3.5 Steps of Developing the PCS . . . 34
- 3.6 Errors Not Detected by the ASD Verification . . . 38
- 3.7 Results of Developing the PCS . . . 41
- 3.8 Concluding Remarks . . . 45

4 Language for Exploring New System Concepts **47**
- 4.1 Motivation for Applying POOSL . . . 47
- 4.2 Fundamentals of POOSL . . . 49
  - 4.2.1 POOSL Modelling Language . . . 49
  - 4.2.2 POOSL Tooling . . . 50

4.3 Application at Philips . . . 51
4.4 Modeling the SU/SD Concept in POOSL . . . 53
4.4.1 Modelling Scope and Simulator . . . 53
4.4.2 Modelling Steps . . . 55
4.4.3 Modelling Devices and Control . . . 56
4.4.4 Extensive Model Testing . . . 57
4.5 Concluding Remarks . . . 60

5 Configuring a Component using DSLs 63
5.1 Motivation for Creating DSLs . . . 63
5.2 Context of the Fieldbus . . . 64
5.3 DSL for Fieldbus Configurations . . . 65
5.4 DSL Instance Validation . . . 66
5.5 DSL to Describe System Configurations . . . 67
5.6 Concluding Remarks . . . 69

6 DSLs Combined with other Model-Based Techniques 71
6.1 Motivation and Global Overview . . . 71
6.2 Context of the PDU . . . 73
6.3 Defining the Behaviour of the Component . . . 77
6.3.1 POOSL . . . 79
6.3.2 SAL . . . 81
6.3.3 Generation of Configuration Files . . . 82
6.4 Testing the Component . . . 83
6.4.1 Test cases . . . 84
6.4.2 Test DSL . . . 85
6.4.3 Automated Test Case Generation . . . 86
6.5 Increasing the Confidence in Models and Generators . . . 89
6.6 Concluding Remarks . . . 91
6.6.1 Results . . . 91
6.6.2 Analysis models . . . 91
6.6.3 Evaluation . . . 92

7 Model Learning to Validate Refactoring 95
7.1 Motivation for the Application of Model Learning . . . 95
7.2 Learning Approach . . . 96
7.3 Context of the PCS . . . 97
7.4 Application of the Learning Approach . . . 98
7.4.1 Design of the Learning Environment . . . 98
7.4.2 Learned Output . . . 99
7.4.3 Checking Equivalence . . . 99
7.4.4 Investigating Counterexamples . . . 100
7.5 Results of Learning the Implementations of the PCS . . . 101
7.5.1 Iteration 1 . . . 101
7.5.2 Iteration 2 . . . 101
7.5.3 Iteration 3 . . . 102
7.5.4 Iteration 4 . . . 102
7.5.5 Iteration 5 . . . 103
7.6 Scalability of the Learning Approach . . . 103
7.6.1 Faster implementations . . . 103

7.6.2 Faster Learning and Testing Algorithms 104
7.6.3 Using Parallelization and Checkpointing 104
7.6.4 Using Abstraction and Restriction 105
7.7 Concluding Remarks 105

8 Refactoring a Legacy Implementation using a DSL 107
8.1 Motivation for the Transformation 107
8.2 Transformation Approach 108
8.3 State Machine Transformations 110
8.3.1 From Rhapsody to ComMA 112
8.3.2 Generating ComMA Instances 115
8.3.3 From ComMA to Dezyne 117
8.4 Increasing Confidence in the Generated Code 117
8.5 Concluding Remarks 119

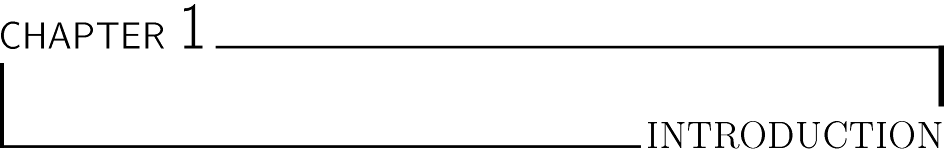
9 Epilogue 121
9.1 Evaluation of Criteria 121
9.2 Lessons Learned 126
9.3 Future Work 127

Bibliography 129

Samenvatting 141

Summary 143

Curriculum Vitae 145



This chapter is the introduction to this thesis. It starts with a description of high-tech systems, the evolution of high-tech systems and the challenges with legacy software. Based on this research context, we formulate the problem statement. Next we define the goal of this thesis and discuss the research approach. Subsequently, the industrial context in which the research took place is presented. Finally we describe the chapter outline of this thesis.

1.1 Context

In the high-tech industry complex products are created. By definition high-tech products are constructed using cutting edge technology. Consequently, embedded system technology plays a major and often even decisive role in such systems [27]. Examples of high-tech products are: cars, trains, airplanes, smart phones, waver steppers, medical devices, et cetera. These products consist for a large part of software. The software in products is constructed using a process called the software life cycle. The software life cycle consists of a number of distinct phases [165]. The first phase concerns the construction of the software. In this phase it is specified, designed, built and tested. When it is accepted by the customer, it is taken into use. In the second phase the software needs to be maintained by the constructor of the system. The system is taken out of use when it has reached end-of-life and it is replaced by its successor.

Bennett et al. [16] define a staged model in which the maintenance phase is split up into an evolution stage and a servicing stage. In the evolution stage, changes are made to cope with changes in functional and non-functional requirements. If evolutionary changes are no longer possible, the system moves to the servicing stage. In the servicing stage only service patches are applied to keep the system alive.

Lientz et al. [93] divided software maintenance activities into the following four categories:

- Adaptive; changes in the software environment,
- Perfective; new user requirements,
- Corrective; fixing errors, and
- Preventive; prevent problems in the future.

According to [158], 50-75 percent of all the effort on a software system is spent in the maintenance phase.

High-tech systems are being made for many decades. While the systems for the customer may be new, they are constructed from a collection of many components from which some can be characterized as legacy components [26].

Legacy components suffer from a combination of the following characteristics: high maintenance costs; no, or few test cases; complex software; source code contains duplications; obsolete support software; maintenance of hardware and software components from third parties has expired; lacking technical expertise; business critical; backlog of change requests; no, poor, or outdated documentation; embedded business knowledge; original developers or users are no longer available; poorly understood by maintainers; and small maintenance tasks require a lot of time [168, 166].

This thesis focuses on high-tech systems in the evolution stage. These systems have legacy components as an additional challenge.

1.2 Problem Statement

For improving the software life cycle most can be gained by improving the time and effort that needs to be spend in the maintenance phase. High-tech systems are the result of decades of development with dozens of man-years invested. The resulting systems are very large with millions of lines of source code. The code might no longer be in line with the original design concepts and the documentation might no longer be in line with the implementation. Descriptions of interfaces might be incomplete, e.g. not describing the behaviour in terms of state or timing. Support for obsolete hardware and other dead code might confuse the engineer that needs to apply changes. Some software components might be created by obsolete tools and languages. The original designers and developers may no longer be available.

High-tech systems also contain a lot of accumulated value. Replacing a legacy system would require more resources than keeping the current implementation alive [168]; scarce resources that could also be used to implement product innovations on the existing system. Hence, these systems need to evolve to bring product innovations to the market. In practice, changing these systems, or in other words performing maintenance activities on these systems, consumes a lot of time and effort. Summarizing, evolving high-tech systems is very hard.

1.3 Goal and Evaluation Criteria

The goal of the described research is to investigate the application of techniques, that are new for Philips, to improve the evolution of high-tech systems with legacy

components. The applicability of the applied techniques is evaluated according to the following criteria.

Scalability High-tech systems are large and complex, therefore it is relevant to evaluate the scalability of new techniques. Scalability has different dimensions, such as: size of engineering artefacts, size and complexity of languages used, and number of engineering artefacts [89]. By evaluating these dimensions of scalability, we check if a new technique can be applied on industry-sized problems.

Integration in an industrial context Applying techniques in industry means that they need to be integrated in the industrial context. This criterion evaluates 1) the learning curve needed before applying the techniques, 2) whether the technique can be incorporated in the current way of working or to which extent the current way of working has to be adapted, 3) whether organisational or cultural changes are required, and 4) whether commercial support is available.

Return On Investment (ROI) Introducing a new technique in a company needs to add value. ROI is defined as operational income divided by assets invested [110]. ROI is used to objectively evaluate the cost compared by the potential gain of introducing a new technique. We exclude learning the technology from the ROI calculations.

Improve system evolution We evaluate the contribution of a technique to improve the evolution of a high-tech system.

Note that to be able to apply the techniques in industry, particular tools have been used. So the evaluation also includes tool aspects.

1.4 Approach

According to Potts [119], the traditional approach of software engineering research consists of distinct sequential activities. First, a researcher defines an industrial problem to investigate, e.g., in a multi-year research program. Second, the researcher incrementally refines the solution. Concurrently, when the researcher is investigating solutions for the problem, the industrial context - and with it the problem - also evolves. Last, by the time the researcher wants to transfer the results back to industry, the solution no longer matches the current state of the problem. Potts calls this approach: "research-then-transfer". As an alternative, he introduces the "industry-as-lab" approach. With the "industry-as-lab" approach, the researcher and the industrial partner tightly work together to define the problem. The research results are iteratively tried and implemented by the industrial partner. Benefits of applying the "industry-as-lab" approach are:

- The problem definition is frequently improved because of a better understanding of the problem domain.

- Research results are incrementally integrated in industry with a short feedback loop.
- Research does not only address technical challenges, but also non-technical challenges such as the transfer of the research results into the organization.
- Research takes into account the scale and complexity industry faces when solving problems.

Davison et al. [41] define action research as an attempt by researchers to solve real-world problems while simultaneously studying the experience of solving the problem. Easterbrook et al. [46] describe: While most empirical research methods attempt to observe the world as it currently exists, action researchers aim to intervene in the studied situations for the explicit purpose of improving the situation.

Furthermore, Easterbrook et al. describe that the problem owner needs to collaborate with the researcher to be able to solve the problem, and that the problem owner and the researcher might be the same person.

The benefits of the "industry-as-lab" approach matches the goals described in Section 1.3. For this reason, the research described here has taken place using an "industry-as-lab" approach. Action research was applied during real development projects in which the problem owner and researcher were the same person. The projects were executed at Philips between 2010 and 2016.

1.4.1 Domain Specific Languages

This thesis focuses on applying techniques for improving the evolution of a high-tech systems. The techniques investigated in this thesis are based on Domain Specific Languages (DSLs) [155], because 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 [156]. According to Michaelson [101] a DSL is defined as a notation oriented to a specific problem domain with specialised types and control structures. Furthermore, Rosen et al. [124] discriminate between a vertical and a horizontal DSLs. A vertical DSL focuses on a certain business domain. Typically, a business domain is specific for a single company. These DSLs are being used by domain experts. Alternatively, a horizontal DSL focuses on a specific problem domain. Examples are SQL, HTTP, et cetera [42]. As one of the disadvantages of DSLs, van Deursen et al. [156] mention the costs of designing, implementing and maintaining a DSL. Since then, large improvements have been achieved in the area of language workbenches. Such tools facilitate the efficient construction of languages, editors, and transformations [57, 163].

1.4.2 Overview Applied Techniques

Figure 1.1 depicts the approach taken. We created multiple DSLs and generators for these DSLs. From an instance of a DSL, a number of artefacts are generated, such as new implementations, e.g. source code or configuration files, and test cases. Formal techniques provide confidence in the behaviour described by

a DSL instance and also visualizations have been generated from DSL instances. Additionally, legacy implementations have been captured by DSL instances after which the previous described techniques could be applied, e.g., to generate new implementations.

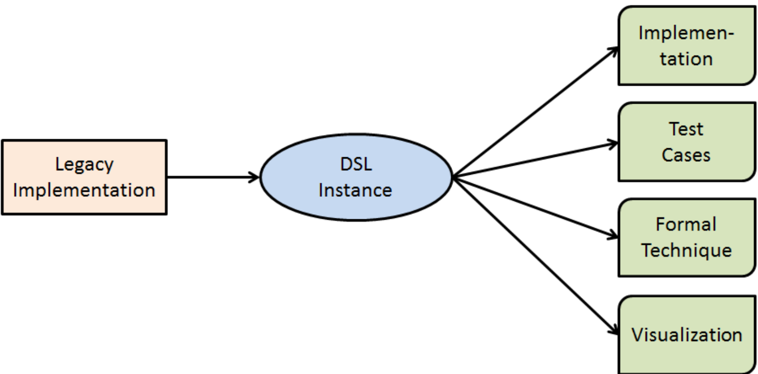


Figure 1.1: *Overview Approach*

Next we provide an overview of the DSLs that have been applied to improve system evolution. The choice for these techniques was pragmatic; we knew the techniques from earlier experience and they seemed suitable for the projects in which they were applied. The application of these DSLs in real industrial development projects is evaluated according the criteria listed in Section 1.3.

Analytical Software Design (ASD) ASD [161] is a horizontal DSL that can be used to describe data-independent control components and interfaces. Instances of the ASD language can be checked, using CSP/FDR2 [29, 60], for a predefined set of properties. Examples of these properties are: determinism of design models, life-lock freedom, dead-lock freedom, and interface compliance. From the language instances, source code is generated and this code is integrated into the product. In addition, visualizations can be generated.

For replacing an obsolete hardware component, a new hardware component was created. ASD has been applied to model a new software component that interfaces with the new hardware component.

Parallel Object-Oriented Specification Language (POOSL) POOSL [147] is a horizontal DSL for describing the behaviour of digital hardware and software components. Instances of the language can be simulated using the Rotalumis simulator [22]. When model checking is no longer feasible because of the large number of states, simulation with POOSL can be chosen as an alternative to get confidence in the behaviour of a system.

POOSL has been applied to model the start-up/shut-down behaviour of a high-tech system. The model includes both hardware and software components and their interactions. Using this model, evolutionary changes to the system have

been explored. Simulation was used to get confidence in the correctness of the new system concepts, before implementing them in the real system.

Vertical DSLs A vertical DSL can be created using the Eclipse [171] plug-in Xtext [18]. From a DSL instance, artefacts can be generated. In addition, errors are prevented by a number of validation checks that are run on DSL instances.

New vertical DSLs have been created to generate configuration files for legacy components. Besides generating configuration files, also SAL models have been generated. SAL [138] is a formal technique which is used to verify the behaviour specified by the DSL. In addition, SAL automatically generates test cases. These test cases are represented in a second DSL to be able to exploit the test cases in multiple frameworks.

Model learning and equivalence checking The externally observable behaviour of an implementation can be learned using LearnLib [76], which is a model learner [153]. It generates a state machine in terms of an instance of the GraphViz's DOT-language [47]. Instances of this horizontal DSL can be used to generate an instance of the mCRL2 horizontal DSL. The equivalence of two learned implementations can be checked using the equivalence checker from the mCRL2 tool-set [38].

Model learning has been applied to learn the behaviour of a legacy component and its new refactored implementation. Next equivalence of these two implementations is checked.

Model transformation DSLs can be applied to perform model transformations. A horizontal DSL can be used to capture instances of one language and to generate an instance of another language.

Model transformations have been used to replace a legacy implementation that has been created with a modelling tool. Model files that were used by the tool to generate implementations are instances of the horizontal DSL and a generator of the DSL can create model instances for a new tool. Because there can be errors in the transformation, model learning and equivalence checking provide confidence in the correctness of the transformation.

1.4.3 General Aspects of Industrial Cases

We applied the techniques described above in real industrial development projects. These projects address the following aspects of system evolution:

- Create a new software component for the replacement of a legacy hardware component.
- Explore new concepts for system evolution.
- Keeping a legacy software component longer in the evolution stage by improving its maintainability and extensibility.
- Acquire confidence that the behaviour of a refactored component is equivalent to its legacy predecessor.

- Transform a legacy component into a new refactored component.

Note that [168, 166] describe these aspects as general industrial challenges for system evolution.

1.5 Industrial Context

The industrial applications described in this thesis have been conducted at Philips in the healthcare domain. Philips is an electronics company founded in 1891 by Gerard Philips in Eindhoven, the Netherlands. The business innovation unit Image Guided Therapy (IGT) Systems makes interventional X-ray systems for several medical segments such as cardiology, radiology, neuro-radiology, electrophysiology, and surgery. The general product name is Allura. There are multiple variations of this product for the different medical segments, depending on the chosen hardware configuration and software packages. In Figure 1.2 a possible configuration of the Allura product is depicted. The common factor of the product variations is that X-ray movies of a patient's body can be made in real-time.

Figure 1.2: *Allura*

The patient lies on the table and is positioned between the X-ray generator and detector which are mounted to the C-arc of the product. The table and C-arc of the product can be manoeuvred by means of a software user interface. At one end of the C-arc the generator transmits an X-ray beam through the patients' body and at the other end of the C-arc the detector receives the residual radiation. This received radiation is transformed into an image which can be processed and viewed by the physician and other operating room personnel. The variations of the product are for a large part determined by the software applications needed

for specific medical segments. If, for example, a physician wants to place a stent into the aorta of a patient, then the product is used to navigate the stent through the patient's arteries to the target position. The arteries of the patient can be made visible by injecting a contrast medium.

Next a part of the architecture and components of the interventional X-ray system are explained as far as needed to understand the applications described in this thesis. The embedded software of the interventional X-ray system is deployed on a cluster of PCs and devices that cooperate to achieve various clinical procedures. The control of power to these components is the responsibility of a central Power Distribution Unit (PDU). Clinical users of an individual PC cannot control the power of the PC without using the PDU, as depicted in Figure 1.3. The PDU also controls communication signals related to the start-up and shut-down of the PCs.

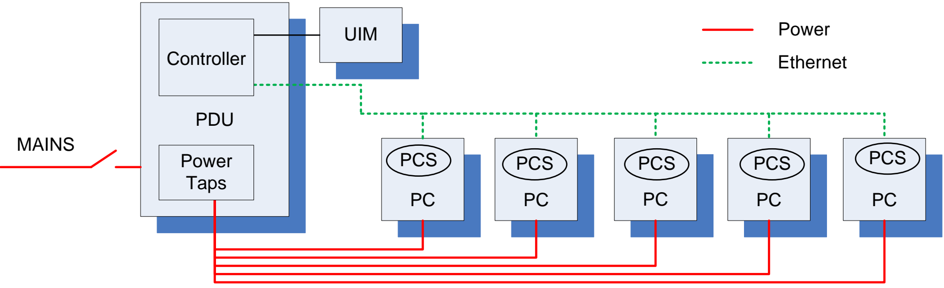


Figure 1.3: *Components Involved in Start-Up and Shut-Down*

As can be seen in Figure 1.3, each PC includes a Power Control Service (PCS) which is a software component used for exchanging power-related communication commands between running applications within a PC and the PDU through an Ethernet network. As a typical example of powering off the system, the PDU sends a message instructing all PCSs to gradually shut-down first the running applications and next the Operating Systems (OS), in an orderly fashion. The PDU is connected to a User Interface Module (UIM).

In this thesis, a number of chapters address the PCS and PDU components, related to different phases of the evolution of these components. Chapter 3 describes the development of a new PCS. This PCS instance, however, was never introduced in the field because the overall project was cancelled. For the successor of the cancelled project, the concepts were different and a new PCS was created by another team. This PCS is for an old version of the PDU, which is described in Chapter 6, since also the PDU has been renewed. The differences between the old and the new PDU are explained in Chapter 4. The refactored instance of the PCS for the new PDU is presented in Chapter 7. The Chapters 5 and 8 describe cases related to the components responsible for positioning the X-ray beam with respect to the patient.

1.6 Thesis Outline

Related work for the techniques used in this thesis is described in Chapter 2. Some conclusions can be found in Chapter 9. An overview of the other chapters is depicted in Figure 1.4 which is a refinement of Figure 1.1.

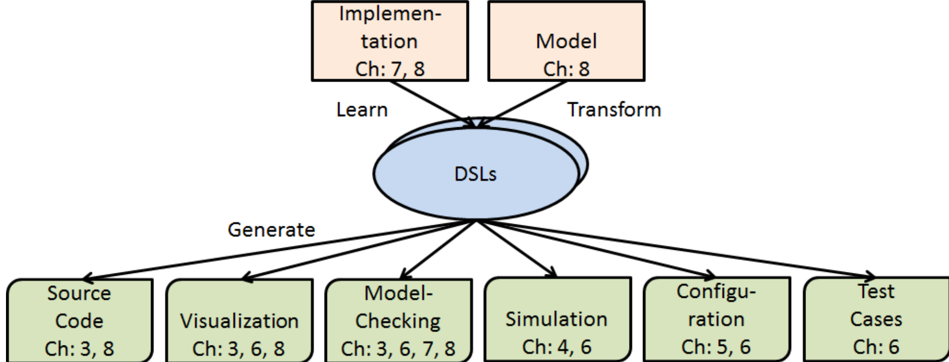


Figure 1.4: *Chapter Overview*

Chapter 3 reports on the application of ASD that has been used to design and integrate a new software component for the replacement of a legacy hardware component. This chapter is based on the following papers:

- Jozef Hooman, Robert Huis in 't Veld, and Mathijs Schuts. *Experiences with a Compositional Model Checker in the Healthcare Domain.* In Foundations of Health Information Engineering and Systems (FHIES 2011), LNCS 7151, pages 93-110, Springer-Verlag, 2012 [75].

- Ammar Osaiweran, Mathijs Schuts, and Jozef Hooman. *Experiences with Incorporating Formal Techniques into Industrial Practice.* In Journal Empirical Software Engineering, Volume 19, Issue 4, Pages 1169-1194, August 2014 [113]. This is the journal version of the following paper:

- Ammar Osaiweran, Mathijs Schuts, Jozef Hooman, and Jacco Wesselius. *Incorporating Formal Techniques into Industrial Practice: an Experience Report.* In Proceedings 9th International Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA 2012), ENTCS, volume 295, pages 49-63, 2013 [115].

The last paper [115] is also included in the PhD thesis of A.A.H. Osaiweran [112], the second paper [113] is not. The application of ASD has been done by the author of this thesis based on his ideas described in [126].

Chapter 4 describes the application of POOSL to explore new concepts for system evolution. It is based on the following two papers:

- Mathijs Schuts and Jozef Hooman. *Formalizing the Concept Phase of Product Development.* In Proceedings FM 2015: Formal Methods, LNCS 9109, pages

605-608, Springer International Publishing, 2015 [128]. This is an extended abstract of the following paper:

- Mathijs Schuts and Jozef Hooman. *Formal Modelling in the Concept Phase of Product Development*. In Proceedings Conference on Software Engineering Research & Practice (SERP 2015), WORLDCOMP'15, pages 3-9, CSREA Press, USA, 2015 [127].

Chapter 5 describes the use of DSLs for keeping a legacy software component longer in the evolution stage by improving its maintainability and extensibility. From DSL instances, configuration files that instrument the legacy component are generated. The contents is from the paper:

- Mathijs Schuts and Jozef Hooman. *Improving Maintenance by Creating a DSL for Configuring a Fieldbus*. In Proceedings of the Workshop on Domain-Specific Modeling (DSM 2016), pages 28-34, ACM, 2016 [130].

Chapter 6 extends the approach from Chapter 5 by the integration of a model-based technique to increase the confidence in DSL instances. The following two papers form the basis of Chapter 6:

- Mathijs Schuts and Jozef Hooman. *Using Domain Specific Languages to Improve the Development of a Power Control Unit*. In Proceedings 2015 Federated Conference on Computer Science and Information Systems, Annals of Computer Science and Information Systems, Volume 5, IEEE, pages 781-788, 2015 [129].

- Mathijs Schuts and Jozef Hooman. *Industrial Application of Domain Specific Languages Combined with Formal Techniques*. In Proceedings Workshop on Real World Domain Specific Languages (RWDSL), The International Symposium on Code Generation and Optimization (CGO), pages 2:1-2:8, 2016 [131].

Chapter 7 presents an approach on how model learning and equivalence checking have been used to acquire confidence that the behaviour of a refactored component is equivalent to its legacy predecessor. This chapter is based on:

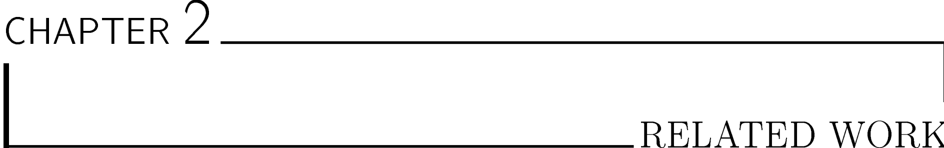
- Mathijs Schuts, Jozef Hooman, and Frits Vaandrager. *Refactoring of Legacy Software using Model Learning and Equivalence Checking: an Industrial Experience Report*. In Proceedings 12th International Conference on integrated Formal Methods (iFM 2016), LNCS 9681, pages 311-325, Springer, 2016 [132].

Chapter 8 describes a technique to automatically transform a legacy component into a new refactored component by means of a DSL. This chapter is based on unpublished work.

The author also contributed to the following papers, which are related to applying new techniques in an industrial context, but are not included in this thesis:

- Ivan Kurtev, Mathijs Schuts, Jozef Hooman, and Dirk-Jan Swagerman. *Integrating Interface Modeling and Analysis in an Industrial Setting*. In Proceedings 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2017), pages 345-352, 2017 [91].

- Ammar Osaiweran, Mathijs Schuts, Jozef Hooman, Jan Friso Groote, and Bart van Rijnsoever. *Evaluating the effect of a lightweight formal technique in industry.* In Journal on Software Tools for Technology Transfer, Volume 18, Issue 1, pages 93-108, February 2016 [114].
- Jan Friso Groote, Ammar Osaiweran, Mathijs Schuts, and Jacco Wesselius. *Investigating the effects of designing industrial control software using push and poll strategies.* In Computer Science Report 11/16, Eindhoven University of Technology, 2012 [66].
- Mathijs Schuts, Feng Zhu, Faranak Heidarian, and Frits Vaandrager. *Modelling Clock Synchronization in the Chess gMAC WSN Protocol.* In Proceedings 1st Workshop on Quantitative Formal Methods (QFM 2009), EPTCS 13, pages 41-54, 2009 [133].



CHAPTER 2 RELATED WORK

This chapter provides an overview of the work related to the techniques that have been applied for the research in this thesis. For the organisation of this chapter, the overview of techniques as described in Section 1.4 is taken.

Analytical Software Design (ASD) The ASD approach has been inspired by the formal Cleanroom software engineering method [94, 121] which is based on systematic step-wise refinement from formal specification to implementation. As observed in [28], the Cleanroom method lacks tool support to perform the required verification of refinement steps. The tool ASD:Suite can be seen as a remedy to this shortcoming. The additional code generation features of the tool make the approach attractive for industry.

Related to this combination of formal verification and code generation are, for instance, the formal language VDM++ [55] and the code generator of the industrial tool VDMTools [159]. Similarly, the B-method [8], which has been used to develop a number of safety-critical systems, is supported by the commercial Atelier B tool [37]. The SCADE Suite [51] provides a formal industry-proven method for critical applications with both code generation and verification. FALKO [25] is a software package created by Siemens. It uses Abstract State Machines [24] to simulate a software design before C++ code is generated. FALKO has been applied by the operator of the Vienna subway system for creating railway process models. SPARK is a formally defined subset of the Ada [78] programming language and has been applied in industry for over 20 years [34].

Compared to ASD, most of these methods are less restricted and, consequently, correctness usually requires interactive theorem proving. ASD is based on a careful restriction to data-independent control components to enable fully automated verification of a limited set of properties. In addition to the above listed work, [172] contains an overview on the application of formal methods in industry including 70 references.

Parallel Object-Oriented Specification Language (POOSL) POOSL fills a gap between expensive commercial modelling tools (like MATLAB [97] and Rational Rhapsody [77]) that require detailed modelling, often close to the level of code, and drawing tools (such as Visio and UML drawing tools) that do not allow simulation. Related to the POOSL approach is the OMG specification called the Semantics of a Foundational Subset for Executable UML Models (fUML) [111] with, e.g., the Cameo Simulation Toolkit [95].

Related to our use of POOSL in the concept phase of development is the application of ACL2 [88] for hardware development [125]. The ACL2 logic is used for the specification of the communication structure of a system on chip. Formal proofs of desirable properties, e.g., messages reach their destination, show the correctness of the specifications. Also at Vanderlande [160], simulation is used in the context of designing warehouses. They have created generic models describing the building blocks. Using these building blocks, a model of a warehouse can be constructed, next these models can be simulated, and the performance of the warehouse can be predicted.

The application of formal techniques early in the development process was already reported in [45]. It describes the application of tools such as PVS [116] to requirements modelling for spacecraft fault protection systems. Although the specification language of PVS appears to be easily understandable by engineers, the interactive proof of properties is far from trivial. Hence, the conclusion of [45] proposes a rapid prototyping approach, where prototypes are tested against high level objectives.

The difficulty to use formal techniques early in the development process, when there are many uncertainties and information changes rapidly, is also observed in [61]. They investigated the use of formal simulations based on rewriting logic, namely Maude executable specifications [36]. The approach has been applied to the design of a new security protocol.

Vertical DSL According to Ward [167], DSLs improve the maintainability of new software due to code size reduction and improved readability. In [13], a DSL for a command-and-control simulator for Army fire support has been defined. They observed an improvement of the maintainability and extensibility by a higher level of abstraction using domain concepts.

There are a number of relevant applications of DSLs 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 [87], 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 [109]. From this DSL, a simulation of hardware behaviour which enables software in the loop simulation has been generated. At ASML [30], they also created another family of DSLs called Control Architecture Reference Model (CARM) to explore the throughput of tasks in lithography systems. From DSL instances, POOSL simulation models were generated to simulate the throughput depending on the number of cores of a CPU. In [33], 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 [3], an extension of the C language has been constructed [164, 162]. Ex-

periences with DSLs at Philips are reported in [106, 105]. A DSL was created to explore new concepts for a collision prevention component which is part of the movement controller. To increase confidence in the new concepts, the generation of analysis models was added to the DSL.

In addition to the topics mentioned above, this thesis also addresses the maintainability of legacy software. Cao et al. [31] report that most DSL research focuses on new development. Concerning legacy software, [52, 49] try to improve the maintainability of large software systems by an evolutionary process that can be used to incrementally refactor an implementation and raise abstraction using an extensible programming language called SugarJ. SugarJ enables the use of multiple small embedded DSLs. These DSLs are embedded in Java source code. They have created DSLs for XML and SQL. Major benefit of these embedded DSLs is that instances can be statically validated, e.g. to check whether an XML file has the right closure. The embedded DSLs are placed in libraries, to enable incremental introduction of generated code in a code base. By importing the DSLs in a source file, the DSL can be used only when required. The SugarJ approach has been applied to a Java Pet Store and the Eclipse sources, a project with a code base of 10 million LOC. These applications show that it is possible to incrementally apply small scale changes on large scale systems. In addition, they created a program called "sweet tooth" to analyse a legacy software code base for identifying new refactor opportunities based on patterns taken from the generated code of the DSLs. Stoel et al. [145] report on designing and applying a specification language called Rebel in the context of the ING bank. Rebel is based on Rascal [12] which is a tool created for meta-programming. Specifications in Rebel can be model checked, new implementations can be generated, and test cases can be generated to test legacy implementations.

At a large Austrian electricity company with more than 140 power plants, a DSL has been developed for a legacy software system. The system describes schedules that are used for trading electricity between companies [142]. They conclude the following: This project report shows that a DSL-based system refactoring can provide benefits in terms of reduced code redundancy for an improved maintainability of a code base.

Bodeveix et al. [20] describe an early experiment to combine DSLs and formal techniques. In this paper, the correctness of instances of a DSL for process scheduling is verified using the B method [8]. To increase the use of formal methods in industry, the paper [82] 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 [108].

Different from the related work above, we additionally used DSLs to improve the use of configuration files. Tolvanen et al. [150] describe over 20 industrial applications of DSLs, including the generation of configuration files. They observed that DSLs are beneficial for design guidance and early error prevention or detection. In addition, they report that DSLs increase productivity due to the raised level of abstraction.

Software Product Line Engineering (SPLE) addresses modelling and analysis of commonality and variability. System configurations can be generated from the resulting models. Berger et al. [17] conducted a survey on the industrial usage

of variability modelling. They conclude that the SPLE community focuses on creating methods and tools for new systems and a shift might be needed towards support for legacy software.

Model learning and equivalence checking Besides LearnLib, which is used in this thesis, there are several other approaches to extract the behaviour of a legacy implementation. Static analysis methods concentrate on the analysis and transformation of source code. For instance, the commercial Design Maintenance System (DMS) has been used in several industrial projects to re-engineer code. DMS is based on abstract syntax tree (AST) representations of programs [9].

Whereas static analysis techniques focus on the internal structure of components, learning techniques aim at capturing the externally visible behaviour of a component. Process mining extracts business logic based on event logs [154]. In [86], a combination of static analysis and process mining has been applied to a financial management system, identifying tasks, actors, and their roles. Process mining can be seen as a passive way of learning which requires an instrumentation of the code to obtain event logs.

Active learning techniques [10, 143] do not require code instrumentation, but need an adapter to interact with a running system. In this approach, a learning algorithm interacts with a software component by sending inputs and observing the resulting output, and uses this information to construct a state machine model. Active learning has, for instance, been successfully applied to learn models of (and to find mistakes in) implementations of protocols such as TCP [54] and TLS [43], to establish correctness of protocol implementations relative to a given reference implementation [7], and to generate models of a telephone switch [96] and a printer controller [141]. Learning-based testing [53] combines active learning and model checking. In this approach, which requires the presence of a formal specification of the system, model checking is used to guide the learning process. In [53] three industrial applications of learning-based testing are described from the web, automotive and finance domains.

Model transformation In their research agenda, van Deursen et al. [157] identify that migration from one model-based development tool to another, for instance, using model to model transformations, is required for system evolution. Mens et al. [99] discuss definitions of model transformation techniques. Model to model transformation techniques are described and compared in [39].

To facilitate the migration from one model-based development tool to another, the Object Management Group (OMG) has defined and specified a modelling transformation language: Query Views Transformations (QVT) [118]. According to [136], many UML modelling tools provide the possibility to import and export models in a XML-based language called XML Metadata Interchange (XMI). In the XMI format these models can be transformed from one modelling tool to another using existing XML manipulation tools, e.g., the Extensible Stylesheet Language Transformation (XSLT) tool. Czarnecki et al. [39] describe that XSLT has scalability issues and that Tratt [151] has overcome this shortcoming by creating a DSL for applying model transformations. Another approach to replace XSLT is the UML Model Transformation (UMT) tool. [64] describes the empirical successes

of UMT by means of a number of transformations, e.g., from UML to J2EE and XDoclet, WSDL-to-UML, UML-to-WSDL and UML-to-BPEL4WS.

Mooij et al. [103] describe an industrial project at Philips in which legacy XML files, used for describing field service procedures, are semi-automatically rejuvenated into DSL model instances. These rejuvenated DSL models are transformed into redesigned DSL models from which new C++ source code is generated. In two other projects Mooij et al. [107] used Abstract Syntax Trees (ASTs) for the rejuvenation of legacy C++ code. The rejuvenated models are captured in a DSL and altered using model transformation, before new C++ code is generated.

The previously described model transformation technologies assume that the models to be transformed are in XML format. However, the modelling tool from which we want to transform models stores the models in a proprietary format. We used a DSL to transform legacy models into models for a new tool. A related approach has been applied at ASML for a language called Logical Action Component Environment (LACE). LACE defines how different sub-systems are allowed to interact with each other. In [149], Tikhonova et al. describe how LACE models are transformed using DSL technology into Event-B models to gain access to multiple specification analysis tools. In [134], an industrial case study at General Motors is described where legacy models are converted to AUTomotive Open System ARchitecture (AUTOSAR) models. For the model to model transformations, the MDWorkbench together with the Atlas Transformation Language (ATL) [84] was used [135]. ATL is a QVT-like transformation language [85].

CHAPTER 3

Language for Creating New Components

In this chapter, we describe our experiences with Analytical Software Design (ASD) during a real industrial development project. First the motivation for applying ASD is given and the ASD approach is introduced as far as needed to understand this chapter. Then we present the workflow that has been used to combine ASD with traditional approaches for developing software components. Next the industrial case is introduced and we describe the application of the presented workflow to this case. Two errors which were found after completing the formal verification using ASD are presented. Finally we discuss the results achieved and our main observations.

3.1 Motivation for Applying ASD

The interventional X-ray system introduced in Section 1.5 is based on a component-based architecture and therefore consists of a large number of hardware and software components. An obsolete hardware component is replaced by a new component. Analytic Software Design (ASD) [161] has been applied to model a new software component that interfaces with the hardware component. ASD is a horizontal DSL, with a tabular notation, that can be used to describe the signature and behaviour of data-independent control components. Instances of the ASD language can be model checked [29, 60], with respect to a predefined set of checks. From the language instances, source code is generated. In addition, visualizations can be generated. The ASD approach is supported by the commercial tool ASD:Suite of the company Verum.

Our focus is on the embedding of the ASD approach in the industrial workflow. As observed in [19, 172], there are quite a number of reports about industrial case studies with formal methods, but very few publications describe second or subsequent use. Similarly, the literature about the incorporation of formal methods in the standard industrial development process is very limited. This limitation is evident in [172] which reports about the use of formal methods in industry and its

references (over 70 publications) which report about the use of formal methods in industry.

Osaiweran et al. [68, 67] describe an analysis of the first usage of the ASD approach at Philips. They show that it leads to the development of components with fewer reported defects compared to components developed with more traditional development approaches. Therefore, formal methods are gradually becoming more credible in developing software within Philips. However, in the healthcare domain this requires validated tools and the incorporation of these new techniques into well-defined development and quality management processes. This requires an answer to a number of questions such as:

Test and integration To what extent does the formal verification affect the test and integration phase? Are certain tests no longer needed? Which tests are still essential to guarantee the quality of components? Can formal interface models be used to generate test cases?

Quality management Which artefacts have to be included in the version management system; do we need the models, the generated code, or also the version of the tool? How does the approach fit into the existing quality management system (e.g., concerning the required review procedures)?

Workflow How can formal techniques be tightly integrated with standard development processes in industry? How to deal with changes; how flexible is the approach?

Design What is the impact of the modelling and formal verification on the project planning? Is more time needed during the design phase? Can the test and integration phase be shortened?

3.2 Fundamentals of ASD

ASD is a component-based, model-driven approach that combines formal mathematical methods with industrial software development methods. The approach is supported by the commercial tool ASD:Suite of the company Verum. The tool supports two types of models which are both based on state machines and described by a similar tabular notation: *interface models* and *design models*.

- The *interface* models are used to define the interaction protocol between important system components in a formal way. An interface model describes not only signatures of methods to be invoked by other components but also the external behavior exposed to client components. Internal interactions with used components are not present in this model.
- The *design* model describes the internal behavior of a component given its interface model and typically uses the interface models of other components. By means of the ASD:Suite it can be verified formally whether the design model refines the interface model. Very important in our industrial context

is that ASD:Suite supports complete code generation from design models to a number of programming languages (C, C++, C#, Java). Hence, design models provide a platform-independent description of internal component behaviour.

ASD uses a Sequence-Based Specification Method [120] to obtain complete and consistent specifications. This means that the response to all possible sequences of input stimuli has to be defined. Sequences that cannot happen must be declared illegal explicitly. The tool ASD:Suite translates the sequence-based specifications into CSP. The FDR2 model checker [56] is used to verify a predefined fixed set of properties such as refinement and absence of deadlock and livelock. Error traces are visualized by means of sequence diagrams.

ASD:Suite hides the CSP and FDR2 details, which is important to enable industrial usage. To enable automated refinement checks, the use of design models is restricted to components with data-independent control decisions. Components that involve data manipulations or algorithms are implemented by other techniques.

Scalability is addressed by the ASD interface model which defines a contract between the ASD design models and the used components. ASD interface models also enable compositional verification [74] in the sense that the formal verification uses only the interfaces of the used components, without knowing their implementation details.

3.2.1 ASD Interface Models

To illustrate the ASD approach we use a small camera example. We start with an ASD interface model which represents the external behaviour of the server (Camera). Clients can use this behaviour. There are two ways of communication between client and server:

- *Procedure calls* from client to server, which are synchronous in the sense that the client has to wait until the server is ready to accept the call. Next the client is blocked until the server returns the call. Hence, all calls to a server are serialized. There are two types of calls:
  - Void calls, which return a void reply to signal the completion of the call
  - Valued calls, which return a value upon completion
- *Callbacks* from server to client, which are asynchronous events that can be sent by the server immediately. The client decouples a callback by putting an event in the queue. Emptying the queue has priority over accepting new synchronous calls. By default a queue of 5 elements deep is used during model checking, but it is possible to change the number of elements of the queue.

To model an interface, e.g., to trigger callbacks, an interface model may also contain:

- *Internal modelling events*, which can be used in interface models to model that an event can happen without an external trigger. Then the trigger is

the modelling event. Modelling events can be optional (meaning that they may happen) or inevitable (meaning that they will happen eventually).

For the camera example we have the following signature, i.e., sets of calls, callbacks, and modelling events:

- APICamera is the name of a grouped set of calls and contains three calls:
  - PowerOn(): valued, with two possible return values: OnOK, OnFailed
  - PowerOff(): void
  - Click([in]exposureTime:int): void, Click has an input argument exposureTime of type int
- CBCamera is the name of a grouped set of callbacks and contains four callbacks:
  - CBPicture([in]photo:image), CBPicture has an input argument photo of type image
  - CBOn()
  - CBEmptyBattery()
  - CBOnFailed()
- INTCamera is the name of a grouped set of modelling events and contains four internal modelling events to trigger the four callbacks above:
  - PictureMade, which is inevitable
  - SwitchedOn, which is inevitable
  - SwitchedOnFailed, which is inevitable
  - BatteryEmpty, which is optional

An interface model is represented as a state machine which defines the behaviour between client and server. Such an ASD interface model plays a similar role as a protocol state machine of UML [23]. An ASD interface not only describes the services offered by the server; it also specifies the calls allowed by the client. So it can be seen as a contract between client and server, similar to the Design by Contract approach [100].

In the Camera example, the ASD interface is shown in Figure 3.1. There are four states: Off, SwitchingOn, On, and TakingPicture. In each state the actions for all possible events are defined. The "+" behind the name of an event indicates that it is a valued call. The tool ASD:Suite generates for each reachable state a so-called canonical sequence which is a minimal sequence of input stimuli leading to the state from the initial state (which is always the first state mentioned). This canonical sequence is written behind the name of each state. Observe that the state machine is non-deterministic; in state Off there are two possible responses to the valued call PowerOn.

In state Off the calls PowerOff and Click are declared to be Illegal which means that the client should not call these functions in this state. If the client makes an

Illegal call, the program will raise an exception and exit. The modelling events are Disabled in state Off. The use of these internal modelling events is illustrated by state SwitchingOn; when modelling event SwitchedOn or SwitchedOnFailed occurs, the camera sends the corresponding callback to the client.

Note that the rules 2, 12, 21 and 30 are hidden; they can be used for invariants which are not discussed in this chapter. Figure 3.2 shows a visual representation of the state machine of this interface, which is generated automatically by ASD:Suite from the table of Figure 3.1.

	Interface	Event	Guard	Actions	State Variable Updates	Target State
1	**Off <>**					
3	APICamera	PowerOn+		APICamera.OnOK		SwitchingOn
4	APICamera	PowerOn+		APICamera.OnFailed		Off
5	APICamera	PowerOff		Illegal		-
6	APICamera	Click(exposureTime)		Illegal		-
7	INTCamera	PicutureMade		Disabled		-
8	INTCamera	SwitchedOn		Disabled		-
9	INTCamera	SwitchedOnFailed		Disabled		-
10	INTCamera	BatteryEmpty		Disabled		-
11	**SwitchingOn <APICamera.PowerOn+>**					
13	APICamera	PowerOn+		Illegal		-
14	APICamera	PowerOff		APICamera.VoidReply		Off
15	APICamera	Click(exposureTime)		Illegal		-
16	INTCamera	PicutureMade		Disabled		-
17	INTCamera	SwitchedOn		CBCamera.CBOn		On
18	INTCamera	SwitchedOnFailed		CBCamera.CBOnFailed		Off
19	INTCamera	BatteryEmpty		CBCamera.CBEmptyBattery		Off
20	**On <APICamera.PowerOn+, INTCamera.SwitchedOn>**					
22	APICamera	PowerOn+		Illegal		-
23	APICamera	PowerOff		APICamera.VoidReply		Off
24	APICamera	Click(exposureTime)		APICamera.VoidReply		TakingPicture
25	INTCamera	PicutureMade		Disabled		-
26	INTCamera	SwitchedOn		Disabled		-
27	INTCamera	SwitchedOnFailed		Disabled		-
28	INTCamera	BatteryEmpty		CBCamera.CBEmptyBattery		Off
29	**TakingPicture <APICamera.PowerOn+, INTCamera.SwitchedOn, APICamera.Click(exposureTime)>**					
31	APICamera	PowerOn+		Illegal		-
32	APICamera	PowerOff		APICamera.VoidReply		Off
33	APICamera	Click(exposureTime)		Illegal		-
34	INTCamera	PicutureMade		CBCamera.CBPicture(photo)		On
35	INTCamera	SwitchedOn		Disabled		-
36	INTCamera	SwitchedOnFailed		Disabled		-
37	INTCamera	BatteryEmpty		CBCamera.CBEmptyBattery		Off

Figure 3.1: *ICamera: Interface Model of the Camera Component*

To design the camera component, two other components will be used: a battery component and a shutter component. The interfaces of these components are

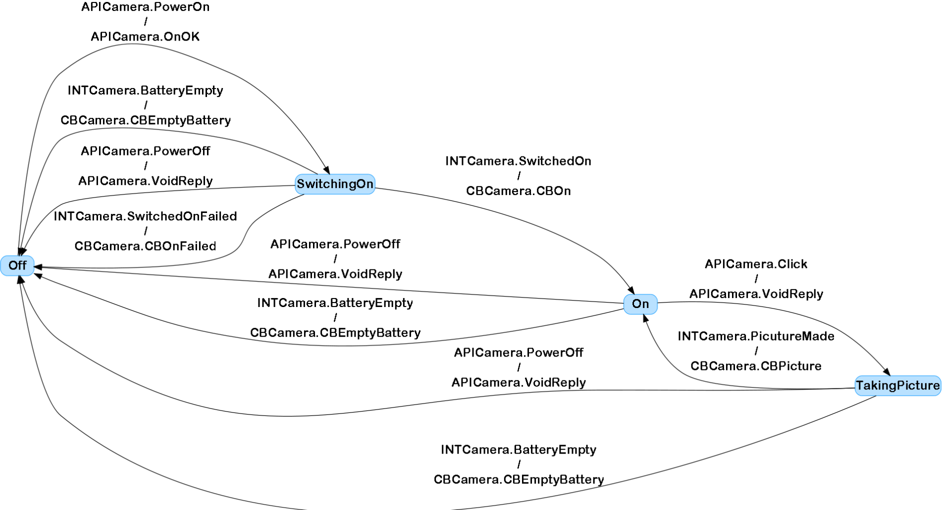


Figure 3.2: *Graphical Representation of the Interface Model of the Camera*

shown in Figure 3.3 and Figure 3.4, respectively, where illegal and disabled events are hidden. Also these two interfaces are non-deterministic. Observe that the battery interface uses a state variable EmptyDetected to describe the set of allowed traces. Rule 16 of Figure 3.3 shows that a Charge call in state BatteryOn does not lead to an externally visible action, but it updates variable EmptyDetected if it is true. Hidden is the rule which expresses that the Charge call is illegal if EmptyDetected equals false.

	Interface	Event	Guard	Actions	State Variable Updates	Target State
1	**BatteryOff** <>					
3	APIBattery	BatteryOn		APIBattery.VoidReply		BatteryOn
5	APIBattery	CheckBattery+		APIBattery.Battery_OK		BatteryOff
6	APIBattery	CheckBattery+		APIBattery.Battery_Empty		BatteryOff
8	INTBattery	Charge	EmptyDetected==true	NoOp	EmptyDetected=false	BatteryOff
9	**BatteryOn <APIBattery.BatteryOn>**					
12	APIBattery	BatteryOff		APIBattery.VoidReply		BatteryOff
13	APIBattery	CheckBattery+		APIBattery.Battery_OK		BatteryOn
14	APIBattery	CheckBattery+		APIBattery.Battery_Empty		BatteryOn
15	INTBattery	EmptyDetected	EmptyDetected==false	CBBattery.CBBatteryEmpty	EmptyDetected=true	BatteryOff
16	INTBattery	Charge	EmptyDetected==true	NoOp	EmptyDetected=false	BatteryOn

Figure 3.3: *IBattery: Interface Model of the Battery Component*

These interfaces can be checked using the built-in model checker of ASD:Suite which verifies a number of properties such as guard completeness, absence of state invariant violations, absence of livelock (a livelock occurs when a component is permanently busy with internal behaviour without any visible response to the client), and absence of deadlock (a deadlock occurs when nothing can happen and the component refuses all communication).

	Interface	Event	Guard	Actions	State Variable Updates	Target State
1	**Off <>**					
3	APIShutterr	SwitchOn+		APIShutterr.OnOK		On
4	APIShutterr	SwitchOn+		APIShutterr.OnFailed		Off
8	**On <APIShutterr.SwitchOn+>**					
11	APIShutterr	SwitchOff		APIShutterr.VoidReply		Off
12	APIShutterr	Click(exposureTime)		APIShutterr.VoidReply		TakingPicture
14	**TakingPicture <APIShutterr.SwitchOn+, APIShutterr.Click(exposureTime)>**					
17	APIShutterr	SwitchOff		APIShutterr.VoidReply		Off
19	INTShutter	PicutureMade		CBShutter.CBPicture(photo)		On

Figure 3.4: *IShutter: Interface Model of the Shutter Component*

3.2.2 ASD Design Models and Model Checking

To implement components, the ASD approach uses so-called design models. Design models are tables similar to interface models with a few differences. A design model must be deterministic and it has a number of associated interface models: an implemented interface model and a number of used interface models.

The model checker of ASD:Suite can be used to check conformance with the implemented interface and consistency with the used interfaces. Complete executable code can be generated from the design model, where a choice can be made between a number of programming languages (currently C, C++, C#, and Java). Next the camera example is used to explain the ASD design models and the model checker.

ASD Design Models

The design of the camera component is depicted in Figure 3.5, where ovals represent ASD interface models and the rectangle denotes an ASD design model. The up arrow denotes that the design model refines the interface model of Figure 3.1, as will be explained later in this section. The design uses the interfaces IBattery of the battery (Figure 3.3) and IShutter of the shutter (Figure 3.4).

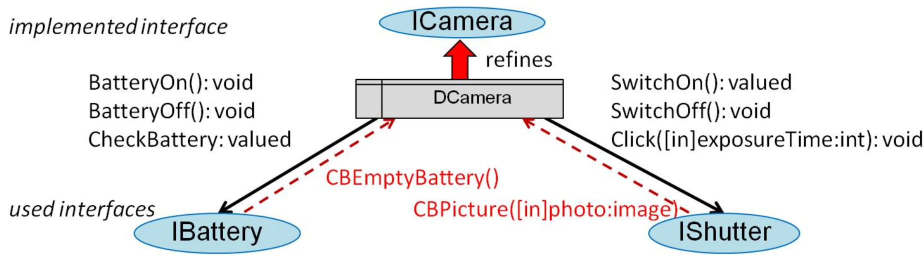


Figure 3.5: *Design of the Camera Component*

Figure 3.6 shows the ASD design model for the camera example, hiding illegal and disabled events. Similar to the interface model, the table must be complete in the sense that for all events an action must be defined (which might be Illegal or Disabled). The events now include callbacks of the used interfaces, such as

the callbacks CBBatteryEmpty from the battery and CBPicture from the shutter. Similarly, the actions may contain calls to used interfaces. For instance, in state Off rule 3 expresses that, as a response to the PowerOn call, the function CheckBattery of the battery is called. As indicated by the "+" behind the name, this is a valued call and in state CheckingBattery the component is waiting for a response. In such a so-called blocking state, all other events are blocked.

	Interface	Event	Guard	Actions	State Variable Updates	Target State
1	**Off <>**					
3	APICamera	PowerOn+		Battery:APIBattery.CheckBattery+		CheckingBattery
8	Battery:CBBattery	CBBatteryEmpty		CBCamera.CBEmptyBattery		Off
12	**CheckingBattery <APICamera.PowerOn+>**					
17	Battery:APIBattery	Battery_Empty		APICamera.OnFailed		Off
18	Battery:APIBattery	Battery_OK		APICamera.OnOK; Battery:APIBattery.BatteryOn; Shutter:APIShutterr.SwitchOn+		SwitchingOn
23	**SwitchingOn <APICamera.PowerOn+, Battery:APIBattery.Battery_OK>**					
31	Shutter:APIShutterr	OnOK		CBCamera.CBOn		On
32	Shutter:APIShutterr	OnFailed		CBCamera.CBOnFailed		Off
34	**On <APICamera.PowerOn+, Battery:APIBattery.Battery_OK, Shutter:APIShutterr.OnOK>**					
37	APICamera	PowerOff		Battery:APIBattery.BatteryOff; Shutter:APIShutterr.SwitchOff; APICamera.VoidReply		Off
38	APICamera	Click(exposureTime)		Shutter:APIShutterr.Click(exposureTime); APICamera.VoidReply		TakingPicture
41	Battery:CBBattery	CBBatteryEmpty		CBCamera.CBEmptyBattery		Off
45	**TakingPicture <APICamera.PowerOn+, Battery:APIBattery.Battery_OK, Shutter:APIShutterr.OnOK, APICamera.Cli**					
48	APICamera	PowerOff		Battery:APIBattery.BatteryOff; Shutter:APIShutterr.SwitchOff; APICamera.VoidReply		Off
52	Battery:CBBattery	CBBatteryEmpty		CBCamera.CBEmptyBattery		Off
55	Shutter:CBShutter	CBPicture(photo)		CBCamera.CBPicture(photo)		On

Figure 3.6: *DCamera: Design Model of the Camera Component*

The design model assumes that CBPicture only occurs in state TakingPicture; in all other states the callback is illegal or blocked. The correctness of this assumption is verified by the model checker using the interface specification of the shutter. CBPicture is a so-called *solicited* callback, because it is received as a response to the Click call to the shutter. On the other hand, CBBatteryEmpty can be received in any non-blocking state and is called an *unsolicited* callback.

In an ASD design model it is not possible to make control decisions based on parameters of function calls. For instance, in the camera example it is not possible to make a case distinction on the exposureTime parameter of the Click call. If control would depend on the value of such a parameter, it has to be sent to a so-called *foreign component*, which is not implemented using ASD. Such a foreign component can analyse the value and return different values or callbacks to indicate the required control.

The semantics of a design model is such that callbacks from used components can always be received and they are put into a so-called *callback queue* (FIFO). Client calls are serialized, that is, at any point in time at most one client call is executed. Callbacks have priority over client calls. Initially, and after the completion of a rule case, first the callback queue is inspected. If this queue is not empty, the rule case corresponding to the first callback is executed. When

the callback queue is empty and no call is being processed, a new call may be accepted. An illegal call or callback leads to a halt of the component.

Compositional Model Checking

The tool ASD:Suite contains a fixed number of checks on design models. Figure 3.7 shows a screenshot of part of the tool with a Verify window. Verification includes

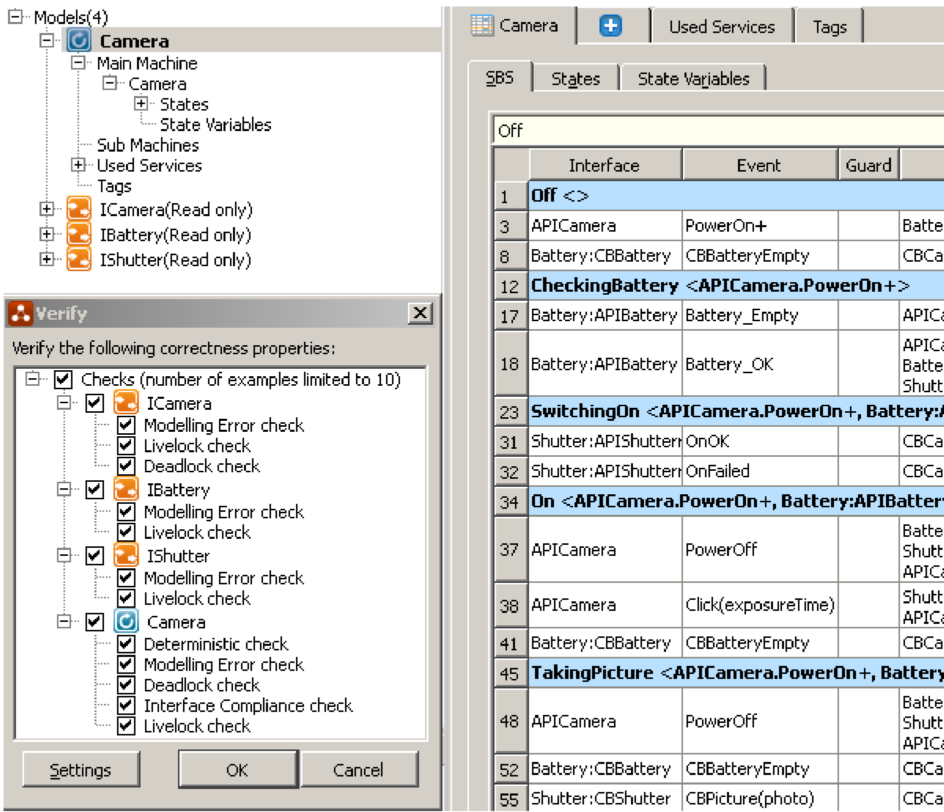


Figure 3.7: *Screenshot ASD:Suite with Verify Window*

the previously discussed checks on all interfaces, i.e., implemented and used interfaces. In addition there are specific checks for design models, such as a check to ensure that the design model is deterministic. Most important is a check on the consistency of all interfaces. The design should adhere to all interface models of used components and it should conform to the implemented interface. Conformance has been defined formally in the failures-divergence model of CSP [123] and is checked with the underlying FDR2 model checker [56]. Note that FDR2, which is an abbreviation of Failures/Divergence Refinement 2, is in fact a refinement checker.

In the camera example, verification revealed quite a number of problems in the models presented above. A few errors found by the model checker:

- In used interface IBattery it is possible to get a BatteryOff call in state BatteryOff; this is a race condition between a PowerOff call and a callback CBBatteryEmpty send by the battery components. Note that the callback is put into the callback queue of the camera component while the BatteryOff call is processed. This problem is corrected by improving the battery interface as shown in Figure 3.8 where rule 4 now allows a BatteryOff call.
- The model checker complained about an attempt to switch the shutter on when it was already on, which is not allowed by the interface of the shutter as specified in Fig 3.4. Analysing this situation, it turned out that in the design model it was forgotten to switch the shutter off when a CBBatteryEmpty has been received (rules 41 and 52 in Figure 3.6).

	Interface	Event	Guard	Actions	State Variable Updates	Target State
1	**BatteryOff <>**					
3	APIBattery	BatteryOn		APIBattery.VoidReply		BatteryOn
5	APIBattery	CheckBattery+		APIBattery.Battery_OK		BatteryOff
6	APIBattery	CheckBattery+		APIBattery.Battery_Empty		BatteryOff
8	INTBattery	Charge	EmptyDetected==true	NoOp	EmptyDetected=false	BatteryOff
9	**BatteryOn <APIBattery.BatteryOn>**					
12	APIBattery	BatteryOff		APIBattery.VoidReply		BatteryOff
13	APIBattery	CheckBattery+		APIBattery.Battery_OK		BatteryOn
14	APIBattery	CheckBattery+		APIBattery.Battery_Empty		BatteryOn
15	INTBattery	EmptyDetected	EmptyDetected==false	CBBattery.CBBatteryEmpty	EmptyDetected=true	BatteryOff
16	INTBattery	Charge	EmptyDetected==true	NoOp	EmptyDetected=false	BatteryOn

Figure 3.8: *Improved Battery Interface IBattery*

- Similarly, it was forgotten to switch the battery off when an attempt to switch the shutter on fails (rule 32 in Figure 3.6).
- As another race condition, the model checker shows that a callback CBPicture might be received in state Off, namely after a CBBatteryEmpty in state TakingPicture. This has been repaired by adding a rule case in the design to receive the callback, but not forwarding it to the client.

ASD:Suite has a nice visualization of error traces, which makes it easy to find the cause of an error. Figure 3.9 shows the visualization of the last problem mentioned above. It shows the lifeline of the Camera component, with a client, the callback queue (called DPC+Q), its used components Battery and Shutter, and an environment which triggers modelling events in the used interfaces.
The corrected design model for the camera is shown in Figure 3.10, with changes in rules 11, 32, 41, and 52.

Observe that the verification is compositional since it uses only the interfaces of the used components. Hence, the used components can be developed independently according to their interface. Also note that there is no obligation to develop these components with ASD. Typically, components that involve data manipulations will be implemented differently and conformance to their ASD interface is checked by means of testing. To support manually implemented components, the ASD:Suite can generate a dummy implementation of which only the body of the calls need to be filled in manually.

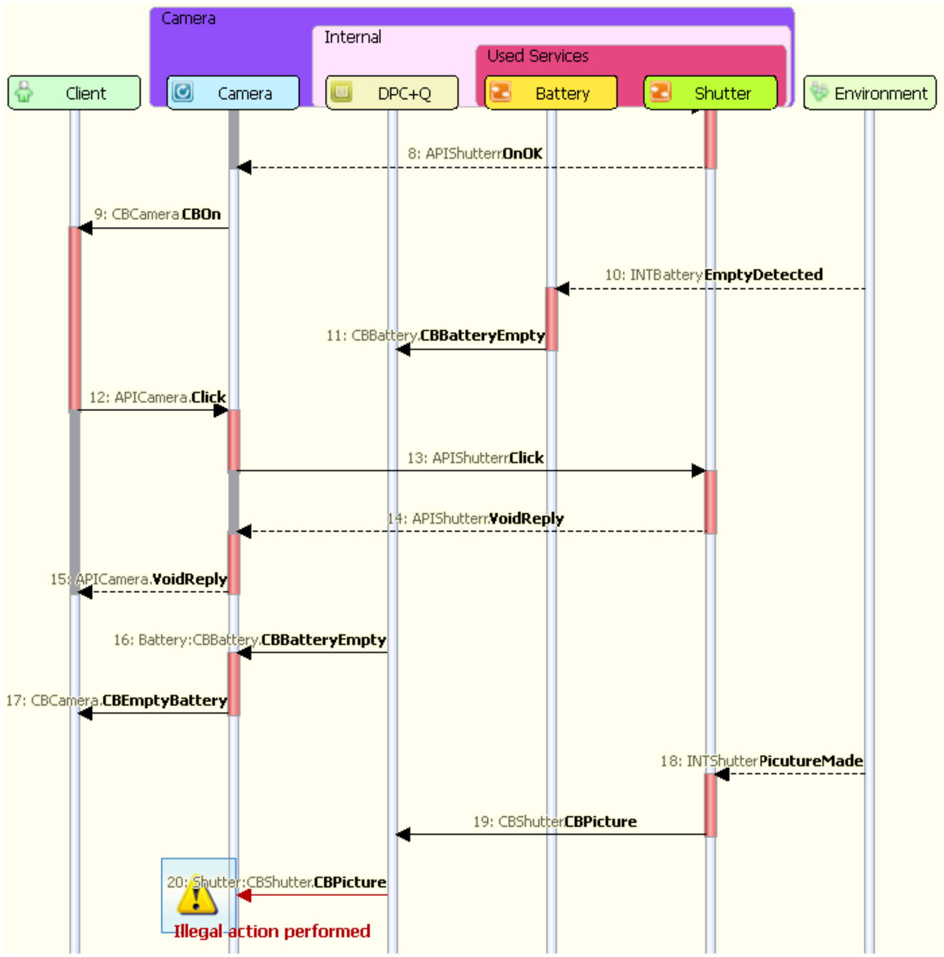


Figure 3.9: *Visualization of an Error Trace*

Finally, observe that the components have a strict communication pattern; a client of a component can only perform synchronous calls and might receive asynchronous callbacks from the component. Similarly, the component itself will only perform synchronous calls on its used components and receive callbacks from them. In this way the absence of deadlocks in communication between components is achieved by construction. Note that a deadlock in a design model, e.g. because it has a terminal state, is detected by the model checker.

3.3 Integrating ASD in Industrial Workflow

The development process of software, used in projects within the context of Philips, is an evolutionary iterative process. That is, the entire software product is developed through accumulative increments, each of which requires regular review

	Interface	Event	Guard	Actions	ariable U	Target State
1	**Off <>**					
3	APICamera	PowerOn+		Battery:APIBattery.CheckBattery+		CheckingBattery
8	Battery:CBBattery	CBBatteryEmpty		CBCamera.CBEmptyBattery		Off
11	Shutter:CBShutter	CBPicture(photo)		NoOp		Off
12	**CheckingBattery <APICamera.PowerOn+>**					
17	Battery:APIBattery	Battery_Empty		APICamera.OnFailed		Off
18	Battery:APIBattery	Battery_OK		APICamera.OnOK; Battery:APIBattery.BatteryOn; Shutter:APIShutterr.SwitchOn+		SwitchingOn
23	**SwitchingOn <APICamera.PowerOn+, Battery:APIBattery.Battery_OK>**					
31	Shutter:APIShutterr	OnOK		CBCamera.CBOn		On
32	Shutter:APIShutterr	OnFailed		CBCamera.CBOnFailed; Battery:APIBattery.BatteryOff		Off
34	**On <APICamera.PowerOn+, Battery:APIBattery.Battery_OK, Shutter:APIShutterr.OnOK>**					
37	APICamera	PowerOff		Battery:APIBattery.BatteryOff; Shutter:APIShutterr.SwitchOff; APICamera.VoidReply		Off
38	APICamera	Click(exposureTime)		Shutter:APIShutterr.Click(exposureTime); APICamera.VoidReply		TakingPicture
41	Battery:CBBattery	CBBatteryEmpty		CBCamera.CBEmptyBattery; Shutter:APIShutterr.SwitchOff		Off
45	**TakingPicture <APICamera.PowerOn+, Battery:APIBattery.Battery_OK, Shutter:APIShutterr.OnOK, APICa**					
48	APICamera	PowerOff		Battery:APIBattery.BatteryOff; Shutter:APIShutterr.SwitchOff; APICamera.VoidReply		Off
52	Battery:CBBattery	CBBatteryEmpty		CBCamera.CBEmptyBattery; Shutter:APIShutterr.SwitchOff		Off
55	Shutter:CBShutter	CBPicture(photo)		CBCamera.CBPicture(photo)		On

Figure 3.10: *Correct Design Model for the Camera*

and acceptance meetings by several stakeholders. When creating components with ASD, not all code can be generated. Hence, an application will always contain foreign components. For the manually crafted code we take the test-driven development (TDD) [14] approach. Figure 3.11 outlines the flow of activities in a development increment, highlighting the steps to incorporate both the ASD and TDD approaches.

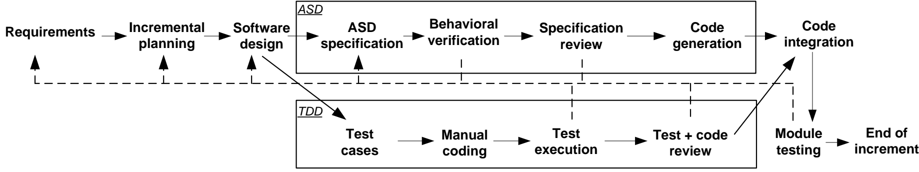


Figure 3.11: *Steps Performed in a Development Increment*

Each increment starts with identifying a list of requirements to be implemented by team members. As soon as requirements are approved by lead architects, the development team is required to provide work breakdown estimations that include, for instance, required functionalities to be implemented, necessary time, potential risks, and efforts.

For planning and tracking a Work Breakdown Structure (WBS) is created. A WBS consists of tasks that need to be completed in a certain order to obtain a finished product. At the beginning of each increment a new WBS for that increment is created. For each task, the time needed to complete the task is estimated with the Wideband Delphi estimation method [144]; this means that the effort needed for every task is estimated by two or more experienced software designers in the first phase. In the second phase, software designers need to get consensus on the estimate. The outcome of the estimate is then used in the planning. Not all tasks of the WBS are estimated; some are derived from historical data. Examples are overhead and average time needed to solve a defect.

Team and project leaders take these work breakdown estimations as an input for preparing an incremental plan, which includes the list of functions to be implemented in a chronological order, tightly scheduled with strict deadlines to realize each of them. The plan is used as a reference during a weekly progress meeting for monitoring the development progress.

The construction of software components starts with an accepted design, i.e., a decomposition into components with clear interfaces and well-defined responsibilities. Usually such a design is the result of iterative design sessions and approved by all team members.

When the aim is to use ASD, a common design practice at Philips is to organize components in a hierarchical control structure. Typically, there is a main component on the top which is responsible for high-level, abstract behaviour, e.g., dealing with the main modes and the transitions between these modes. More detailed behaviour is delegated to lower-level components which deal with a particular mode or part of the functionality.

The control components that include state machines are then developed using ASD, whereas TDD is used for the other type of components such as those used for data processing since ASD may not be very suitable for developing components responsible for data computations or manipulations.

The ASD and the TDD approaches are explained below, describing the well-known TDD approach only briefly.

3.3.1 The TDD Approach

The TDD approach starts each increment with the definition of a set of test cases. To validate the test set, it is checked whether all tests fail on an empty implementation. Next the components are developed iteratively, gradually increasing the set of passed test cases. When all tests succeed, the code of the components is reviewed by the team before it is integrated with the code generated by the ASD approach.

3.3.2 The ASD Approach

An overview of the activities in the ASD approach we followed is depicted in Figure 3.12. Starting point is a structure of the components as described above.

ASD components can be developed in a top-down, bottom-up or middle-out fashion. Each component is developed using ASD according to the steps 1 through

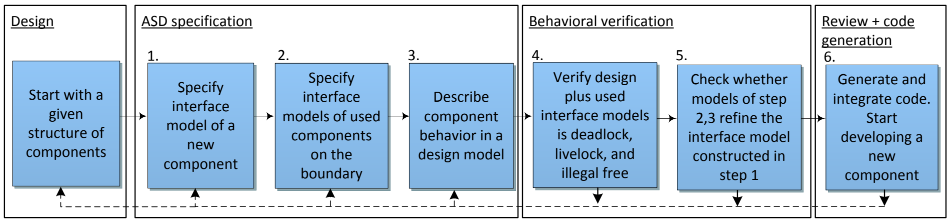


Figure 3.12: *The ASD Approach to Develop Components*

6 of Figure 3.12:

1. Specification of externally visible behaviour. At first, an ASD interface model of the component being developed is created. This interface model might already exist if the component is used by a component that has been developed already, as explained in the next step.

2. Specification of external behaviour of used components. Similarly, ASD interface models are constructed to formalize the external behaviour of components that are used by the component under development.

3. Model component design. An ASD design model of the component is created; it describes the complete behaviour of the component, including calls to used interface models (as created in step 2) to realize proper responses to client calls.

4. Formal verification of the design model. Using the FDR2 model checker controlled by the ASD:Suite tool, the design model is exhaustively checked on the absence of deadlocks, livelocks, and illegal interactions with the used interface models. When an error is detected by FDR2, ASD:Suite presents a nice sequence diagram and allows users to trace the source of the error in the models.

5. Formal refinement check. ASD:Suite is used to check whether the design model created in step 4 is a correct refinement of the interface model of step 1. As in the previous steps, errors are visualized and related to the models to allow easy debugging.

6. Code generation and integration. After all formal verification checks are successfully accomplished, source code can be generated from the model.

3.4 Context of the PCS

Interventional X-ray systems have been introduced in Section 1.5. As can be seen in Figure 1.3 of Section 1.5, each PC includes a Power Control Service (PCS) which is used for exchanging power-related communication commands between running applications within a PC and the Power Distribution Unit (PDU) through an internal Ethernet control network. A typical scenario is powering off the system; the user presses the off button on the User Interface Module (UIM), PDU then sends a message instructing all PCSs to gradually shut-down first the running applications and next the operating systems (OS), in an orderly fashion.

Figure 3.13 sketches the PCS in a PC as a black-box, surrounded by a number of internal and external concurrent components, located inside and outside the PC. For instance, the PDU interacts with the PCS to reboot or shut-down the

PC. Moreover, the PCS can also send events to the PDU to enable or disable a number of buttons on the UIM.

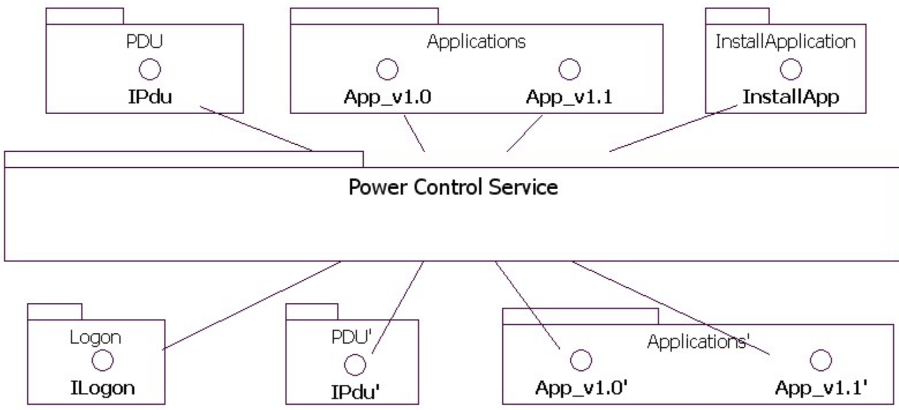


Figure 3.13: *The PCS as a Black-Box Surrounded by Concurrent Components*

Another example of a concurrent component is the *InstallApplication* which is an external component used to install and upgrade software on the PC. During the installation of software on a PC, the PCS instructs the running applications to stop, start or restart.

The main function of the PCS is to coordinate all requests to and from these concurrent components. Due to the concurrent execution, controlling the flow of events among the components is rather complex, and the architecture sketched in Figure 3.13 is prone to deadlocks, livelocks, race conditions and illegal interactions. Since the PCS is deployed on every PC, any error is replicated on every PC and potentially leads to serious problems for the entire system.

Moreover, the PCS may lose connection with other applications at any time due to a failure of one of these applications or with the PDU (e.g., due to a network outage). The PCS has to be robust against such failures, especially when the PCS is in the middle of executing a particular scenario. When the PCS detects that the system is in a faulty state, it should take appropriate actions and log the events for further diagnostics by the field service engineer. As soon as the cause of malfunctions has disappeared, the PCS ensures that all its internal components are synchronized back with other external components to a predefined state.

Due to the complex behaviour of the PCS and the many possible regular and exceptional execution scenarios that need to be considered carefully, the ASD technology has been used to develop the control part of the service, and to specify the external behaviour of the components on the boundary of the PCS. The TDD approach has been applied to develop the non-control part of the service and the components on the boundary of the PCS.

3.5 Steps of Developing the PCS

In this section we report about the component-based development of the PCS by one developer working full-time on the project from October 2010 till October 2011. The development process contained five increments, each implementing part of the PCS functionality. The ASD-based development of control components and the development of other components using TDD has been carried out in parallel, as depicted in Figure 3.11. Below we describe the development process in more detail, concentrating on the ASD part, since the TDD approach is more conventional.

Requirements and incremental planning The development process was started by identifying the scope and the requirements of the PCS. At early stages of development it was difficult to reach agreement with all stakeholders, since they had different wishes concerning the required functionality. The process of getting consensus took up to two-thirds of the total time. During this negotiation phase, requirements and design documents were iteratively written and reviewed by team members to reflect the current view of the solution and as input for further discussions.

Hence, the development process initially took place in a context where scope and requirements were very uncertain and changed frequently - even within a single increment. Additionally, the features required to be implemented in every increment were only known at a very abstract level, such as: “In increment 2 automatic logon of the default user of a PC has to be implemented". The requirements of each increment were only acquired just at the beginning of the increment, which put more pressure on meeting the strict deadlines.

Software design The design of the PCS consists of a hierarchy of components, as depicted in Figure 3.14. In this decomposition, ASD components are depicted in a gray colour, whereas light coloured components have been developed using TDD. Not shown in the picture are commonly used components such as tracing (to facilitate in-house diagnostics by developers) and logging (to facilitate diagnostic by field service engineers in the field).

The decomposition of PCS components was accomplished top-down in steps, such that each lower level comprises components with more detailed behaviour. Below we describe each ASD component individually, sketching briefly their related responsibilities.

- The *PduEventController* component mainly serves commands issued by the external components: the PDU and the InstallApplication, for instance. It contains a top-level state machine that captures overall global states (or modes) of a PC: normal mode, installing, starting/stopping applications, operational, et cetera.

- The *InstallTransitioning* component implements the detailed behaviour of the installation mode of the top-level state machine. The component is responsible of safeguarding the detailed transitions from normal mode to installation mode, and vice versa.

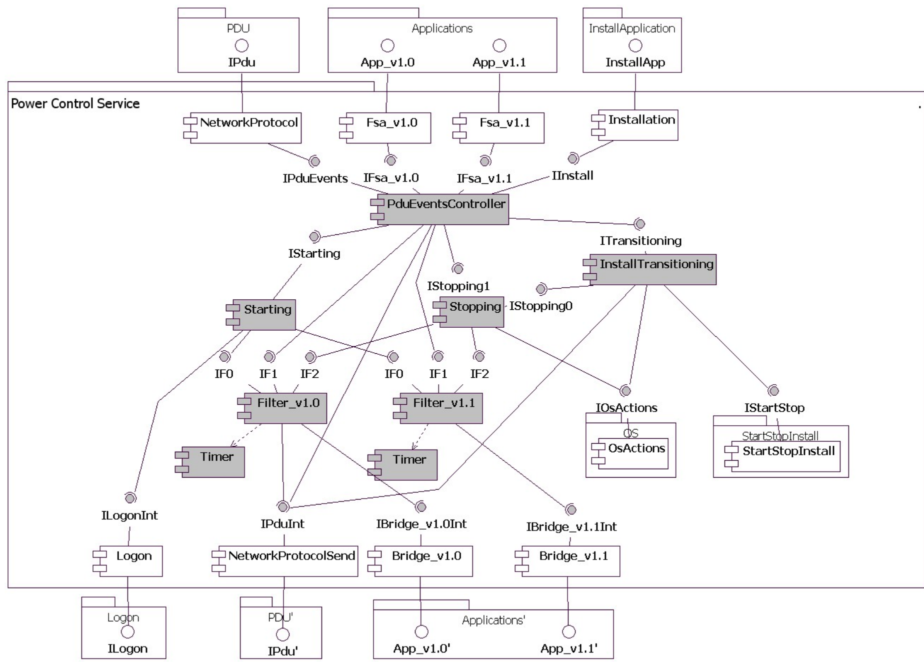


Figure 3.14: *Components of the PCS*

- The *Starting* component launches the clinical applications of a PC and logs-on/off the default clinical user. It ensures that clinical applications are successfully started.

- The *Stopping* component is responsible for ensuring that closing the running applications and then shutting down or rebooting the OS is done sequentially.

- The *Filter* components are responsible for starting, restarting, and stopping the applications within a predefined fixed time. They are the facade to the components located outside the boundary of the PCS.

Experience shows that most novice ASD users tend to design rather large components leading to large ASD models [126, 67], in terms of many hundreds of rules. Although this might be acceptable in traditional development methods, it leads to serious problems when using formal techniques such as ASD:Suite. In fact, formal techniques such as ASD would show the consequences of poor designs early during development while in conventional practices these consequences would appear at later stages when the developed system becomes overly complex and difficult to maintain. In general, the key issues encountered with large models were as follows.

- Verifiability: while verifying large models one quickly runs into the main limitation of model checking, namely the state-space explosion problem. Veri-

fication may take a large number of hours or might even be impossible for large models.

- Maintainability: design models which contain a substantial number of input stimuli and states are difficult to adapt or to extend. This leads to problems when requirements change or functionality has to be added.
- Readability: large design models are hard to read and to understand. Design reviews will consume a large amount of development time.

During the development of the PCS, the first point was the main concern. Earlier experience showed that as soon as the state space explosion problem is faced, the development process is blocked and components have to be refined and redesigned from scratch. Since code generation is only allowed when the formal verification checks succeed, this causes some visible deviations between hours estimated in the WBS's and actual hours spent for development.

Therefore, from the start we designed the PCS to be decomposed into rather small components, described using small models. Although the ASD approach shown in Figure 3.12 does not prescribe an order in which the components are realized, we used a top-down, step-wise refinement approach. This effectively helped us distributing responsibilities and maintaining a proper degree of abstraction among all components. In this way we obtained a set of formally verifiable components.

Specification and formal verification of ASD models The ASD models were specified using the ASD:Suite version 6.2.0, following the ASD approach. Each component was modelled in isolation with interfaces of boundary components. An example structure of ASD models related to the *Stopping* component is depicted in Figure 3.15.

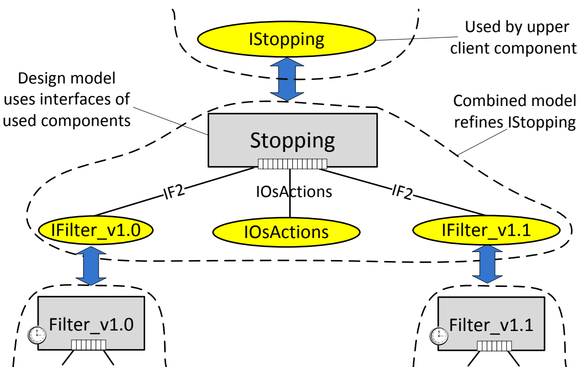


Figure 3.15: *Structure of ASD Models of the* Stopping *Component*

The Figure depicts the interface model *IStopping* that describes the external behaviour of the *Stopping* component excluding related lower-level interactions. As shown in the figure, the interface is refined by a design model and a number of interface models that represents lower-level ASD and non-ASD components.

Upon the completion of their specification, the interface and design models were verified also in isolation. The formal verification was performed on a remote server located at the company Verum.

The ASD formal properties introduced in Section 3.2.2 were performed step by step for the models of each component. We first started checking correctness of interface models. When this check succeeded, we searched for illegal scenarios and then for deadlocks in the design model. After that we checked determinism and finally refinement of designs against the interfaces.

Note that although we followed this order, the entire verification process is rather iterative. That is, when a property fails and certain changes to the models are required, we re-check all previously succeeded properties.

Usually, this reveals quite a number of errors, both in design and interface models, e.g. illegal interactions. Since changes in interface models affect other boundary components this sometimes leads to a chain of changes. However, since our components are kept small, it is easy and fast (usually less than a second) to re-check these other components.

Specification review, code generation and integration Although the formal verification is very useful to detect errors, it does not guarantee that the design model realizes the intended behaviour. For instance, the correct relation between client calls and calls to used components is not checked. Also the value of parameters is not verified. Hence, when all formal checks succeed, the ASD models were reviewed by the project team. The review process performed for the ASD models was similar to the review process of any normal source code developed manually. After the team review, including corrections and a re-check of the formal verification, C# source code was generated automatically using ASD:Suite. This code is then integrated with the manually coded components.

Testing At the end of each increment the ASD generated code plus the manually coded components were exposed to black-box testing. Corresponding test cases were specified and implemented before and in parallel to the implementation of the increment. As a result of the black-box testing, a total of three errors were found, two of which were related to ASD components and one to the manually coded components. Note that the manually coded components are rather straightforward and less complex than the control part developed in ASD. The error in the manually coded components was due to the existence of a null reference exception. We detail ASD errors in the next section.

The entire PCS code was exposed to further testing on module level at the end of all increments. After that, both manually written code and test code were carefully reviewed by team members. As a result of review, minor issues were identified and immediately resolved. Test cases were rerun in order to assure that the rework after review did not break the intended behaviour of the service.

3.6 Errors Not Detected by the ASD Verification

As a result of the black-box testing, two errors were found in the ASD code throughout all increments. We refer to the two errors as:

- the *ordering* error, since it concerns the ordering of messages of multiple components, and
- the *multi-client* error, since it results from the interaction between multiple clients.

As described earlier the model checker does not detect functional errors. Therefore, we expected to find functional errors during black-box testing.

Below we explain the details of these errors, highlighting their sources and potential solutions.

The ordering error This error was not found because of the impossibility to specify and verify properties about the order of messages of two components in ASD. In our project, this concerns the *Stopping* component and the *Filter* components. Considering Figure 3.14, the *Stopping* component can receive a request to shutdown the PC from the *PduEventsController* component. The *Stopping* component first instructs the *Filter* component to stop the running applications and then waits for the result before it instructs the *OsActions* component to shutdown the OS.

As specified in rule case 19 in Figure 3.16 of the *Filter* design model, the *Filter* component starts its timer, instructs the clinical applications to stop, and transits to the *Stopping* state waiting some seconds for a notification from the applications indicating the completion of the stop request. Meanwhile, if the timer expires while waiting for the notification, the *Filter* notifies the *Stopping* component using the *Stopped* callback and then logs a "FinishedStoppingAfterTimeOut" message; see rule case 28 in Figure 3.16.

When the *Stopping* component receives the notification from the *Filter*, it instructs the *OsActions* component to shutdown the operating system and then logs a "Shutdown" message indicating that the system is shutting down.

A test case was implemented which requires the log messages to be received in a logical order. That is, the "FinishedStoppingAfterTimeOut" is received followed by the "Shutdown" message. But the test case failed since it unexpectedly received the messages in the reverse order.

The reason of this error was that when the timer expired, the *Filter* component sent the *Stopped* callback to the queue of the *Stopping* component and then tries to log the "FinishedStoppingAfterTimeOut" message. Since the queue runs in a separate execution thread, the execution context was switched such that the *Stopping* component quickly de-queued the callback, sent the shutdown request to the OS and immediately logged the "Shutdown" message before the *Filter* component logged the "FinishedStoppingAfterTimeOut" message; see the sequence diagram of Figure 3.17.

This error was hard to reproduce due to its concurrent nature. Once the error occurred, it was easy to find the cause by examining the logging produced by the

	Channel	Stimulus event	Predicate	Response	State update	Next state	Comment
1	Init<>						
2	PdsEvents_v11	Initialize		PdsEvents_v11.NullRet; Log:ILog.Initialize(suSdLogger)		Stopped	
3	PdsEvents_v11	Restart		PdsEvents_v11.NullRet		Init	
4	Starting_v11	Start		Starting_v11.NullRet		Init	
5	Stopping_v11	Stop		Stopping_v11.NullRet		Init	
6	Pm_v11:IStartupShutdownCB_v11	FinishedStopping		Illegal		-	
7	Timer:ITimerCB	Timeout		Illegal		-	
8	Stopped<PdsEvents_v11.Initialize>						
9	PdsEvents_v11	Initialize		PdsEvents_v11.NullRet		Stopped	
10	PdsEvents_v11	Restart		PdsEvents_v11.NullRet		Stopped	
11	Starting_v11	Start		Starting_v11.NullRet; Pm_v11:IStatupShutdown_v11.Start		StartedOrStarting	
12	Stopping_v11	Stop		Stopping_v11.NullRet; StoppingCB_v11.Stopped		Stopped	
13	Pm_v11:IStartupShutdownCB_v11	FinishedStopping		Log:ILog.Log($"FinishedStoppingAfterTimeOut"$)		Stopped	
14	Timer:ITimerCB	Timeout		Illegal		-	
15	StartedOrStarting<PdsEvents_v11.Initialize,Starting_v11.Start>						
16	PdsEvents_v11	Initialize		Illegal		-	
17	PdsEvents_v11	Restart		PdsEvents_v11.NullRet; Pm_v11:IStatupShutdown_v11.Restart		StartedOrStarting	
18	Starting_v11	Start		Starting_v11.NullRet		StartedOrStarting	
19	Stopping_v11	Stop		Stopping_v11.NullRet; Timer:ITimer.CreateTimerMSec(<<waitForPM); Pm_v11:IStatupShutdown_v11.Stop		Stopping	
20	Pm_v11:IStartupShutdownCB_v11	FinishedStopping		Illegal		-	
21	Timer:ITimerCB	Timeout		Illegal		-	
22	Stopping<PdsEvents_v11.Initialize,Starting_v11.Start,Stopping_v11.Stop>						
23	PdsEvents_v11	Initialize		Illegal		-	
24	PdsEvents_v11	Restart		Illegal		-	
25	Starting_v11	Start		Illegal		-	
26	Stopping_v11	Stop		Illegal		-	
27	Pm_v11:IStartupShutdownCB_v11	FinishedStopping		Timer:ITimer.CancelTimer; StoppingCB_v11.Stopped		Stopped	
28	Timer:ITimerCB	Timeout		Timer:ITimer.CancelTimer; StoppingCB_v11.Stopped; Log:ILog.Log($"FinishedStoppingAfterTimeOut"$)		Stopped	

Figure 3.16: *Design Model of the* Filter *Component*

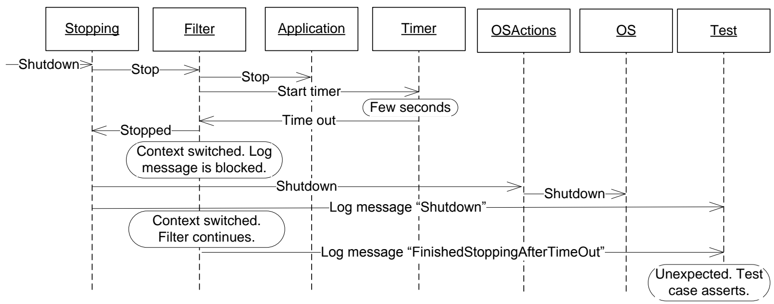


Figure 3.17: *Error Caused by Concurrent Execution of Events Due to Wrong Ordering*

application. The scenario was not detected by the model checker due to the way ASD performs compositional verification. That is, verification of the *Filter* design

model did not include the design of the *Stopping* design model.

Fixing the error was straightforward. We changed the order of responses in rule case 28 of the *Filter* component such that the “FinishedStoppingAfterTimeOut" message is logged before notifying the *Stopping* component.

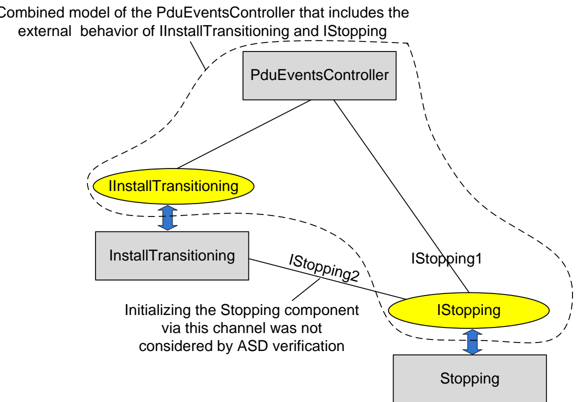


Figure 3.18: *Model Checker Could Not Detect the Error Due to a Hidden Dependency*

The multi-client error Although the model checker of ASD:Suite verified the absence of illegal events, testing showed an illegal event during the execution of the PCS. Figure 3.18 depicts the structure of the three components involved in the error: the *PduEventController*, the *InstallTransitioning* and the *Stopping* components. The *Stopping* component was initially in the *Created* state, waiting to be initialized by its client components. Upon receiving the *initialize* call, it initializes other lower-level components and then transits to the *Initialized* state, where any other *initialize* call is illegal. However, the *Stopping* component received the first initialize call from the *PduEventController* component, and then the second call from the *InstallTransitioning* component, causing the illegal error in the *Initialized* state.

The reason of not detecting this error using model checking when verifying the *PduEventController* component is that the interface model of the *InstallTransitioning* component exposes only the interaction with the client *PduEventController* component, excluding any interaction with the *Stopping* component; see Figure 3.18. More precisely, the *initialize* call from *InstallTransitioning* to the *Stopping* component is excluded from the specification and formal verification, causing a hidden dependency between the *InstallTransitioning* and *Stopping* components not visible to the *PduEventController*.

We have observed that the courses and the information Verum provides about ASD:Suite only addresses the usage of the tool. This information does not describe how to create suitable designs. The tool supports the generation of multi-threaded components. These components allow the interaction with several clients. From experience we know that it is common to have a diamond-like structure of components. The reason is that on the top of an application their is a single interface to

the client of the application and at the bottom of the application, the application is a client for another application. Since we want to check the interface between the application and another application that acts as a server to this application we need to capture the interface to the server with a component. An example of this is depicted in Figure 3.14; the PCS application interfaces with, e.g., App_v1.0 applications. The App_v1.0 applications act as a server for the PCS. In the body of the application we need multiple small components between the two previously described components on the boundary of the application that interface with other applications. For this reason, at the bottom of the application we need multiple components to be the client of the boundary component. In Figure 3.14 the Filter_v1.0 component plays this role. However, the underlying CSP/FDR2 model checker does not test for this situation and only supports checking a tree-like structure of components. Hence, additional black-box testing is needed to catch errors that might occur because of the deviation from the tree structure.

Similar to the first error, solving this issue was also straightforward. We ignored any initialize request in the *Initialized* state instead of assigning illegal responses. We manually searched for similar occurrences in other components and corrected them similarly.

3.7 Results of Developing the PCS

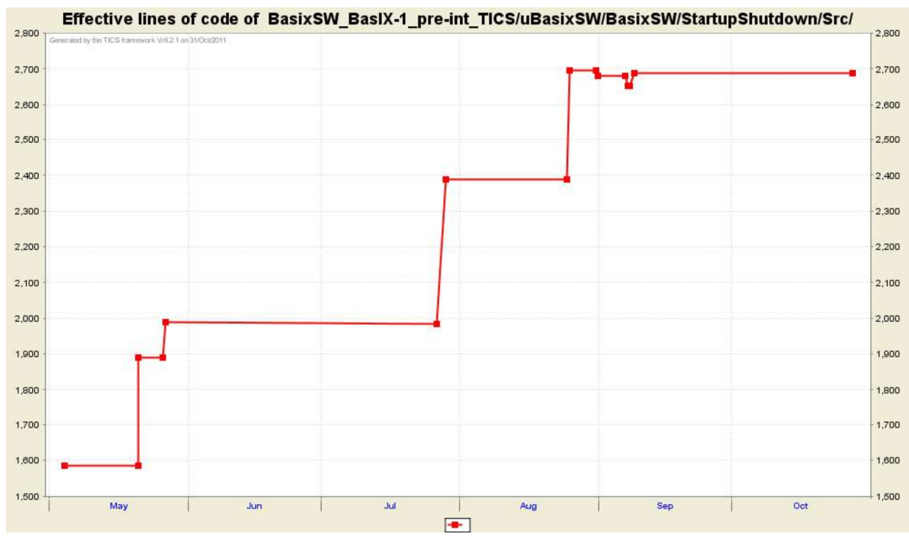


Figure 3.19: *Evolution of the Manually Coded Components*

Figure 3.19 depicts code evolution of the manually coded components, after mining the code repository using TIOBE software [70]. The figure shows only the effective lines of code (ELOC), i.e., all blank and comment lines are excluded from calculations. The code was officially placed in the repository at the start of May 2011, with approximately 1,600 ELOC of previously coded components. As

can be seen from the figure, the construction of the manually coded components was smooth and gradually evolved throughout all increments. The figure also indicates that there were no major redesign activities causing any removal of the implemented code in any increment.

Similarly, Figure 3.20 depicts the evolution of test code. The reason of having more testing code than product code is that the manually coded components were developed under the control of the TDD technique. As mentioned earlier, the TDD approach implies that test cases have to be written first, before the product code.

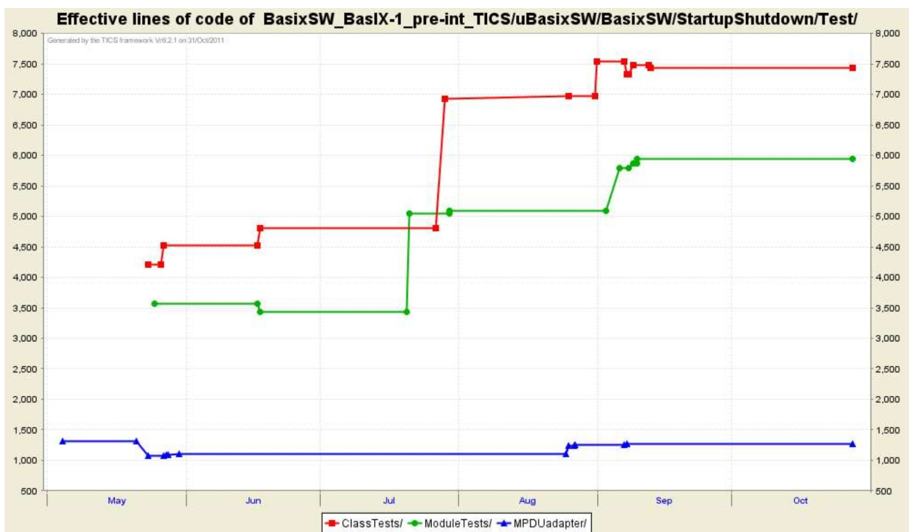


Figure 3.20: *Evolution of Test Code*

Figure 3.21 sketches the evolution of the ASD generated code, highlighting 5 versions from 5 stable baselines at the end of each increment, taken from a code management system, called IBM ClearCase [15]. We extracted such figures manually since the ASD code did not comply to the coding standard enforced by the TIOBE technology and hence was excluded from calculation by the technology since the early phase of the development process. As can be seen from the figure, the PCS appeared to already be stable since the start of increment 3. In previous projects where ASD was used [67, 68], major redesigns were needed due to the state space explosion problem. This did not happen in the PCS project since all ASD components are kept small and fit within the limits of the model checker.

In Table 3.1 we provide statistical data of the final developed ASD components after increment 5, listing all corresponding interface and design models. The first and second column include all ASD interface and design models (IM and DM respectively). The third column shows the number of rule cases of each model. These rule cases have been reviewed thoroughly by team members. The fourth and fifth column reveal the states and transitions reported from the model checker FDR2 to check deadlock freedom (which holds for all models). For the other checks

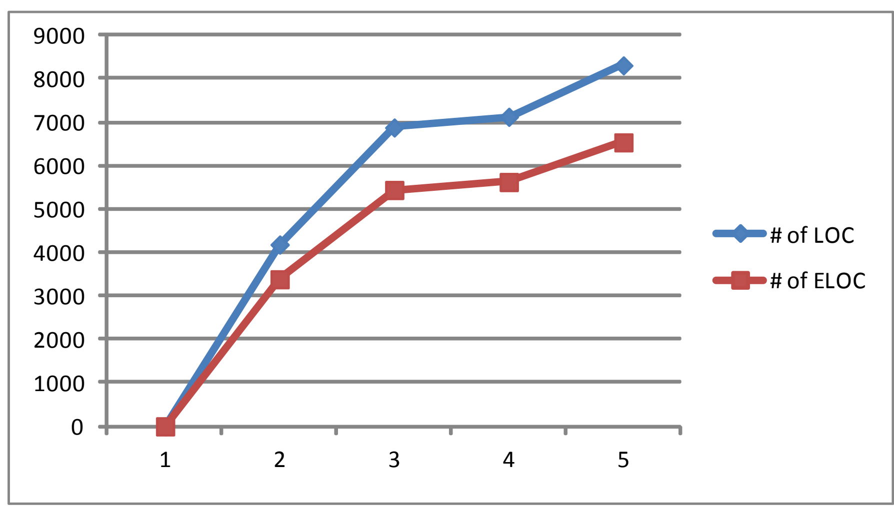


Figure 3.21: *Evolution of ASD Code*

Model	Type	Rule cases	States	Transitions	LOC	ELOC
IPdsEventController	IM	102	55	139	112	58
PdsEventController	DM	242	141	225	2891	2165
IPmFilter_v10	IM	33	17	29	37	28
IStarting	IM	10	3	4	36	13
IStopping	IM	24	9	16	117	41
IPmFilter_v11	IM	28	13	21	36	27
IPdsAdapter	IM	12	3	6	21	12
IInstallTransitioning	IM	45	11	14	61	22
ILog	IM	8	3	4	35	12
InstallTransitioning	IM	78	59	62	989	830
IStartStopInstall	IM	10	3	4	20	11
IOsActions	IM	14	3	7	22	13
PmFilter_v10	DM	46	79	113	859	712
IPm_v10	IM	25	9	13	50	19
ITimer	IM	14	5	9	26	17
PmFilter_v11	DM	32	45	59	651	549
IPm_v11	IM	18	7	8	26	17
Starting	DM	12	12	13	435	379
ICpActions	IM	8	3	3	19	10
Stopping	DM	78	51	58	1065	903
ASD runtime	-	-	-	-	803	701
Total	5D + 15I	839	-	-	8311	6539

Table 3.1: *The ASD Models of the PCS*

we obtained similar numbers.

Each interface model was verified separately, whereas every design model was verified as a combined model that includes all interface models of used components. The verification of all ASD models was conducted on a remote server at the company Verum, the provider of ASD:Suite. All models were checked in less than one second by FDR2, covering all possible execution scenarios. Compared to more traditional testing this reduced both time and effort.

Last two columns present the total number of generated lines of generated code (LOC), in the C# language. The LOC column denotes the sum of all generated source code lines, including blank and comment lines.

Table 3.2 depicts metrics related to all manually developed code. It includes the sum of all total and executable lines of code written for the product and test code.

Code	LOC	ELOC
Manual Code	8,915	3,828
Simulator Code	2,553	1,275
Class Test Code	15,180	7,437
Module Test Code	12,531	5,946

Table 3.2: *Statistical Data of the PCS*

The entire service includes 17,226 lines of ASD generated and manually written code. It includes a total of 30,264 LOC of test code. The end quality result of the PCS service is remarkable, as the entire service exhibited only 0.17 defect per KLOC, according the definition of defect described in [62]. This level of quality is much better than the industry standard defect rate of 1-25 defects per KLOC [98].

Table 3.3 depicts the hours spent during each increment. The total hours spent for developing the entire service is 1787, with average productivity of 5.8 effective lines of code per hour.

Increment	inc1	inc2	inc3	inc4	inc5
Requirements Specification	13	64	1	15	8
Design Specification	18	96	4	4	40
TDD/ASD	101	167	67.5	103	88
Verification Specification	49.5	46.5	40.5	22.5	4
Verification Report	18.5	5		2	
Test code	182.5	91	94	91.5	42
Simulator	55.5	18			16
Other		24.5	63.5	33	97.5
Total	438	512	270.5	271	295.5

Table 3.3: *Hours Spent on the PCS*

The PCS service was deployed on all PCs in the product, and further tested by independent teams who are responsible of developing the clinical applications

on each PC. The result of testing was that no errors were found and the service appeared to function correctly on every PC, from the first run.

Feedback received from the independent test teams was very positive, and the service seems to be stable and reliable. Team members of the PCS appreciated the quality of the service, and decided to further incorporate the ASD technology to the development of other parts of the system. The behavioural verification and the firm specification and code reviews provided a suitable framework for increasing the quality, assisting the work, and decreasing potential efforts devoted to bug fixing at later stages of the project.

3.8 Concluding Remarks

We have described our experiences with the PCS case at Philips with a component-based development method which is supported by the commercial formal tool ASD:Suite. The proposed workflow also includes test-driven development. This approach has been used for the development of a basic power control service. We list our main observations and lessons learned.

Test and integration Concerning the code generated by ASD:Suite, TDD tests can be safely discarded since all possible execution scenarios have been covered by the model checker of this tool. However, it is important to test the combination of ASD components and hand-written components. In the PCS project this revealed a few errors.

Observations from other projects at Philips using more conventional approaches shows that integrating concurrent components is usually a challenging task. It is often the case that components work correctly on their own, but do not function as expected when they are integrated with one another. Sometimes, errors are profound in length, hard to analyse and often tough to reproduce due to the concurrent nature of components.

Our experience with ASD differs from the observations mentioned above. Design errors were detected by the model checker early and automatically before any single line of code is being written or generated. The behavioural verification thoroughly checked the correctness behaviour of components under all circumstances of use. It was often the case that fixing an error caused other errors to emerge, which were deeper in length and complexity than a previous one, but these design errors were detected with the click of a button. Fixing these errors was done iteratively until components became neat and free from all sources of errors. Since formal verification of each ASD design model was done with the interface specification of the boundary components, integrating the code of all ASD design models is often quick and accomplished without errors.

Quality management While applying the proposed workflow, we observed a few tensions with the current quality management system. The code generated by ASD:Suite does not comply to the required coding standards provided by the TIOBE technology. Moreover, the fact that ASD forces the designer to define the response to all possible stimuli in all states leads to very robust code, but

it decreases the test coverage. In our case, it is acceptable for quality managers to exclude ASD generated code from coverage metrics and coding standards. In fact, the quality of the generated code turned out to be very good, since the PCS components have been used frequently by several parts of the system without any problem report.

In the version management system, ASD models and code are stored. Code is used for a fast build process, independent of the ASD:Suite tool. The models are used for maintenance and to include change requests. New versions of the ASD:Suite tool accepts models from previous versions.

Workflow In the PCS project a lot of time was needed to clarify the requirements, since there were many stakeholders at different sites. We believe that in such a situation the formal ASD interface models are very useful. Since ASD requires complete interface models, requirements have to be complete and clear. Discussions to clarify the requirements resulted into new and changed requirements and certainly improved the quality of the requirements.

Moreover, after identifying parts of the system that are most likely rather stable, these parts can already be implemented using ASD in parallel with ongoing discussions about unclear requirements. If the design is based on a set of small components this can be done, since adapting and extending small ASD models has proven to be easy. When large models are being used, this could prove to be cumbersome. Further, the definition of ASD interfaces enables concurrent engineering of components.

As mentioned above, an important benefit of the proposed workflow is that the test and integration phase becomes more predictable.

Design The use of ASD has a clear impact on the design and the definition of components. Because formal verification and code generation with ASD is only possible for control components, the design should make a clear separation between data and control. Control components are generated using ASD:Suite whereas test-driven development is used for the data components. Especially for designers used to object-oriented design this requires a paradigm shift.

Another important aspect is that large ASD components should be broken down into smaller components to enable efficient model checking; as a guideline a design model should not contain more than 250 rule cases, a few asynchronous callbacks, leading to not more than approximately 3000 lines of code. With these restrictions, the formal technique is rather easy to use without much training and models are easy to understand and to modify. This is similar to traditional programming where breaking a program down into smaller parts is highly recommended to reduce the effort of applying changes.

CHAPTER 4

LANGUAGE FOR EXPLORING NEW SYSTEM CONCEPTS

This chapter describes a case where we applied the Parallel Object Oriented Specification Language (POOSL). We propose an approach to explore new system concepts and motivate the use of POOSL. The POOSL models are simulated and tested to check their intended behaviour.

4.1 Motivation for Applying POOSL

In industry, the traditional development process from concept to a validated product is depicted in Figure 4.1, see for instance [90]. It describes six distinct phases between concept and product. During the concept phase of a new sys-

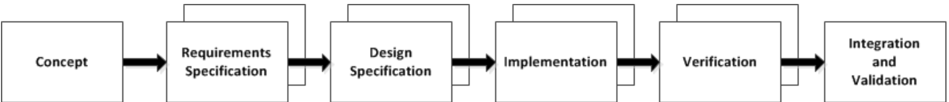


Figure 4.1: *Traditional Process Framework*

tem release an informal document is being created with a high level description of the concept. This document is reviewed and agreed upon by all stakeholders. The document consists of a decomposition of the developed product, the different hardware and software components it consists of, the responsibilities per component, and the interaction between the components, possibly with an informal interface description. From the concept description, different development groups concurrently start developing the component they are responsible for. This may also include third party components developed by other companies.

The process framework depicted in Figure 4.1 provides a structured way to come from concept to product and allows the concept to be decomposed into different components such that multiple development groups can concurrently work

on the different components. A frequently occurring problem in industry, however, is that the integration and validation phase takes a large amount of time and is rather uncontrollable because many problems are detected in this phase and might require a redesign of components.

An important reason for these problems is the informal nature of the concept phase [35]. Clearly, this leads to ambiguities and inconsistencies. Moreover, only a part of the complete behaviour is described in an informal document, often only a part of the basic functional behaviour without taking errors or non-functional aspects into account. The complete behaviour is defined during the implementation phase of the different components. Hence, a large part of system behaviour is implicitly defined during the implementation phase. If multiple development groups work in parallel in realizing the concept, the integration phase can take a lot of time because the independently developed components do not work together seamlessly. Another problem is that during the integration phase sometimes issues are found in which hardware is involved. Then it is usually too late to change the hardware and a workaround in software has to be found.

To prevent these types of problems, like others [92, 117], we propose the use of formal modelling techniques in the concept phase, because it is early in the process and all consecutive phases can benefit from an improved unambiguous concept description. Moreover, errors made in this phase are very costly to repair in a later phase [21, 169].

By making a formal model of the system in the concept phase, ambiguities, contradictions and errors are removed from the informal concept description. During modelling one is forced to think about the exceptional behaviour, such as a failing component, early in the development process. Many questions needs to be answered which would be implicitly defined during the implementation phase otherwise. Moreover, by formalizing interface descriptions, less problems during the integration phase are expected. Figure 4.2 depicts a graphical representation of the proposed extension of the product realisation framework.

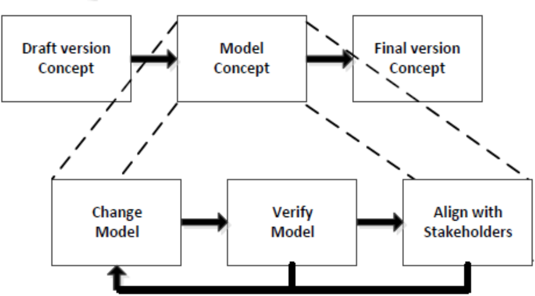


Figure 4.2: *Model-Based Concept Phase*

The formal model is developed incrementally to allow updates after aligning with stakeholders and to incorporate new insights frequently. Before choosing a formal method, we first list the aspects that are important in the concept phase:

- The definition of complete system behaviour, including error scenarios.

- A clear and unambiguous definition of interfaces and design concepts to support parallel development in subsequent phases.
- The possibilities to explore concepts and design decisions fast.
- Communication with stakeholders to obtain agreement on the concepts and externally visible behaviour of the product.
- The possibility to model a combination of hardware and software components.

Furthermore, the formal method should be easy to use by industrial engineers and scalable to large and complex systems. Based on earlier experiences, see, e.g., [66], we decided not to aim for exhaustive model checking. Since our applications consist of many asynchronous components with queues and also timing aspects are important, one would almost immediately run into state-space explosion problems.

As an alternative to increase the confidence in the model, we will use simulation. Formal models are expressed using the Parallel Object Oriented Specification Language (POOSL). The language is supported by a simulator and a new Eclipse Integrated Development Environment (IDE). The tooling can easily be combined with a dedicated Graphical User Interface (GUI) to support communication with all stakeholders.

4.2 Fundamentals of POOSL

One of the goals of the POOSL tooling is to shorten the development time of complex high-tech systems by providing a light-weight modelling and simulation approach. It is targeted at the early phases of system development, where requirements might not yet be very clear and many decisions have to be taken about the structure of the system, the responsibilities and behaviour of the components, and their interaction. Another goal of the POOSL tooling, not described in this chapter, is the support for performance analysis of a system.

In Section 4.2.1 we introduce the POOSL modelling language and describe the available tool support in Section 4.2.2.

4.2.1 POOSL Modelling Language

POOSL is a modelling language for systems that include both software and digital hardware. It is not intended for continuous aspects, e.g., modelling physical processes by differential equations is not possible. POOSL is an object-oriented modelling language with the following aspects:

- *Concurrent parallel processes* A system consists of a number of parallel processes. A process is an instance of a process class which describes the behaviour of the process by means of an imperative language. A process has a number of external ports for message-based communication with its environment.

- *Hierarchical structure* A number of processes and clusters can be grouped into a cluster. A cluster is an instance of a cluster class which has a number of external ports and specifies how the ports of its processes are connected.
- *System definition* A system is defined by a number of instances of processes and clusters and the connections between the ports of its instances.
- *Synchronization* Processes communicate externally by synchronous message passing along ports, similar to CCS [102]. That is, both sender and receiver of a message have to wait until a corresponding communication statement is ready to execute. A process may contain parallel statements which internally communicate by shared data objects.
- *Timing* Progress of time can be represented by statements of the form *delay d*. It postpones the execution of the process by *d* time units. All other statements do not take time. Delay statements are only executed if no other statement can be executed.
- *Object-oriented data structures* Processes may use data objects that are instances of data classes. Data objects are passive sequential entities which can be created dynamically. A number of structures can be accessed from a library, such as set, queue, stack, matrix, etc.
- *Stochastic behaviour* A library provides support for stochastic distribution functions; a large number of standard distribution functions are supported, such as DiscreteUniform, Exponential, Normal, and Weibull.

The formal semantics of POOSL has been defined in [22] by means of a probabilistic structural operational semantics for the process layer and a probabilistic denotational semantics for the data layer.

4.2.2 POOSL Tooling

As explained in [22], the operational semantics of POOSL has been implemented in a high-speed simulation engine called Rotalumis. It supports the Software/Hardware Engineering (SHE) methodology [139]. The tool SHESim [59] is intended for editing POOSL models and validating them by interactive simulation. Recently, a modern Eclipse IDE has been developed on top of an improved Rotalumis simulation engine. The combination of the last two tools have been used for the application described in this chapter.

The Eclipse-based POOSL IDE is freely available [147] and supports advanced textual editing with early validation and extensive model debugging possibilities. It is easy to use for industrial users and scalable to industrial-sized systems. The tool contains on-line explanation and documentation[1].

Model validation is convenient to detect modelling errors early, before they appear during simulation. It includes checks on undeclared variables and ports, types, unconnected ports, and mismatches between send and receive statements. The debugging view, as shown in Figure 4.3, allows step-wise execution of models,

[1] poosl.esi.nl

inspection of variables, setting of breakpoints, and a running sequence diagram during simulation.

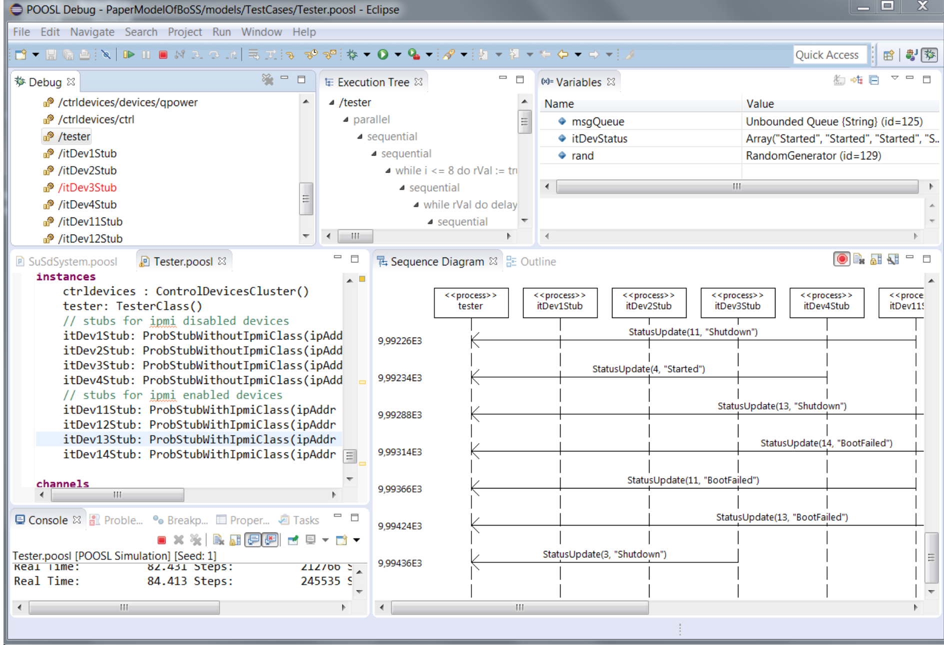


Figure 4.3: *Debug Mode of the POOSL Eclipse IDE*

4.3 Application at Philips

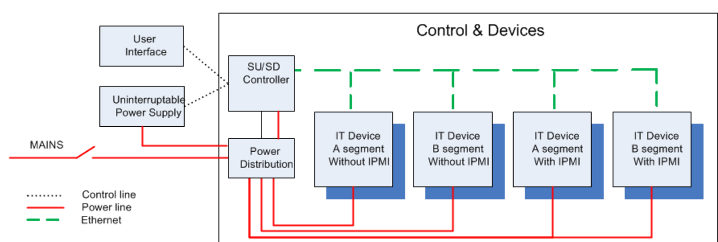


Figure 4.4: *System Overview*

The proposed approach has been applied at Philips, in the context of interventional X-ray systems which are introduced in Section 1.5. For a new product release, we have explored evolutionary changes to create a new concept for starting up and shutting down the system. This section briefly describes the informal concepts of the new start-up/shut-down (SU/SD) behaviour.

As shown in Figure 4.4, which is a refinement of Figure 1.3, the system is partitioned into two segments: A and B (for reasons of confidentiality, some aspects have been renamed). This partitioning is mainly used in the case of a power failure. When all segments are powered and the mains power is lost, the UPS takes over. Once this happens, the A segment is shut down in a controlled way, leaving the B segment powered by the battery of the UPS. If the battery energy level of the UPS becomes critical, also the B segment is shut down in a controlled way. Usually, the diesel generator of the hospital will provide power before this happens. An IT device is part of either the A segment or the B segment.

One of the evolutionary changes is the use of Intelligent Platform Management Interface (IPMI) [79], a standard interface to manage and monitor IT devices in a network. The IT devices in the system are either IPMI enabled or IPMI disabled.

- IPMI disabled IT devices are started and stopped directly by switching the power tap on or off.
- IPMI enabled IT devices are on a power tap that is continuously powered. To start-up these IT devices, the SU/SD controller sends a command via IPMI to them.

Combined with the two types of segments, this leads to four types of IT devices, as depicted in Figure 4.4.

This figure also shows that there are several communication mechanisms between the components

- Power lines for turning the power on and off.
- Control lines to connect the controller to the UI and the UPS.
- The internal Ethernet network, which is used for different purposes:
  - By the IT devices, to request the SU/SD state of the SU/SD controller and to receive SU/SD notification messages from this controller.
  - By the SU/SD controller, to ping the Operating System (OS) of an IPMI disabled IT device to observe its shut down.
  - By the SU/SD controller, to turn on an IPMI enabled IT device and to observe the shut down of the device.

A mains disconnector switch (MDS) can be used to power the complete system. An example of a SU/SD scenario is the shut-down scenario. When all segments are powered and the SU/SD controller detects that the AllSegmentOff button is pressed by the user, it will send an AllSegmentOff-pressed notification to all registered IT devices. Next all IT devices go through the following shut-down phases:

- The applications and services running on the IT device are stopped.
- The IPMI disabled IT devices will register themselves and ask the SU/SD controller to observe their shut-down. This is needed because the controller does not know which IPMI disabled devices are connected to a power tap. The IPMI enabled devices are known to the controller by configuration.

- Once the applications and services are stopped, the OS will be shut down.

The scenario ends when the SU/SD controller has detected that all IT devices are shut down. IPMI disabled IT devices are pinged to observe that they are shut down and IPMI enabled IT devices are requested for their state via IPMI to detect that they are shut down. Next the SU/SD controller will instruct the power distribution component to turn off the switchable power taps with which the IPMI disabled IT devices are powered. The IT tap that powers the IPMI enabled IT devices remains powered while these devices are in the standby state.

In [66], an abstract model of the current start-up and shut-down concept for a simpler version of the system has been made for three model checkers: mCRL2 [65], FDR2 [146] and CADP [58]. For reasons of comparison, exactly the same model was made for all three tools, leading to 78,088,550 states and 122,354,296 transitions. Model checking such a model easily takes hours. The new concept described here is far more complex because of the many asynchronous IT devices that all exhibit different behaviour. For example, the IT devices can sometimes fail to start-up or shut down. Also the timing and order in which they start-up and shut down might be different. Hence, we assume that the new concept is too complex to model check. Consequently, we decided to model the system in POOSL and used simulation to increase the confidence in the concepts.

4.4 Modeling the SU/SD Concept in POOSL

This section describes an incremental approach to model the SU/SD concepts in POOSL. The scope of the model and the simulation environment is described in Section 4.4.1. Section 4.4.2 contains the modelling steps. A few details of the POOSL models can be found in Section 4.4.3. Our approach to test models automatically is presented in Section 4.4.4.

4.4.1 Modelling Scope and Simulator

The aim was to model the Control & Devices part of Figure 4.4 in POOSL. Besides the SU/SD Controller and the Power Distribution, the model should contain all four types of IT devices, i.e., all combinations of segments (A and B) and IPMI support. Moreover, to capture as much as possible of the timing and ordering behaviour, we decided to include two instances of each type.

To be able to discuss the main concepts to stakeholders, we connect the POOSL model to a simulation of the environment of the Control & Devices part. We created a Simulator in Java with the use of WindowBuilder in Eclipse to allow the manual execution of scenarios. It allows sending commands from the User Interface and power components to the model and displaying information received from the model. Additionally, one can observe the status of IT devices and even influence the behaviour of these devices, e.g., to validate scenarios in which one or more IT devices do not start-up or shut down properly. Figure 4.5 shows a screenshot of the SU/SD simulator.

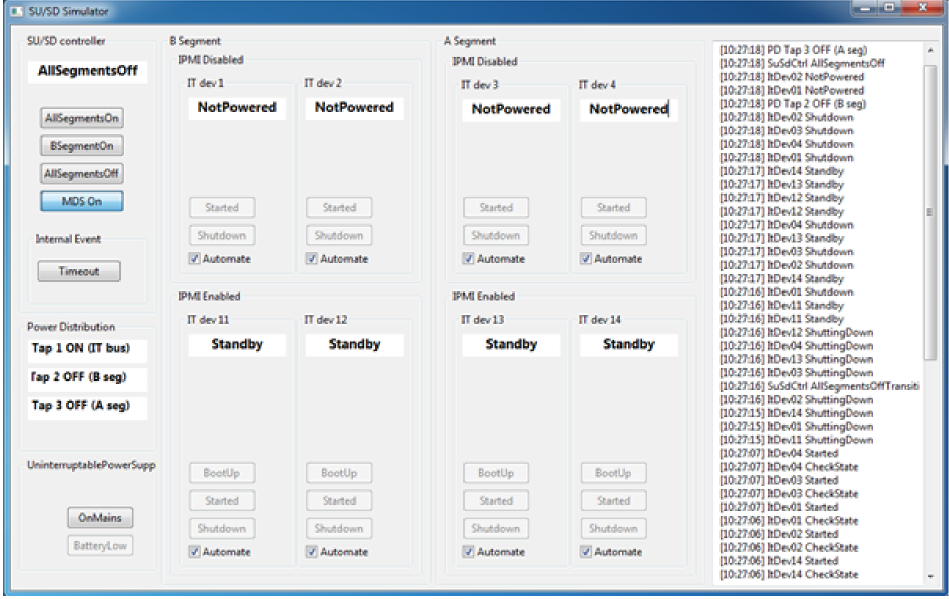


Figure 4.5: *GUI Simulator*

There are three main columns:

- The left column contains three parts:
  - On the top, the state and the UI buttons to control the SU/SD controller are displayed.
  - In the middle, the tap state of the segments is displayed.
  - On the bottom, the UPS triggers are displayed.
- The middle part contains a column for the B segment and one for the A segment; each contains a row for the IPMI disabled IT devices and one for the IPMI enabled IT devices. For each IT device the state is displayed. The start-up and shut-down behaviour of an IT device can be simulated automatically or it can be set to manual to simulate error scenarios, where the system might fall into a Timeout (see the Internal Event in the column of the SU/SD controller).
- In the right column, the status updates of the model are displayed.

The Java simulation is connected to POOSL by means of a socket. The structure of the POOSL system model is shown in Figure 4.6.

The system part to be modelled (the Control & Devices part) is represented by cluster ControlDevicesCluster. It has 10 external ports, one to communicate with the SU/SD controller (simc), one for power commands (simqp) and 8 for the IT

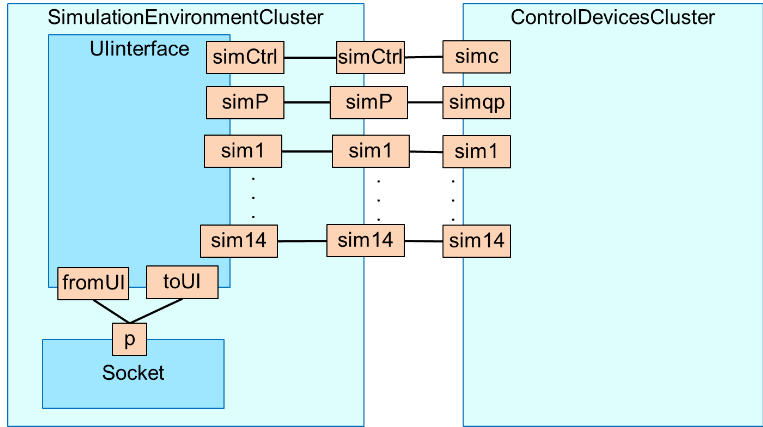


Figure 4.6: *Simulation Environment*

devices: sim1, sim2, sim3, and sim4 for IPMI disabled devices; sim11, sim12, sim13, sim14 for IPMI enabled devices. These ports are connected to corresponding ports of the SimulationEnvironmentCluster. This cluster contains an instance of the standard Socket process class provided by the POOSL library. Class UI-interface is responsible for the translation between strings of the socket interface and the SU/SD system interface.

4.4.2 Modelling Steps

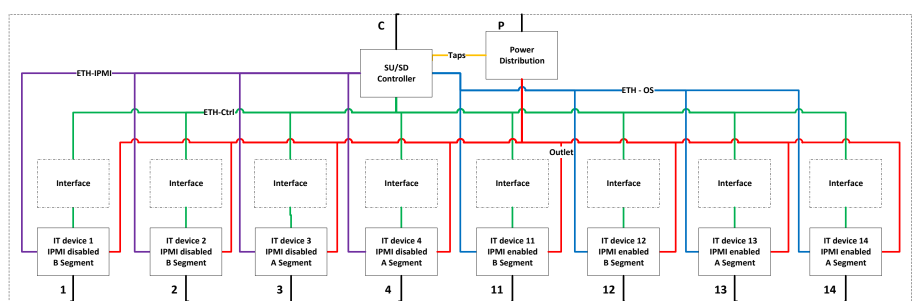


Figure 4.7: *Structure of the POOSL Model of the ControlDevicesCluster*

After the simulator was built, the ControlDevicesCluster was gradually defined in POOSL. The proposed framework, in Figure 4.2, defines an incremental approach to build the model of the concept. We have used the simulator to validate the intermediate models and align the behaviour with internal stakeholders.

We started with a model of an IPMI disabled IT device and a model of the SU/SD controller for shutting down these IT devices of the A segment. In this model there were two instantiations of IPMI disabled IT devices. Note that POOSL supports a partial model where not all ports are used.

This model has been extended gradually to a model where all 8 instances of IT devices are present. Next, the SU/SD controller was extended with error behaviour to verify, for instance, that the system is always in a defined state after shut-down, which is an important requirement.

Finally, we added a model of the interface between the IT device and the SU/SD controller, because these two components will be developed concurrently. Hence, it is important to specify this contract formally and to verify it. Every IT device has an instance of the same interface model, which is implemented in such a way that the system will deadlock if the formal interface is violated. Hence, interface compliance is verified continuously during simulation.

The structure of the resulting model of this incremental approach is depicted in Figure 4.7.

4.4.3 Modelling Devices and Control

This section provides some details of the POOSL models. The first part of the model of an IT device with IPMI is shown in Figure 4.8. It imports a library which, e.g., defines queues. Next the process class is defined, including two parameters for the IP address and the segment. All IT devices have an IP address to be able to connect them to the same network. In the model we only use the least significant byte of the IP address used in the system. Subsequently, the ports, the messages (only one is shown here), the variables and the initial method call are defined. Note that the variables define two queues.

In the initial method *init()()*, the queues are initialized, which are FIFO by default. Next the method defines three parallel activities. The first activity defines a state machine, where the states are represented by methods. It starts the state machine by calling the initial state *ItDevNotPowered()()*.

Figure 4.9 shows a typical definition of a state, in this case state *ItDevShuttingDown()()*. The state is defined as a method with local variable *m*. It selects the next state based on the contents of the *ipmiQueue* or the receipt (indicated by "*?*") of a particular message on one of its ports. Since switching a power tap on or off is instantaneous and cannot be refused by a process, all states allow the receipt of messages *On* and *Off* via port *outlet*.

The other two parallel activities of the *init()()* method are used to model the asynchronous nature of the Ethernet communication. Method *MsgReceiveBuffer* receives messages on port *con* and stores them in queue *msgQueue*, as shown in Figure 4.10.

Note that POOSL allows a condition on the receive statement to express that only messages with the corresponding IP address are received. Similarly, method *IpmiReceiveBuffer* stores messages in *ipmiQueue*.

```
import "../libraries/structures.poosl"
process class ItDevWithIpmiClass(ipAddr : Integer,
                                  segment: String)
ports
    outlet, con, ipmi, sim
messages
    outlet ? On,…
variables
    msgQueue : Queue,
    ipmiQueue: Queue
init
    init()()
methods
init()() /* initial method */
    msgQueue := new(Queue);
    ipmiQueue := new(Queue);
    par
        ItDevNotPowered()()
    and
        MsgReceiveBuffer()()
    and
        IpmiReceiveBuffer()()
    rap
```

Figure 4.8: *POOSL Definition IPMI Enabled Device*

```
ItDevShuttingDown()() | m : String |
sel
    [!(ipmiQueue isEmpty())] m := ipmiQueue remove();
    if m = "status" then ipmi ! On(ipAddr) fi;
    ItDevShuttingDown()()
or sim ? Shutdown; ItDevPowered()()
or outlet ? On; ItDevShuttingDown()()
or outlet ? Off; ItDevNotPowered()()
or sim ? Started; ItDevShuttingDown()()
or sim ? BootUp; ItDevShuttingDown()()
les
```

Figure 4.9: *State ItDevShuttingDown*

```
MsgReceiveBuffer()() | m : String, ip : Integer |
    con ? RecvEvent(m, ip | ip = ipAddr);
    msgQueue add(m);
    delay(1);
    MsgReceiveBuffer()()
```

Figure 4.10: *Receive Buffer*

4.4.4 Extensive Model Testing

The simulator has been used to align the behaviour with internal stakeholders and to get confidence in the correctness of the behaviour. To increase the confidence without the need of many manual mouse clicks, we created a separate test environment in POOSL. Therefore, a stub is connected to every IT device. A stub is a

process which randomizes the start-up and shut-down timing of an IT device. In addition, a stub randomly decides if a device fails to start-up or shut-down. Also in these random cases the system has to respond well and it needs to be forced into defined states. The POOSL fragment of Figure 4.11 depicts how the random timing and random behaviour is implemented in the Stub.

```
process class ProbStubWithoutIpmiClass
                          (ipAddr      : Integer,
                           StartUpProp : Real,
                           ShutDownProp: Real)
ports
    sim, tester
    ...

Loop()() | message : String |
[!(msgQueue isEmpty())] message := msgQueue remove();
if message = "Booting" then
    delay(rand random * 5.0);
    if rand random <= StartUpProp then
        sim ! Started;
        tester ! StatusUpdate(ipAddr, "Started")
    else
        tester ! StatusUpdate(ipAddr, "StartFailed")
    fi
fi;
```

Figure 4.11: *Stub Used for Testing the Model*

```
// stubs for ipmi disabled devices
itDev1Stub: ProbStubWithoutIpmiClass(ipAddr := 1,
                                     StartUpProp := 0.9,
                                     ShutDownProp := 0.9)
itDev2Stub: ProbStubWithoutIpmiClass(ipAddr := 2, □
itDev3Stub: ProbStubWithoutIpmiClass(ipAddr := 3, □
itDev4Stub: ProbStubWithoutIpmiClass(ipAddr := 4, □
// stubs for ipmi enabled devices
itDev11Stub: ProbStubWithIpmiClass(ipAddr := 11,
                                   StartUpProp := 0.9,
                                   ShutDownProp := 0.9)
itDev12Stub: ProbStubWithIpmiClass(ipAddr := 12, □
itDev13Stub: ProbStubWithIpmiClass(ipAddr := 13, □
itDev14Stub: ProbStubWithIpmiClass(ipAddr := 14, □
```

Figure 4.12: *Stub Instances*

The stubs are configured such that they fail to start-up or shut-down in 10% of the cases, as shown in Figure 4.12.

In reality the IT devices are quite reliable, but to reduce testing time it is more convenient to make the IT devices less reliable. Moreover, we are interested in the error handling behaviour of the system and not in the statistical behaviour.

For the execution of scenarios initiated by a user and the UPS, a Tester process has been created to automatically drive the system. Every stub has a feedback channel to the Tester to report the status of an IT device. Figure 4.13 depicts how the Tester and Stubs are connected to the system.

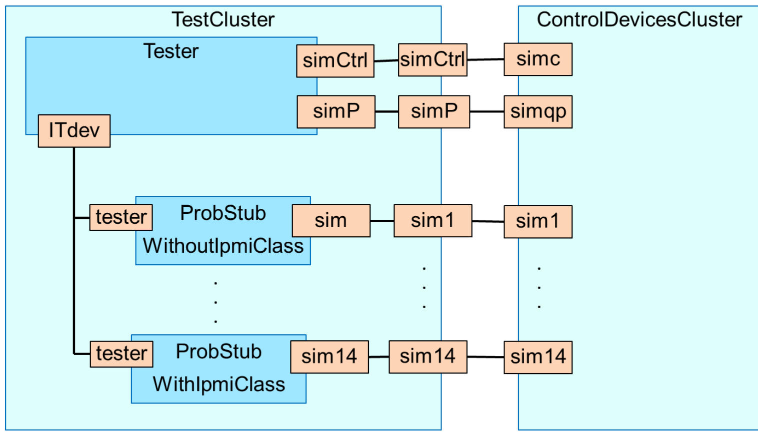


Figure 4.13: *System Composition with Tester*

Name	Value
delayedBatteryLow	false
ipmiQueue	Empty
Occupation	0
PrimQueue	nil
QueuingPolicy	"FIFO"
Size	-1
Unbounded	-1
osQueue	Empty

Figure 4.14: *Variable Values from the POOSL IDE's Debug Window*

The definition of the Tester is such that it leads to a deadlock when the SU/SD controller or the IT devices do not behave as intended. Deadlock and livelock can be detected by the lack of progress in the debug mode as shown in Figure 4.3. Already during the first simulation run we experienced a deadlock. The cause of the problem was found using the debug possibilities of the new POOSL IDE. We simulated the model in debug mode and inspected the sequence diagram (see

also Figure 4.3) when the deadlock occurred. In this sequence diagram we saw a problem with a message about the IPMI status of an IT device. Next we inspected the variables window as shown in Figure 4.14.

It revealed that the ipmiQueue was empty, which was not expected at this point in the execution. When checking the code that handles the IPMI queue, we found that the queue was emptied after the IPMI status request had been sent. The race condition was fixed by changing the order; first empty the queue and then send the IPMI status request. After fixing the race condition, the model has been executed 100,000 random start-up and shut-down cycles without experiencing a single deadlock.

4.5 Concluding Remarks

In the concept phase of product definition, we have used a formal system description in POOSL in combination with a graphical user interface to align stakeholders and to get confidence in the behaviour of the system. We have added a model with a formal interface description between two important components of the system that will be developed concurrently. To increase the confidence in the concept, we created an automated test driver for the system with stubs that exhibit random behaviour and random timing.

While modelling, we found several issues that were not foreseen in the draft concept. We had to address issues that would otherwise have been postponed to the implementation phase and which might easily lead to integration problems. We observed that the definition of a formal executable model of the SU/SD system required a number of design choices. We give two examples of such choices.

- If all segments are on and the UPS indicates that the mains power input fails, then the system will shut down the A segment. If, however, during this transition one or more of the IPMI enabled IT devices fail to shut down, then the SU/SD controller has no way to force these IT devices into the right state. This could be solved by an additional tap, but given the costs of an extra tap and the small chance that this will happen (both mains power and shut down of an IT device should fail), we have decided to leave it this way. If the user experiences unexpected behaviour of the system, the user can always recover the system by turning it off and on again.

- An early version of the SU/SD controller did not track if an IPMI enabled IT device did in fact start up. However, if something is wrong with the start-up or shut-down of an IPMI enabled IT device, we want to toggle the power during shut-down in the hope that a reset will solve the issue. Once we found the described issue with the simulator, we extended the model of the SU/SD controller with a storage of the start-up status of an IPMI enabled IT device.

In addition, the model triggered many discussions about the combined behaviour of the hardware and software involved in start-up and shut-down. This resulted in a clear description of responsibilities in the final concept. Also the exceptional system behaviour when errors occur has been elaborated much more

compared to the traditional approach. Note that the modelling approach required a relatively small investment. The main POOSL model and the Java simulator were made in 40 hours; the tester and the stubs required another 10 hours.

The application of exhaustive model checking techniques to the full model is not feasible, give the large number of concurrent processes and the use of queues for asynchronous communication. However, it might be possible to apply these techniques to verify certain aspects on an abstraction of the model.

CHAPTER 5

CONFIGURING A COMPONENT USING DSLS

In this chapter we investigate the maintainability improvement of a legacy component by vertical DSLs. We consider a legacy component which requires low-level configuration files and create a DSL to generate these files. Validation checks on language instances ensure correctness of the generated files. To be able to deal with a large number of configurations, a second DSL has been created which has a higher level of abstraction.

5.1 Motivation for Creating DSLs

Legacy components are often the result of decades of development with dozens of man-years invested. Creating new implementations would require a similar investment and typical more resources than keeping the legacy implementation alive; scarce resources that could also be used to implement product innovations [168].

The aim of this chapter is to investigate whether a vertical DSL could improve the maintainability and extensibility of a legacy component. In particular, we investigate the definition of low-level configuration files for a legacy component. Our experience is that constructing such configuration files is time consuming and error prone. To abstract from low-level details, a DSL has been created. The configuration files are generated from instances of the DSL. In addition, validation checks have been defined to prevent errors in language instances.

We would like to get an answer to the following questions:

- Is it financially feasible to extend the life of a legacy component using a DSL?
- What are the pros and cons of using a DSL compared to the current way of working?

In this chapter, we try to answer these research questions based on our experiences.

5.2 Context of the Fieldbus

The interventional X-ray system, as introduced in Section 1.5 and depicted in Figure 1.2, consists of a number of building blocks such as the patient table, one or two stands which hold an X-ray generator and a detector, and a stand mover that can position the stands away from the table. Each building block has a number of axes that are used to position the X-ray beam with respect to the patient. The axes are controlled by motion drives which are connected by means of a fieldbus. A fieldbus is an industrial network used for real-time distributed control.

For each building block there are a number of variations, e.g., they may have a different number of axes or might come from different third-party vendors. For example, the table can have one, two, three, four, five or six moveable axes. Moreover, there are many possible combinations of these building blocks, leading to many systems configurations depending on the wishes of the customer. The fieldbus needs a separate topology description for all these combinations. In addition, past and future configurations need to be supported.

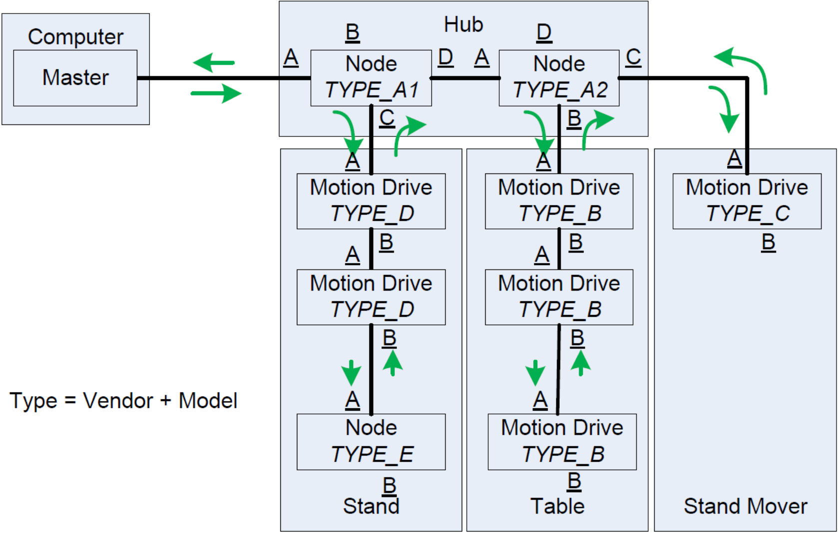


Figure 5.1: *Fieldbus Topology*

Figure 5.1 depicts an example network topology. A number of components can be distinguished: a Computer, a Hub, a Stand, a Table, and a Stand Mover. The Computer runs the fieldbus master and also hosts the motion application. The fieldbus Hub is an embedded device that supports the use of tree topologies. The Hub is optional; for instance, a configuration with only a Table does not need a Hub. Because of the Hub, different cable sets toward the Table and Stand(s) can be bundled. A Stand consists of two motion drives, each controlling a number of axes, and a node that is used to prevent collisions between the Stand and other objects in the room. The Table has a motion drive for every axis. So the Table has up to six motion drives. The Stand Mover has a motion drive that controls a

number of axes to move the Stand away from the Table.

Every node or motion drive in the fieldbus has a certain type. A type is a combination of a vendor and model. The concept is that every motion drive in the system can be replaced by a compatible type from another vendor. The Hub has two nodes of TYPE_A1 and TYPE_A2 respectively. A Stand uses devices of TYPE_D and TYPE_E. The Table uses motion drives from TYPE_B and the Stand Mover from TYPE_C.

The nodes of the Hub have four ports. The node of TYPE_A1 is connected to a master via port A. The nodes TYPE_A1 and TYPE_A2 are internally connected via port D of TYPE_A1 and port A of TYPE_A2. The motion drives have two ports: A and B. At the end of a branch it is possible that a port is not connected to another device.

The arrows in Figure 5.1 describe the flow of messages over the fieldbus. The master sends a packet to the node it is connected to. A packet consists of different fields. Every node in the network has its own field. Every node reads and writes its field of the message. At the end, the master receives a message with all updates of the nodes. The master sends messages with a time interval of 2 milliseconds.

During the start-up of the network, the master reads a configuration file that describes the physical network. The master then starts the network by programming the nodes. The nodes need to be programmed such that they know their field, the elements of this field, and the address of the elements.

To create the configuration files of the master, a commercial tool is used. A tutorial of 29 pages describes how a configuration file needs to be created. The first two pages explain how to install the tool and some basic explanation of the topologies used in the system. The remaining 27 pages describe what needs to be filled in when making network topologies for the system.

Currently, the interventional X-ray system needs about 40 different topology configuration files for the product family. All files are created with the commercial tool. To test the configuration files, it is too expensive to physically build 40 different complete systems, in terms of effort, lead time and system cost. Hence, for testing a lab set-up is created. In the lab set-up the master is started using a stripped version of the motion application. The nodes, such as the motion drives, are placed on a board. Using the board it is possible to reroute the cables to test different configurations.

The configuration files are formatted using eXtensible Markup Language (XML). Today, the simplest configuration file consists of 2147 lines and the most extensive configuration contains 13128 lines.

5.3 DSL for Fieldbus Configurations

The number of configurations explodes when multiple suppliers and motion drive types have to be supported. The commercial tool that is used to create network topologies has many settings and options. However, for the system configurations we want to describe, the settings and options are always the same. The only variation between the different configurations is the number of nodes, their type and how they are connected to each other.

To handle these differences, a DSL is created to generate configuration files for the fieldbus. The DSL only describes the variations; the fixed options and settings are defined by the generator. For creating the language, Eclipse [171] is used with Xtext and Xtend [18]. Figure 5.2 presents an example instance of the language. Each topology needs to have a name which is used as the file name for the generated configuration. After the *network* keyword the ordering of the nodes is described. The language has predefined node types. In the language used at Philips the types have more meaningful names, but for confidentiality reasons we use abstract names in this chapter.

```
topology
name        ExampleTopology
network     TYPE_A1 <-> TYPE_A2 <-> TYPE_B
                   <-> TYPE_B <-> TYPE_B
            prev TYPE_A2 port C TYPE_C
            prev TYPE_A1 port C TYPE_D
                   <-> TYPE_D <-> TYPE_E
```

Figure 5.2: *Example Topology Description*

Figure 5.2 presents the network topology of the system configuration depicted by Figure 5.1. The example presents a single network topology, but typically a DSL instance consists of multiple topology definitions. The master is always present in a topology and hence omitted in the DSL instances. The nodes TYPE_A1 and TYPE_A2 of the Hub are connected to each other. Because the connection between TYPE_A1 and TYPE_A2 is hardwired inside the Hub, this information does need to be provided when creating a DSL instance. The other devices (nodes and motion drives) are implicitly connected via port A to port B of the previous device. Hence, this information does not have to be described in a DSL instance. With “prev TYPE_A2 port C TYPE_C” a branch is created by connecting port C of TYPE_A2 to port A of TYPE_C. Similarly, a branch is created from TYPE_A1 to TYPE_D.

From every topology in a DSL instance, an XML configuration file is generated. The XML configuration file generator has been defined using multi-line template expressions [18]. The settings that are always the same for network topologies are part of the template. The configuration settings that vary, e.g., the position of the nodes and the fields, are calculated and filled in the right position. A DSL instance of 5 lines describing an existing topology leads to an XML file of 13128 lines. The output of the generator has been validated by generating ten existing network topologies and comparing the configuration files with the ones that are produced with the commercial tool.

5.4 DSL Instance Validation

To prevent that the user of the language makes faults in describing network topologies, validation rules have been added to check the validity of a network topology. For example, it is physically impossible to connect two branches of motion drives to the same port of the Hub. Figure 5.3 shows the validation rule which expresses

```
@Check
def CheckTagPortUnique(Topology topo) {
	for (var i = 0; i < topo.nextNode.length; i++) {
		var nodeA = topo.nextNode.get(i)
		for (var j = 0; j < topo.nextNode.length; j++) {
			var nodeB = topo.nextNode.get(j)
			if (nodeA.prev != null && nodeB.prev != null) {
				if (i != j && nodeA.prev.name == nodeB.prev.name &&
					nodeA.portOfPrev == nodeB.portOfPrev) {
					error("A port can only be used once", null)
				}
			}
		}
	}
}
```

Figure 5.3: *Validation Rule*

that a port of a specific node can only be used once within a topology. The rule checks for every topology and for every pair of nodes which have the same predecessor that they are connected to different ports of this predecessor.

In addition to the rule in Figure 5.3, there are validation rules to check that:

- Within a DSL instance, a topology has a unique name.
- Within a topology, the types TYPE_A1 and TYPE_A2 are paired. TYPE_A1 and TYPE_A2 are either both present or both absent.
- Within a topology, TYPE_A1 comes before TYPE_A2 and is connected to TYPE_A2.

Using the commercial tool there are many ways to produce a faulty configuration file. The DSL and the above described validation rules provide enough confidence in the validity of the produced configuration files. Hence, the language, including its validation rules, is restricted in such a way that only valid configuration files can be produced. Creating a hardware set-up in the lab to check the correctness of a network configuration is no longer needed.

5.5 DSL to Describe System Configurations

Once a year a new system release is made. Because the system is a medical device, for such a release all functionality needs to be verified and validated using strict rules of authorities. Hence, all supported network topology configurations are part of the annual release and all of them are installed on every system. A system configuration prescribes which configuration file is used for a particular system instance.

When in the future many network topology configuration files are needed, it is still a gigantic and error-prone task to create them all. For this reason, we investigated the possibility to further raise the abstraction level. The result is a second language to represent system level configurations and generate a DSL instance of the previously described network topologies. The system configuration DSL consists of two parts: the first part describes building block definitions and the second

part describes system configuration descriptions which consists of combinations of the building blocks.

```
building blocks
building block
id          BB1
type        CeilingStand
vendor(s)   VENDOR_A

building block
id          BB2
type        Table
vendor(s)   VENDOR_A VENDOR_B
```

Figure 5.4: *Building Block Definitions*

Figure 5.4 shows a fragment of the building block definitions. Every building block has a unique id. Building blocks have a type, for instance, a Stand can be based on the floor or the ceiling. Also the Table is a building block.

Depending on which options a customer chooses, the Table can have one up to six motorized degrees of freedom. Hence, there are six combinations of motion drives for a Table. For certain building blocks, nodes from multiple vendors can be used. Recall that in the topology descriptions of the first DSL, nodes of a certain type are used. The relation between vendors and types has been encoded in the generator, e.g., VENDOR_A corresponds to types TYPE_C and TYPE_D. Also information about which port of a building block needs to be connected is hard coded into the generator. The items are fixed in the generator because the system's reference architecture fixes these items and therefore they are not expected to change.

```
configurations
configuration
name              Configuration1
building blocks   BB1

configuration
name              Configuration2
building blocks   BB1 BB2
```

Figure 5.5: *System Configuration Descriptions*

A fragment of two system configuration descriptions is shown in Figure 5.5:

- Configuration1 describes a system configuration consisting of a ceiling stand with motion drives of vendor A. This is a very basic example that results in a single network topology.

- Configuration2 consists of a ceiling stand and a table. The table can have from 1 up to 6 motion drives and each of these motion drives can be either from VENDOR_A or VENDOR_B. If a table has one motion drive it can be of two different vendors, leading to two network topologies. If a table has two motion drives, then four different combinations are possible, et cetera. When we sum all possibilities, we get 126 network topologies.

The Configuration2 example makes the need for the system configuration language clear. The number of network topologies grows exponentially with the number of different vendors that need to be supported.

Using the system configuration language, the generation of the configuration files takes a two step approach. In the first step, an instance of the system configuration language generates an instance with network topology descriptions. From the network topologies, XML configuration files are generated.

5.6 Concluding Remarks

We have presented an approach to improve the maintenance of a legacy component using two DSLs. The first DSL describes network topologies from which XML files are generated for the master of a fieldbus network. Because of the expected large number of topologies in the future, we further raised the abstraction level by means of a second DSL that describes system configurations and generates an instance of the first network topology DSL. The experiences with these DSLs at Philips leads to the following observations.

- *Is it financially feasible to extend the life of a legacy component using a DSL?* We calculate the Return On Investment (ROI) for the presented DSL. First we compute the required investment for the DSL approach. To learn the domain and the structure of the configuration XML files took 10 hours. The construction of the DSL took about 40 hours including the creation of the validation rules. In total it took about 50 hours to create the DSL. We expect a new vendor in the future and estimate that it will take 30 hours to extend the DSL framework with support for this vendor.

  Next we compare the DSL approach with the current way-of-working. We estimate that approximately 8 hours are required to manually create a topology file, build a physical hardware set-up and test if the master can start the fieldbus. Of the 8 hours approximately half an hour is needed required to create the topology file. If we multiply these 8 hours of work with the the 2000 network topologies we need in the future, it takes 16000 hours which is 10 man-years.

  Using the DSL we expect it takes around 20 hours to create instances describing the system configurations for the 2000 topologies. These 20 hours plus support for new vendors (30 hours) plus the 50 hours to create the DSL itself leads to 100 hours of investment. ROI = (gain from investment − cost of investment) / cost of investment = (16000 - 100) / 100 = 159. Hence, the DSL has a high ROI which indicates that the investment in the DSL will be preferred above keeping the current way-of-working.

- *What are the pros and cons of using a DSL compared to the current way of working?*
  We list a number of advantages and disadvantages of using a DSL compared to the current way of working. We start with the advantages, in addition to the large ROI computed in the previous point:

  - Our DSLs are simple and easy to use; the users of the commercial tool, which are software engineers, should be able to create a new network topology and a new system configuration in a short amount of time.
  - Creating a network topology can be done in less time than with the current way of working, i.e., using the commercial tool.
  - The validation rules check if a network topology is valid, while with the commercial tool faults can be introduced that can only be found when a topology is build and tested.

  Below a list of disadvantages:

  - The generators of the DSLs contain additional code that needs to be archived, supported and maintained.
  - C++ is the programming language that is used at Philips. The generators of the DSLs can be programmed in Xtend and/or Java. The switch in programming language will create a barrier for some software engineers although the generator only needs to be supported by a few software engineers. There will be more users for the language than there are software engineers that need to maintain the language.
  - The preferred Integrated Development Environment (IDE) at Philips is Microsoft Visual Studio (MSVS). We have investigated and compared multiple solutions to create DSLs using MSVS, but the outcome of the investigation is that we can only use Eclipse for our needs. Installing a second IDE and switching between IDEs is a disadvantage.

At Philips, we clearly have a maintenance challenge when 2000 network topologies need to be supported. In this case, the DSL approach has a large ROI and, despite a few drawbacks, provides a very good solution for this future maintenance problem. In general, due to the challenges with maintaining legacy components and the experiences presented in this chapter, Philips will continue with the DSL approach.

CHAPTER 6

DSLS COMBINED WITH OTHER MODEL-BASED TECHNIQUES

The industrial case described in this chapter is a legacy component which uses configuration files that are hard to maintain. To improve the situation we used an approach in which we combine DSLs with other model-based techniques. DSLs have been created to describe the behaviour and the test cases of the component. In addition, we created cross-checks to increase confidence in our approach.

6.1 Motivation and Global Overview

The Power Distribution Unit (PDU) of an interventional X-ray system is responsible for executing power control scenarios, such as start-up, shut-down and power failure. During such a scenario the PDU is the master of the system and all other hardware and software components follow the instructions of the PDU.

The PDU 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, based on the behaviour defined by configuration, for every release a separate test set is required.

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 PDU. 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 [72, 83]. 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 PDU 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 6.1. 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.

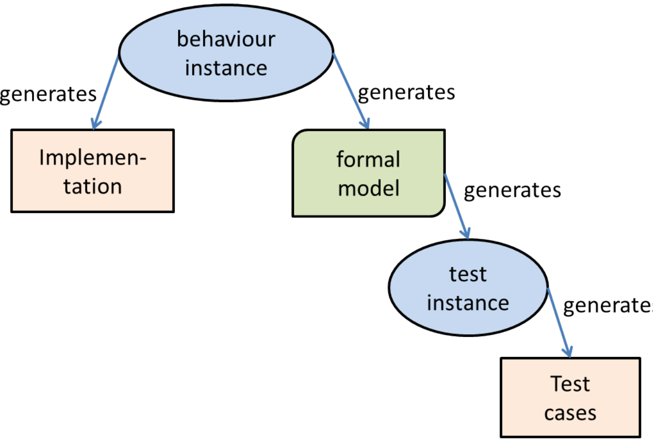


Figure 6.1: *DSL's Combined with Formal Techniques*

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 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.

To construct DSLs we used the Xtext plugin of Eclipse [18], supported by a manual [104]. To be able to simulate the behaviour of the PDU based on a DSL instance, we use a translation to POOSL [148], see Chapter 4 for more details about POOSL. Formal verification has been done by means of SAL [138], because it also includes convenient support for test generation from a formal model.

We present our experiences with this approach in a real development project. The business goal of the development project is to improve the maintainability and extendibility of the PDU. The aim of our work is to investigate whether DSLs could provide a solution to improve the maintainability and testability of the PDU. 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 model checkers?
- What are the benefits of introducing these new techniques compared to the current way of working?

6.2 Context of the PDU

An interventional X-ray system, as introduced in Section 1.5, 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 PDU, see Figure 1.3. This component is installed in a technical room together with a number of cabinets which contain the supporting hardware components.

The PDU consists of a controller that has three interfaces, as shown in Figure 6.2:

- 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.

The controller and all distribution panels have a 16 bit micro-controller running an embedded application and they communicate with each other via LonWorks [2] 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 6.1, defining recall TermStandby. Everything behind a # sign is a comment.

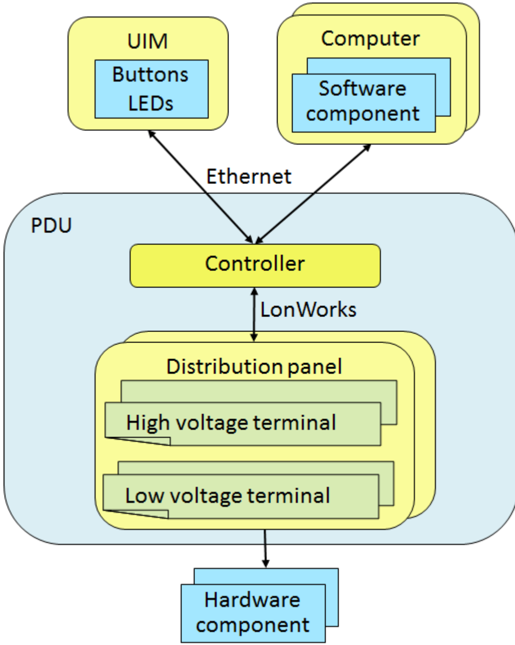


Figure 6.2: *Overview Power Distribution Unit*

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. The state machine consists of two parts:

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

The scenarios configuration file is used by the high-level state machine to perform the configuration-specific transitions. The high-level state machine is implemented with VisualState [5] and describes the main states and the associated LED behaviour when transitioning between these main states. These main states are:

- Off: the PDU is not powered;
- Init: represents the start-up of the PDU, in this state a Power On Self Test (POST) is executed;

```
# 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 6.1: *Fragment of the Recalls Configuration File*

- Standby: the system is off for the user, but PDU 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 PDU 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 PDU 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 the recalls configuration file.

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. Note that in case of an emergency power off, hardware immediately cuts off all power.

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

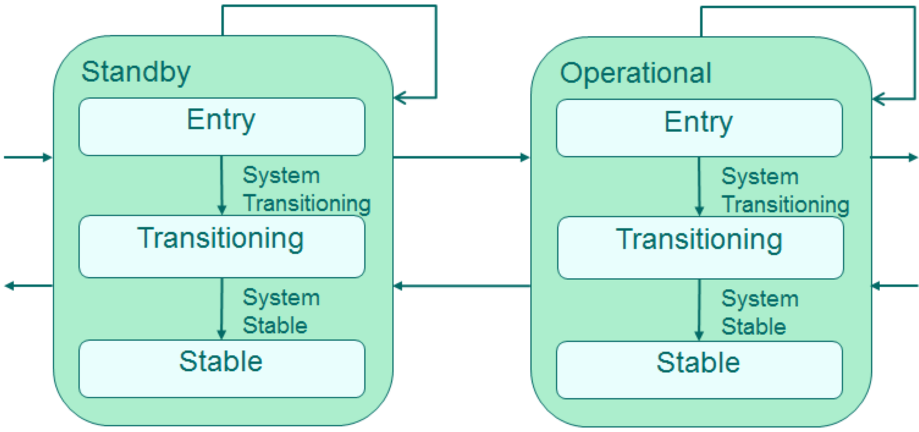


Figure 6.3: *Two Main States and Their Substates*

Table 6.2: *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, otherwise it is Stable, except for the first recall where it stays in Entry.

- 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.

Figure 6.3 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 recalls configuration file. The low-level state machine and the recalls are different for every system release. The scenarios 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. 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 6.2 shows an example of a line in the scenarios configuration file. Everything after a # is a comment.
The first three columns of Table 6.2 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.

6.3 Defining the Behaviour of the Component

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 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 6.3.1, and verification models (SAL), described in Section 6.3.2.

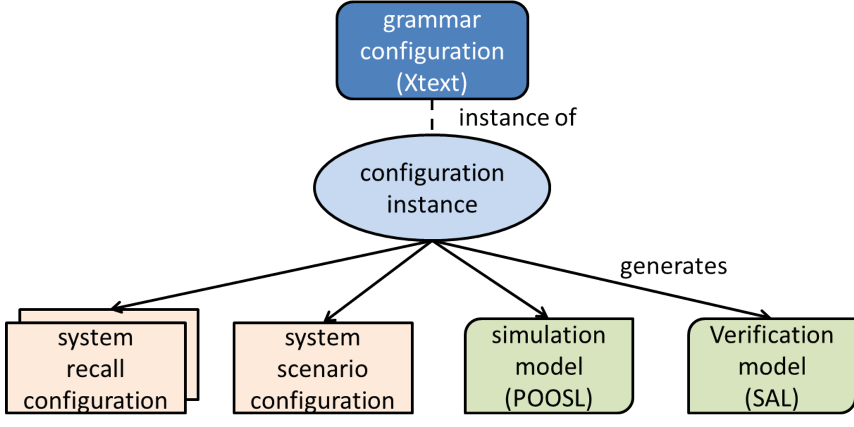


Figure 6.4: *Overview Configuration DSL Transformations*

An overview is given in Figure 6.4. 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 which results in different hardware components (and their associated behaviour).

The grammar for the DSL is expressed in Xtext. The Xtend 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.

Since the main states and their substates are always the same, there is no need to define them explicitly in the DSL. The main purpose is to define the recalls and their transitions. Figure 6.5 depicts a fragment of the first section of an instance.

```
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 6.5: *First Section of an Instance of the Configuration DSL*

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 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.

The grammar for this language has been expressed in Xtext; a fragment is depicted in Figure 6.6. 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 Figure 6.5.

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

Figure 6.6: *Grammar of the Configuration DSL*

Figure 6.7 shows the last section 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 6.7) the setting of another configuration (e.g., `setup`) can be used and only the differences have to be specified.

6.3.1 POOSL

To increase the confidence in the correctness of the behaviour of the PDU, we simulate the specified behaviour using POOSL. 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 6.8, 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

```
...
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 6.7: *Last Section of an Instance of the Configuration DSL*

stakeholders. In this way, we detected problems in the DSL instance, e.g., the simulation in POOSL revealed a missing condition.

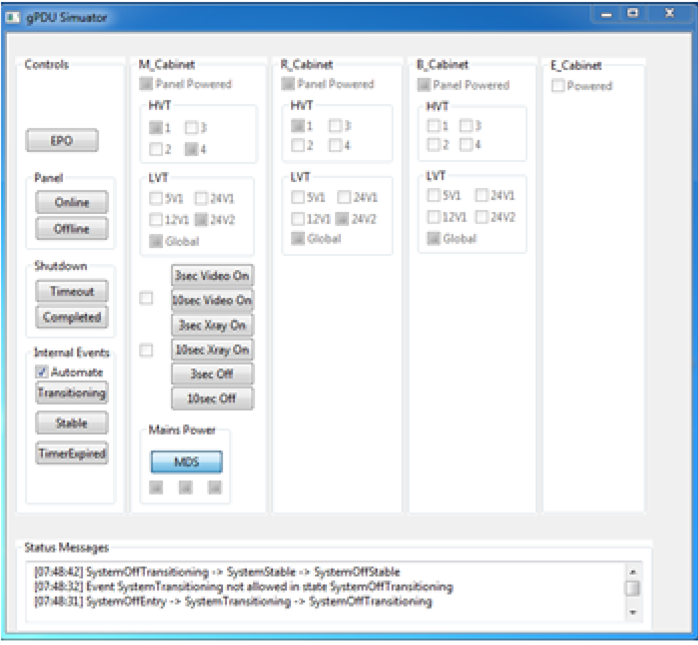


Figure 6.8: *GUI of the Simulation of the Power Distribution Unit*

6.3.2 SAL

To increase our confidence in the intended behaviour of the PDU, we generated a SAL model [71, 137] to enable formal model checking. Table 6.3 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) 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 6.3: *Fragment of a SAL Model*

Table 6.4 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 the SystemOffStable state, then it also provides power in all future steps when it is in the SystemPartlyOnStable state. Similarly, with SystemPartlyOnStable and SystemOnStable.

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.

```
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 6.4: *SAL Property*

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.

6.3.3 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 6.5. Since the state machine described by the DSL is ordered on the states,

```
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 6.5: *Fragment of the Generated Scenarios Configuration File*

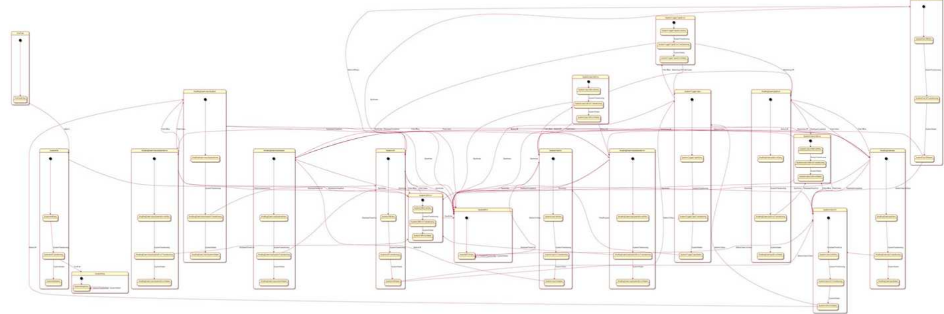


Figure 6.9: *State Diagram of the Configuration DSL*

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. The first eight columns, till the # sign, are the same as the manually created configuration file described previously. In the comment part, the generator writes the first letter of the source main state, the source termstatus name, the stimulus, followed by the first letter of the target main state and the target termstatus name. Similarly, for all system configurations, a recalls configuration file has been generated.

Additionally, we have generators that yield for every language instance a set of UML state diagrams, at several levels of abstraction, using PlantUML [4]. An impression of a generated diagram is given in given in Figure 6.9 (not readable for reasons of confidentiality).

6.4 Testing the Component

The PDU 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 the requirements. Clearly, this is needed for every new product re-

PDS:QUE9:PAR-1	NoErr	30000	1000	2000	On Button
PDS:SYST?	3:2:2	1000	1000	2000	System On

Table 6.6: *Part of a Test Case*

lease, 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 6.4.1. A DSL to improve maintainability of the test cases is presented in Section 6.4.2. Our approach to generating test cases using SAL is explained in Section 6.4.3.

6.4.1 Test cases

To test the PDU, there is an automated test tool running on a companion PC that connects to the controller of the PDU via Ethernet. It can inject stimuli and ask the current state. A test case is a comma separated (CVS) file. All test cases start and end in the same state which makes it possible to execute them consecutively. Table 6.6 shows two lines of a test case.

1. The first column is the network command that is send from the test tool to the PDU (QUE injects a stimulus into the state machine and SYST asks the current state).
2. The second column is the expected response from the PDU 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 PDU (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 PDU; within this amount of time the PDU 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 PDU (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.

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.

Every night these test cases are executed multiple times on two PDUs with Jenkins [1] 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 PDU or the test case. Test cases often fail because

```
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
...
```

Figure 6.10: *Test DSL*

of timing issues in which the PDU 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.

6.4.2 Test DSL

To describe test cases for the PDU in a maintainable way, a test DSL has been created. The test DSL is explained using the instance fragment depicted in Figure 6.10. 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 substates, by adding *Entry*, *Transitioning*, and *Stable* behind the 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 6.7, for each trace in the language instance. Every Transitioning and Stable termstatus

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

Table 6.7: *Part of a Generated Test Case*

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 6.6; instead of a waiting time, now also an event can be specified. Such an event is used to synchronize with the PDU 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 Figure 6.11.

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. Figure 6.12 shows a fragment of a coverage file.

6.4.3 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 6.3 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 6.8 depicts a fragment of an updated SAL model.

The SAL test generator is fed with three files:

- the SAL model described in Table 6.8 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.

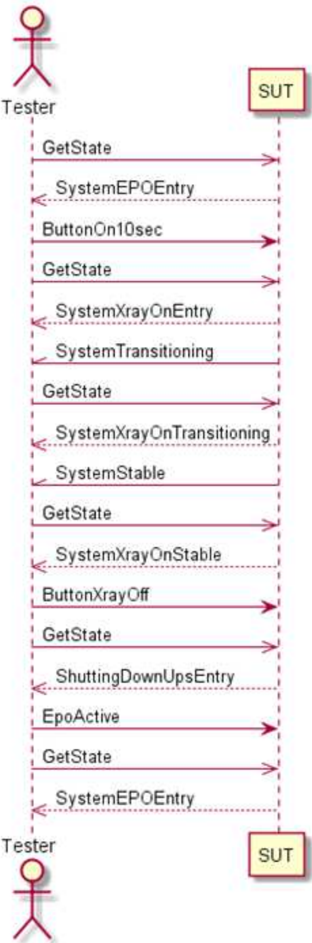


Figure 6.11: *Generated Sequence Diagram of a Test Case*

Based on this input, the SAL test generator yields 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.

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 PDU 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 PDU. Outside the weekend, the full test set is run every night.

With the generated test cases, approximately twice as many transitions are covered as with the manually written test cases. Theses manually written test

```
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
 ...
```

Figure 6.12: *Part of a Coverage File*

```
...
 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 6.8: *Fragment of a SAL Model*

cases only make 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 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 PDU and immediately resumes the test case once the PDU has reached the desired state.

6.5 Increasing the Confidence in 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 PDU 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 6.13.

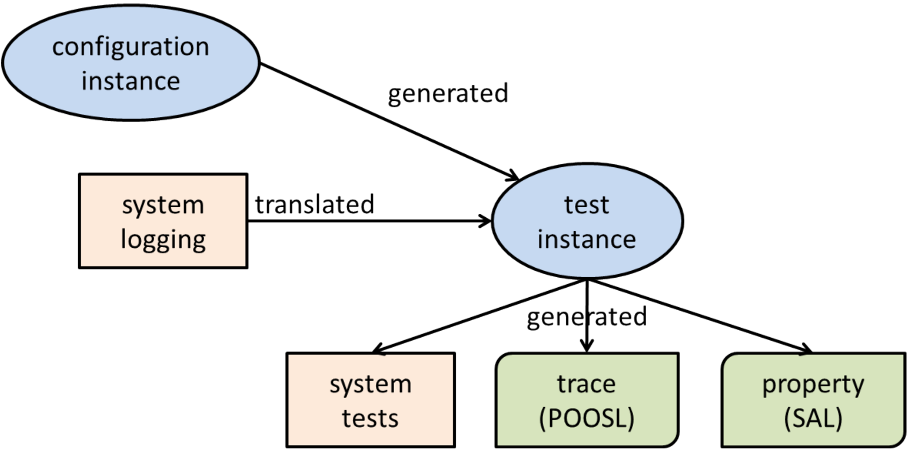


Figure 6.13: *Overview Test DSL Transformations*

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 6.3.1). 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 6.9 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 6.9: *Fragment of a Trace Expressed in LTL*

Finally, Figure 6.14 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.

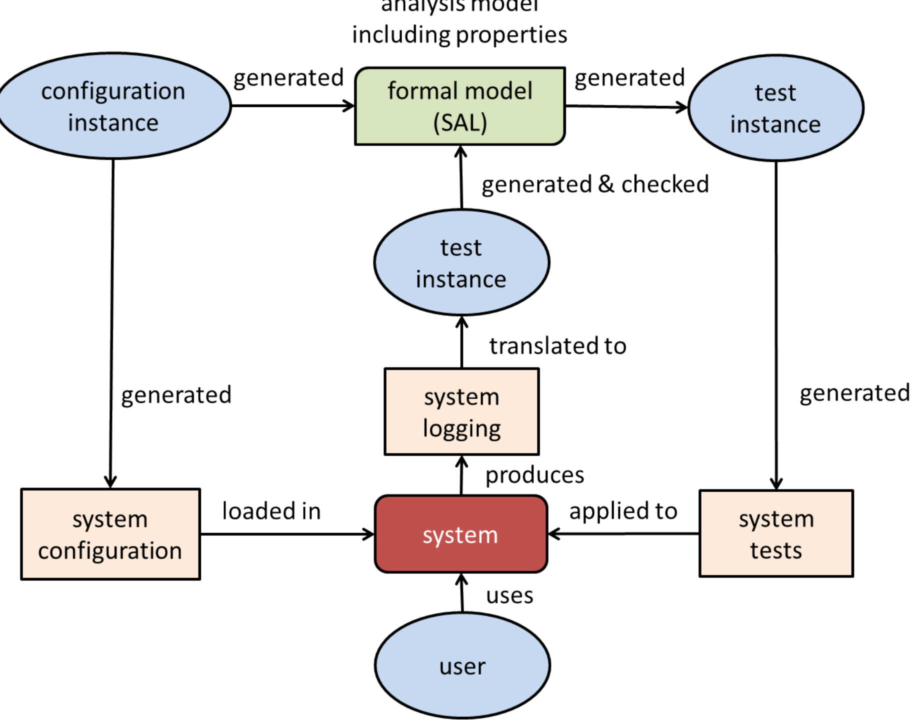


Figure 6.14: *SAL Overview*

6.6 Concluding Remarks

We summarize the results in Section 6.6.1. The additional generation of analysis models is described in Section 6.6.2. Section 6.6.3 addresses the research questions.

6.6.1 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 make transitions from the Stable substates. The generated test cases also make transitions from the Entry and Transitioning substates, which leads to twice as many 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 PDU and immediately resumes the test case once the PDU has reached the desired state.

6.6.2 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 and the model checker SAL. The generators for SAL 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 shut-down 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.

SAL was very useful in generating an improved test set with twice as many transition coverage. However, model checking with SAL is quite time consuming. For instance, defining and checking 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 i7 with 6 cores with 20GB of 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.

Using POOSL and SAL, we detected problems in the DSL instance, e.g., the simulation of a POOSL model revealed a missing condition. Moreover, we can detect whether a termstatus occurs in multiple main states. With SAL we found an error in the HVT power status of the recall configuration file.

6.6.3 Evaluation

We discuss the questions listed:

- *How much time is needed to learn the tools and techniques?*
  Clearly, the learning curve for new techniques depends on previous education and knowledge. With a Master's in Computer Science, including courses 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.

- *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 PDU 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, 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?*
  Currently, the configuration files need to be crafted manually. Because of the format of these configuration files, this is a difficult and error prone task. 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. It took only 45 hours to be ready for this increasing complexity. We can now create the configuration and test cases for the PDU in a readable, easy to change and maintainable format. The tests are now 3 times faster with a double coverage. It is expected that creating a new instance of the

configuration DSL and integrating the generated artefacts 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 [170]. It shows that most successful applications of model-driven development use small DSLs.

CHAPTER 7

MODEL LEARNING TO VALIDATE REFACTORING

This chapter describes the use of model learning to check the correctness of a refactored component. First we describe the motivation for applying model learning. Next we explain the learning approach in more detail. Then the industrial case is described to the extent needed for understanding this chapter. This is followed by the results of model learning and model checking to compare the legacy implementation with the new implementation. Lastly we discuss the scalability of our approach.

7.1 Motivation for the Application of Model Learning

In this chapter, we report about a novel industrial application of model learning to gain confidence in a refactored legacy component. We decided to use a combination of tools similar to the approach of [63, 7]. The model learning tool LearnLib [76] was used to learn models of the legacy component and the refactored implementation. The version of LearnLib we used can actively learn derministic Mealy machines. These models were then compared to check if the two implementations are equivalent. Since the manual comparison of large models is not feasible, we used an equivalence checker from the mCRL2 toolset [38] for this task.

We report about our experiences with the described approach on a real development project at Philips. The project concerns the introduction of a new hardware component, the Power Distribution Unit (PDU). As described in Section 1.5, the PDU is used to start-up and shut-down an interventional X-ray system. All computers in the system have a software component, the Power Control Service (PCS) which communicates with the PDU over an internal control network during the execution of start-up and shut-down scenarios. To deal with the new hardware of the PDU, which has a different interface, a new implementation of the PCS is needed. Since different configurations have to be supported, with old and new

PDU hardware, the old and new PCS software should have exactly the same externally visible behaviour.

7.2 Learning Approach

The learning approach can be described as follows (see also Figure 7.1):

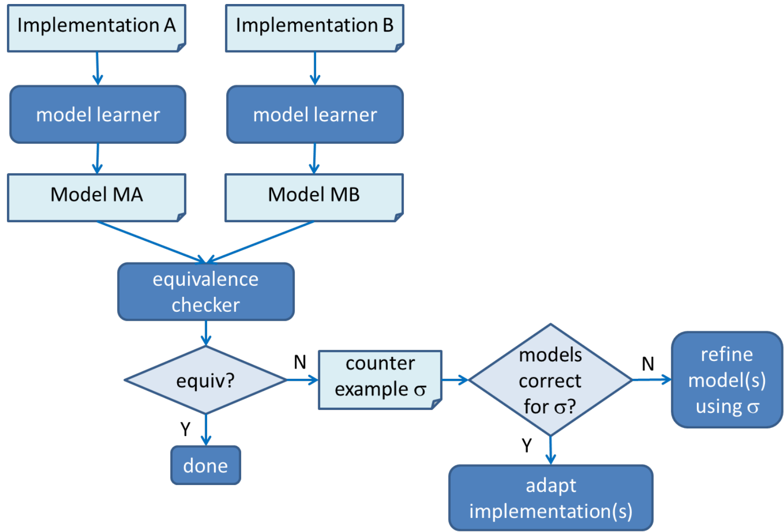


Figure 7.1: *Approach to Compare Legacy Component and Refactored Version*

1. Implementation A (the legacy component) is explored by a model learner. The output of the model learner is converted to an input format for the equivalence checker, model MA.

2. Implementation B (the refactored component) is explored by a model learner. The output of the model learner is converted to an input format for the equivalence checker, model MB.

3. The two models are checked by the equivalence checker. The result of the equivalence checker can be:

   - The two models are equivalent. In this case we are done.
   - The two models are not equivalent and a counterexample is provided: a sequence of inputs σ for which the outputs produced by the two models are different. In this case we proceed to step 4.

4. Because models A and B have been obtained through a finite number of tests, we can never be sure that they correctly describe implementations A and B, respectively. Therefore, if we find a counterexample σ for the equivalence of models MA and MB, we first check whether implementation A and model MA behave the same for σ, and whether implementation B and model MB

behave the same for σ. If there is a discrepancy between a model and the corresponding implementation, this means that the model is incorrect and we ask the model learner to construct a new model based on counterexample σ, that is, we go back to step 1 or 2. Otherwise, counterexample σ exhibits a difference between the two implementations. In this case we need to change at least one of the implementations, depending on which output triggered in response to input σ is considered unsatisfactory behaviour. Note that also the legacy component A might be changed, because the counterexample might indicate an unsatisfactory behaviour of A. After the change, a corrected implementation needs to be learned again, i.e., we go back to step 1 or 2 for relearning the corrected implementation.

Since the learning of an implementation can take a substantial amount of time, we start with a limited subset of input stimuli for the model learner and increase the number of stimuli once the implementations are equivalent for a smaller number of stimuli. Hence, the approach needs to be executed iteratively.

7.3 Context of the PCS

The PCS is a software component that is used to start and stop subsystems via their Session Managers (SMs). In addition to the start-up and shut-down scenarios executed by the PDU, the PCS is also involved during service scenarios such as upgrading the subsystem's software.

The PCS implementation for the old PDU is event-based. An event is handled differently based on the value of global flags in the source code. Hence, all state behaviour is implicitly coded by these flags, which makes the implementation unmaintainable. The development of a new implementation for supporting the new PDU is an opportunity to create a maintainable implementation. The new implementation makes the state behaviour explicit by a manually crafted state machine.

To be able to support both the old and the new PDU, the PCS software has been refactored such that the common behaviour for both PDUs is extracted. Figure 7.2(a) depicts the PCS before refactoring. The Host implements the IHost interface that is used by the service application. The implementation of the PCS after refactoring is show in Figure 7.2(b).

(a) Before refactoring (b) After refactoring

Figure 7.2: *Class Diagrams of PCS Design*

The PcsCommon class implements the ISessionManager interface to control

the SMs. The OldPduSupport class contains the legacy implementation for the old PDU whereas a NewPduSupport class deals with the new PDU. Both classes inherit from the PcsCommon class to achieve the same internal interface for the Host. Depending on the configuration, the Host creates an instance of either the OldPduSupport or the NewPduSupport class.

The PCS as depicted in Figure 7.2(b) is written in C++ and consists of a total of 3365 Lines Of Code (LOC): Host has 741 LOC, PcsCommon has 376 LOC, OldPduSupport has 911 LOC, and NewPduSupport has 1337 LOC.

The unit test cases were adapted to include tests for the new implementation. It was known that the unit test set is far from complete. Hence, we investigated the possibility to use model learning to get more confidence in the equivalence of the old and new implementations.

7.4 Application of the Learning Approach

To learn models of our implementations, we used the LearnLib tool [122], see `http://learnlib.de/`. For a detailed introduction into LearnLib we refer to [143]. In our application we used the development 1.0-SNAPSHOT of LearnLib and its MealyLearner which is connected to the System Under Learning (SUL) by means of an adapter and a TCP/IP connection.

7.4.1 Design of the Learning Environment

Figure 7.3 depicts the design used for learning the PCS component. Creating an initial version of the adapter took about 8 hours, because the test primitives of the existing unit test environment could be re-used.

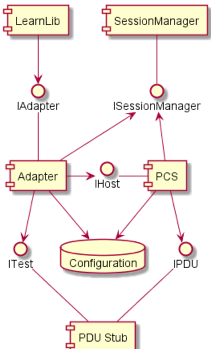


Figure 7.3: *Design Learning Environment*

With this design, the PCS can be learned for both the old and the new PDU. The adapter automatically changes the configuration of the PCS such that the PCS knows if it needs to instantiate the old or the new implementation. Depending on the old or new PDU, the adapter instantiates a different PDU stub.

7.4.2 Learned Output

The Mealy machine that is the result of a LearnLib session is represented as a "dot" file, which can be visualized using Graphviz[1]. A fragment of a model is shown in Table 7.1.

```
digraph g {
start0 [label="" shape="none"];

s0 [shape="circle" label="0"];
s1 [shape="circle" label="1"];
s2 [shape="circle" label="2"];
s3 [shape="circle" label="3"];
s4 [shape="circle" label="4"];
s5 [shape="circle" label="5"];
s6 [shape="circle" label="6"];
s7 [shape="circle" label="7"];
s8 [shape="circle" label="8"];
s0 -> s1 [label="|PDU(StateSystemOn)| / |PCS(Running);SM1(Running);SM2(Running)|"];
s0 -> s2 [label="|PDU(StateSystemOff)| / |PCS(Running);SM1(Stopped);SM2(Stopped);Dev(Shutdown)|"];
s1 -> s2 [label="|PDU(ButtonSystemOff)| / |PCS(Running);SM1(Stopped);SM2(Stopped);Dev(Shutdown)|"];
s1 -> s3 [label="|Host(goToOpenProfile)| / |PCS(Stopped);SM1(Stopped);SM2(Stopped);Dev(OpenProfile)|"];
...
start0 -> s0;
}
```

Table 7.1: *Fragment of a Learned DOT-File*

7.4.3 Checking Equivalence

For models with more than five states it is difficult to compare the graphical output of LearnLib for different implementations. Therefore, an equivalence checker is used to perform the comparison. In our case, we used the tool support for mCRL2 (micro Common Representation Language 2) which is a specification language that can be used for specifying system behaviour. The mCRL2 language comes with a rich set of supporting programs for analysing the behaviour of a modelled system [38].

Once the implementation is learned, a small script is used to convert the output from LearnLib to a mCRL2 model. Basically, the learned Mealy machine is represented as an mCRL2 process `Spec(s:States)`. As an example, the two transitions of state $s0$ in the dot-file

```
s0 -> s1 [label="|PDU(StateSystemOn)| / |PCS(Running);SM1(Running);SM2(Running)|"];
s0 -> s2 [label="|PDU(StateSystemOff)| / |PCS(Running);SM1(Stopped);SM2(Stopped);Dev(Shutdown)|"];
```

are translated into the following process algebra construction:

```
(s==s0) -> (
 (PDU(StateSystemOn) . PCS(Running) . SM1(Running) . SM2(Running) . Spec(s1)) +
 (PDU(StateSystemOff) . PCS(Running) . SM1(Stopped) . SM2(Stopped) . Dev(Shutdown) . Spec(s2))
)
```

A part of the result of translating the model of Table 7.1 to mCRL2 is shown in Table 7.2.

[1] www.graphviz.org/

```
sort States = struct s0 | s1 | s2 | s3 | s4 | s5 | s6 | s7 | s8;
OsStim = struct StartPcs | StopPcs;
PDUStim = struct StateSystemOn | StateSystemOff | ...;
HostStim = struct stopForInstallation | startAfterInstallation | ...;
ServiceStates = struct Running | Stopped;
DevStates = struct OpenProfile | Shutdown;

act OS:OsStim;
act PDU:PDUStim;
act Host:HostStim;
act PCS:ServiceStates;
act SM1:ServiceStates;
act SM2:ServiceStates;
act Dev:DevStates;

proc Spec(s:States)=
 (s==s0) -> (
  (PDU(StateSystemOn) . PCS(Running) . SM1(Running) . SM2(Running) . Spec(s1)) +
  (PDU(StateSystemOff) . PCS(Running) . SM1(Stopped) . SM2(Stopped) . Dev(Shutdown) . Spec(s2))
 ) +
 (s==s1) -> (
  (PDU(ButtonSystemOff) . PCS(Running) . SM1(Stopped) . SM2(Stopped) . Dev(Shutdown) . Spec(s2)) +
  (Host(goToOpenProfile) . PCS(Stopped) . SM1(Stopped) . SM2(Stopped) . Dev(OpenProfile) . Spec(s3)) +
  (Host(goToClosedProfile) . PCS(Stopped) . SM1(Stopped) . SM2(Stopped) . Spec(s4)) +
  (Host(openProfileStartApplication) . PCS(Running) . SM1(Running) . SM2(Running) . Spec(s1)) +
  (Host(openProfileStopApplication) . PCS(Running) . SM1(Running) . SM2(Running) . Spec(s1)) +
  (OS(StartPcs) . PCS(Running) . SM1(Running) . SM2(Running) . Spec(s1)) +
  (OS(StopPcs) . PCS(Stopped) . SM1(Stopped) . SM2(Stopped) . Spec(s4))
 ) +
 (s==s2) -> (
 ...
 );

init Spec(s0);
```

Table 7.2: *Fragment of mCRL2 Model*

Given two (deterministic) Mealy machines, the labelled transition systems for the associated mCRL2 processes are also deterministic. Since the labelled transition systems also do not contain any τ-transitions, trace equivalence and bisimulation equivalence coincide, and there is no difference between weak and strong equivalences [48]. Thus, two Mealy machines are equivalent iff the associated mCRL2 processes are (strong) trace equivalent, and the mCRL2 processes are (strong) trace equivalent iff they are (strong) bisimulation equivalent.

7.4.4 Investigating Counterexamples

When the equivalence check indicates that the two models are not equivalent, the mCRL2 tool provides a counterexample. To investigate counterexamples, we created a program that reads a produced counterexample and executes this on the implementations. In the design depicted in Figure 7.3, the LearnLib component has been replaced by the counterexample program. As before, switching between the two implementations can be done by instructing the adapter. In this way, the standard logging facilities of program execution are exploited to study the counterexample.

7.5 Results of Learning the Implementations of the PCS

In this section we describe the results of applying the approach to the implementations of the PCS component.

7.5.1 Iteration 1

The first iteration was used to realize the learning environment as is described in Section 7.4.1. An adapter was created to interface between the PCS and LearnLib. Because the communication between the PCS and the adapter is asynchronous, the adapter has to wait some time before the state of the PCS can be examined. In this iteration we performed a few try runs to tweak the wait time needed before taking a sample. In addition, the first iteration was used to get an impression on how long learning the PCS takes with different numbers of stimuli. The necessary waiting time of 10 seconds after a stimulus for learning the PCS is quite long, and this greatly influenced the time needed for learning models.

7.5.2 Iteration 2

After a first analysis of the time needed for model learning in iteration 1, we decided to start learning with 9 stimuli. These 9 stimuli were all related to basic start-up/shut-down and service scenarios. We learned the PCS implementation for the old PDU and the PCS implementation for the new PDU. The results are presented in Table 7.3. The table has a column for the number of stimuli, for the number of states and transitions found, and for the time it took for LearnLib to learn the implementations.

	Stimuli	States	Transitions	Time (in seconds)
PCS impl. for old PDU	9	8	43	32531
PCS impl. for new PDU	9	3	8	1231

Table 7.3: *Results Learning PCS with 9 Stimuli*

Note that learning a model for the old implementation took 9 hours. (This excludes the time used to test the correctness of the final model.) As described in Section 7.4.3, the learned models were converted to mCRL2 processes. Next, the mCRL2 tools found a counterexample starting with:

PDU(StateSystemOn), PCS(Running), SM1(Running), SM2(Running), ...

We investigated this counterexample and found an issue in the PCS implementation for the new PDU. The new implementation did not make a distinction between the SystemOff event, and the ServiceStop and ServiceShutdown events.

Note that before performing the learning experiment the new and old implementations were checked using the existing regression test cases. This issue was not found by the existing unit test cases.

7.5.3 Iteration 3

In the third iteration, the PCS implementation for the new PDU was re-learned after solving the fix. Table 7.4 describes the results.

	Stimuli	States	Transitions	Time (in seconds)
PCS impl. for old PDU	9	8	43	32531
PCS impl. for new PDU	9	7	36	8187

Table 7.4: *Results Learning PCS with 9 Stimuli*

An equivalence check with the mCRL2 tools resulted in a new counterexample of 23 commands:
PDU(StateSystemOn), PCS(Running), SM1(Running), SM2(Running), Host(goToOpenProfile), PCS(Stopped), SM1(Stopped), SM2(Stopped), Dev(OpenProfile), OS(StartPcs), PCS(Running), SM1(Stopped), SM2(Running), Dev(OpenProfile), Host(openProfileStopApplication), PCS(Running), SM1(Stopped), SM2(Running), Dev(OpenProfile), PDU(ButtonSystemOff), PCS(Running), SM1(Stopped), SM2(Running).

When we executed this counterexample on the PCS implementation for the old PDU, we found the following statement in the logging of the PCS: "Off button not handled because of PCS state (Stopping)". A quick search in the source code revealed that the stopSessionManagers method prints this statement when the Stopping flag is active. This is clearly wrong, because this flag is set by the previous stimulus, i.e., the openProfileStopApplication stimulus. The PCS implementation for the old PDU was adapted to reset the Stopping flag after handling the openProfileStopApplication stimulus.

7.5.4 Iteration 4

In the fourth iteration, the PCS implementation for the old PDU was re-learned after solving the fix. Table 7.5 describes the results after re-learning. Note that, after correcting the error, learning the model for the old implementation only takes slightly more than one hour. When checking the equivalence, the mCRL2 tool reports that the two implementation are (strong) trace equivalent for these 9 stimuli.

	Stimuli	States	Transitions	Time (in seconds)
PCS impl. for old PDU	9	7	36	4141
PCS impl. for new PDU	9	7	36	8187

Table 7.5: *Results Learning PCS with 9 Stimuli*

7.5.5 Iteration 5

As a next step we re-learned the implementations for the complete set of 12 stimuli; the results are shown in Table 7.6. Note that learning the new implementation takes approximately 3.5 hours. The mCRL2 tools report that the two obtained models with 12 stimuli are trace equivalence and bisimulation equivalent.

	Stimuli	States	Transitions	Time (in seconds)
PCS impl. for old PDU	12	9	65	10059
PCS impl. for new PDU	12	9	65	12615

Table 7.6: *Results Learning PCS with 12 Stimuli*

7.6 Scalability of the Learning Approach

Using model learning we found issues in both a legacy software component and in a refactored implementation. After fixing these issues, model learning helped to increase confidence that the old and the new implementations behave the same. Although this is a genuine industrial project, the learned Mealy machine models are very small. Nevertheless, learning these tiny models already took up to 9 hours. For applying these techniques in industry there is an obvious need to make model learning more efficient in terms of the time needed to explore a system under learning. Clearly, our approach has been highly effective for the PCS. But will it scale?

Below we present an overview of some recent results that make us optimistic that indeed our approach can be scaled to a large class of more complex legacy systems.

7.6.1 Faster implementations

The main reason why model learning takes so long for the PCS is the long waiting time in between input events. As a result, running a single test sequence (a.k.a. membership query) took on average about 10 seconds and a reset of the implementation took about 5 seconds. In another industrial project a model for a printer controller was learned with 3410 states and 77 stimuli [141]. Even though more than 60 million test sequences were needed to learn it, the task could be completed within 9 hours because on average running a single test sequence took only 0.0005 seconds. For most software components the waiting times can be much smaller than for the PCS component studied in this chapter. In addition, if the waiting times are too long then sometimes it may be possible to modify the components (just for the purpose of the model learning) and reduce the response times. For our PCS project such an approach is difficult. The PCS controls the Session Managers (SMs), which are Windows services. After an input event we want to observe the resulting state change of the SMs, but due to the unreliable timing of the OS we need to wait quite long. In order to reduce waiting times we would need to speed up Windows.

7.6.2 Faster Learning and Testing Algorithms

There has been much progress recently in developing new algorithms for automata learning. In particular, the new TTT learning algorithm that has been introduced by Isberner [80] is much faster than the variant of Angluin's L^* algorithm [10] that we used in our experiments. Since the models for the PCS components are so simple, the L^* algorithm does not need any intermediate hypothesis: the first model that L^* learns is always correct (that is, extensive testing did not reveal any counterexample). The TTT algorithm typically generates many more intermediate hypotheses than L^*. This means that it becomes more important which testing algorithm is being used. But also in the area of conformance testing there has been much progress recently [44, 141]. Figure 7.4 displays the results of some experiments that we did using an implementation of the TTT algorithm that has become available very recently in LearnLib, in combination with a range of testing algorithms from [44, 141]. As one can see, irrespective of the test method that is used, the TTT algorithm reduces the total number of input events needed to learn the final PCS model with a factor of about 3.

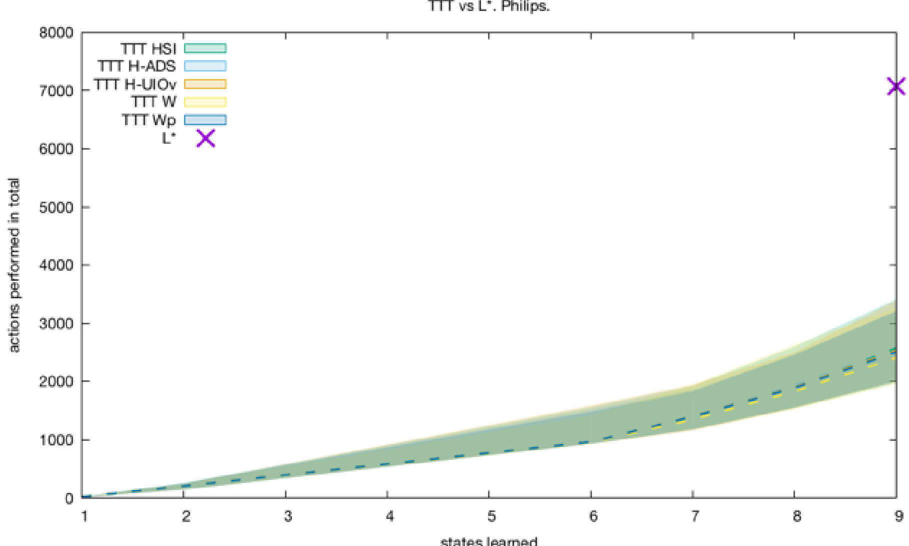


Figure 7.4: *Experiments with TTT algorithm for final PCS implementation for new PDU. The used test methods (W, Wp, hybrid adaptive distinguishing sequences, hybrid UIOv) were all randomised. For each test method 100 runs were performed. In each case 95% of the runs were in the shaded area. The dotted lines give the median run for a given test method.*

7.6.3 Using Parallelization and Checkpointing

Learning and testing can be easily parallelized by running multiple instances of the system under learning (in our case the PCS implementation) at the same

time. Henrix [73] reports on experiments in which doubling the number of parallel instances nearly doubles the execution speed (on average with a factor 1.83). Another technique that may speed-up learning is to save and restore software states of the system under learning (checkpointing). The benefit is that if the learner wants to explore different outgoing transitions from a saved state q it only needs to restore q, which usually is much faster than resetting the system and bringing it back to q by an appropriate sequence of inputs. Henrix [73] reports on experiments in which checkpointing with DMTCP [11] speeds up the learning process with a factor of about 1.7.

7.6.4 Using Abstraction and Restriction

The number of test/membership queries of most learning algorithms grows linearly with the number of inputs. However, these algorithms usually assume an oracle that provides counterexamples for incorrect hypothesis models. Such an oracle is typically implemented using a conformance testing algorithm. In practice, conformance testing often becomes a bottleneck when the number of inputs gets larger. Thus we seek methods that help us to reduce the number of inputs.

To get confidence that two implementations with a large number of stimuli exhibit the same behaviour, a simple but practical approach is to apply model learning for multiple smaller subsets of stimuli. This will significantly reduce the learning complexity, also because the set of reachable states will typically be smaller for a restricted number of stimuli. Models learned for a subset of the inputs may then be used to generate counterexamples while learning models for larger subsets on inputs. Smeenk [140] reports on some successful experiments in which this heuristic was used.

A different approach, which has been applied successfully in many case studies, is to apply abstraction techniques that replace multiple concrete inputs by a single abstract input. One may, for instance, forget certain parameters of an input event, or only record the sign of an integer parameter. We refer to [6, 32] for recent overviews of these techniques.

7.7 Concluding Remarks

We presented an approach to get confidence that a refactored software component has equivalent external control behaviour as its non-refactored legacy software implementation. From both the refactored implementation and its legacy implementation, a model is obtained by using model learning. Both learned models are then compared using an equivalence checker. The implementations are learned and checked iteratively with increasing sets of stimuli to handle scalability. By using this approach we found issues in both the refactored and the legacy implementation in an early stage of the development, before the component was integrated. In this way, we avoided costly rework in a later phase of development.

CHAPTER 8

REFACTORING A LEGACY IMPLEMENTATION USING A DSL

In this chapter, we describe a case in which models created with a legacy tool are transformed to models that can be used by another tool. The transformation is established by means of a DSL for the legacy models. The model transformations are defined as a generator of this DSL and we explain how confidence in these transformations has been obtained.

8.1 Motivation for the Transformation

Rhapsody [77] is an UML-based modelling tool that is used at Philips to develop software components. In Rhapsody, state machines can be modelled graphically. From these models, the Rhapsody tooling generates source code in C++ which can be integrated into the product. The generated source code depends on run-time libraries from Rhapsody that are required to execute the state machine code.

Unfortunately, the use of Rhapsody did not bring the expected benefits. Partly this is due to a wrong use of the tool, but it is also not very convenient to use the Rhapsody tooling in the way of working at Philips. Code, e.g., to describe actions on transitions, has to be entered via a small window and it is difficult to get a good overview. The main problem is the difficulty to merge development streams, which easily leads to product faults.

This led to the wish to remove the dependency on Rhapsody and to migrate Rhapsody models to Dezyne models. Dezyne is a commercial modelling tool created by the company Verum. It is the successor of the ASD approach described in Chapter 3. Whereas ASD models have a tabular notation, Dezyne models are described in a text-based horizontal DSL. All model checking capabilities present in ASD are also available in Dezyne. From Dezyne models source code can be generated. Dezyne is already used at Philips, hence a transformation of Rhapsody to Dezyne would also reduce the number of tools that have to be supported.

As an intermediate step in the transformation, we use ComMA models [91]. Component Modelling & Analysis (ComMA) is a horizontal DSL created at Philips. The main business driver for this DSL was the need to test and monitor interfaces of components. For instance, to check interface compliance of third-party components. Interface definitions in ComMA include state machines to describe the allowed interactions, similar to UML's protocol state machines. In the approach of this chapter, the state machine notation of ComMA is used to describe the internal behaviour of components and to generate a Dezyne model from it. In this way, we avoid a strong dependency on Dezyne and allow the direct generation of source code from ComMA models in the future.

In this chapter we investigate the following two research questions:

- Can a DSL be used to migrate from one model to another model?

- Is it financially feasible to transform a legacy model using a DSL?

8.2 Transformation Approach

The migration from Rhapsody to Dezyne is based on two steps:

1. From a Rhapsody model, an implementation is generated. The generated C++ source code is placed in a Microsoft Visual Studio project. After this, the implementation can be maintained without the use of Rhapsody.

2. The implementation generated in step 1 still depends on Rhapsody run-time libraries for the Rhapsody generated state machines. To be able to remove the Rhapsody dependencies, the Rhapsody state machines are transformed into Dezyne state machines. From Dezyne state machine source code is generated and integrated into the Visual Studio project of step 1.

The approach to migrate Rhapsody models into Dezyne models is depicted in Figure 8.1. From a Rhapsody model, the Rhapsody tool can generate an implementation in source code that includes the Rhapsody state machines. After transforming the state machines in a Rhapsody model to a Dezyne model, the Dezyne tool can generate a state machine implementation. The Dezyne state machines replaces the Rhapsody state machines in the Rhapsody generated implementation. An example of a Rhapsody model instance is shown in Figure 8.2. It describes a transition from one state (indicated by *_itsSource*) to another state (indicated by *_itsTarget*). The transition is triggered by *tgPostDone*. When triggered and the guard (*AllSlavePostExecuted*) evaluates to true, the action body is executed. A DSL, called Rhapsody DSL, is defined such that it accepts such Rhapsody models as language instances. A generator of this DSL transforms the model into a ComMA instance. This transformation is described formally in Section 8.3.

Next a Dezyne model is generated from a ComMA instance, as described in Section 8.3.3. Finally, from a Dezyne model a state machine implementation in C++ is generated which replaces the Rhapsody generated state machine.

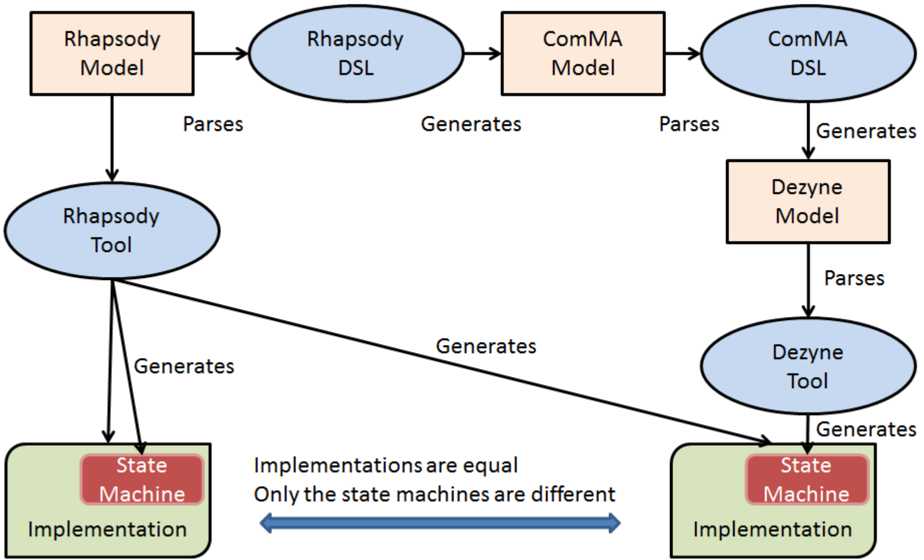


Figure 8.1: *Transformation Approach*

```
        { ITransition
          - _id = GUID b84b7817-9034-4188-8840-e5512cfc7cc3;
          - _myState = 2048;
          - _name = "transition_22";
          - _itsLabel = { ILabel
            - _id = GUID 54724896-ea95-419d-a743-614a16e67224;
            - _itsTrigger = { IInterfaceItemTrigger
              - _id = GUID fe220d97-f1b7-49ec-94a4-74199cf9ea10;
              - _body = "tgPostDone";
              - _info = "";
              - _itsInterfaceItem = { IHandle
                - _m2Class = "ITriggered";
                - _id = GUID d1519062-f7b1-47df-8f30-48fed1243784;
              }
            }
            - _itsGuard = { IGuard
              - _id = GUID c301d2e1-d81d-4417-b944-ca3cb51aa999;
              - _body = "AllSlavePostsExecuted()";
            }
            - _itsAction = { IAction
              - _id = GUID 79ff1540-ae32-4bcc-b9c4-2b917c2637aa;
              - _body = "
TracePrintf(MCI_traceLevelLow, \"All POSTs executed\");
m_PostExecuteSemaphore->signal();";
            }
          }
          - _itsTarget = GUID a6e1e650-3930-41c1-b4f9-2a388e6f7e83;
          - _staticReaction = 0;
          - _itsSource = GUID 29147b0c-6222-4973-b3fa-ea1bc9175465;
        }
```

Figure 8.2: *Fragment of a Rhapsody Model*

8.3 State Machine Transformations

In this section, we describe the essence of the transformation from Rhapsody state machines to ComMA state machines. The aim is to define this transformation in such a way that the generation from ComMA to Dezyne is trivial. Our notations are inspired by [40, 50].

State machine representation A state machine (SM) is represented using a tuple $< s_d, S, T, Sub >$ where

- S is a set of *states*. A state s has a name, denoted by $name(s)$. Optionally, a state may have a Global Unique IDentifier (GUID), denoted by $GUID(s)$. The set S may contain so-called connector states (denoted by $s_c, s_{c_1}, \ldots$). A connector state is a state that can be used to model choices on transitions.
- $s_d \in S$ is the *default state*, also called the initial state, and denoted by *default(SM).*
- T is a set of *transitions*. A transition has the from $s_0 \xrightarrow{[g]act/a} s_1$ where
  - $s_0 \in S$ is the *source state*.
  - g is a *guard*, which is a boolean expression consisting of method calls that evaluate to either true or false. The guard is optional, i.e., can be omitted.
  - act is an *activation* which can be a *trigger* which is a synchronous method or an asynchronous *event*. For a synchronous method, the caller has to wait until the callee is available. Events are queued. The activation part is optional.
  - a is a list of *actions*. An action is a method call. The action part is optional.
  - $s_1 \in S$ is the *target state*.

  For such a transition t we use $source(t) = s_0$, $guard(t) = g$, $activation(t) = act$, $action(t) = a$, and $target(t) = s_1$.
- Sub is set of sub state machines. A sub state machine is a tuple of the form $< s_i, SM_i >$ where $s_i \in S$ and SM_i is a state machine. Note that Sub may be empty.
  The states s_i occurring in sub state machines are called *super states*. $Super(s)$ is true iff s is a super state.

Figure 8.3 depicts an example state machine (SM) $< s_d, S, T, Sub >$ where

- The *default state* is $s_d = s_0$.
- The set of states S is $\{s_0, s_1, s_2, s_3, s_4\}$.
- The set of transitions T is $\{t_0, t_1, t_2, t_3\}$ for which:
  - $t_0 = s_0 \xrightarrow{[g_0]act_0/a_0} s_1$

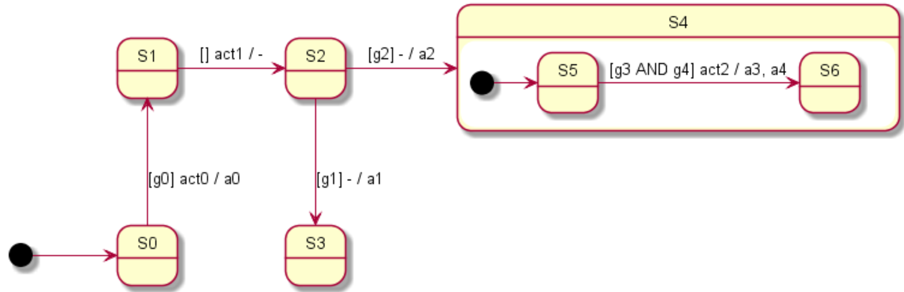


Figure 8.3: *Example of a State Machine*

- $t_1 = s_1 \xrightarrow{[]act_1/-} s_2$
- $t_2 = s_2 \xrightarrow{[g_1]-/a_1} s_3$
- $t_3 = s_2 \xrightarrow{[g_2]-/a_2} s_4$

- Sub is $< s_4, SM_4 >$.

For sub-state machine SM_4 is defined by $< s_{d_4}, S_4, T_4, Sub_4 >$ where

- The *default state* is $s_d = s_5$.
- The set of states $S_4 = \{s_5, s_6\}$.
- The set of transitions $T_4 = \{s_5 \xrightarrow{[g_3 \wedge g_4]act_2/a_3,a_4} s_6\}$.
- $Sub_4 = \varnothing$.

Rhapsody models The Rhapsody models considered for the transformations discussed in this chapter are state machines, but they satisfy a number of constraints. First of all, states and GUIDs are unique:

- Each state occurs at most once in the hierarchy of states. The same holds for GUIDs. However, there can be multiple states with the same name.

Moreover, for each transition t:

- If t has an action list, then the list has length 1.
- If t has a guard, then this is a single method call which returns a boolean value.
- If $source(t)$ is a connector state then t has no activation. It may have a guard and may have an action. Moreover, $target(t) \neq source(t)$, i.e., connector states do not have self transitions.
- If $source(t)$ is not a connector state and $target(t)$ is a connector state, then t has an activation, but no guard and no action.

- For each connector state s_c there is one incoming transition (for which s_c is the target) and two outgoing transitions (for which s_c is the source). Moreover, the guard of one outgoing transition is the negation of the guard on the other transition.

- The source of t is not a super state, i.e., $\neg Super(source(t))$.

A state machine which satisfies our restrictions on a Rhapsody model is denoted by SM_r.

ComMA models A state machine in ComMA satisfies the following restrictions:

- States only have a name, no GUID. Names are unique, i.e., if $s_1 \neq s_2$, then $name(s_1) \neq name(s_2)$.

- There are no connector states.

- There are no sub state machines (and hence no super states), i.e., $Sub = \varnothing$.

Note that for readability we choose to use state names instead of GUIDs for the states in ComMA. To allow a straightforward translation from ComMA models to Dezyne we add one additional restriction on the ComMA models used in this chapter:

- All transitions have an activation.

A state machine which satisfies the restrictions above on a ComMA model is denoted by SM_c.

8.3.1 From Rhapsody to ComMA

In the previous subsection the Rhapsody and ComMA models are formally described. In this subsection, we describe transformations needed to come from a Rhapsody to a ComMA model. The following transformations are required to convert the models:

- Rename duplicate state names to allow removal of GUIDs.

- Remove hierarchy, i.e., allow removal of sub state machines.

- Remove connector states.

- Make input enabled. Implicitly, all inputs are allowed in all states in Rhapsody models.

- Add an activation to transitions without it.

The transformations are applied to a Rhapsody state machine $SM_r =< s_d, S, T, Sub >$.

Rename duplicate state names In the first transformation the duplicate states are renamed using the *RenameDuplicateNames* method. We use + for string concatenation and *asString* to convert a number to a string.

RenameDuplicateNames(SM) ::=
FORALL $s \in S$ **DO**
  $nr \leftarrow 1$
  FORALL $s' \in S, s' \neq s$ **DO**
    IF $name(s) = name(s')$ **THEN**
      $name(s') \leftarrow name(s') + asString(nr)$
      $nr \leftarrow nr + 1$
    FI
  OD
OD
RETURN SM

Remove hierarchy Removing the hierarchy is done in a number of steps. To remove the sub state machines, we first remove transitions to super states as follows:

RemoveSuperStates(SM) ::=
FORALL $< s_i, SM_i > \in Sub$ **DO**
  FORALL $t \in T$ **AND** $target(t) = s_i$ **DO**
    $target(t) \leftarrow default(SM_i)$
  OD
  $S \leftarrow S \setminus s_i$
  RemoveSuperStates(SM_i)
OD
RETURN SM

Next, all states and all transitions are collected by the following recursive definitions for a state machine SM:

- if $Sub = \emptyset$, then define $S_{tot}(SM) = S$ and $T_{tot}(SM) = T$.
- Otherwise, for $Sub = \{< s_1, SM_1 >, \ldots, < s_n, SM_n >\}$ define
  $S_{tot}(SM) = S \cup S_{tot}(SM_1) \cup \ldots \cup S_{tot}(SM_n)$ and
  $T_{tot}(SM) = T \cup T_{tot}(SM_1) \cup \ldots \cup T_{tot}(SM_n)$.

This is used in the following transformation:

RemoveHierarchy(SM) ::=
$SM' \leftarrow$ *RemoveSuperStates*(SM)
$S' \leftarrow S_{tot}(SM')$
$T' \leftarrow T_{tot}(SM')$
$Sub' \leftarrow \varnothing$
RETURN SM'

Remove connector states The connector states are removed using the transformation *RemoveConnectorStates*. Note that, by the restrictions on Rhapsody state machines, an outgoing transition has no activation.

$RemoveConnectorStates(SM)$ ::=
FORALL connector state $s_c \in S$ **DO**
  FORALL $t \in T$ with $target(t) = s_c$ **DO**
    FORALL $t_1 \in T$ with $source(t) = s_c$ **DO**
      $g \leftarrow guard(t) \wedge guard(t_1)$
      $a \leftarrow action(t), action(t_1)$
      $T \leftarrow T \cup \{source(t) \xrightarrow{[g]activation(t)/a} target(t_1)\}$
      $T \leftarrow T \setminus t_1$
    OD
    $T \leftarrow T \setminus t$
  OD
  $S \leftarrow S \setminus s_c$
OD
RETURN SM

Make input enabled To ensure that in all states there is a transition for every activation, *EnableInput* adds self transitions with activations to states that are not input enabled. The reason for this transformation is explained in Section 8.4. *EnableInput* uses two helper methods:

- $Act_{tot}(T) = \{activation(t) \mid t \in T\}$
- $Act_{state}(s)$ to collect all activations used in a certain state, defined by $Act_{state}(s, T) = \{activation(t) \mid t \in T \wedge source(t) = s\}$

$EnableInput(SM)$ ::=
FORALL $s \in S$ **DO**
  $Act \leftarrow Act_{tot}(T) \setminus Act_{state}(s, T)$
  FORALL $act \in Act$ **DO**
    $T \leftarrow T \cup \{s \xrightarrow{act/-} s\}$
  OD
OD
RETURN SM

Add an activation to transitions To ensure that all transitions have an activation, every transition t without an activation gets an activation event ($event_{E_n}$), where n is some unique number. All incoming transitions of $source(t)$ get an additional action ($action_{E_n}$). This action places an event ($event_{E_n}$) in the queue. This is formalized in transformation *HandleActivation*.

$HandleActivation(SM)$ ::=
$nr \leftarrow 1$
FORALL $t \in T$ without an activation **DO**
  $activation(t) \leftarrow event_{E_{nr}}$

FORALL $t' \in T$ with $target(t') = source(t)$ **DO**
$action(t') \leftarrow add(action(t'), action_{E_{nr}})$
OD
$nr \leftarrow nr + 1$
OD
RETURN SM

This leads to complete transformation of a Rhapsody state machine SM_r to a ComMA model SM_c as follows:
$SM_1 = RenameDuplicateNames(SM_r)$
$SM_2 = RemoveHierarchy(SM_1)$
$SM_3 = RemoveConnectorStates(SM_2)$
$SM_4 = EnableInput(SM_3)$
$SM_c = HandleActivation(SM_4)$
After the described transformations, the state machine of Figure 8.3 is transformed into the state machine shown in Figure 8.4.

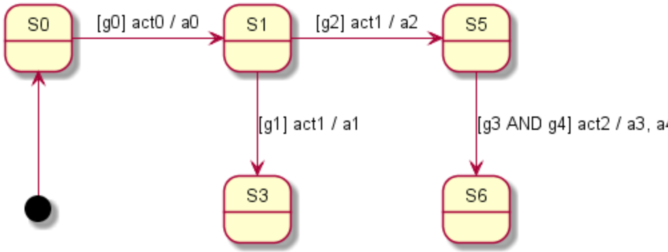


Figure 8.4: *Result of Transforming the State Machine of Figure 8.3*

8.3.2 Generating ComMA Instances

The mathematical description of the transformations in the previous subsection has been encoded in generators of the Rhapsody DSL. The resulting ComMA model is defined by three files, as shown in Figure 8.5:

- The .if file describes the interfaces signatures. In the .if file two interfaces are defined: the first interface (IImpl) includes all activations, and the second interface (IUsed) includes all actions.
- The .sm file contains the state machine behaviour specification.
- The .mp file provides a mapping from the guards and actions to the source code implementation of these methods.

Figures 8.6 and 8.7 show an example of an interface signatures file and a state machine behaviour specification file, respectively. Both examples correspond to the state machine as depicted in Figure 8.4.

Figure 8.8 provides an example of the mapping from actions and guards to the source code implementation. In this example, we assume that the transition

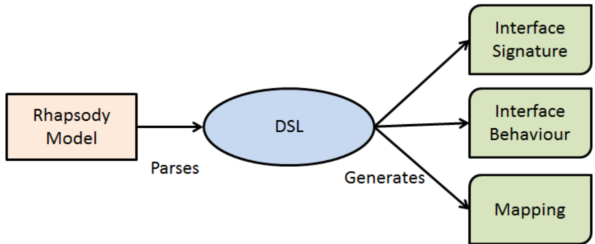


Figure 8.5: *Generating Two ComMA Files and a Mapping File*

$S_0 \xrightarrow{[g0]act0/a0} S_1$ of Figure 8.4 corresponds to the transition in the Rhapsody model fragment depicted in Figure 8.2. The mappings are separated by ###. Note that in the source code there are no lines starting with = or ###.

```
interface IImpl {
    commands
        void act0
        void act1
        void act2
}

interface IUsed {
    commands
        void a0
        void a1
        void a2
        void a3
        void a4
}
```

Figure 8.6: *Example of a ComMA Interface File*

```
import "Interfaces.if"
Behavior:
    machine StateMachine provides IImpl requires IUsed {
        // ...
        initial
        state S0 {
            transition trigger: IImpl::act0
                guard: (g0)
                do:    IUsed::a0
                    next state: S1
        }
        state S1 {
            transition trigger: IImpl::act1
                guard: (g1)
                do:    IUsed::a1
                    next state: S3
            transition trigger: IImpl::act1
                guard: (g2)
                do:    IUsed::a2
                    next state: S5
        }
        // ...
    }
```

Figure 8.7: *Example of a Specification of Interface Behaviour in ComMA*

```
g0
=
AllSlavePostsExecuted()
###
a0
=
TracePrintf(MCI_traceLevelLow, \"All POSTs executed\");
m_PostExecuteSemaphore->signal();
###
```

Figure 8.8: *Example of a Mapping File*

8.3.3 From ComMA to Dezyne

In the ComMA framework, we have defined a generator which transforms a ComMA model (i.e., an interface file and a state machine file) and a mapping file into a Dezyne model.

The ComMA interface of Figure 8.6 is transformed into Dezyne interfaces as shown in Figure 8.9. Observe that, different from the ComMA model, the guards are part of the IUsed interface and return an enum value of type RVal which contains the values Ok and Nok.

```
interface IImpl {
    in void act0();
    in void act1();
    in void act2();
    behaviour {}
}
interface IUsed {
    in void a0();
    in void a1();
    in void a2();
    in void a3();
    in void a4();
    enum RVal {Ok, Nok};
    in RVal g0();
    in RVal g1();
    in RVal g2();
    in RVal g3();
    in RVal g4();
    behaviour {}
}
```

Figure 8.9: *Dezyne Interfaces*

Figure 8.10 shows part of the Dezyne model which is generated from the ComMA state machine of Figure 8.7. Note that by our restrictions on a Rhapsody state machine, we have $g2 = \neg g1$. In the Dezyne model this is generated as an else clause.

8.4 Increasing Confidence in the Generated Code

After model checking the Dezyne models, source code in C++ is generated and integrated into a Visual Studio project. The existing regression test set is executed

```
component StateMachine {
    provides IImpl impl;
    requires IUsed used;
    behaviour   {
        enum State {S0, S1, S2, S3, S4, S5};
        State state = State.S0;
        [state.S0] {
            on impl.act0(): {
                IUsed.RVal rVal = used.g0();
                if (rVal == IUsed.RVal.Ok) {
                    used.a0();
                    state = State.S1;
                }
            }
        }
        [state.S1] {
            on impl.act1(): {
                IUsed.RVal rVal = used.g1();
                if (rVal == IUsed.RVal.Ok) {
                    used.a1();
                    state = State.S3;
                } else { // g2
                    used.a2();
                    state = State.S5;
                }
            }
        }
        // et cetera
    }
}
```

Figure 8.10: *Dezyne Instance*

to get confidence in the generated code. To further increase the confidence, we applied the learning approach of Chapter 7.

Figure 8.11 depicts the use of learning in our approach. The two versions of state machine code, generated by Rhapsody and Dezyne, are stimulated by all possible inputs and the resulting outputs are examined by the model learner. The two models that are the result of the learning phase are compared by the equivalence checker of the mCRL2 tool set.

With the learning approach, we found two errors in the Dezyne models that were not detected by the existing regression test set. The next two paragraphs describe these errors.

Make the model input enabled By learning and comparing the Rhapsody and Dezyne state machines, we found out that the Rhapsody generated implementation is implicitly made input enabled by the Rhapsody run-time libraries. This behaviour is not described in the Rhapsody models. In the Dezyne models all transitions have to be defined explicitly. Before we started learning the implementations, the *EnableInput* method was not present in the transformation from Rhapsody to Dezyne. After adding the *EnableInput* method, re-learning confirmed

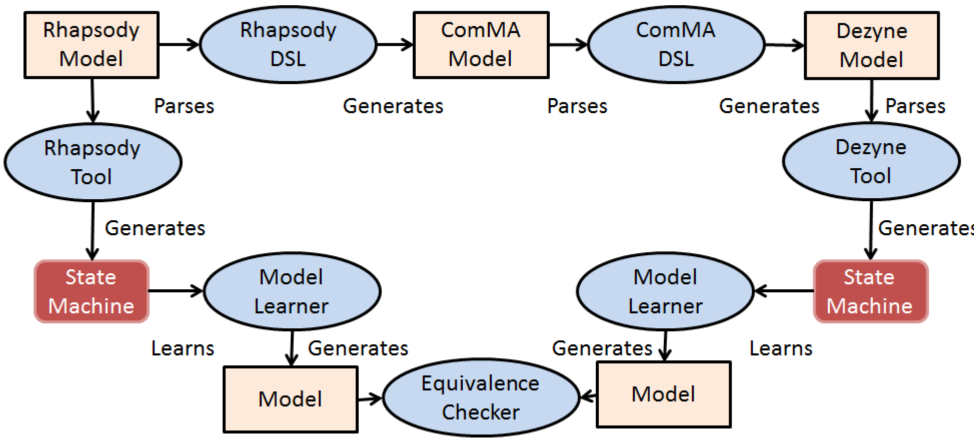


Figure 8.11: *Learning Approach*

that both state machines are input enabled.

Add an activation to transitions The first solution for the *HandleActivation* method had a different implementation. Because every state with a transition without an activation had a self-transition with only an activation, these two transitions were combined to get a transition with an activation and the self-transition was removed. When we found out that Rhapsody models are input enabled, the *HandleActivation* method was modified according to the implementation described in Subsection 8.3.1.

After fixing the two issues found, re-learning confirmed that both state machines exhibit equivalent external behaviour. As proposed in Chapter 7, we used an iterative approach. We did two learning experiments for learning two state machines. The first state machine was learned with 14 stimuli and the second state machine was learned with 22 stimuli. Table 8.1 lists the final results of these two learning experiments with the number of states and the number of transitions found.

Experiment	Stimuli	States	Transitions
1	14	34	476
2	22	58	1276

Table 8.1: *Learning Results*

8.5 Concluding Remarks

In this chapter, we presented an approach to use a DSL to transform models from one tool to another tool. We applied the approach to transform Rhapsody models into Dezyne models. To gain confidence in the transformation, model learning

was applied. The experiences with this approach at Philips lead to the following answers to the research questions of Section 8.1.

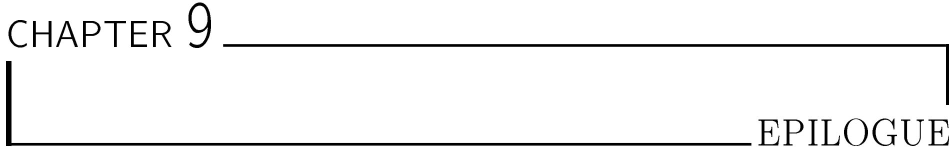
- *Can a DSL be used to migrate from one model to another model?*
  When the model that needs to be converted is in a textual and human readable format, a DSL can be created that accepts this model as language instance. Next a generator of the DSL can transform this instance into the target model format. Given the existence of powerful techniques such as Xtext and Xtend, this approach can be realized quickly and conveniently. In addition, it is rather easy and fast to create additional generators for instance for visualisations. An alternative approach would be to write a script, for instance using Perl or Python, that does the parsing and transformation. We expect that this would consume more time and requires more expertise, e.g., concerning parsing.

- *Is it financially feasible to transform a legacy model using a DSL?*
  We calculate the Return On Investment (ROI) for the presented DSL. First we compute the required investment for the model transformation approach. It took about 60 hours to learn the Rhapsody model structure, create the Rhapsody DSL and add a Dezyne generator to ComMA. Increasing confidence into the transformations by model learning and equivalence checking costed an additional 15 hours. Placing the generated code into a Visual Studio project took another 25 hours. Hence, the total investment was approximately 100 hours.

  Next we compare the approach with the current way-of-working. We estimate that on every development stream the annual inefficiencies of applying Rhapsody is more than 50 hours. We have one stream for new development and two streams for maintenance, leading to an inefficiency of at least 150 hours per year. A product in the field needs to be supported for more than 10 years. This leads to a total of 1500 hours of inefficiency.

  ROI = (gain from investment − cost of investment) / cost of investment = (1500 - 100) / 100 = 14. This positive ROI value indicates that transforming the legacy implementation is preferred above keeping the current Rhapsody models.

The approach was applied on one software component. In the future, we want to convert other components that use Rhapsody models. These components have three Rhapsody state machines. For these other components we think that we can generate the new implementations in 5 hours and integrate it into Visual Studio projects in 20 hours. By also converting these software components we can further improve our ROI.



This epilogue starts with an evaluation of the applied techniques. Subsequently, we describe our lessons learned for industrial engineers. We conclude the epilogue with a discussion about possible future work.

9.1 Evaluation of Criteria

In this thesis we investigated the application of techniques that were new for Philips to improve the evolution of interventional X-ray systems with legacy components. More specifically, we looked at Domain Specific Languages (DSLs). The applied techniques are Analytical Software Design (ASD), Parallel Object-Oriented Specification Language (POOSL), vertical DSLs, the combination of model learning and equivalence checking, and model transformation.

These techniques were applied on real industrial development projects and were evaluated with the criteria listed in Section 1.3: scalability, integration in an industrial context, Return On Investment (ROI), and improve system evolution. The following conclusions are based on our observations made while executing these projects. They might generalize, but we do not have evidence for a generalization. Note that the evaluation also includes tool aspects, because in order to apply a technique one or more tools have been used.

Scalability We evaluate the scalability of the techniques by checking whether a technique can be applied on industry-sized problems. We list our conclusions.

- The first technique we applied was ASD. Large applications can be created with ASD when exploiting the compositional approach using small components. ASD does not scale for large, complex components because the model checker might hit the state-space explosion problem, e.g., because of a large number of callbacks. For this reason, ASD requires a design with small components. Another way in which ASD prevents the state-space explosion

is that ASD checks for a limited set of properties. Using this approach in the project described in Chapter 3, we never encountered the state-space explosion problem.

- A number of language constructs make that POOSL is scalable, e.g. object-orientation and the import construct. Simulation of POOSL models was used because exhaustive model checking, e.g. with mCRL2, of the full model was not feasible, given the large number of concurrent processes and the use of queues for asynchronous communication.

- We applied the vertical DSL approach on two projects. When instances of a DSL become too large, it might be possible to raise the abstraction level by creating a more abstract language that generates instances of the prior DSL. In one of the projects we created a second more abstract DSL. Another way to improve scalability is to work modular with imports. In this way, instances are spread over multiple files. Using imports also enables the ability to reuse partial instances. In addition, a large language can be split into multiple smaller languages. This approach has been used in the ComMA framework for interface specifications. Hence, an usage tree of small and simple languages can be created. In the projects described in this thesis, we did not encounter scalability issues.

- The fourth technique applied is a combination of model learning with LearnLib and equivalence checking with mCRL2. For this combination of tools, we expect to encounter scalability issues with LearnLib before we hit the state space explosion in model checking with mCRL2. The number of queries needed to learn a System Under Learning (SUL) with the L^* algorithm depends on the number of possible inputs and the number of states. In Section 7.6 we listed a number of recent research results that decrease the number of stimuli needed to learn an implementation. In addition the time required to learn the implementation depends on the time required to query and reset the SUL. There were no scalability issues encountered in the project described in Chapter 7. The implementations are learned and checked iteratively with increasing sets of stimuli to handle scalability. Although this was a reasonable small implementation, learning took a substantial amount of time.

- In Chapter 8, we created a horizontal DSL for the transformations from Rhapsody model instances to Dezyne model instances. More specifically, we transformed the state machines defined in Rhapsody models. Regarding scalability, the transformations from one state machine model into another model did not increase the number of states and the transformations are executed in less than a second. In addition, we applied the model learning technique of the previous point to gain additional confidence in the transformations. Because the implementations (SULs) responded in a few milliseconds on queries and could be reset in milliseconds, learning was faster than in the project of Chapter 7.

Summarizing, our experience is that model learning takes a substantial amount of time. Scalability issues can be expected for learning large implementations or

implementations for which resetting and querying takes a significant amount of time.

Integration in an industrial context Applying new techniques in industry means that they need to be integrated within the industrial context.

- Regarding ASD, the company that creates the ASD:Suite provides two 2-day courses on ASD to get started and to master more advanced aspects. Our experience is that these courses are a good introduction to the tool. However, they do not provide the information needed to create a design that is verifiable. The company also provides commercial support for modelling and usage of the ASD:Suite.

  The use of ASD has a clear impact on the design and the definition of components. Because formal verification and code generation is only possible for control components, the design has to make a clear separation between data and control. Especially for designers familiar with object-oriented design, ASD requires a paradigm shift [126].

- POOSL comes with example models and a website explains the language constructs. Our experience is that these two sources provide enough information to quickly apply the tooling. The tooling is not commercially supported.

  To apply POOSL in an industrial context we presented an adaptation of the concept phase of the industrial development process to model new features.

- Vertical and horizontal DSLs have been created based on a manual [104]. 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. The tooling is commercially supported by Itemis [81] and commercially trainings are provided by TypeFox [152].

  Our experience is that the DSLs we developed are simple and easy to use. The main benefit of the DSLs is that they provide a way to solve industrial problems in less time than with the current way of working, i.e., using a commercial tool for configuring a fieldbus as described in Chapter 5. Validation rules can be added to check some correctness properties of language instances. The main disadvantage of applying the techniques we used is that the generators of the DSLs are programmed in Xtend and/or Java while C++ is the programming language that is used at Philips. The switch in programming language will create a barrier for some software engineers although the generator only needs to be supported by a few software engineers. There will be more users for the language than there are software engineers that need to maintain the language.

- Setting up LearnLib can be done by following a Wiki. For mCRL2 many examples and tutorials are available, see for instance [65]. Both tools are not commercially supported.

All techniques we applied could be integrated in our industrial context.

Return On Investment (ROI) Introducing a new technique in a company needs to add value. ROI is operational income divided by assets invested [110]. ROI is used to evaluate objectively the cost compared by the potential gain of introducing a new technique. Note that to simplify our calculations we use person hours instead of income in monetary terms, because the ROI does not change if we multiple the person hours with a fixed hourly rate.

- The first technique we applied is ASD. In the 17,000 lines of source code (LOC) produced in the project where we applied ASD, we found only 1 defect. Hence, the entire project exhibited only 0.17 defect per KLOC. This level of quality is much better than the industry standard defect rate of 1-25 defects per KLOC [98]. When this project would have been done with the normal way of working, the number of defects would have been at least 17 assuming 1 defect per KLOC. Based on historical data at Philips, solving a single error takes on average 8 hours, which leads to 8 times 17 is 136 hours. The average productivity of our project was 5.8 effective lines of code per hour. The industry standard is 1-2 effective lines of code (ELOC) per hour, including all non-coding overhead [98]. Next we calculate how many hours it would have taken to create a similar component from the same size in terms of ELOC. To be on the safe side we use 2 ELOC in our calculations. Hence, without applying ASD the 1787 hours (see Section 3.7) shall be multiplied with a factor 5.8 ELOC / 2 ELOC = 2.9 leads to 2.9 * 1787 hours = 5182 hours. The gain from the investment is 5182 plus 136 (for solving defects) is 5318 and the cost of the investment is 1787 plus 8 is 1795. ROI = (gain from investment – cost of investment) / cost of investment = (5318 - 1795) / 1795 = 2.

- POOSL is the second technique we applied. The modelling approach required a relatively small investment. The POOSL models and the Java simulator were made in 50 hours. While modelling, we found several issues that were not foreseen initially. We had to address issues that would otherwise have been postponed to the implementation phase and which might easily lead to integration problems. We observed that the definition of a formal executable model required a number of design choices that in the standard way of working would not have been made in the concept phase but in a later phase. The investment it took to make the POOSL models can be quantified (50 hours). We cannot quantify the savings we made by preventing issues in later phases. Hence, we cannot quantify the ROI for this technique.

- Vertical DSLs were applied in two projects. For the fieldbus configuration project described in Chapter 5, we calculated the ROI in Section 5.6. ROI = (16000 - 100) / 100 = 159.

  In Chapter 6 we described the PDU project. We use the hours presented in Section 6.6.3 in the ROI calculation. The costs are 45 hours plus 3 times 8 hours is 69 hours. The current way of working would require 3 times 60 hours is 180 hours. ROI = (180 - 69) / 69 = 1.6. In this calculation we only included the work to make all the artefacts for a new release, not

the prevented issues with the new way of working because this is hard to quantify.

In [150], over 20 industrial cases are analyzed and they conclude that productivity is improved by the higher level of abstraction of DSLs.

- The fourth technique applied is a combination of model learning with LearnLib and equivalence checking using the mCRL2 toolkit. Because we do not known when the defects found by applying these techniques would have been found otherwise, we cannot calculate a ROI value for applying these techniques based on this project.

- The ROI for the presented horizontal DSL applied for the transformation of legacy Rhapsody models described in Section 8.5 is (1500 - 100) / 100 = 14.

The techniques for which we could perform a calculation leads to a ROI greater than 1 which indicates that the projects benefited from the technique.

Improve system evolution We evaluate the contribution of the techniques to improve the evolution of interventional X-ray systems.

- A common activity in system evolution is replacing a legacy hardware component of the current system by a new hardware component for a new system release. In this context, we applied ASD for making a new software component that interfaces with the new hardware component. In the described case we improved on productivity in terms of lines of code per hour and we improved on the number of defects per lines of code.

  Our observations are in line with other reported projects at Philips about the application of ASD [114]. In addition, [114] refers to many industrial projects about the application of model based techniques and concludes that productivity and quality improved when applying these techniques.

- When evolving a system, new concepts need to be added. To explore new system concepts we modelled the existing system and added the new concepts for evaluation. By modelling the new concepts in the first phase of product development questions were raised early and costly rework at later phases was avoided. Hence, by modelling the system concepts with POOSL we could improve system evolution for this specific case.

- Another aspect of system evolution is to deal with legacy components. We considered components that need to be instrumented by configuration files. We applied DSLs to improve the maintainability and extensibility of two legacy software components. From languages that expresses domain concepts, we can generate the required configuration files. Before we generate configuration files, the language instances are checked to acquire confidence that they describe the intended behaviour. With DSLs we could keep these two components longer in the evolution stage. This is consistent with was was reported in e.g. [167, 13] about improved maintainability using a DSL.

- As part of system evolution, a legacy software component should be replaced by a new component, e.g., to support future extensions. When replacing a component it is important that its behaviour is exactly the same as its predecessor to ensure that other interfacing system components are unaffected. We proposed a combination of model learning and equivalence checking to acquire confidence in the refactored component. We applied this combination of techniques in two projects. In both projects we found issues in an early stage that otherwise would be found at a later stage or could potentially become a field issue.

- Another aspect is the usage of obsolete tools. In Chapter 8, we have created a DSL that accepts model instances of an obsolete modelling tool. From a DSL instance, a new implementation could be generated. In this way, we automatically replaced the implementation of a component. This avoids the need to make a new implementation from scratch.

The applied techniques had a positive contribution in the considered system evolution cases on which we applied the techniques.

9.2 Lessons Learned

In this section, we list a number of lessons learned. Goal of this section is to provide guidance to industrial engineers about the application of the presented techniques.

- When using ASD or its successor Dezyne, make a design with small components to avoid hitting the state-space explosion problem. Using small components has the additional advantage that when modifications are required planning effort is more predictable and modifying many small components is less difficult than changing one large component.

  ASD checks interface conformance of components, but it does not verify the functionality of an application. ASD does not make the use of testing obsolete. On the other hand, ASD can detect race-conditions that are difficult to find using a regular test approach [69].

- Our experience is that creating a DSL can be beneficial, if the following can be accomplished.

  - Raise abstraction by hiding lower level details; an example of this has been presented in Chapter 5.
  - Repetition in the target artefacts in terms of copy-paste or some pattern that is used over and over again; see for instance Chapters 5 and 6.
  - Target artefacts are poorly readable; as in Chapter 6.

  The investment of making small DSLs was earned back on first or second usage. Hence, even if the DSL is no longer used in the near future, the investment had a positive effect. Also [170] reports that the most successful applications of model-driven development use small DSLs.

- For the combination of model learning and equivalence checking one needs to note that creating an adapter can be time consuming as well as the time the model learner needs to learn a software component. If the adapter needs to observe an asynchronous event, this can only be done by letting the adapter wait before sending the output to the model learner. Hence, the time needed for learning depends also on the response time of the SUL. Note that learned models may be incorrect.

In general, when considering the application of one of the techniques described in this thesis, one need to consider if learning and applying the technique is an investment that can be earned back. Another consideration is that applying new tools creates tomorrows legacy. In case of a DSL, a new generator can be created to migrate to a new technique when required in the future.

9.3 Future Work

The results of this thesis have been acquired using an action research approach applied in industry. Taking the industry-as-lab approach has some limitations. We applied a number of techniques in a limited number of industrial projects. For every project it was only feasible to apply a single technique. Because we only applied one technique per project, we did not compare the techniques with each other on the same project. Moreover, we applied every technique on only one or two projects. So we cannot generalize our findings for the application of these techniques for other projects.

In this thesis we have described the usage of some techniques to improve system evolution. To get a better view on how these techniques can help the high-tech industry, future work could improve this view by:

- applying the techniques described in this thesis in a different industrial setting, and
- applying different techniques for cases comparable with the cases described in this thesis.

A ROI calculation is a way to compare multiple techniques and helps with making a choice which technique to use [110]. The higher the ROI value, the higher the gain. We applied different techniques on different projects. For this reason, we cannot compare the ROI values of the techniques. By performing additional research we could, for instance, compare the ROI when applying different techniques for comparable cases. In addition we could compare the usage of a single technique in different industrial settings.

This thesis presented the usage of new techniques for system evolution with the focus on legacy components. The applied techniques helped improving system evolution for the project they were applied in. However, the new techniques today are tomorrows legacy. Future work could address how to maintain DSLs and how to create maintainable DSLs.

In future work new model learning algorithms such as TTT [80] could be applied to reduce the number of queries and resets needed, and by doing so less time

is needed to learn an implementation. TTT and future algorithms could improve the scalability of model learning for industrial sized applications.

ASD checks a predefined limited set of properties. Checking functional verification would be a valuable addition. The challenge is to make these techniques applicable by an average industrial engineer. Such a new technique should for instance hide the details of, e.g., LTL and CTL like model checking techniques.

In this thesis we applied a number of DSLs, each of which stores instances of the language in a different format to file. For example, ASD stores instances to a XML file, Rhapsody has its own proprietary format, and POOSL, Dezyne and the vertical DSLs store instances one-to-one to file. At Philips, there are multiple (maintenance) streams in our archive and our experience is that merging ASD and Rhapsody instances is cumbersome and error prone. Even though these techniques come with tools to automate merging, manual merging is the preferred approach. We choose to create new textual DSLs instead of graphical DSLs for these reasons. If graphical DSLs could write an instance to a file in an easily manually mergeable format, then these graphical DSLs would be worth considering to use in industry.

In Chapter 8, ComMA was introduced and we described the extension of the ComMA framework with a generator for the Dezyne tool. In the future, we want to extend ComMA further by adding generators for model-checking and model-based testing. In addition we want to create new DSLs for the generation of ComMA models from other technologies than Rhapsody.

BIBLIOGRAPHY

[1] Jenkins. www.jenkins-ci.org/, 2015. Cited on page 84.

[2] LonWorks. www.echelon.com/technology/lonworks/, 2015. Cited on page 73.

[3] Meta programming system (MPS). www.jetbrains.com/mps, 2015. Cited on page 14.

[4] PlantUML. plantuml.sourceforge.net/, 2015. Cited on page 83.

[5] VisualState. www.iar.com/Products/IAR-visualSTATE/, 2015. Cited on page 74.

[6] F. Aarts, B. Jonsson, J. Uijen, and F. Vaandrager. Generating models of infinite-state communication protocols using regular inference with abstraction. *Formal Methods in System Design*, 46(1):1–41, 2015. Cited on page 105.

[7] F. Aarts, H. Kuppens, G. Tretmans, F. Vaandrager, and S. Verwer. Improving active Mealy machine learning for protocol conformance testing. *Machine Learning*, 96(1–2):189–224, 2014. Cited on pages 16 and 95.

[8] J.-R. Abrial. *The B-book: assigning programs to meanings.* Cambridge University Press, 2005. Cited on pages 13 and 15.

[9] R. Akers, I. Baxter, M. Mehlich, B. Ellis, and K. Luecke. Case study: Re-engineering C++ component models via automatic program transformation. *Information and Software Technology*, 49(3):275 – 291, 2007. Cited on page 16.

[10] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2), 1987. Cited on pages 16 and 104.

[11] J. Ansel, K. Arya, and G. Cooperman. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *IEEE Parallel and Distributed Processing Symposium*, 2009. Cited on page 105.

[12] B. Basten, J. van den Bos, M. Hills, P. Klint, A. Lankamp, B. Lisser, A. van der Ploeg, T. van der Storm, and J. Vinju. Modular language implementation in rascal–experience report. *Science of Computer Programming*, 114:7–19, 2015. Cited on page 15.

[13] D. Batory, C. Johnson, B. MacDonald, and D. Von Heeder. Achieving extensibility through product-lines and domain-specific languages: A case study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):191–214, 2002. Cited on pages 14 and 125.

[14] K. Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003. Cited on page 30.

[15] D. Bellagio and T. Milligan. *Software configuration management strategies and ibm® rational® clearcase®: a practical introduction*. IBM Press, 2005. Cited on page 42.

[16] K. Bennett and V. Rajlich. Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 73–87. ACM, 2000. Cited on page 1.

[17] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wąsowski. A survey of variability modeling in industrial practice. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, page 7. ACM, 2013. Cited on page 15.

[18] L. Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd, 2013. Cited on pages 6, 66, and 72.

[19] J. C. Bicarregui, J. S. Fitzgerald, P. G. Larsen, and J. Woodcock. Industrial practice in formal methods: A review. In *International Symposium on Formal Methods*, pages 810–813. Springer, 2009. Cited on page 19.

[20] J.-P. Bodeveix, M. Filali, J. Lawall, and G. Muller. Formal methods meet domain specific languages. In *Integrated Formal Methods*, volume 3771 of *LNCS*, pages 187–206. Springer, 2005. Cited on page 15.

[21] B. Boehm and V. Basili. Software defect reduction top 10 list. *IEEE Computer*, 34(1):135–137, 2001. Cited on page 48.

[22] L. Bokhoven. *Constructive tool design for formal languages: from semantics to executing models*. PhD thesis, Technische Universiteit Eindhoven, 2002. Cited on pages 5 and 50.

[23] G. Booch, J. Rumbaugh, and I. Jacobson. *The unified modeling language user guide - the ultimate tutorial to the UML from the original designers*. Addison-Wesley object technology series. Addison-Wesley-Longman, 1999. Cited on page 22.

[24] E. Börger and J. Huggins. Abstract state machines 1988–1998: Commented asm bibliography. In *Bulletin of EATCS*. Citeseer, 1998. Cited on page 13.

[25] E. Börger, P. Päppinghaus, and J. Schmid. Report on a practical application of asms in software design. In *International Workshop on Abstract State Machines*, pages 361–366. Springer, 2000. Cited on page 13.

[26] H. Breivold, I. Crnkovic, and M. Larsson. A systematic review of software architecture evolution research. *Information and Software Technology*, 54(1):16–40, 2012. Cited on page 2.

[27] E. Brinksma and J. Hooman. *Dependability for high-tech systems: an industry-as-laboratory approach*. IEEE, 2008. Cited on page 1.

[28] G. Broadfoot and P. Hopcroft. Introducing formal methods into industry using cleanroom and csp. *Dedicated Systems Magazine Q*, 1:2005, 2005. Cited on page 13.

[29] S. Brookes, C. Hoare, and A. Roscoe. A theory of communicating sequential processes. *Journal of the ACM (JACM)*, 31(3):560–599, 1984. Cited on pages 5 and 19.

[30] V. Camelo. *Multi-core CPU exploration for CARM host in ASML technology*. Master thesis, Eindhoven University of Technology, 2012. Cited on page 14.

[31] L. Cao, B. Ramesh, and M. Rossi. Are domain-specific models easier to maintain than uml models? *IEEE software*, 26(4):19–21, 2009. Cited on page 15.

[32] S. Cassel. *Learning Component Behavior from Tests: Theory and Algorithms for Automata with Data*. PhD thesis, University of Uppsala, 2015. Cited on page 105.

[33] 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. Cited on page 14.

[34] R. Chapman and F. Schanda. Are we there yet? 20 years of industrial theorem proving with spark. In *International Conference on Interactive Theorem Proving*, pages 17–26. Springer, 2014. Cited on page 13.

[35] B. Cheng, S. Easterbrook, R. France, and B. Rumpe. Integrating formal and informal specification techniques. why? how? 1998. Cited on page 48.

[36] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187 – 243, 2002. Cited on page 14.

[37] A. ClearSy. Industrial tool supporting the b method, 2012. Cited on page 13.

[38] 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)*, pages 199–213. Springer, 2013. Cited on pages 6, 95, and 99.

[39] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006. Cited on page 16.

[40] A. David, J. Deneux, and J. d'Orso. A formal semantics for UML statecharts. Technical Report 2003-010, Uppsala University, 2003. Cited on page 110.

[41] R. Davison, M. Martinsons, and N. Kock. Principles of canonical action research. *Information systems journal*, 14(1):65–86, 2004. Cited on page 4.

[42] G. De Geest, A. Savelkoul, and A. Alikoski. Building a framework to support domain-specific language evolution using microsoft dsl tools. In *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modelling*, 2007. Cited on page 4.

[43] J. de Ruiter and E. Poll. Protocol state fuzzing of tls implementations. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 193–206. USENIX Association, 2015. Cited on page 16.

[44] R. Dorofeeva, K. El-Fakih, S. Maag, A. Cavalli, and N. Yevtushenko. FSM-based conformance testing methods: A survey annotated with experimental evaluation. *Information & Software Technology*, 52(12):1286–1297, 2010. Cited on page 104.

[45] S. Easterbrook, R. Lutz, R. Covington, J. Kelly, Y. Ampo, and D. Hamilton. Experiences using lightweight formal methods for requirements modeling. *IEEE Trans. Software Eng.*, 24(1):4–14, 1998. Cited on page 14.

[46] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian. Selecting empirical methods for software engineering research. In *Guide to advanced empirical software engineering*, pages 285–311. Springer, 2008. Cited on page 4.

[47] J. Ellson, E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull. Graphviz–open source graph drawing tools. In *International Symposium on Graph Drawing*, pages 483–484. Springer, 2001. Cited on page 6.

[48] J. Engelfriet. Determinacy - (observation equivalence = trace equivalence). *Theoretical Computer Science*, 36:21–25, 1985. Cited on page 100.

[49] S. Erdweg, S. Fehrenbach, and K. Ostermann. Evolution of software systems with extensible languages and dsls. *IEEE Software*, 31(5):68–75, 2014. Cited on page 15.

[50] R. Eshuis. Reconciling statechart semantics. *Sci. Comput. Program.*, 74(3):65–99, Jan. 2009. Cited on page 110.

[51] Esterel Technologies. *SCADE Suite*, 2011. Model based development environment dedicated to critical embedded software, `www.esterel-technologies.com/products/scade-suite/`. Cited on page 13.

[52] S. Fehrenbach, S. Erdweg, and K. Ostermann. Software evolution to domain-specific languages. In *International Conference on Software Language Engineering*, pages 96–116. Springer, 2013. Cited on page 15.

[53] L. Feng, S. Lundmark, K. Meinke, F. Niu, M. Sindhu, and P. Wong. Case studies in learning-based testing. In H. Yenigün, C. Yilmaz, and A. Ulrich, editors, *ICTSS 2013*, LNCS, vol. 8254, pages 164–179. Springer, Heidelberg, 2013. Cited on page 16.

[54] P. Fiterău-Broştean, R. Janssen, and F. Vaandrager. Learning fragments of the TCP network protocol. In F. Lang and F. Flammini, editors, *FMICS 2014*, LNCS, vol. 8718, pages 78–93. Springer, Heidelberg, 2014. Cited on page 16.

[55] J. Fitzgerald, P. G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef. *Validated Designs for Object-oriented Systems*. Springer, New York, 2005. Examples are available at www.vdmbook.com. Cited on page 13.

[56] Formal Systems (Europe) Ltd. *FDR2 model checker*, 2011. www.fsel.com/. Cited on pages 21 and 27.

[57] M. Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 2010. Cited on page 4.

[58] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2011: a toolbox for the construction and analysis of distributed processes. *International Journal on Software Tools for Technology Transfer*, 15(2):89–107, 2013. Cited on page 53.

[59] M. Geilen. Formal techniques for verification of complex real-time systems. Phd thesis, Eindhoven University of Technology, the Netherlands, 2002. Cited on page 50.

[60] M. Goldsmith, B. Roscoe, and P. Armstrong. Failures-divergence refinement-fdr2 user manual, 2005. Cited on pages 5 and 19.

[61] A. Goodloe, C. Gunter, and M.-O. Stehr. Formal prototyping in early stages of protocol design. In *Proc. of the 2005 Workshop on Issues in the Theory of Security*, WITS '05, pages 67–80. ACM, 2005. Cited on page 14.

[62] D. Graham, E. Van Veenendaal, and I. Evans. *Foundations of software testing: ISTQB certification*. Cengage Learning EMEA, 2008. Cited on page 44.

[63] A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. *Logic Journal of IGPL*, 14(5):729–744, 2006. Cited on page 95.

[64] R. Grønmo and J. Oldevik. An empirical study of the uml model transformation tool (umt). *Proc. First Interoperability of Enterprise Software and Applications, Geneva, Switzerland*, 2005. Cited on page 16.

[65] J. Groote, J. Keiren, A. Mathijssen, B. Ploeger, F. Stappers, C. Tankink, Y. Usenko, M. v. Weerdenburg, W. Wesselink, T. Willemse, and J. v. d. Wulp. The mCRL2 toolset. In *Proceedings of the International Workshop on Advanced Software Development Tools and Techniques (WASDeTT 2008)*, 2008. Cited on pages 53 and 123.

[66] J. Groote, A. Osaiweran, M. Schuts, and J. Wesselius. Investigating the effects of designing industrial control software using push and poll strategies. Computer Science Report 11/16, Eindhoven University of Technology, the Netherlands, 2011. Cited on pages 11, 49, and 53.

[67] J. Groote, A. Osaiweran, and J. Wesselius. Analyzing the effects of formal methods on the development of industrial control software. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 467–472. IEEE, 2011. Cited on pages 20, 35, and 42.

[68] J. Groote, A. Osaiweran, and J. Wesselius. Experience report on developing the front-end client unit under the control of formal methods. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 1183–1190. ACM, 2012. Cited on pages 20 and 42.

[69] M. Grottke and K. Trivedi. Fighting bugs: Remove, retry, replicate, and rejuvenate. *Computer*, 40(2), 2007. Cited on page 126.

[70] T. Group et al. Tiobe index for ranking the popularity of programming languages, 2013. Cited on page 41.

[71] G. Hamon, L. de Moura, and J. Rushby. Automated test generation with SAL. CSL Technical Note, SRI International, January 2005. Cited on page 81.

[72] C. Heitmeyer. On the need for practical formal methods. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1486 of *LNCS*, pages 18–26. Springer, 1998. Cited on page 71.

[73] M. Henrix. *Performance improvement in automata learning*. Master thesis, Radboud University, Nijmegen, 2015. Cited on page 105.

[74] J. Hooman. *Specification and Compositional Verification of Real-Time Systems*, volume 558 of *LNCS*. Springer, 1991. Cited on page 21.

[75] J. Hooman, R. Huis in 't Veld, and M. Schuts. Experiences with a compositional model checker in the healthcare domain. In *Foundations of Health Information Engineering and Systems*, number 7151 in LNCS, pages 93–110. Springer–Verlag, 2012. Cited on page 9.

[76] F. Howar, M. Isberner, M. Merten, and B. Steffen. Learnlib tutorial: From finite automata to register interface programs. In T. Margaria, editor, *ISoLA 2012*, LNCS, vol. 7609, pages 587–590. Springer, Heidelberg, 2012. Cited on pages 6 and 95.

[77] IBM. Rational Rhapsody. `www.ibm.com/software/products/en/ratirhapfami`, 2015. Cited on pages 14 and 107.

[78] J. Ichbiah. *Rationale for the design of the Ada programming language*. Cambridge University Press, 1991. Cited on page 13.

[79] Intel. Intelligent Platform Management Interface (IPMI) - specifications. `www.intel.com/content/www/us/en/servers/ipmi/ipmi-specifications.html`, 2015. Cited on page 52.

[80] M. Isberner. *Foundations of Active Automata Learning: An Algorithmic Perspective*. PhD thesis, Technical University of Dortmund, 2015. Cited on pages 104 and 127.

[81] Itemis. *Xtext*, 2016. `xtext.itemis.com`. Cited on page 123.

[82] 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. Cited on page 15.

[83] C. Jones, D. Jackson, and J. Wing. Formal methods light. *Computer*, 29(4):20–22, 1996. Cited on page 71.

[84] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. Atl: A model transformation tool. *Science of computer programming*, 72(1):31–39, 2008. Cited on page 17.

[85] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez. Atl: a qvt-like transformation language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 719–720. ACM, 2006. Cited on page 17.

[86] A. Kalsing, G. do Nascimento, C. Iochpe, and L. Thom. An incremental process mining approach to extract knowledge from legacy systems. In *Enterprise Distributed Object Computing Conference (EDOC)*, pages 79–88, 2010. Cited on page 16.

[87] 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. Cited on page 14.

[88] M. Kaufmann, J. Moore, and P. Manolios. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000. Cited on page 14.

[89] D. Kolovos, L. Rose, N. Matragkas, R. Paige, E. Guerra, J. Cuadrado, J. De Lara, I. Ráth, D. Varró, M. Tisi, et al. A research roadmap towards achieving scalability in model driven engineering. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*, page 2. ACM, 2013. Cited on page 3.

[90] S. Koo, H. Son, and P. Seong. Nusee: Nuclear software engineering environment. In *Reliability and Risk Issues in Large Scale Safety-critical Digital Control Systems*, Springer Series in Reliability Engineering, pages 121–135. Springer London, 2009. Cited on page 47.

[91] I. Kurtev, M. Schuts, J. Hooman, and D.-J. Swagerman. Integrating interface modeling and analysis in an industrial setting. In *Proceedings 5th International Conference on Model-Driven Engineering and Software Development*, pages 345–352, 2017. Cited on pages 10 and 108.

[92] P. Larsen, J. Fitzgerald, and T. Brookes. Applying formal specification in industry. *IEEE software*, 13(3):48–56, 1996. Cited on page 48.

[93] B. Lientz and E. Swanson. *Software Maintenance Management.* Addison Wesley, 1980. Cited on page 1.

[94] R. C. Linger. Cleanroom process model. *IEEE Software*, 11(2):50, 1994. Cited on page 13.

[95] MagicDraw. Cameo simulation toolkit. `www.nomagic.com/products/magicdraw-addons/cameo-simulation-toolkit.html`, 2015. Cited on page 14.

[96] T. Margaria, O. Niese, H. Raffelt, and B. Steffen. Efficient test-based model generation for legacy reactive systems. In *9th IEEE Int. High-Level Design Validation and Test Workshop*, pages 95–100, 2004. Cited on page 16.

[97] MathWorks. Matlab and Simulink. `www.mathworks.com`, 2015. Cited on page 14.

[98] S. McConnell. *Code complete.* Pearson Education, 2004. Cited on pages 44 and 124.

[99] T. Mens and P. Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006. Cited on page 16.

[100] B. Meyer. Applying “design by contract”. *IEEE Computer*, 25(10):40–51, 1992. Cited on page 22.

[101] G. Michaelson. Are there domain specific languages? In *Proceedings of the 1st International Workshop on Real World Domain Specific Languages*, page 1. ACM, 2016. Cited on page 4.

[102] R. Milner. *A Calculus of Communicating Systems.* Springer-Verlag, 1980. Cited on page 50.

[103] A. Mooij, G. Eggen, J. Hooman, and H. van Wezep. Cost-effective industrial software rejuvenation using domain-specific models. In *International Conference on Theory and Practice of Model Transformations*, pages 66–81. Springer, 2015. Cited on page 17.

[104] A. Mooij and J. Hooman. Creating a Domain Specific Language (DSL) with Xtext. `www.cs.ru.nl/J.Hooman/DSL/`, 2015. Cited on pages 72 and 123.

[105] A. Mooij, J. Hooman, and R. Albers. Early fault detection using design models for collision prevention in medical equipment. In *International Symposium on Foundations of Health Informatics Engineering and Systems*, pages 170–187. Springer, 2013. Cited on page 15.

[106] A. Mooij, J. Hooman, and R. Albers. Gaining industrial confidence for the introduction of domain-specific languages. In *Computer Software and Applications Conference Workshops (COMPSACW), 2013 IEEE 37th Annual*, pages 662–667. IEEE, 2013. Cited on page 15.

[107] A. Mooij, M. Joy, G. Eggen, P. Janson, and A. Rădulescu. Industrial software rejuvenation using open-source parsers. In *International Conference on Theory and Practice of Model Transformations*, pages 157–172. Springer, 2016. Cited on page 17.

[108] P. Mosses, editor. *CASL Reference Manual*, volume 2960 of *LNCS*. Springer, 2004. Cited on page 15.

[109] 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. Cited on page 14.

[110] B. Needles, M. Powers, and S. Crosson. *Principles of accounting*. Cengage Learning, 2013. Cited on pages 3, 124, and 127.

[111] OMG. Semantics of a foundational subset for executable UML models (fUML). `www.omg.org/spec/FUML/`, 2015. Cited on page 14.

[112] A. Osaiweran. *Formal development of control software in the medical systems domain*. PhD thesis, Eindhoven University of Technology, 2012. Cited on page 9.

[113] A. Osaiweran, M. Schuts, and J. Hooman. Incorporating formal techniques into industrial practice. *Empirical Software Engineering*, 19:1169–1194, 2014. Cited on page 9.

[114] A. Osaiweran, M. Schuts, J. Hooman, J. Groote, and B. van Rijnsoever. Evaluating the effect of a lightweight formal technique in industry. *International Journal on Software Tools for Technology Transfer*, 18(1):93–108, 2016. Cited on pages 11 and 125.

[115] A. Osaiweran, M. Schuts, J. Hooman, and J. Wesselius. Incorporating formal techniques into industrial practice: an experience report. In *Proceedings 9th International Workshop on Formal Engineering approaches to Software Components and Architectures*, volume 295 of *ENTCS*, pages 49–63, 2013. Cited on page 9.

[116] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: prolegomena to the design of pvs. *Software Engineering, IEEE Transactions on*, 21(2):107–125, Feb 1995. Cited on page 14.

[117] G. Palshikar. Applying formal specifications to real-world software development. *IEEE Software*, 18(6):89–97, 2001. Cited on page 48.

[118] R. Pérez-Castillo, I. De Guzman, and M. Piattini. Knowledge discovery metamodel-iso/iec 19506: A standard to modernize legacy systems. *Computer Standards & Interfaces*, 33(6):519–532, 2011. Cited on page 16.

[119] C. Potts. Software-engineering research revisited. *IEEE software*, 10(5):19–28, 1993. Cited on page 3.

[120] S. Prowell and J. Poore. Foundations of sequence-based software specification. *IEEE Transactions on Software Engineering*, 29:417–429, 2003. Cited on page 21.

[121] S. J. Prowell, C. J. Trammell, R. C. Linger, and J. H. Poore. *Cleanroom software engineering: technology and process*. Pearson Education, 1999. Cited on page 13.

[122] H. Raffelt, B. Steffen, T. Berg, and T. Margaria. LearnLib: a framework for extrapolating behavioral models. *STTT*, 11(5):393–407, 2009. Cited on page 98.

[123] A. Roscoe. *The theory and practice of concurrency*. Prentice Hall, 1998. Cited on page 27.

[124] M. Rosen, B. Lublinsky, K. Smith, and M. Balcer. *Applied SOA: service-oriented architecture and design strategies*. John Wiley & Sons, 2012. Cited on page 4.

[125] J. Schmaltz and D. Borrione. A functional approach to the formal specification of networks on chip. In *Formal Methods in Computer-Aided Design*, number 3312 in LNCS, pages 52–66. Springer–Verlag, 2004. Cited on page 14.

[126] M. Schuts. *Improving software development: The introduction and implementation of ASD at Philips Healthcare*. Master thesis, 2010. Cited on pages 9, 35, and 123.

[127] M. Schuts and J. Hooman. Formal modelling in the concept phase of product development. In *Software Engineering Research & Practice*, WORLDCOMP'15, pages 3–9. CSREA Press, 2015. Cited on page 10.

[128] M. Schuts and J. Hooman. Formalizing the concept phase of product development. In *Formal Methods*, number 9109 in LNCS, pages 605–608. Springer–Verlag, 2015. Cited on page 10.

[129] M. Schuts and J. Hooman. Using domain specific languages to improve the development of a power control unit. In *Proceedings 2015 Federated Conference on Computer Science and Information Systems*, volume 5 of *Annals of Computer Science and Information Systems*, pages 781–788. IEEE, 2015. Cited on page 10.

[130] M. Schuts and J. Hooman. Improving maintenance by creating a dsl for configuring a fieldbus. In *Proceedings of the International Workshop on Domain-Specific Modeling*, DSM 2016, pages 28–34. ACM, 2016. Cited on page 10.

[131] M. Schuts and J. Hooman. Industrial application of domain specific languages combined with formal techniques. In *Proceedings Workshop on Real World Domain Specific Languages*, pages 2:1–2:8. ACM, 2016. Cited on page 10.

[132] M. Schuts, J. Hooman, and F. Vaandrager. Refactoring of legacy software using model learning and equivalence checking: an industrial experience report. In *integrated Formal Methods*, number 9681 in LNCS, pages 311–325. Springer–Verlag, 2016. Cited on page 10.

[133] M. Schuts, F. Zhu, F. Heidarian, and F. Vaandrager. Modelling clock synchronization in the chess gmac wsn protocol. In *Proceedings 1st Workshop on Quantitative Formal Methods (QFM 2009)*, volume 13 of *EPTCS*, pages 41–54. Cited on page 11.

[134] G. Selim, S. Wang, J. Cordy, and J. Dingel. Model transformations for migrating legacy models: an industrial case study. In *European Conference on Modelling Foundations and Applications*, pages 90–101. Springer, 2012. Cited on page 17.

[135] G. Selim, S. Wang, J. Cordy, and J. Dingel. Model transformations for migrating legacy deployment models in the automotive industry. *Software & Systems Modeling*, 14(1):365–381, 2015. Cited on page 17.

[136] S. Sendall and W. Kozaczynski. Model transformation the heart and soul of model-driven software development. Technical report, 2003. Cited on page 16.

[137] N. Shankar. Combining theorem proving and model checking through symbolic analysis. In *CONCUR'00: Concurrency Theory*, number 1877 in LNCS, pages 1–16. Springer, 2000. Cited on page 81.

[138] N. Shankar. Symbolic analysis of transition systems. In *Abstract State Machines: Theory and Applications (ASM 2000)*, volume 1912 of *LNCS*, pages 287–302. Springer, 2000. Cited on pages 6 and 72.

[139] SHE. System-level design with the SHE methodology. `www.es.ele.tue.nl/she/`, 2015. Cited on page 50.

[140] W. Smeenk. *Applying Automata Learning to Complex Industrial Software*. Master thesis, Radboud University, Nijmegen, Sept. 2012. Cited on page 105.

[141] W. Smeenk, J. Moerman, F. Vaandrager, and D. Jansen. Applying automata learning to embedded control software. In M. Butler, S. Conchon, and F. Zaïdi, editors, *ICFEM 2015*, LNCS, vol. 9407, pages 67–83. Springer, Heidelberg, 2015. Cited on pages 16, 103, and 104.

[142] S. Sobernig, M. Strembeck, and A. Beck. Developing a domain-specific language for scheduling in the european energy sector. In *International Conference on Software Language Engineering*, pages 19–35. Springer, 2013. Cited on page 15.

[143] B. Steffen, F. Howar, and M. Merten. Introduction to active automata learning from a practical perspective. In M. Bernardo and V. Issarny, editors, *SFM 2011*, LNCS, vol. 6659, pages 256–296. Springer, Heidelberg, 2011. Cited on pages 16 and 98.

[144] A. Stellman and J. Greene. *Applied software project management.* " O'Reilly Media, Inc.", 2005. Cited on page 31.

[145] J. Stoel, T. v. d. Storm, J. Vinju, and J. Bosman. Solving the bank with rebel: on the design of the rebel specification language and its application inside a bank. In *Proceedings of the 1st Industry Track on Software Language Engineering*, pages 13–20. ACM, 2016. Cited on page 15.

[146] F. Systems. Failures-Divergences Refinement (FDR). `www.fsel.com`, 2015. Cited on page 53.

[147] B. Theelen, O. Florescu, M. Geilen, J. Huang, P. van der Putten, and J. P. Voeten. Software/hardware engineering with the parallel object-oriented specification language. In *Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign*, pages 139–148. IEEE Computer Society, 2007. Cited on pages 5 and 50.

[148] 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*, pages 139–148. IEEE, 2007. Cited on page 72.

[149] U. Tikhonova, M. Manders, M. van den Brand, S. Andova, and T. Verhoeff. Applying model transformation and event-b for specifying an industrial dsl. In *MoDeVVa@ MoDELS*, pages 41–50, 2013. Cited on page 17.

[150] J.-P. Tolvanen and S. Kelly. Defining domain-specific modeling languages to automate product derivation: Collected experiences. In *International Conference on Software Product Lines*, pages 198–209. Springer, 2005. Cited on pages 15 and 125.

[151] L. Tratt. The mt model transformation language. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 1296–1303. ACM, 2006. Cited on page 16.

[152] TypeFox. *DSL trainings*, 2016. `www.typefox.io/trainings-2`. Cited on page 123.

[153] F. Vaandrager. Model learning. *Communications of the ACM*, 60(2):86–95, 2017. Cited on page 6.

[154] W. van der Aalst. *Process Mining - Discovery, Conformance and Enhancement of Business Processes.* Springer-Verlag Berlin Heidelberg, 2011. Cited on page 16.

[155] A. Van Deursen and P. Klint. Little languages: little maintenance? *Journal of software maintenance*, 10(2):75–92, 1998. Cited on page 4.

[156] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000. Cited on page 4.

[157] A. Van Deursen, E. Visser, and J. Warmer. Model-driven software evolution: A research agenda. Technical report, Delft University of Technology, Software Engineering Research Group, 2007. Cited on page 16.

[158] H. Van Vliet, H. Van Vliet, and J. Van Vliet. *Software Engineering: Principles and Practice*. John Wiley & Sons, 2008. Cited on page 2.

[159] VDMTools. Industrial tool of CSK systems corporation supporting VDM++. www.vdmtools.jp/en, 2015. Cited on page 13.

[160] J. Verriet, R. Hamberg, J. Caarls, and B. van Wijngaarden. Warehouse simulation through model configuration. In *ECMS*, pages 629–635, 2013. Cited on page 14.

[161] Verum. *ASD:Suite*, 2011. `www.verum.com/`. Cited on pages 5 and 19.

[162] M. Voelter. *Generic Tools, Specific Languages*. PhD thesis, Delft University of Technology, 2014. Cited on page 14.

[163] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. Kats, E. Visser, and G. Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013. Cited on page 4.

[164] 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)*, pages 121–140. ACM, 2012. Cited on page 14.

[165] V. Vyatkin. Software engineering in industrial automation: State-of-the-art review. *IEEE Transactions on Industrial Informatics*, 9(3):1234–1249, 2013. Cited on page 1.

[166] C. Wagner. *Model-Driven Software Migration: A Methodology*. Springer Vieweg, 2014. Cited on pages 2 and 7.

[167] M. Ward. Language-oriented programming. *Software-Concepts and Tools*, 15(4):147–161, 1994. Cited on pages 14 and 125.

[168] I. Warren. *The renaissance of legacy systems: method support for software-system evolution*. Springer Science & Business Media, 2012. Cited on pages 2, 7, and 63.

[169] J. Westland. The cost of errors in software development: evidence from industry. *The Journal of Systems and Software*, 62:1–9, 2002. Cited on page 48.

[170] J. Whittle, J. Hutchinson, and M. Rouncefiled. The state of practice in model-driven engineering. In *IEEE Software*, pages 79–85. IEEE, 2014. Cited on pages 93 and 126.

[171] J. Wiegand et al. Eclipse: A platform for integrating development tools. *IBM Systems Journal*, 43(2):371–383, 2004. Cited on pages 6 and 66.

[172] J. Woodcock, P. Larsen, J. Bicarregui, and J. Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys*, 41(4):1–36, 2009. Cited on pages 13 and 19.

SAMENVATTING

Complexe systemen zoals vliegtuigen, auto's, en medische apparaten bestaan voor een groot gedeelte uit complexe software. Het aanpassen en uitbreiden van dergelijke apparaten is een uitdagende taak. Neem als voorbeeld de Image Guided Therapy (IGT) systemen van Philips. Deze systemen worden gebruikt voor minimaal invensieve operaties van, b.v., hart- en vaatziekten. De software die verantwoordelijk is voor de aansturing van de hardware van het systeem bestaat uit miljoenen regels broncode. Voor een nieuw systeem worden grote delen van eerder uitgebrachte systemen hergebruikt. Omdat de systemen van IGT al meer dan dertig jaar bestaan, is een groot gedeelte van de broncode jaren geleden geschreven.

Het onderhoud van complexe systemen wordt belemmerd doordat een groot gedeelte bestaat uit oude broncode. Vaak zijn de oorspronkelijke ontwikkelaars vertrokken. De documentatie beschrijft niet de huidige toestand van de broncode en er zijn onvoldoende testen om te garanderen dat er niets omvalt na het aanbrengen van wijzigingen in de broncode. Hierdoor kunnen wijzigingen gemakkelijk leiden tot fouten. Anderzijds zijn er voordurend vragen om aanpassingen te maken voor nieuwe toepassingen, zoals voor nieuwe medische procedures.

Om te onderzoeken of nieuwe technieken de onderhoudbaarheid van complexe systemen kunnen verbeteren hebben we een aantal van deze technieken toegepast in ontwikkelprojecten bij Philips. Bij alle technieken zijn Domain Specific Languages (DSLs) toegepast. Een DSL heeft een beperkte grammatica, zodat alleen domain concepten kunnen worden beschreven. Hierdoor ontstaan abstracte modellen die gemakkelijk zijn te lezen en te onderhouden. Vanuit een DSL kunnen verschillende artefacten automatisch worden gegenereerd, zoals simulatie modellen, formele analyse modellen, en broncode.

Een techniek die we bij IGT hebben toegepast is Analytical Software Design (ASD). ASD is gebruikt voor het ontwerpen en integreren van een nieuwe software component die nodig was voor de vervanging van een oude hardware component. ASD instanties kunnen formeel worden gecontroleerd en daarna kan broncode worden gegenereerd. Door het gebruik van ASD is het aantal fouten afgenomen tot slechts 0,17 fouten per 1000 regels broncode, terwijl de industriestandaard tussen 1 en 25 ligt. De ASD aanpak kan echter alleen worden toegepast voor kleine, data-onafhankelijke besturingscomponenten.

Omdat de formele controle van ASD niet kan worden toegepast voor grote, generieke componenten, hebben we geëxperimenteerd met de Parallel Object Oriented Specification Language (POOSL). Met deze taal is het ook mogelijk om tijd

en probabilistisch gedrag te beschrijven. POOSL modellen kunnen worden gesimuleerd voor de validatie van een ontwerp. We hebben POOSL toegepast om ontwerp ideeën te valideren voor de introductie van een nieuwe hardware component.

Recente ontwikkelingen in de DSL technologie maakt het mogelijk om snel een taal en de bijbehorende generatoren te creëren. We hebben een taal gemaakt voor het configureren van een oude component. Vanuit deze beknopte taal, die alleen de essentiële domain concepten beschrijft, kunnen we automatische grote configuratiebestanden genereren. De geldigheid van taal instanties wordt automatisch gecontroleerd. Na het genereren van 13 configuratiebestanden is de benodigde investering om de taal te maken al terugverdiend. In totaal verwachten we ongeveer 2000 files op deze manier te gaan genereren.

Deze aanpak is ook toegepast op een andere, oude component die numerieke, moeilijk te onderhouden configuratiebestanden gebruikt. We hebben een leesbare taal gemaakt van waaruit niet alleen de configuratiebestanden kunnen worden gegenereerd, maar ook analyse modellen. Met deze modellen kunnen we het beschreven gedrag formeel controleren en we kunnen test reeksen genereren. De gegenereerde testen zijn een instantie van een tweede test DSL. Vanuit de test DSL genereren we bestanden voor een testprogramma die de hardware en software van een component aftest. Ook hebben we generatoren toegevoegd voor de generatie van grafische plaatjes en testen om de analyse modellen te valideren. Daarnaast gebruiken we logbestanden die het gebruik van het systeem vastleggen. Deze logbestanden kunnen worden omgezet naar instanties van de test DSL. Op deze manier kunnen we checken of het gedrag van de analyse modellen overeenstemt met het gedrag van het systeem. Door deze kruisverbanden te bekijken krijgen we vertrouwen in de DSL instanties en de gebruikte generatoren.

Een andere software component moest worden aangepast om te kunnen communiceren met een nieuwe hardware component. Verder moet het kunnen communiceren met de bestaande, oude hardware component. Het gedrag van de software component moest identiek zijn voor beide versies van de hardware. Om dit te valideren hebben we gebruik gemaakt van een techniek die een model kan leren uit het gedrag van een bestaande implementatie en een programma dat kan checken of twee modellen identiek zijn. Dit leidde tot een aantal verschillen die niet waren gevonden met de bestaande tests. Door het verwijderen van de verschillen hebben we de kwaliteit van de aangepastte software component.

Als laatste toepassing hebben we oude modellen die gebruikt werden om broncode te genereren vervangen door modellen voor een nieuw programma dat ook broncode kan genereren. De oude modellen zijn omgezet met een DSL. Om vertrouwen te krijgen in de omzetting hebben we de hierboven beschreven leertechniek opnieuw toegepast.

Samenvattend kan gesteld worden dat alle technieken hebben bijgedragen aan de onderhoudbaarheid van het systeem. ASD en POOSL hebben bijgedragen aan de softwarekwaliteit door in een vroeg stadium van het ontwikkelproces fouten te voorkomen. De investering om zelf een DSL te maken wordt ruimschoots terugverdiend wanneer veel instanties nodig zijn, bijvoorbeeld in het geval van configuratiebestanden. Door vanuit een enkele bron veel verschillende zaken te genereren, zoals analyse modellen en broncode, worden inconsistentie en fouten op een pragmatische manier vermeden.

SUMMARY

High-tech systems such as air planes, cars, and medical systems contain large amounts of complex software. Adapting and extending such systems is a very challenging task. As an example, consider the Image Guided Therapy (IGT) systems of Philips. These systems are used for minimally invasive treatment of, e.g., cardio and vascular diseases. The software that is responsible for managing and controlling the hardware of the system contains millions of lines of source code. Hence, a new system release will reuse large parts of previous releases. Since IGT systems have been developed for over thirty years, they contain large parts of old software that has been written many years ago.

In general, the maintainability of complex high-tech systems is hampered by the existence of large amounts of old software. Often the original developers have left the development group, the documentation describing the source code is not up to date, and there are not sufficient tests to guarantee correctness after modifications. Hence, changes are difficult and can easily lead to system failures. On the other hand, there is a continuous stream of change requests to deal with new usage scenarios, e.g., new medical procedures.

To investigate whether new techniques can improve the maintainability of complex high-tech systems, we have applied them in a number of real development projects at Philips. Common to all techniques is the use of Domain Specific Languages (DSLs). A DSL has a restricted grammar to express the essential domain concepts only. This leads to abstract models that are easy to read and to maintain. Based on a single model, several artefacts can be generated such as simulation models, formal analysis models, and code.

The first applications of DSL-based techniques at IGT concerned Analytical Software Design (ASD). ASD has been used to design and integrate a new software component for the replacement of an old hardware component. Instances of the ASD language can be verified formally and - from the same instance - source code can be generated automatically. By applying ASD, the number of defects dropped dramatically to 0.17 defects per 1000 lines of source code, while industry standard is between 1 and 25. The approach can be used, however, for a restricted type of small data-independent control components only. To deal with larger components of a more general type, where formal verification is not feasible, we have experimented with the Parallel Object Oriented Specification Language (POOSL). This language allows the expression of timing and probabilistic behaviour. Models can be simulated to explore and validate design concepts. POOSL has been applied

to explore new concepts for the introduction of a new hardware component.

Recent DSL technology makes it possible to create custom languages and generators within a limited amount of time. This enables fast prototyping with dedicated languages in industry. As an application, we have addressed the large configuration files of an old component. We have created a concise language, containing only the essential concepts, and automatically generate the large configuration files. The validity of language instances can be checked automatically before the configuration files are generated. The investment of creating the language can be earned back after creating 13 configuration files, while we expect to need approximately 2000 configuration files.

The approach to create a dedicated DSL was also applied to another old component where the configuration files were completely numerical and consequently difficult to maintain. We created a more readable language from which the low-level configuration files can be generated. In addition, we also generated POOSL models and models for the Symbolic Analysis Laboratory (SAL). SAL allows formal verification of the model but also the generation of test cases. These test cases are an instance of a so-called test DSL. From this test DSL we generate input for the test tool that is used to test the combination of hardware and software. We also generate graphical representations and checks to validate the POOSL and SAL models. Moreover, log files from the usage of the real system are transformed into instances of the test DSL and we check if they conform to the model of the behaviour. By means of these cross checks between the different generated artefacts, we increased the confidence in the correctness of the language instances and the generators.

In another application we considered a software component which was updated to communicate with new hardware. Since it should be able to work with both old and new hardware, the updated software should exhibit the same behaviour as the original software. To validate this, we applied model learning, a technique where automatically a state machine model is created by repeatedly stimulating an implementation and observing the resulting output. The learned models of original and updated software have been transformed into the language of an equivalence checker. This led to a number of differences which were not found with the existing regression test set and it improved the quality of the updated software component.

As a last application we addressed the transformation of old software models, from which code is generated, to new model-based techniques that also allow code generation. The old models have been transformed using DSL technology. To improve the confidence in the model transformations, we applied the previously described model learning technique.

Summarizing, we observe that all techniques improved the evolvability of the system. ASD and POOSL improved software quality by preventing errors early in the development process. DSLs show a very good return on investment when there are many language instances, e.g., in case of configuration files. Having a single source for all artefacts, such as analysis models and code, avoids error prone manual transformations. Moreover, the use of cross checks between different artefacts ensures consistency and correctness in a pragmatic way.

CURRICULUM VITAE

M.T.W. Schuts

1994 – 1998:
Norbertus MAVO Tilburg, the Netherlands.

1998 – 2002:
Professional education in Electronics,
ROC Tilburg, the Netherlands.

2002 – 2006:
Bachelor in Technical Informatics,
Fontys University of Applied Sciences Eindhoven, the Netherlands.

2007 – 2010:
Master of Science in Computing Science,
Radboud University Nijmegen, the Netherlands.

2010 – present:
Software Designer,
Philips Medical Systems Best, the Netherlands.