# Formal Development of Control Software in the Medical Systems Domain

## PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Eindhoven, op gezag van de
rector magnificus, prof.dr.ir. C.J. van Duijn, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op maandag 17 december 2012 om 16.00 uur

door

Ammar Ahmed Hasan Osaiweran

geboren te Ta’izz, Jemen

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr.ir. J.F. Groote
en
prof.dr. J.J.M. Hooman

**Formal Development of Control Software in the Medical Systems Domain /**
By A.A.H. Osaiweran.

This work has been sponsored by Philips Healthcare, Best, The Netherlands.
Thesis lay-out and page size of 160 x 450 mm are based on Philips style.

Cover design: Ammar Osaiweran

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).
IPA dissertation series 2012–15

A catalogue record is available from the Eindhoven University of Technology Library
ISBN: 978-90-386-3276-6

# Contents

# Preface

After the completion of my Master study at the Eindhoven University of Technology and finishing my Master graduation project at Philips Healthcare, I was puzzled between working as a software engineer in industry or pursuing a PhD study in academia. The choice was initially very difficult as both trajectories are very attractive to me. Suddenly, I got an email from my supervisor Jan Friso Groote informing me that people at Philips Healthcare are willing to support my PhD study and whether I would be interested to continue my research on the running projects of Philips.

Without a doubt I was very happy and the offer attracted me since through this opportunity I felt that I could gain lots of experience from both industry and academia. Therefore, I decided to continue having fun biking every day all the way from Eindhoven to Best and accepted the PhD position.

During the last four years, I worked with many people at Philips Healthcare. First, I would like to thank Marcel Boosten, who was my first supervisor at Philips Healthcare, and Wim Pasman, both for arranging all required procedures towards sponsoring my PhD study by Philips. Especial thanks go to Marcel for adapting me to the Frontend team and for the technical and non-technical advices.

I would like to thank everybody I worked with in the Frontend team, especially during my first year. Although at the middle of that period there were many obstacles hindered the progress of this thesis due to changes in the organization, there were certainly many technical lessons and experiences gained when working with the team.

From the Frontend team I would like to thank everyone who had in some way or another contributed in the content of this thesis. I am thankful to Jan Borstrok, Hans van Wezep, Andre Postma, Johan Gielen and Ben Nijhuis, for sharing their expertise and domain knowledge, especially at the period of preparing the reference architecture of the Frontend controller. I am also thankful to Ad Jurriens and Mahmoud Elbanna for providing me the necessary information that helped constructing some of the work presented in this thesis.

At the beginning of my second year, I started a very productive period when joining the workflow team at the Backend. I can not express my deep thanks to my second supervisor at Philips Jacco Wesselius. I am thankful to him for adapting me to the Backend team and for his rigorous comments and valued feedbacks on most of the jointly published papers.

During the work with the workflow team I met many people who favorably contributed to this thesis. I am grateful to Paul Alexander and Bert Folmers for assigning me the development tasks that fit both my research interest and at the same time contribute to the end product. It was very pleasant sharing one office with Roel van Velzen. I am thankful to him for sharing his knowledge regarding the design of the Backend subsystem during the design phase.

I would like to express my thanks to the people I worked with, one by one, including Ron Swinkles, Amol Kakhandki, Amit Ray, Harry Kuipers, Tom Fransen, Paul Langemeijer, Marco van der Wijst and Astrid Morselt. Many thanks go to you all since you never hesitated to help or clarify any issue I faced despite the workload and the limited time you had. It was very pleasant working with you all. I am thankful for Bart van Rijnsover for being my Philips supervisor at the end of my study period and for his support.

From Verum, I would like to thank Leon Bouwmeester and Guy Broadfoot for their valued feedback on the joint work we did at early stage of my study regarding the suitability of ASD to model object-oriented designs.

From the BasiX project, I would like to thank Ivo Canjels, Marc Loos and Rob Kleihorst for explaining the architecture of the PDU and providing me the necessary documents and directions. Also many thanks go to them for their valued feedbacks.

I cannot express my deep thanks to Mathijs Schuts with whom I wrote some of the papers constitute this thesis. Many thanks for his kindness and sharing his knowledge and expertise. I enjoyed accompanying him in Tallinn when presenting one of our papers there.

Many thanks also go to my second promotor Jozef Hooman for assisting me making my work more mature. This thesis would not take its current shape without his directions and valued comments.

I would like to deeply thank my first promotor Jan Friso Groote. Without his advice and directions this thesis may never come to exist. Thank you Jan Friso for helping me to develop as a researcher during the entire 4 years.

Finally, but certainly not last, I thank my parents and wife for their endless support. Especial thanks go to my wife for her patience while preparing this thesis and for the limitless emotional support.

Thank you all!

Ammar Osaiweran, October 2012

# Chapter 1

# Introduction

## 1.1 The Industrial Context

Philips Healthcare develops a number of highly sophisticated medical systems, used for various clinical applications. One of these systems is the interventional X-ray (iXR) system, which is used for minimally invasive surgeries. An example of this type of systems is depicted in Figure 1.1.

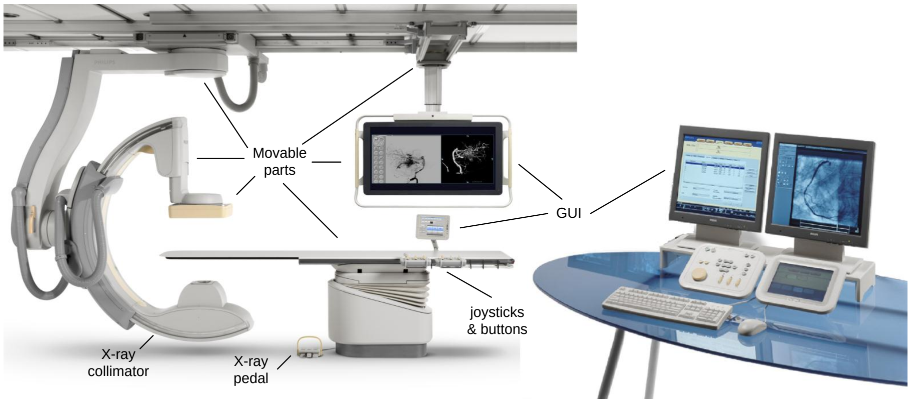


**Figure 1.1** An interventional X-ray system

As can be seen in the figure, the system includes a number of graphical user interfaces, used for managing patients' personal data, exam, and X-ray details. It also comprises a number of motorized movable parts such as the table where patients can lay and the stand where the X-ray generator and the image detector are fixed. Furthermore, the system contains a number of PCs and devices that host the software that controls the entire system.

Using this type of systems various clinical procedures can be accomplished. As an example application, the system provides a practical means to avoid open-heart, invasive surgeries. For instance, if a heart artery of a patient is blocked or narrowed, the surgeon inserts a very thin flexible catheter (tube) to the artery of the patient and directs it to the blockage in the affected artery. When the catheter reaches the blockage, a small balloon and a stent are inflated to reopen the artery and flatten the blockage into the artery wall. Along the surgery, the physicians are guided by high quality X-ray images. Figure 1.2 illustrates this type of clinical procedure.

The main benefits of using these systems in hospitals are that they provide more effective and safe treatments, higher success rates, and shorter hospital stays. Therefore, many of these systems are widely used in hospitals, saving patients lives every day, and everywhere.

However, since the healthcare domain is quickly evolving, many challenges are imposed

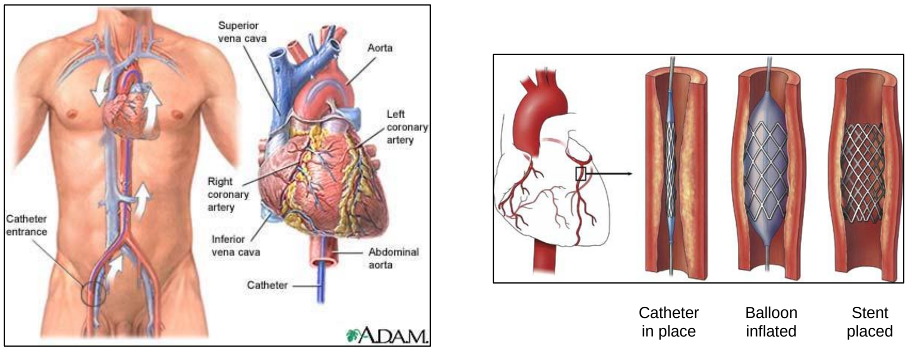


**Figure 1.2** At the left, the insertion point of the catheter and its course to the heart, clinical users are guided by X-ray images. At the right, treatment of the blockage.

on such type of X-ray systems. This includes, for example, rapidly supporting the increasing amount of medical innovations, new complex and highly sophisticated clinical procedures and smooth integration with the increasing products of third party suppliers. Indeed, this requires a flexible software architecture that can be easily extended and maintained without the need of constructing the software from scratch.

For this reason, the software practitioners at Philips Healthcare are constantly seeking approaches, tools and techniques to advance current software development processes. The purpose is to improve the quality of developed code, enhance productivity, lower development costs, shorten the time to market, and increase end-user satisfaction.

To achieve a fast realization of the above goals, Philips Healthcare is gradually shifting to a component-based architecture with formally specified and verified interfaces. New components are developed according to this paradigm, and existing parts are gradually replaced by components with well-defined formal interfaces.

The software architecture divides the X-ray system into three main subsystems called, the Backend, the Frontend and the Image Processing subsystems. We detail their responsibilities in the subsequent chapters. In turn, the software of each subsystem is divided into a number of software units. Each unit comprises a number of modules that include software components with interfaces.

At Philips Healthcare, the component-based development approach is based on a formal approach called Analytical Software Design (ASD) [20, 21, 22]. This approach is supported by the commercial tool ASD:Suite of the company Verum [83]. ASD enables the application of formal methods into industrial practice by a combination of the Box Structure Development Method [65], CSP [79] and FDR2 [34, 79]. The tool ASD:Suite supports the automatic generation of code in high-level languages, such as C++ or C#, from formally verified models.

The aim of using the ASD approach at Philips Healthcare is to build high quality components that are mathematically verified at the design phase by eliminating defects as early as possible in the development life cycle, and thus reducing the effort and shortening the time devoted to integration and testing.

## 1.2 The subject matter

This thesis is concerned with evaluating the use of the ASD formal techniques in industry, and investigating whether the use of the techniques resulted in better quality software compared to traditional development. It details in depth the experiences and the challenges encountered during the application of the techniques to the development of the iXR system, providing practical solutions to the encountered shortcomings. In particular, it evaluates and details the formal development of various software components of the iXR system and investigates the added value of such techniques to the end quality and the productivity of the developed software.

The concept of quality in this thesis refers to the error density in terms of the number of reported defects per thousand lines of code while productivity denotes the number of lines of code produced per staff-hour.

The thesis also discusses a number of design steps and guidelines that facilitate the construction of formally verified software components, based on the experiences gained from designing various industrial cases. To detail these guidelines we employed another formal technology called mCRL2, which we used to formally specify and verify a number of design cases. For each guideline two designs that capture a same application intent are compared in terms of the number of states they produce, given that one design follows the guideline while the other does not. Moreover, we show that these guidelines can provide an effective framework to design verifiable components at Philips Healthcare by applying them to formally specify and verify an industrial case, developed at Philips Healthcare.

This work was established in an industrial context, dealing with real industrial projects and a real product. The results are very conclusive in the sense that, in the context of Philips Healthcare, the used formal techniques could deliver better quality code compared to the code developed in more traditional development methods. Also, the results show that the productivity of the formally developed code is better than the productivity of code developed by projects of Philips Healthcare. Our findings indicate the possibility of a 10 fold reduction in number of errors and a threefold increase in productivity, as a result of applying these techniques.

As observed in [86, 14], there are quite a number of reports detailing the application of formal methods to industrial case studies, but very few published reports of formal methods applications describe second or subsequent use. Similarly, the literature about the incorporation of formal methods in the standard industrial development process is very limited. Therefore, in this thesis, we discuss how the ASD formal techniques were tightly incorporated in the development process of projects established at Philips Healthcare, and we introduce in the following chapters a series of industrial cases, demonstrating continues and subsequent usages of these formal techniques in industry.

This work concerns a PhD thesis funded by Philips Healthcare. The author of this thesis worked 4 days a week at Philips Healthcare as a member of various development teams, participating, monitoring and analyzing the progress of applying formal methods to the development of various components of the iXR product.

The effort was more concentrated on developing the control part of various components using ASD, and investigating the fundamental issues behind the technology and its applicability in industry. The work was accomplished in close contact with team members that include project and team leaders, lead architects, main responsible designers of the units, and the software and test engineers. Regarding the application of formal techniques in industry, the thesis provides answers to various research questions that were raised during the development process of the software. The research questions address various aspects encountered when integrating the formal techniques in industry.

With respect to the accomplished work, the thesis reports about the experiences gained from designing and developing real industrial products and not from designing case studies. The thesis evaluates the ASD technology not only in one project but in a series of projects. To answer some of the research questions raised for some of the projects where the author was partially involved, the author conducted plenty of meetings with the main responsible leaders, architects and designers and then evaluated the projects from the data available in many sources at Philips Healthcare such as version control and the defect management system. After a careful analysis of the data, the data and the results are communicated to the responsible teams for further feedback and comments. The author published the results externally [43, 70, 74, 69, 42, 75, 44] after they are being reviewed and confirmed by the responsible teams. The details are provided in the chapters of this thesis.

## 1.3 Overview of the thesis

The thesis is divided into two parts and consists of seven chapters. Each chapter concentrates on different aspects of applying formal techniques in industry. The thesis is organized as follows.

The first part details the application of the ASD technology to develop various parts of the X-ray system. It includes four chapters in total.

Chapter 2 introduces the formal methods, tools and techniques that were used in this work. It provides the preliminary concepts needed for understanding the rest of the thesis. First, we start by detailing the fundamentals of the ASD approach, illustrating how the formal approach can be tightly integrated with the standard development processes in industry. Then, we briefly introduce relevant concepts of the mCRL2 language, which we use to introduce the design guidelines to avoid the state space explosion problem of model checking in Chapter 6. Also we employe the language to formalize a number of industrial designs, specified using the design guidelines in Chapter 7, to show that the guidelines are effective in industrial settings.

Chapter 3 is mainly concerned with detailing experiences of incorporating the ASD technology in industrial practice during a project of the iXR, called the BasiX project. The chapter details two design cases. With the first case we show how the ASD technology was used to formalize and verify a controller of a power distribution unit (PDU) used to distribute electrical power and network signals to the PCs and the devices of the X-ray system. Using this design we elaborate more on how ASD models can be specified, the ASD notations used to model the components, and the formal checks established by FDR2. We discuss how the use of ASD formal techniques resulted in detecting some veiled errors and subsequently improved the quality of the controller.
The second design case is related to the development of a number of services deployed on the PCs of the X-ray system. The services communicate with the PDU across the network to facilitate the automatic start-up and shutdown of the system. Using this case we illustrate how the ASD technology was incorporated with the test-driven development method to develop the software of the service. Since any method or technology used in industry has to be compatible with other used methods, tools and techniques, we illustrate the main challenges and key issues encountered when integrating the ASD technology with available techniques and tools used at Philips, providing some practical solutions for the faced shortcomings. Furthermore, we detail the benefits of applying these formal techniques to the quality of the developed product. The details of this chapter is based on the following publications:

> J.F Groote, A.A.H. Osaiweran and J.H. Wesselius. *Analyzing a controller of a power distribution unit using formal methods*. Proceedings of *the Fifth International Conference on Software Testing, Verification and Validation (ICST 2012, Montreal, Canada, April 18-20, 2012)*, (pp. 420-428). IEEE.
>
> A.A.H. Osaiweran, M.T.W. Schuts, J.J.M. Hooman and J.H. Wesselius. *Incorporating formal techniques into industrial practice: an experience report*. Proceedings of *the 9th International Workshop on Formal Engineering Approaches to Software Components and Architectures (FESCA'12, Tallinn, Estonia, March 31, ), (Electronic Proceedings in Theoretical Computer Science, ..., pp. ...-...)*, submitted / in press.

Chapter 4 introduces experiences of developing sizable software units of the Backend subsystem. The first unit is the FEClient, using which we detail the effort spent to develop the control part using ASD and the percentage of time consumed for every development process: requirements, formal modeling and verification, for instance. The second unit is the Orchestration, by which we demonstrate the steps performed to obtain structured components and the peculiarities that made the components easily verified using model checking. For both cases, we discuss the typical errors found during the construction of the units and we show that the errors that could escape the formal verification were easy to find and to fix, not very deep interface or design errors. Finally, compared to the industrial standard, we show that the end quality result of the two units was remarkable, and the units were robust against the frequent changes in the requirements. The details of this chapter is based on the following publications:

> J.F. Groote, A.A.H. Osaiweran and J.H. Wesselius. *Experience report on developing the Front-end Client unit under the control of formal methods*. Proceedings of *the 27th ACM Symposium on Applied Computing, The Software Engineering Track (ACM SAC-SE 2012, Riva del Garda, Italy, March 25-29, 2012)*, (pp. 1183-1190).

> A.A.H. Osaiweran, T. Fransen, J.F. Groote and B.J. van Rijnsoever. *Experience report on designing and developing control components using formal methods*. Proceedings of *the 18th international Symposium of formal methods. Cnam, Paris, France, 27-31 August, 2012. (pp. ...-...)*, submitted / in press

Chapter 5 evaluates the use of the ASD formal techniques in both the Frontend and the Backend subsystems. The chapter highlights the main obstacles encountered during the application of the ASD technology. Furthermore, we compare the end quality and productivity of the units that incorporate ASD with others that were developed in a more traditional development method. We demonstrate the steps followed to empirically evaluate the ASD developed units. The details of this chapter is based on the following publications:

> A.A.H. Osaiweran, M.T.W. Schuts, J.F. Groote, J.J.M. Hooman and B.J. van Rijnsoever. *Evaluating the effect of formal techniques in industry*. (Computer Science Report, No. 12-13). Eindhoven: Technische Universiteit Eindhoven, 21 pp.

> J.F. Groote, A.A.H. Osaiweran and J.H. Wesselius. *Analyzing the effects of formal methods on the development of industrial control software*. Proceedings of *the 27th IEEE International Conference on Software Maintenance (ICSM 2011, Williamsburg VA, USA, September 25-30, 2011)*, (pp. 467-472). IEEE.

The second part is concerned with introducing a number of specification and design guidelines to circumvent the state space explosion problem and detailing their applicability to design real industrial cases. This part of the thesis consists of two chapters.

Chapter 6 introduces the guidelines. For each design guideline we introduce two different designs that both maintain the same application intent. The first design does not consider the guideline so it subsequently produces a large state space. The second design uses the guideline so the resulting state space is less compared to the first design. The details of this chapter is based on the following publication:

> J.F. Groote, T.W.D.M. Kouters and A.A.H. Osaiweran. *Specification guidelines to avoid the state space explosion problem. Fundamentals of Software Engineering (4th IPM International Conference, FSEN 2011, Tehran, Iran, April 20-22, 2011. Revised Selected Papers)*, (Lecture Notes in Computer Science, 7141, pp. 112-126). Berlin: Springer.

Chapter 7 demonstrates the application of some of the proposed guidelines of Chapter 6 to the design and the verification of the PDU controller, addressed in Chapter 4. We show that the guidelines were effective for designing the controller, and hence could provide a suitable framework to design verifiable components of real industrial cases. The details of this chapter is based on the following publication:

> J.F. Groote, A.A.H. Osaiweran, M.T.W. Schuts and J.H. Wesselius. *Investigating the effects of designing industrial control software using push and poll strategies*. (Computer Science Report, No. 11-16). Eindhoven: Technische Universiteit Eindhoven, 19+36 pp.

# Chapter 2

# Methods, Tools and Techniques

## 2.1 Introduction

This chapter is concerned with a concise introduction to the formalisms, tools and techniques used for modeling, verifying and developing components of the industrial designs presented along the rest of this thesis. First, we introduce the ASD approach. We illustrate its main principles and how the formal techniques supplied by the approach were tightly integrated with the standard development processes of software in iXR projects. The ASD approach was used to develop software components of the industrial projects presented in Chapter 3,4, and 5.

Second, we introduce the mCRL2 language [63] in Section 2.5. We use the mCRL2 toolset to formally analyze a number of design cases. The purpose is to illustrate the effectiveness of various design guidelines and styles that we propose to circumvent the state explosion problem in Chapter 6 and 7.

The reason of choosing mCRL2 as an alternative technology of ASD is that we found that some of the guidelines may not be directly realized using the current version of the ASD:Suite. The mCRL2 toolset is used to merely specify and formally verify the behavior of a number of designs constructed following the proposed guidelines. Code generation of the specified formal models is not of importance here as the main goal is to illustrate how the guidelines may circumvent the state explosion of model checking and lead to construct verifiable components.

## 2.2 Analytical Software Design

Philips Healthcare introduced the ASD technology to its development context in 2006. The technology was initially used as a formal means to formally specify the interaction protocol between the subsystems of the X-ray machine. Then, in 2008 the technology was used as a formal approach to develop various software components of the system. Below we briefly discuss the fundamentals of the approach and how it is being exploited in Philips Healthcare.

ASD is a model-based technology that combines formal mathematical methods such as Sequence-based Specification (SBS) [52] technique, Communicating Sequential Processes (CSP) [79] and the model checker FDR2 (Failures-Divergence Refinement) [79, 34] with component-based software development [26].

A key principle of ASD is to identify a design of software as interacting components, communicating with each other or their environment via channels (interfaces). A common ASD practice is to distribute system functionalities among components in levels (e.g., hierarchical structure ), to allow a systematic construction and verification of each component separately. Each ASD component includes a state machine that comprises input stimuli and output responses.

Figure 2.1 at the left depicts an example structure of components, which includes a con-

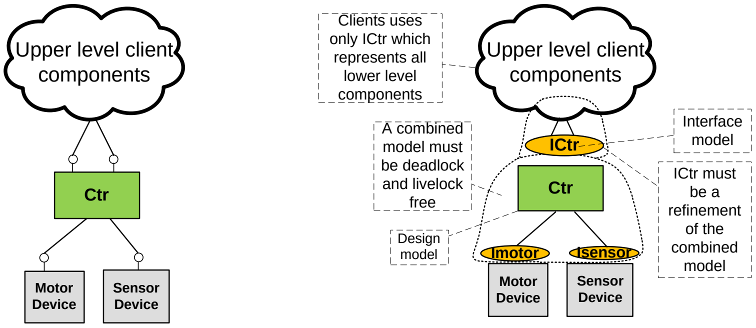

Upper level client components
Ctr
Motor Device
Sensor Device
Clients uses only ICtr which represents all lower level components
Upper level client components
A combined model must be deadlock and livelock free
ICtr
Interface model
Ctr
ICtr must be a refinement of the combined model
Design model
Imotor
Isensor
Motor Device
Sensor Device

**Figure 2.1** Example of structured components

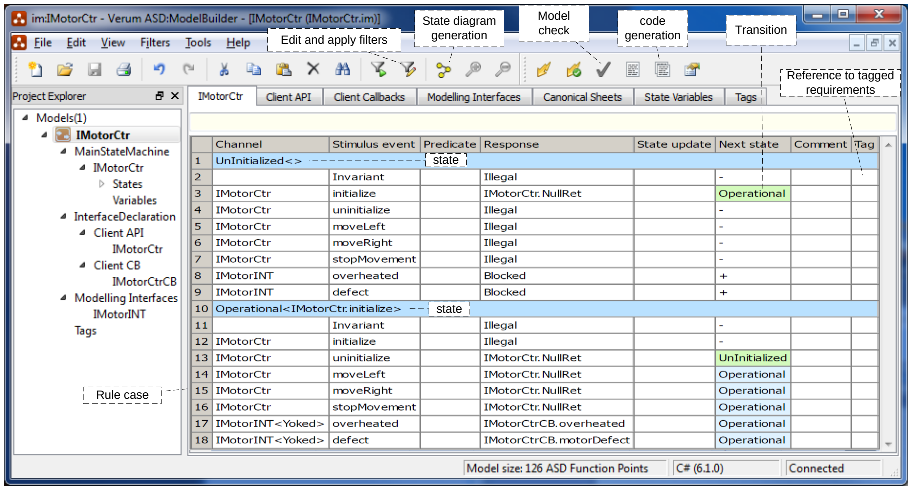
troller (Ctr) that controls a motor and a sensor. We assume that the motor is responsible for moving the patient table to the left and to the right while the sensor is used to detect any object during the movement to prevent potential collisions. We use this example system along the rest of this chapter.

### 2.2.1 The specification of ASD models

Any ASD component is realized by constructing two types of models specified in a similar tabular notation: an interface model and a design model. The two models are exploited as follows:

- The *interface* model is an initial abstract state machine of the component being developed. The model is used as a formal means to describe not only the methods to be invoked on the component but also the external behavior with respect to clients. Interactions with components located at a lower level of the component being developed are abstracted away from this interface specification.
- The *design* model of a component refines and extends the interface model with more internal details. From the behavioural perspective, the design model is not an abstract state machine but a concrete state machine in the sense that all detailed behavior is described. Usually, the design model uses the interface models of other ASD and non ASD components (i.e., the handwritten components). These interfaces, in turn, are independently refined by other design models (perhaps by other teams), facilitating multi-site, parallel development of components.

The state machine of the ASD interface model provides a technical description between client and server components, affording a shared understanding of the required behavior. The model represents the protocol of interactions similar to a protocol state machine of



| | Channel | Stimulus event | Predicate | Response | State update | Next state | Comment | Tag |
|---|---|---|---|---|---|---|---|---|
| 1 | UnInitialized<> | | | | | | | |
| 2 | | Invariant | | Illegal | | - | | |
| 3 | IMotorCtr | initialize | | IMotorCtr.NullRet | | Operational | | |
| 4 | IMotorCtr | uninitialize | | Illegal | | - | | |
| 5 | IMotorCtr | moveLeft | | Illegal | | - | | |
| 6 | IMotorCtr | moveRight | | Illegal | | - | | |
| 7 | IMotorCtr | stopMovement | | Illegal | | - | | |
| 8 | IMotorINT | overheated | | Blocked | | + | | |
| 9 | IMotorINT | defect | | Blocked | | + | | |
| 10 | Operational<IMotorCtr.initialize> | | | | | | | |
| 11 | | Invariant | | Illegal | | - | | |
| 12 | IMotorCtr | initialize | | Illegal | | - | | |
| 13 | IMotorCtr | uninitialize | | IMotorCtr.NullRet | | UnInitialized | | |
| 14 | IMotorCtr | moveLeft | | IMotorCtr.NullRet | | Operational | | |
| 15 | IMotorCtr | moveRight | | IMotorCtr.NullRet | | Operational | | |
| 16 | IMotorCtr | stopMovement | | IMotorCtr.NullRet | | Operational | | |
| 17 | IMotorINT<Yoked> | overheated | | IMotorCtrCB.overheated | | Operational | | |
| 18 | IMotorINT<Yoked> | defect | | IMotorCtrCB.motorDefect | | Operational | | |

**Figure 2.2** The ASD interface model of the Motor in the ASD:Suite

UML [16]. When a design is decomposed into a number of components in levels, a client component uses the interface model of the server component. So a server at one level becomes a client of another component located at a lower level, similar to the Design by Contract approach [64].

To ensure complete and consistent specifications, the models are described using the sequence-based specification technique [72]. That is, any ASD model includes a complete specification in the sense that responses to every possible input stimulus in every state must be described.

An example of the tabular specification related to the interface model of the motor component of Figure 2.1 is depicted in Figure 2.2. As can be seen from the specification, each table is a state, where all potential input stimuli are listed in rule cases (rows of the tables). A rule case comprises a number of items, each of which includes an interface (channel), a stimulus, predicate (conditions on the stimulus), responses, state (or predicate) updates, a next state, comments, and tags of informal requirements.

The set of stimuli of a component consists of events invoked by clients located at an upper level plus callback events sent by used components at a lower level. The set of responses includes events sent to used components plus callback events sent to upper client components. Calls from client to used components are always synchronous, whereas callback events sent by used components to the client components are asynchronous and stored locally in a FIFO queue of the target client component.

To ensure specification completeness, the set of user-defined responses is extended with special purpose responses: *Illegal*, *Blocked*, and *Null*. The *Illegal* response denotes that invoking a stimulus is illegal. The *Blocked* response denotes that the corresponding stimulus cannot happen. The *Null* response denotes that no action is required when the stimulus event is invoked: consuming a call, for instance.

In all presented models throughout this thesis the *NullRet* response indicates the completion of the client request (i.e., the call is returned to the client). A channel postfixed by *INT* denotes an internal channel, not visible to the client. The corresponding stimulus event of an internal channel denotes a spontaneous event internally generated by the component, not synchronized with any component. A channel postfixed by *CB* indicates a callback stimulus/response event received/sent from/to a queue of the client.

To illustrate the above modeling conventions, consider Figure 2.2. The *NullRet* of rule case 3 indicates that the *initialize* event is successfully completed, and the motor transits to the *Operational* state. The spontaneous *defect* stimulus of the *IMotorINT* channel of rule case 18 denotes that the motor may fail internally for some internal reasons, and as a response the motor notifies the upper controller by sending the *IMotorCB.motorDefect* callback event to the queue of the controller.

The addition <yoked> of rule case 17 and 18 indicates that the number of allowed callbacks (listed in the corresponding response list of the rule case) in the queue is restricted. In our example specification, if the yoking threshold is set to 1 for each of the two callbacks then this means that the queue will include only 2 callbacks at maximum. This is one way to circumvent queue overflow cases in ASD, during formal verification using model checking, since the CSP processes will deadlock when the queue is full.

The specification of ASD design models is restricted to components with data-independent control decisions. This means that the correctness of parameter values of methods is not checked by the tool, and components responsible of data manipulations or algorithms should be implemented by other techniques. The technology is also restricted to the development of components with discrete behavior and does not support the development of timed systems.

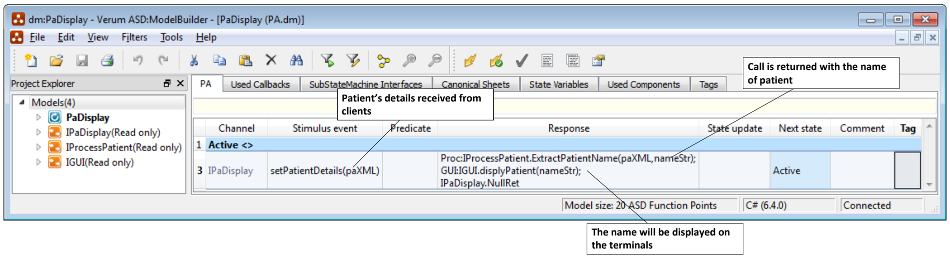
To ensure consistency and correctness, mathematical CSP models [79] are automatically generated from the ASD model. To enable better industrial usage, all CSP models are hidden from end-users and the models are verified at a remote server located at Verum. The details of such translation is outside the scope of this thesis.

A vital and very attractive feature in the ASD:Suite is the support of a comprehensive code generation from formally verified design models to a number of programming languages (C, C++, C#, Java), following the state machine pattern of [35]. The details of such translations are irrelevant for this thesis. The translation details are part of the ASD patent described in [22].

### 2.2.2 The use of data parameters in ASD models

The interface model is used to declare calls received from clients (the implemented interface of the component) and the callbacks sent to client components. Calls and callbacks might include data parameters that must be compatible with the target programming language of code generation.

In the design model these parameters are used as control independent parameters. That is, the parameters, for instance, can not be used in the model as state variables (in the predicates), i.e., they do not affect the behavior of the component as they are only used to pass the data across the component. The design model abstracts from the data details and how the data is processed. Processing the values of these parameters is done by other components written manually.


dm:PaDisplay - Verum ASD:ModelBuilder - [PaDisplay (PA.dm)]
File Edit View Filters Tools Help
Project Explorer
Models(4)
PaDisplay
IPaDisplay(Read only)
IProcessPatient(Read only)
IGUI(Read only)
PA
Used Callbacks
SubStateMachine Interfaces
Canonical Sheets
State Variables
Used Components
Tags
Model size: 20 ASD Function Points
C# (6.4.0)
Connected

| | Channel | Stimulus event | Predicate | Response | State update | Next state | Comment | Tag |
|---|---|---|---|---|---|---|---|---|
| 1 | Active <> | | | | | | | |
| 3 | IPaDisplay | setPatientDetails(paXML) | | Proc:IProcessPatient.ExtractPatientName(paXML,nameStr);<br>GUI:IGUI.displyPatient(nameStr);<br>IPaDisplay.NullRet | | Active | | |

Call is returned with the name of patient

Patient's details received from clients

The name will be displayed on the terminals

**Figure 2.3** Use of data parameters in a design model

To illustrate the parameter usage in ASD consider the example specification of the design model depicted in Figure 2.3. It corresponds to an ASD component that is responsible of displaying a patient name on a screen. The design model uses two interface models representing the external behavior of two handwritten components: *IProcessPatient* and *IGUI* interface models, depicted at the left of the figure and used in the Responses column. The component is used by a client component and receives a stimulus event (*setPatientDetails*) that contains a parameter of an xml string which includes the details of a patient. We assume that the GUI accepts only a string representing the name of the patient to be displayed.

To obtain the name from the xml string the ASD component passes the received string to another handwritten component which extracts the name and returns it to the ASD component with the call. For instance, the parameter *paXML* (patient xml) may be passed to the handwritten code by value while *nameStr* is passed by reference so that *nameStr* will contain the name as soon as the call is returned to the ASD component. When the call is returned from the handwritten code, the ASD component passes the name to the component responsible of displaying the name on the terminal.

The values of parameters can be stored in the ASD components at some states and retrieved later at some other states, if needed. Currently, the parameters and their values are not considered during the formal verification using model checking so correctness are checked by other means, for instance testing. Furthermore, checking refinement of the handwritten code against the interface model is not supported.

## 2.3 Related work

The ASD approach has been inspired by the formal Cleanroom software engineering method [59, 71] which is based on systematic stepwise refinement from formal specification to implementation. As observed in [19], the 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++ [32] and the code generator of the industrial tool VDMTools [27]. Similarly, the B-method [3], which has been used to develop a number of safety-critical systems, is supported by the commercial Atelier B tool [24]. The tool ProB provides facilities to model-check and animate models specified in the B specification. ProB was used in a number of critical systems, most notably railway control. The SCADE Suite [30] provides a formal industry-proven method for designing critical applications with both code generation and verification. Compared to ASD, 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.

## 2.4 Incorporating ASD in industrial practices

The development process of software used in projects within the context of iXR is an evolutionary iterative process. That is, the entire software product is developed through accumulative increments, each of which requires regular review and acceptance meetings by several stakeholders. Figure 2.4 outlines the flow of activities in a development increment, including the steps required to develop ASD components.

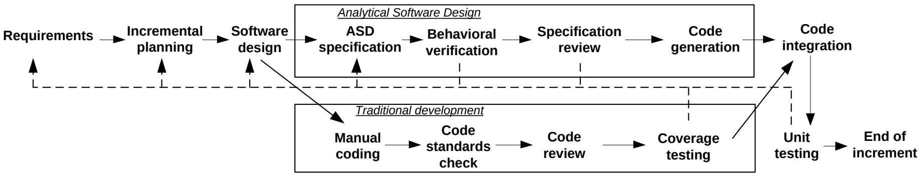


**Figure 2.4** Steps performed in a development increment

Developers are divided into groups, each of which is responsible for developing a software unit. Each unit is developed by a team of 2-5 members. Each unit comprises modules which in turn consist of components with well-defined interfaces and responsibilities.

Each increment starts after lead architects and project leaders identify a list of features to be implemented by the development team. As soon as the features and the corresponding requirements are approved, 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.

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.

The design clearly distinguishes between control (state machines) and non control components (e.g., data manipulation, computation and algorithms). Non control components are always developed using conventional development methods, while control component are usually constructed using ASD.

Given a software design, the manually written components and ASD components can be constructed in parallel, as depicted in Figure 2.4. For the manually coded components, checking coding standards is mandatory. Such a check is performed automatically using the TIOBE technology [81], and required by quality management. After that, the code is thoroughly reviewed by team members before it becomes a target of coverage testing.

Development teams are required to provide at least 80% statement coverage and 100% function coverage for the manually written code. Upon the completion of coverage testing, the code is integrated with the rest of product code, including the automatically generated code from ASD models. The steps performed to develop ASD components are introduced in the subsequent section.

After that, the entire unit becomes a target of unit test, usually accomplished as a blackbox testing. Then, the entire code is delivered to the main code archive, managed by the IBM clearcase technology, where the code is integrated with the code delivered by other team members responsible for developing other units.

At the end of each increment, the units of the Frontend including the ASD components are thoroughly and extensively tested by specialized test teams, using various type of testing such as smoke test, regression test, performance test, statistical test etc, of which details are outside the scope of this thesis. Testing usually reveals some coding errors which are communicated to and resolved by the responsible team.

In the subsequent section, we describe the steps required to develop ASD components in a given design.

### 2.4.1 Steps of developing ASD components

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

ASD components are created and verified in isolation. The compositional design and verification of isolated components in ASD is essential to circumvent the state space explosion problem. The typical steps required for developing an ASD component are summarized below, according to the steps 1 through 6 of Figure 2.5. We consider developing the *Ctr* component depicted in Figure 2.1 at the right as an example.

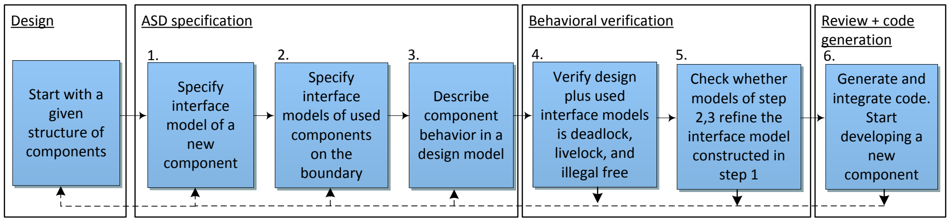

Design
Start with a given structure of components
ASD specification
1.
Specify interface model of a new component
2.
Specify interface models of used components on the boundary
3.
Describe component behavior in a design model
Behavioral verification
4.
Verify design plus used interface models is deadlock, livelock, and illegal free
5.
Check whether models of step 2,3 refine the interface model constructed in step 1
Review + code generation
6.
Generate and integrate code. Start developing a new component

**Figure 2.5** Steps to develop a component using the ASD approach

1. *Specification of externally visible behavior.* Initially, an ASD interface model of the component being developed is created. Note that 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.

   For instance *ICtr* in Figure 2.1 is the interface model of the *Ctr* component, where concrete interactions with the sensor and the motor interfaces are not included. *ICtr* specifies how the clients are expected to use *Ctr*.

2. *Specification of external behavior of used components.* Similarly, ASD interface models are constructed to formalize the external behavior of components that are used by the component under development. These models describe also the external behavior exposed to the component being developed.

   For example, the *Imotor* and *Isensor* interface models describe the external behavior related to the *Ctr* component. All other internal interactions not visible to *Ctr* are not present.

3. *Model component design.* On completion of the external behavior specification, an ASD design model of the component is created. It describes the complete behavior of the component, including calls to used interface models (as created in step 2) to realize proper responses to client calls.

   For instance the design model of *Ctr* includes method invocations from and to the lower level *Motor* and *Sensor* components.

4. *Formal verification of the design model.* Through this step CSP processes are generated automatically from the interface and design models created earlier by the ASD:Suite. These processes form a combined model that includes the parallel composition of the design model process plus the processes of the used interface models. The combined model is used to verify that the design model uses its used interfaces correctly. This is done by checking a fixed set of properties including searching for deadlocks, livelocks, and illegal invocations using FDR2. We detail these properties in Section 3.2. The ASD verification is compositional [49] in the sense that the design model is verified only with the used interfaces, which are refined by other design models.

   To clarify this step using the *Ctr* component example, a combined model that includes *Ctr* and *Imotor* and *Isensor* is constructed. The behavioral verification checks whether *Ctr* uses the motor and the sensor interfaces correctly, such that no deadlocks, livelocks, race conditions, etc. are present.

5. *Formal refinement check.* In this step the 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. When the check succeed, the interface model of step 1 replaces all lower level components. It can be used when constructing upper level components. The refinement check is formally established in ASD using the failure or failures-divergence refinement supplied by the FDR2 model checker, where the interface process is the specification and the combined model is the implementation.

   For instance, when the combined model constructed in step 4 is a valid refinement of the *ICtr* process, *ICtr* formally represents all lower level components during the verification with upper-level client components.

6. *Code generation and integration.* After all formal verification checks are successfully accomplished, source code can be generated from the model. The generated code can be integrated with the rest of the code in the target programming language. Furthermore, for every interface model constructed earlier, the previous steps can be repeated until all components are developed.

### 2.4.2 Roles of the interface models within iXR

The ASD interface model plays important roles along the development of software in iXR projects. Below, some of these roles are summarized.

1. The interface model is used as a formal document that specifies the protocol of interaction between two or more components or subsystems (e.g., between the Backend and the Frontend subsystems). It provides a shared understanding and simplifies communication among independent software architects, designers and engineers, responsible of developing different parts of the system.
2. The interface model is used to formally represent all lower level components. Since all internal details are abstracted away, the interface behavior is often easy to understand. Furthermore, the formal description of the interface reduces the risk of misunderstanding of certain behavior or critical design decisions.
3. When the interface model is formally refined by other lower-level complex models, the model is used for verification with the design models of upper level clients. The interface behavior tends to be simpler than its corresponding composed model, so that verification of clients using model checking can be a straightforward activity. For verification substantially fewer states are generated compared to combining all design models at once.
4. When a formally refined interface model of a component is used correctly by ASD clients, the integration of clients' code with the ASD code of the component is typically done without errors (especially during the compilation and the building process of the code).
5. The interface model can represent foreign components (hardware devices, legacy code or handwritten code) developed outside ASD by describing the externally visible behavior. Doing this simplifies understanding and implementing the internal code, regardless of the programming language being used.

## 2.5 A short introduction into mCRL2

We give a short exposition of the specification language mCRL2. We only restrict ourselves to the parts of the language that we need in this thesis. At `www.mcrl2.org` the toolset for mCRL2 is available, as well as lots of documentation and examples.

The abbreviation mCRL2 stands for micro Common Representation Language 2. It is a specification language that can be used to specify and analyse the behavior of distributed systems and protocols. mCRL2 is based on the Algebra of Communicating Processes (ACP, [8]), which is extended to include data and time.

We first describe the data types. Data types consist of sorts and functions working upon these sorts. There are standard data types such as the booleans ($\mathbb{B}$), the positive numbers ($\mathbb{N}^+$) and the natural numbers ($\mathbb{N}$). All sorts represent their mathematical counterpart. E.g. the number of natural numbers is unbounded.

All common operators on the standard data sorts are available. We use $\approx$ for equality between elements of a data type in order to avoid confusion with $=$ which we use as equality between processes. We also use $\mathit{if}(c, t, u)$ representing the term $t$ if the condition $c$ holds, and $u$ if $c$ is not valid.

For any sort $D$, the sorts $\mathit{List}(D)$ and $\mathit{Set}(D)$ contain the lists and sets over domain $D$. Prepending an element $d$ to a list $l$ is denoted by $d \triangleright l$. Getting the last element of a list is denoted as $\mathit{rhead}(l)$. The remainder of the list after removing the last element is denoted as $\mathit{rtail}(l)$. The length of a list is denoted by $\#(l)$. Testing whether an element is in a set $s$ is denoted as $d{\in}s$. The set with only element $d$ is denoted by $\{d\}$. Set union is written as $s_1{\cup}s_2$ and set difference as $s_1 \backslash s_2$.

Given two sorts $D_1$ and $D_2$, the sort $D_1{\rightarrow}D_2$ contains all functions from the elements from $D_1$ to elements of $D_2$. We use standard lambda notation to represent functions. E.g. $\lambda x{:}\mathbb{N}.x{+}1$ is the function that adds 1 to its argument. For a function $f$ we use the notation $f[t{\rightarrow}u]$ to represent the function $f$, except that if $f[t{\rightarrow}u]$ is applied to $t$, the value $u$ is returned. We call $f[t{\rightarrow}u]$ a function update.

Besides using standard types and type constructors such as *List* and *Set*, users can define their own sorts. In this thesis we most often use user defined sorts with a finite number of elements. A typical example is the declaration of a sort containing the three aspects $\mathit{green}$, $\mathit{yellow}$ and $\mathit{red}$ of a traffic light.

**sort** $\quad \mathit{Aspect} = \mathbf{struct}\ \mathit{green} \mid \mathit{yellow} \mid \mathit{red};$

A more complex user defined sort that we use is a message containing a number that can either be active or passive. The number in each message can be obtained by applying the function $\mathit{get\_number}$ to a message. The function $\mathit{is\_active}$ is true when applied to a message of the form $\mathit{active}(n)$ and false otherwise.

**sort** $\quad \mathit{Message} = \mathbf{struct}\ \mathit{active}(\mathit{get\_number}{:}\mathbb{N})?\mathit{is\_active} \mid \mathit{passive}(\mathit{get\_number}{:}\mathbb{N});$

Using the **map** keyword elements of data domains can be declared. By introducing an equation the element can be declared equal to some expression. An example of its use is the following. The constant $n$ is declared to be equal to 3 and $f$ is equal to the function that returns false for any natural number.

**map** $\quad n : \mathbb{N};$
$\quad\quad f : \mathbb{N} \rightarrow \mathbb{B};$
**eqn** $\quad n = 3;$
$\quad\quad f = \lambda\, x{:}\mathbb{N}.\mathit{false};$

This concise explanation of data types is enough to understand the specifications presented in this thesis.

The behavior of systems is characterised by atomic actions. Actions can represent any elementary activity. Here, they typically represent setting a traffic light to a particular colour, getting a signal from a sensor or communicating among components. Actions can carry data parameters. For example $\mathit{trig}(\mathit{id}, \mathit{false})$ could typically represent that the sensor with identifier $\mathit{id}$ was not triggered (indicated by the boolean $\mathit{false}$).

In an mCRL2 specification, actions must be declared as indicated below, where the types indicate the sorts of the data parameters that they carry.

**act** $trig : \mathbb{N} \times \mathbb{B};$
$send : Message;$
$my\_turn;$

In the examples in this thesis we have omitted such declarations as they are clear from the context.

If two actions $a$ and $b$ happen at the same time, then this is called a multi-action, which is denoted as $a|b$. The operator '|' is called the multi-action composition operator. Any number of actions can be combined into a multi-action. The order in which the actions occur has no significance. So, $a|b|c$ is the same multi-action as $c|a|b$. The empty multi-action is written as $\tau$. It is an action that can happen, but which cannot directly be observed. It is also called the hidden or internal action. The use of multi-actions can be quite helpful in reducing the state space, as indicated in guideline II in section 6.4.

Actions and multi-actions can be composed to form processes. The choice operator, used as $p + q$ for processes $p$ and $q$, allows the process to choose between two processes. The first action that is done determines the choice. The sequential operator, denoted by a dot ('$\cdot$'), puts two behaviors in sequence. So, the process $a{\cdot}b + c{\cdot}d$ can either perform action $a$ followed by $b$, or $c$ followed by $d$.

The *if-then-else* operator, $c \rightarrow p \diamond q$, allows the condition $c$ to determine whether the process $p$ or $q$ is selected. The else part can always be omitted. We then get the conditional operator of the form $c \rightarrow p$. If $c$ is not valid, this process cannot perform an action. It deadlocks. This does not need to be a problem because using the $+$ operator alternative behavior may be possible.

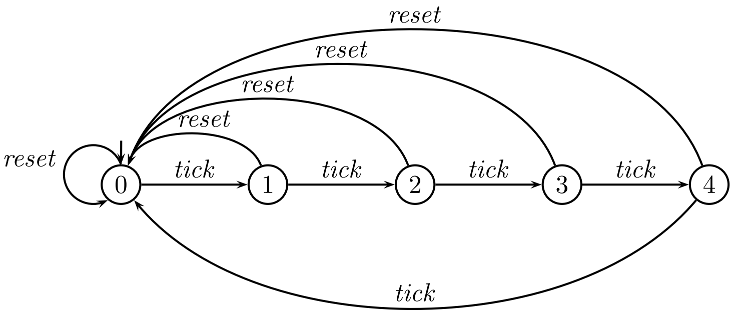


**Figure 2.6** The transition system of the process $Counter$

The following example shows how to specify a simple recursive process. It is declared using the keyword **proc**. It is a timer that cyclically counts up till four using the action *tick*, and can be *reset* at any time. Note that the name of a process, in this case *Counter*, can carry data parameters. The initial state of the process is $Counter(0)$, i.e., the counter starting with argument $0$. Initial states are declared using the keyword **init**. As explained below, we underline actions, if they are not involved in communication between processes.

$$\begin{array}{ll}\textbf{proc} & Counter(n{:}\mathbb{N}) \\ & \quad = (n{<}4) \rightarrow \underline{tick}\cdot Counter(n{+}1) \diamond \underline{tick}\cdot Counter(0) \\ & \quad + \underline{reset}\cdot Counter(0); \\ \textbf{init} & Counter(0);\end{array}$$

In Figure 2.6 the transition system of the counter is depicted. It consists of five states and ten transitions. By following the transitions from state to state a run through the system can be made. Note that many different runs are possible. A transition system represents all possible behaviors of the system, rather than one or a few single runs. The initial state is state $0$, which has a small incoming arrow to indicate this. The transition systems referred to in this article are all generated using the mCRL2 toolset [63].

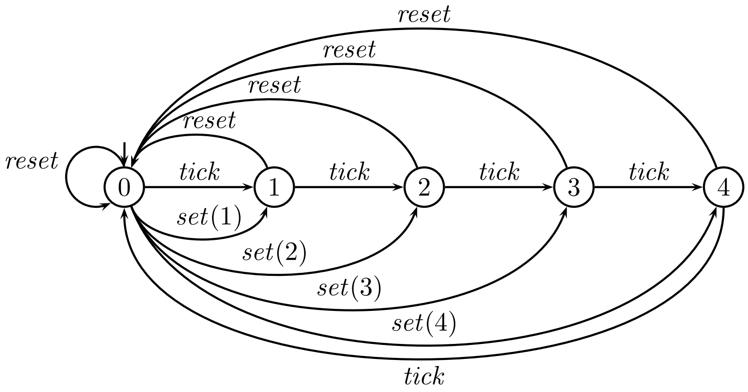


**Figure 2.7** The $Counter$ extended with $set$ transitions

Sometimes, it is required to allow a choice in behavior, depending on data. E.g., for the counter it can be convenient to allow to set it to any value larger than zero and smaller than five. Using the choice operator this can be written as

$$\underline{set}(1)\cdot Counter(1) + \underline{set}(2)\cdot Counter(2) + \underline{set}(3)\cdot Counter(3) + \underline{set}(4)\cdot Counter(4)$$

Especially, for larger values this is inconvenient. Therefore, the sum operator has been introduced. It is written as $\sum_{x:\mathbb{N}} p(x)$ and it represents a choice among all processes $p(x)$ for any value of $x$. The sort $\mathbb{N}$ is just provided here as an example, but can be any arbitrary sort. Note that the sort in the sum operator can be infinite. To generate a finite state space, this infinite range must be restricted, for instance by a condition. The example above uses such a restriction and becomes:

$$\sum_{x:\mathbb{N}}(0{<}x \wedge x{<}5) \rightarrow \underline{set}(x){\cdot}Counter(x)$$

Just for the sake of completeness, we formulate the example of the counter again, but now with this additional option to set the counter, which can only take place if $n$ equals $0$. This example is a very typical sequential process (sequential in the meaning of not parallel). In Figure 2.7 we provide the state space of the extended counter.

**proc** $Counter(n{:}\mathbb{N})$
$= (n{<}4) \rightarrow \underline{tick}{\cdot}Counter(n{+}1) \diamond \underline{tick}{\cdot}Counter(0)$
$+ \sum_{x:\mathbb{N}}(n{\approx}0 \wedge 0{<}x \wedge x{<}5) \rightarrow \underline{set}(x){\cdot}Counter(x)$
$+ \underline{reset}{\cdot}Counter(0);$
**init** $Counter(0);$

Processes can be put in parallel with the parallel operator $\|$ to model a concurrent system. The behavior of $p \parallel q$ represents that the behavior of $p$ and $q$ is parallel. It is an interleaving of the actions of $p$ and $q$ where it is also possible that the actions of $p$ and $q$ happen at the same time in which case a multi-action occurs. So, $a \parallel b$ represents that actions $a$ and $b$ are executed in parallel. This behavior is equal to $a{\cdot}b + b{\cdot}a + a|b$.

Parallel behavior is the second main source of a state space explosion. The number of states of $p \parallel q$ is the product of the number of states of $p$ and $q$. The state space of $n$ processes that each have $m$ states is $m^n$. For $n$ and $m$ larger than 10 this is too big to be stored in the memory of almost any computer in an uncompressed way. Using the allow operator introduced in the next paragraph, the number of reachable states can be reduced substantially. But without care the number of states of parallel systems can easily grow out of control.

In order to let two parallel components communicate, the communication operator $\Gamma_C$ and the allow operator $\nabla_V$ are used where $C$ is a set of communications and $V$ is a set of data free multi-actions. The idea behind communication is that if two actions happen at the same time, and carry the same data parameters, they can communicate to one action. In this article we use the convention that actions with a subscript $r$ (from receive) communicate to actions with a subscript $s$ (from send) into an action with subscript $c$ (from communicate). Typically, we write $\Gamma_{\{a_r|a_s \rightarrow a_c\}}(p \parallel q)$ to allow action $a_r$ to communicate with $a_s$ resulting in $a_c$ in a process $p \parallel q$. In order to make the distinction between internal communicating actions and external actions clearer, we underline all external actions in specifications (but not in the text or in the diagrams). External actions are those actions communicating with entities outside the described system, whereas internal actions happen internally in components of the system or are communications among those components.

To enforce communication, we must also express that actions $a_s$ and $a_r$ cannot happen on their own. The allow operator explicitly allows certain multi-actions to happen, and blocks all others. So, in the example from the previous paragraph, we must add $\nabla_{\{a_c\}}$ to block $a_r$ and $a_s$ enforcing them to communicate into $a_c$. So, a typical expression putting behaviors $p$ and $q$ in parallel, letting them communicate via action $a$, is:

$$\nabla_{\{a_c\}}(\Gamma_{\{a_r|a_s \to a_c\}}(p \parallel q))$$

Of course, more processes can be put in parallel, and more actions can be allowed to communicate.

Actions that are the result of a communication are in general internal actions in the sense that they take place between components of the system and do not communicate with the outside world. Using the hiding operator $\tau_I$ actions can be made invisible. So, for a process that consists of a single action $a$, $\tau_{\{a\}}(a)$ is the empty multi-action $\tau$, an action that does happen, but which cannot directly be observed.

If a system has internal actions, then the behavior can be reduced. For instance in the process $a{\cdot}\tau{\cdot}p$ it is impossible to observe the $\tau$, and this behavior is equivalent to $a{\cdot}p$. The most common behavioral reductions are weak bisimulation and branching bisimulation [67, 82]. We will not explain these equivalences here in detail. For us it suffices to know that they reduce the behavior of a system to a unique minimal transition system preserving the essence of the external behavior. This result is called the transition system modulo weak/branching bisimulation. This reduction is often substantial.

Finally, the mCRL2 toolset supports analyzing systems using a number of nice visualization tools and the possibility of verifying properties of systems using the modal $\mu$-calculus. Since these means were not used in this work we instead refer to [38, 40, 56] for more information.

# Chapter 3

# Formal Methods in the BasiX Project

[1] Analyzing a Controller of a Power Distribution Unit Using Formal Methods. In: Proceedings of the 5th International Conference on Software Testing, Verification and Validation (ICST 2012)

[2] Incorporating Formal Techniques into Industrial Practice: an Experience Report. In: Proceedings of the 9th International Workshop on Formal Engineering Approaches to Software Components and Architectures (FESCA 2012)

## 3.1 Introduction

The BasiX project aims at developing a number of infrastructural services of the X-ray machine, such as the systematic startup/shutdown of devices, installation and upgrading of software. The startup/shutdown service is responsible of providing facilities related to the powering aspects between a central power distribution unit (PDU) and a number of PCs that host the clinical applications through a power and a control network.

The BasiX also provides other important services used for automatic installation of both required operating systems and clinical applications on the PCs through the network. Moreover, it provides generic services for logging to allow easy debugging by field service engineers, and for tracing to facilitate debugging by the in-house developers.

In this chapter we introduce two industrial cases where the ASD technology has been applied to the development of control components. In each case we concentrate on distinct aspects related to the application of ASD to the development process.

The first industrial case is introduced in Section 3.2 and is concerned with detailing the steps performed to specify and formally verify a controller of the PDU that supplies the X-ray machine with the required power. In this industrial case, there was no code generated from the specified models; therefore, the ASD formal techniques were merely used for formal specification and verification.

We exploit the PDU controller design to show how interface and design models were specified using the ASD:Suite, how ASD components interact with one another and what types of formal properties are provided by the ASD technology. As a result, two errors were detected in the design of the controller, although the design was thoroughly reviewed by development team before ASD was used.

The second industrial case is introduced in Section 3.3 and is concerned with detailing the processes accomplished for developing the power control service (PCS). This service is replicated and deployed on the PCs of the X-ray machine and interact with the PDU through a communication network. Through this project we propose a workflow to combine ASD and the test driven development approach, answering a few questions regarding the challenges of incorporating ASD to industrial practices and integrating ASD with other tools and methods being currently applied in industry.

## 3.2 The Power Distribution Unit

### 3.2.1 Introduction

The X-Ray system consists of a number of distributed devices and computers that require a reliable source of power control. The distribution of power to these components is controlled by a power distribution unit (PDU) attached to the main source of power in hospitals.

External users of the system interact with the PDU through an external console, which includes a number of buttons. The console is attached to an embedded controller that controls the flow of power to the components through a number of power taps. The controller communicates with the devices via a network regarding required changes to the powering status of the system. If the PDU does not function correctly components may unpredictably be with or without power when they desire, rendering the system useless or even dangerous.

Throughout this section we illustrate the verification steps of the PDU controller using the ASD approach. Although the design of the controller was thoroughly reviewed, the design included two previously uncovered errors, detected via model checking and specification review enforced by ASD, prior to implementation, at the phase where designers and architects explored various design alternatives.

As will be demonstrated below, specification completeness, specification review and formal behavioral verification provided a key benefit by easily locating design errors in the PDU controller that would be hard to find through conventional testing.

We report about the work accomplished for verifying the controller in this industrial project as follows. Section 3.2.2 introduces the context of the PDU controller. Our experimental method of modeling and verifying the PDU controller is demonstrated in Section 3.2.3, where we further explain the unveiled errors and how precisely they had been discovered. The efforts of modeling and verifying the PDU controller are described in Section 3.2.4.

### 3.2.2 The design description

We provide an overview of the design context, hardware and software components, and the signals exchanged through the system. Figure 3.1 demonstrates the structure of the X-ray system depicting a number of distributed devices and PCs connected to a Power Distribution Unit (PDU).

The PDU is attached to an external power source, via a mains switch. The PDU is responsible for distributing power and related communication signals to the attached devices and PCs. Below we detail these components to the extent relevant for this work.

**The Power Distribution Unit** The PDU interacts with its external users by a user console, which contains a number of buttons: *PowerOn*, *PowerOff* and *EmergencyOff*. The PDU also comprises a base module that hosts a number of power taps. It further houses internal units: a Power Control Unit, which controls the flow of network messages, and a Power Distribution Unit, which controls the distribution of power (Figure 3.1).

Through switching the power taps the PDU controls the flow of power to the devices and the PCs. The type of power taps can either be switchable or permanent. The switchable taps can potentially be switched on/off by the PDU upon requests of external users, issued

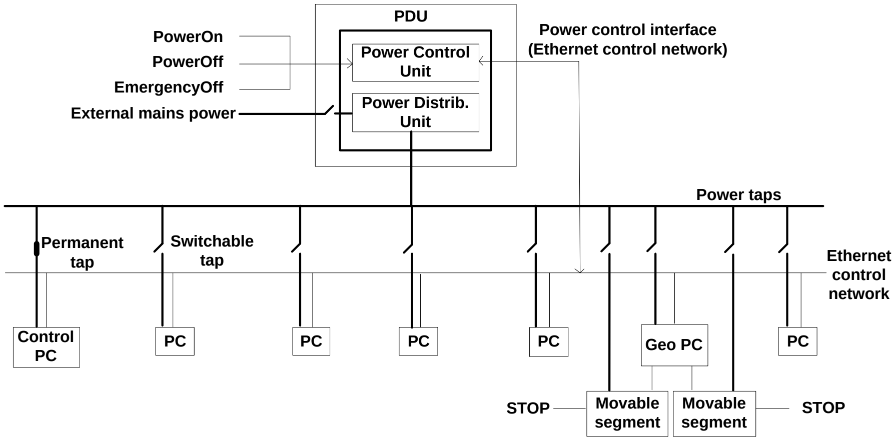


**Figure 3.1** Power, network, and device distribution

by pressing the buttons. For example, when the X-ray system is operational and an external user presses the *PowerOff* button for 3 seconds, all switchable taps are switched off so that all attached components are powered off except those attached to the permanent taps (powering off the system in an orderly fashion).

The permanent taps constantly supply power to a number of components that must always be up-and-running (e.g., for remote access purposes). The permanent taps can also be switched off in some special cases. For example the PDU switches off all taps when the external user presses the *PowerOff* button for 10 seconds (forcing all components to be powered off).

The PDU comprises a controller that includes a state machine for maintaining the states of the system. The state machine is introduced below.

For supplying power to the system in case of failure of the main source of power in the field, an uninterruptible power supply (UPS) is attached to the PDU.

**Devices and PCs** A number of devices and PCs are connected to the PDU, each of which has distinct responsibilities for achieving the required clinical application. All components exhibit the same start-up and shutdown behavior (e.g., powering up, starting the operating system (OS) and the clinical applications, shutting down OS, etc.), controlled systematically by the PDU. The high-level behavior of these devices by means of state machines are introduced below.

All PCs depicted in Figure 3.1 are attached to switchable taps except the ControlPC which is attached to a permanent tap.

The GeoPC (Geometry PC) is responsible for controlling motorized movements of a number of movable parts, such as the table where patients can lay and the X-ray stands. The movable parts are supplied with emergency *Stop* buttons, attached to their bodies. The clinical users can press these buttons to stop any movement in order to avoid any potential damage that might occur because of the motorized movements. Upon pressing a *Stop* button, the GeoPC instructs the PDU to switch off the taps connected to the motor drives of the movable parts.

**External user commands** As a consequence of pressing the buttons on the user console, user commands are generated and received by the PDU controller. The controller processes the commands and, depending on the command and the state, decides to send messages (introduced below) around to the devices and the PCs through the network or to switch the taps on/off.

By pressing the *PowerOn* button, a *powerOn* command is fired. Pressing the *PowerOff* button for 3 seconds generates a *powerOff* command while pushing the button for more than 10 seconds fires a *forcedPowerOff* command. The *EmergencyOff* completely cuts down any source of power (including the UPS) to the system, via an internal switch, positioned inside the PDU. The *EmergencyOff* button is pressed to ensure that the system is immediately powered off in the presence of calamities.

**Internal system messages** The PDU can send and receive the following messages through the network: *shutdown*, *restart*, *controlPowerOff*, and *stop*.

The *shutdown* and *restart* are broadcast messages sent from the PDU to the PCs. The *shutdown* message instructs the devices to gradually shutdown their running applications and then the operating systems. The *restart* message requests the PCs to reboot their operating systems.

*ControlPowerOff* is a message sent from the ControlPC to the PDU, while *stop* is a message sent from the GeoPC to the PDU. Through the *controlPowerOff* message users of the ControlPC can instruct the PDU to systematically power off the entire system. The *stop* signal is sent by the GeoPC when any of the *Stop* buttons on the movable parts is pressed, so the PDU directly switches off all taps that supply the motor drives. The motor drives are powered on again when the user presses the *powerOn* button on the console.

**The state transition diagrams** Figure 3.2 depicts the high-level behavior of the PDU controller from a system-level perception after implementation details are abstracted away [60]. The user and system signals introduced earlier constitute stimuli and responses of the state machine. In addition to these events, the *PDUswitchOn* and *PDUswitchOff* events are used for modeling purposes to indicate switching the external mains switch on and off, respectively.

The effects of these signals on the behavior of the system differ upon the present state of the PDU. For example, when the PDU is in *System_Standby* and the *powerOn* signal is

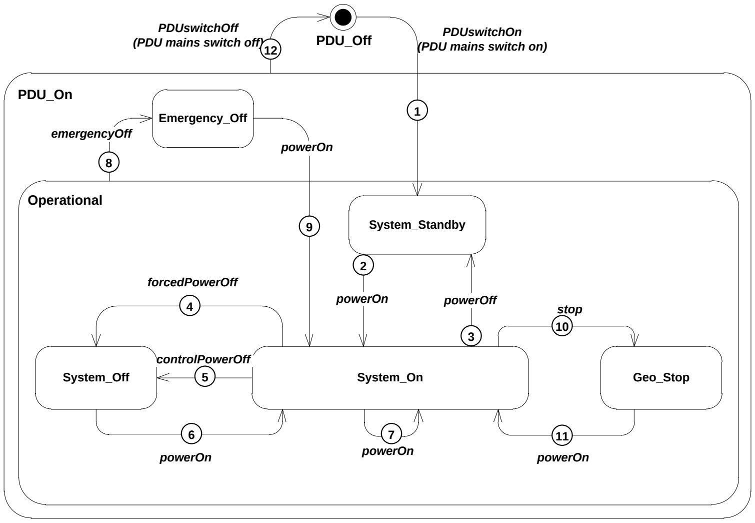


**Figure 3.2** The high-level behavior of the PDU [60]

received, it switches on the switchable taps, so that the attached PCs and devices start up. But, if the PDU is in the *System_On* state and the *powerOn* signal is received, then the PDU broadcasts the *restart* message across the network. The detailed activities required for each transition of the state machine of Figure 3.2 are depicted in Table 3.1.

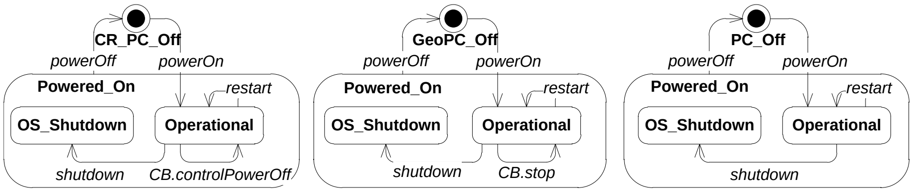


**Figure 3.3** The external behavior of all PCs

Here, the PDU, the devices and the PCs are assumed to be well functioning. All error handling details or recovery operations are removed from the state machine.

The state machine of Figure 3.2 depicts only the *stable* states of the system. The transiting states between any two stable states are excluded from the diagram. For example, when the PDU is off and then is switched on, the PDU transits to the *System_Standby* state, where the ControlPC is assumed to be successfully started and fully operational. The intermediate transiting state between the *PDU_Off* and the *System_Standby* state that

| Transition | Activity |
|---|---|
| 1 | Boot PDU; the PDU switches on all permanent power taps; the ControlPC is operational. |
| 2 | The PDU switches on all switchable taps, one by one to avoid a big inrush current; all devices are operational. |
| 3 | The PDU broadcasts a "shutdown" message to shutdown all control devices except the ControlPC; the PDU switches off all switchable taps when power load is below a threshold or when the timer expires. |
| 4 | The PDU immediately switches off all power taps. |
| 5 | The PDU broadcasts a "shutdown" message to shutdown all control devices including the ControlPC; the PDU switches off all taps when power load is below threshold or when the timer expires. |
| 6 | The PDU switches on all taps, one by one to avoid a big inrush current; all devices are started (all devices are operational including the ControlPC). |
| 7 | The PDU broadcasts a "restart" message; the operating systems of all control devices are restarted. |
| 8 | Disconnect the PDU internal power bus and UPS. |
| 9 | The PDU switches on all taps, one by one to avoid a big inrush current; all devices are started (all devices are operational including the ControlPC). |
| 10 | The PDU switches off the power taps that supply motor drives of movable parts. |
| 11 | The PDU switch on the power taps that supply motor drives of movable parts. |
| 12 | The PDU is switched off; all taps are switched off. |

**Table 3.1** The activities required for each transition of the PDU state machine [60]

ensures that the ControlPC is fully operational is removed. The same assumption applies to other states. For example, the *System_On* state implies the situation where all PCs are fully operational. All intermediate transiting states, which ensure that the PCs are fully operational, leading to *System_On* are removed.

We introduce the external behavior of the PCs with respect to the PDU. All PCs exhibit almost the same startup and shutdown behavior, see Figure 3.3. Initially, they are all in the *Off* state. Once a tap of a PC is switched on, the PC automatically launches its operating system and then starts up its clinical applications. When the applications are successfully started, the PC transits to the *Operational* state; this is indicated by the *powerOn* transition from the *Off* to the *Operational* states. If a PC receives a *restart* message at the *Operational* state, it restarts the operating system and the applications. But, if the *shutdown* message is received, the PC closes all running applications and shuts the operating system down.

The ControlPC and the GeoPC have two additional transitions:*CB.controlPowerOff* and *CB.stop*. The *CB.controlPowerOff* and *CB.stop* signals are callback (CB) events sent to the PDU, where the first indicates that the user of the ControlPC has requested the PDU to entirely power off the system, and the second indicates that the *Stop* button on a movable segment has been pressed.

The movable parts can be powered on or off by the PDU. They don't receive or send the PDU any signal through the network. The behavior of these segments is straightforward, and hence the corresponding specification is omitted (two actions of *powerOff* and *powerOn* affecting two states *Segment*$_x$*_Off* and *Segment*$_x$*_On*, where $_x$ is the device id).

### 3.2.3 Modeling and analyzing the PDU behavior

In this section our experimental methodology is sketched, summarizing the series of steps followed through the experiment of verifying the PDU design. We used the ASD:Suite version 6.1.0 for describing the behavior of our components. The features supplied by this version seemed to be a good fit to our aim at modeling the PDU behavior.

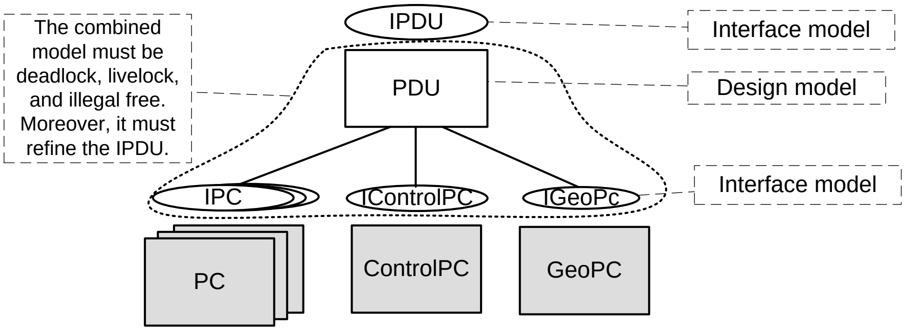
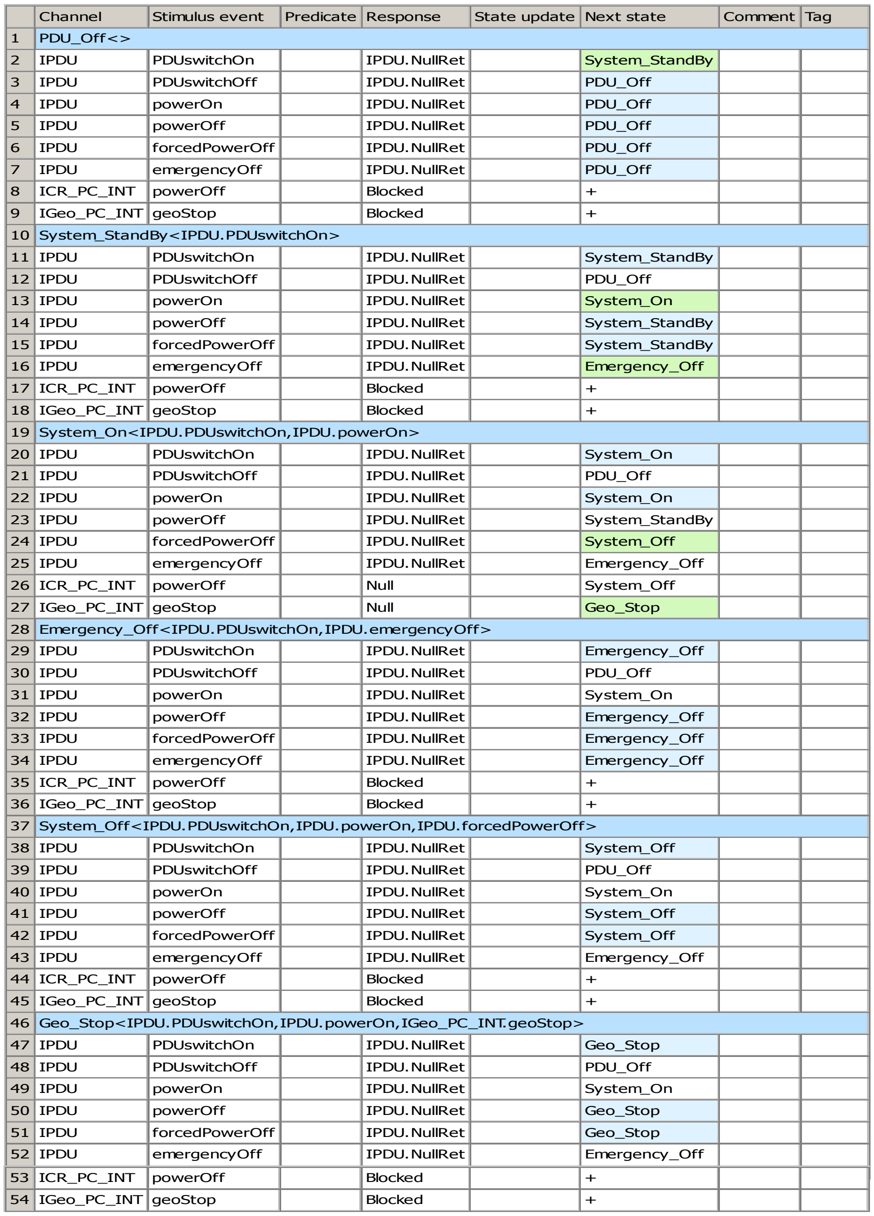

The combined model must be deadlock, livelock, and illegal free. Moreover, it must refine the IPDU.
IPDU
Interface model
PDU
Design model
IPC
IControlPC
IGeoPc
Interface model
PC
ControlPC
GeoPC

**Figure 3.4** The structure of ASD models

The structure of the ASD models of the PDU is depicted in Figure 3.4. Following the ASD recipe we modeled the external behavior of the PDU first. Then, we separately described the external behavior of the PCs located at the subsequent level of the PDU. After that, we modeled the PDU design such that it refines the IPDU specification and includes all interactions with the PCs. Below we individually introduce these models in more detail.

**The external behavior of the PDU (the IPDU interface model).** The specification in Figure 3.5 describes the external behavior with respect to the external users of the PDU, according to the original state machine of Figure 3.2. The specification is described using an ASD interface model. The specification is straightforward and self-explainable.

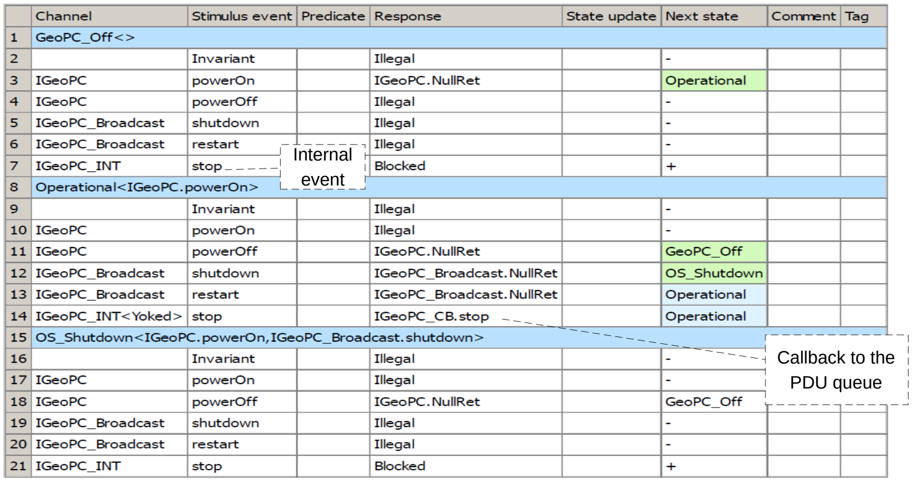


| | Channel | Stimulus event | Predicate | Response | State update | Next state | Comment | Tag |
|---|---|---|---|---|---|---|---|---|
| 1 | PDU_Off<> | | | | | | | |
| 2 | IPDU | PDUswitchOn | | IPDU.NullRet | | System_StandBy | | |
| 3 | IPDU | PDUswitchOff | | IPDU.NullRet | | PDU_Off | | |
| 4 | IPDU | powerOn | | IPDU.NullRet | | PDU_Off | | |
| 5 | IPDU | powerOff | | IPDU.NullRet | | PDU_Off | | |
| 6 | IPDU | forcedPowerOff | | IPDU.NullRet | | PDU_Off | | |
| 7 | IPDU | emergencyOff | | IPDU.NullRet | | PDU_Off | | |
| 8 | ICR_PC_INT | powerOff | | Blocked | | + | | |
| 9 | IGeo_PC_INT | geoStop | | Blocked | | + | | |
| 10 | System_StandBy<IPDU.PDUswitchOn> | | | | | | | |
| 11 | IPDU | PDUswitchOn | | IPDU.NullRet | | System_StandBy | | |
| 12 | IPDU | PDUswitchOff | | IPDU.NullRet | | PDU_Off | | |
| 13 | IPDU | powerOn | | IPDU.NullRet | | System_On | | |
| 14 | IPDU | powerOff | | IPDU.NullRet | | System_StandBy | | |
| 15 | IPDU | forcedPowerOff | | IPDU.NullRet | | System_StandBy | | |
| 16 | IPDU | emergencyOff | | IPDU.NullRet | | Emergency_Off | | |
| 17 | ICR_PC_INT | powerOff | | Blocked | | + | | |
| 18 | IGeo_PC_INT | geoStop | | Blocked | | + | | |
| 19 | System_On<IPDU.PDUswitchOn,IPDU.powerOn> | | | | | | | |
| 20 | IPDU | PDUswitchOn | | IPDU.NullRet | | System_On | | |
| 21 | IPDU | PDUswitchOff | | IPDU.NullRet | | PDU_Off | | |
| 22 | IPDU | powerOn | | IPDU.NullRet | | System_On | | |
| 23 | IPDU | powerOff | | IPDU.NullRet | | System_StandBy | | |
| 24 | IPDU | forcedPowerOff | | IPDU.NullRet | | System_Off | | |
| 25 | IPDU | emergencyOff | | IPDU.NullRet | | Emergency_Off | | |
| 26 | ICR_PC_INT | powerOff | | Null | | System_Off | | |
| 27 | IGeo_PC_INT | geoStop | | Null | | Geo_Stop | | |
| 28 | Emergency_Off<IPDU.PDUswitchOn,IPDU.emergencyOff> | | | | | | | |
| 29 | IPDU | PDUswitchOn | | IPDU.NullRet | | Emergency_Off | | |
| 30 | IPDU | PDUswitchOff | | IPDU.NullRet | | PDU_Off | | |
| 31 | IPDU | powerOn | | IPDU.NullRet | | System_On | | |
| 32 | IPDU | powerOff | | IPDU.NullRet | | Emergency_Off | | |
| 33 | IPDU | forcedPowerOff | | IPDU.NullRet | | Emergency_Off | | |
| 34 | IPDU | emergencyOff | | IPDU.NullRet | | Emergency_Off | | |
| 35 | ICR_PC_INT | powerOff | | Blocked | | + | | |
| 36 | IGeo_PC_INT | geoStop | | Blocked | | + | | |
| 37 | System_Off<IPDU.PDUswitchOn,IPDU.powerOn,IPDU.forcedPowerOff> | | | | | | | |
| 38 | IPDU | PDUswitchOn | | IPDU.NullRet | | System_Off | | |
| 39 | IPDU | PDUswitchOff | | IPDU.NullRet | | PDU_Off | | |
| 40 | IPDU | powerOn | | IPDU.NullRet | | System_On | | |
| 41 | IPDU | powerOff | | IPDU.NullRet | | System_Off | | |
| 42 | IPDU | forcedPowerOff | | IPDU.NullRet | | System_Off | | |
| 43 | IPDU | emergencyOff | | IPDU.NullRet | | Emergency_Off | | |
| 44 | ICR_PC_INT | powerOff | | Blocked | | + | | |
| 45 | IGeo_PC_INT | geoStop | | Blocked | | + | | |
| 46 | Geo_Stop<IPDU.PDUswitchOn,IPDU.powerOn,IGeo_PC_INT.geoStop> | | | | | | | |
| 47 | IPDU | PDUswitchOn | | IPDU.NullRet | | Geo_Stop | | |
| 48 | IPDU | PDUswitchOff | | IPDU.NullRet | | PDU_Off | | |
| 49 | IPDU | powerOn | | IPDU.NullRet | | System_On | | |
| 50 | IPDU | powerOff | | IPDU.NullRet | | Geo_Stop | | |
| 51 | IPDU | forcedPowerOff | | IPDU.NullRet | | Geo_Stop | | |
| 52 | IPDU | emergencyOff | | IPDU.NullRet | | Emergency_Off | | |
| 53 | ICR_PC_INT | powerOff | | Blocked | | + | | |
| 54 | IGeo_PC_INT | geoStop | | Blocked | | + | | |

**Figure 3.5** The external behavior of the PDU towards the external users

Internal events of the PDU interface model represent detailed activities that may internally occur by the implemented system, not visible to the external world. For example, rule case 26 indicates that something internally might happen in the system that lead the PDU to transit to the *System_Off* state. The detailed behavior that matches this internal event in the design of the PDU is that the ControlPC may send the *controlPowerOff* callback to the PDU and then the PDU will process this callback by sending the shutdown message and powering off all PCs.

During the refinement check using FDR2 established by the ASD:Suite, all events not visible to the client component will be hidden from the interface model: the callback events *ICR_PC_INT.powerOff* and the *IGeo_PC_INT.geoStop*, for instance. To reflect these internal activities on the external specification, one needs to add visible callbacks to the interface. For example, we can indicate to the user that the system is powered off in the *System_On* state due to internal activities by replacing the *Null* response of rule case 26 by an extra callback (say *IUserIndicationCB.systemOff*). This way the deep internal modes of the system can be reflected on the external specification, making the specification more strict. We omit such extensions from our specification since we are interested more in verifying the correctness of the state machine of the internal PDU design.

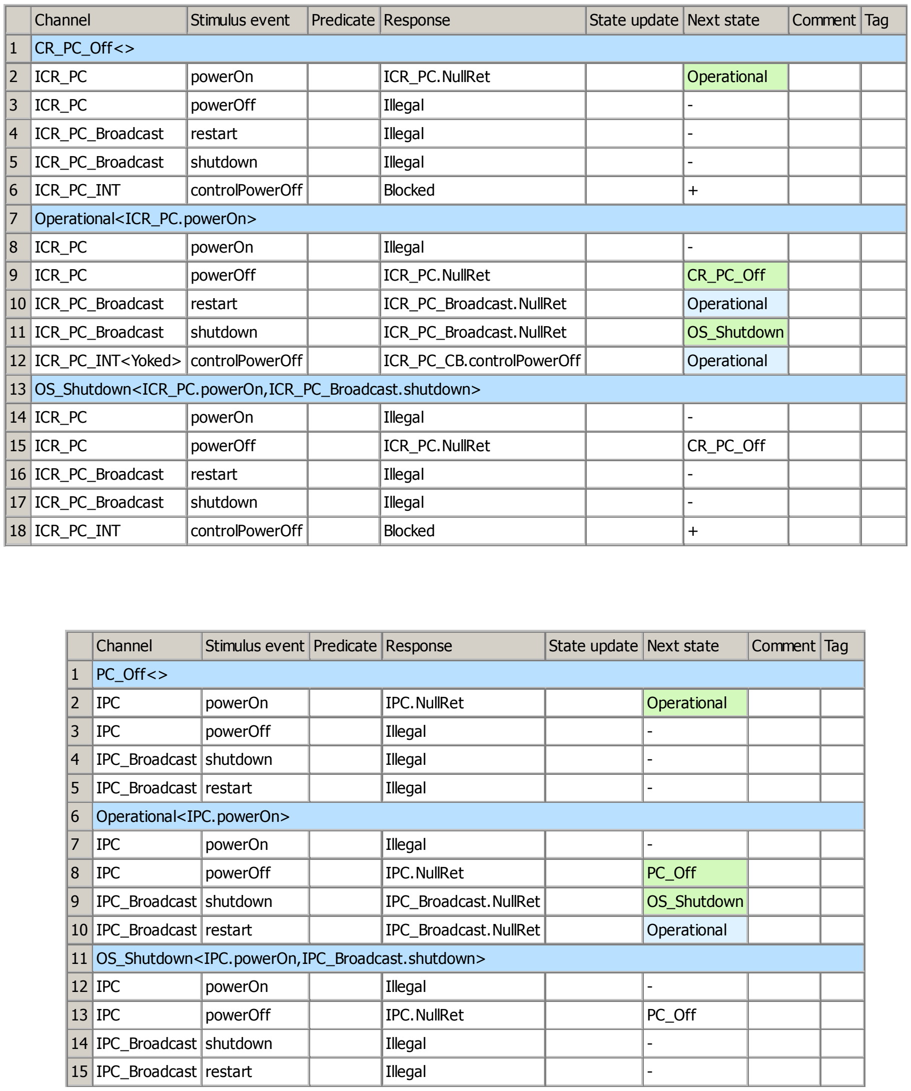


| | Channel | Stimulus event | Predicate | Response | State update | Next state | Comment | Tag |
|---|---|---|---|---|---|---|---|---|
| 1 | GeoPC_Off<> | | | | | | | |
| 2 | | Invariant | | Illegal | | - | | |
| 3 | IGeoPC | powerOn | | IGeoPC.NullRet | | Operational | | |
| 4 | IGeoPC | powerOff | | Illegal | | - | | |
| 5 | IGeoPC_Broadcast | shutdown | | Illegal | | - | | |
| 6 | IGeoPC_Broadcast | restart | | Illegal | | - | | |
| 7 | IGeoPC_INT | stop | | Blocked | | + | | |
| 8 | Operational<IGeoPC.powerOn> | | | | | | | |
| 9 | | Invariant | | Illegal | | - | | |
| 10 | IGeoPC | powerOn | | Illegal | | - | | |
| 11 | IGeoPC | powerOff | | IGeoPC.NullRet | | GeoPC_Off | | |
| 12 | IGeoPC_Broadcast | shutdown | | IGeoPC_Broadcast.NullRet | | OS_Shutdown | | |
| 13 | IGeoPC_Broadcast | restart | | IGeoPC_Broadcast.NullRet | | Operational | | |
| 14 | IGeoPC_INT<Yoked> | stop | | IGeoPC_CB.stop | | Operational | | |
| 15 | OS_Shutdown<IGeoPC.powerOn,IGeoPC_Broadcast.shutdown> | | | | | | | |
| 16 | | Invariant | | Illegal | | - | | |
| 17 | IGeoPC | powerOn | | Illegal | | - | | |
| 18 | IGeoPC | powerOff | | IGeoPC.NullRet | | GeoPC_Off | | |
| 19 | IGeoPC_Broadcast | shutdown | | Illegal | | - | | |
| 20 | IGeoPC_Broadcast | restart | | Illegal | | - | | |
| 21 | IGeoPC_INT | stop | | Blocked | | + | | |

**Figure 3.6** The ASD interface model of the GeoPC

**The external behavior of the PCs.** The external behavior of the PCs was separately described using ASD interface models in Figure 3.6 and 3.7, matching the state machines introduced earlier in Figure 3.3.

All PCs receive a number of messages synchronously via a number of channels: *ICR_PC*, *ICR_PC_Broadcast* in the ControlPC interface, for instance. The ControlPC interface

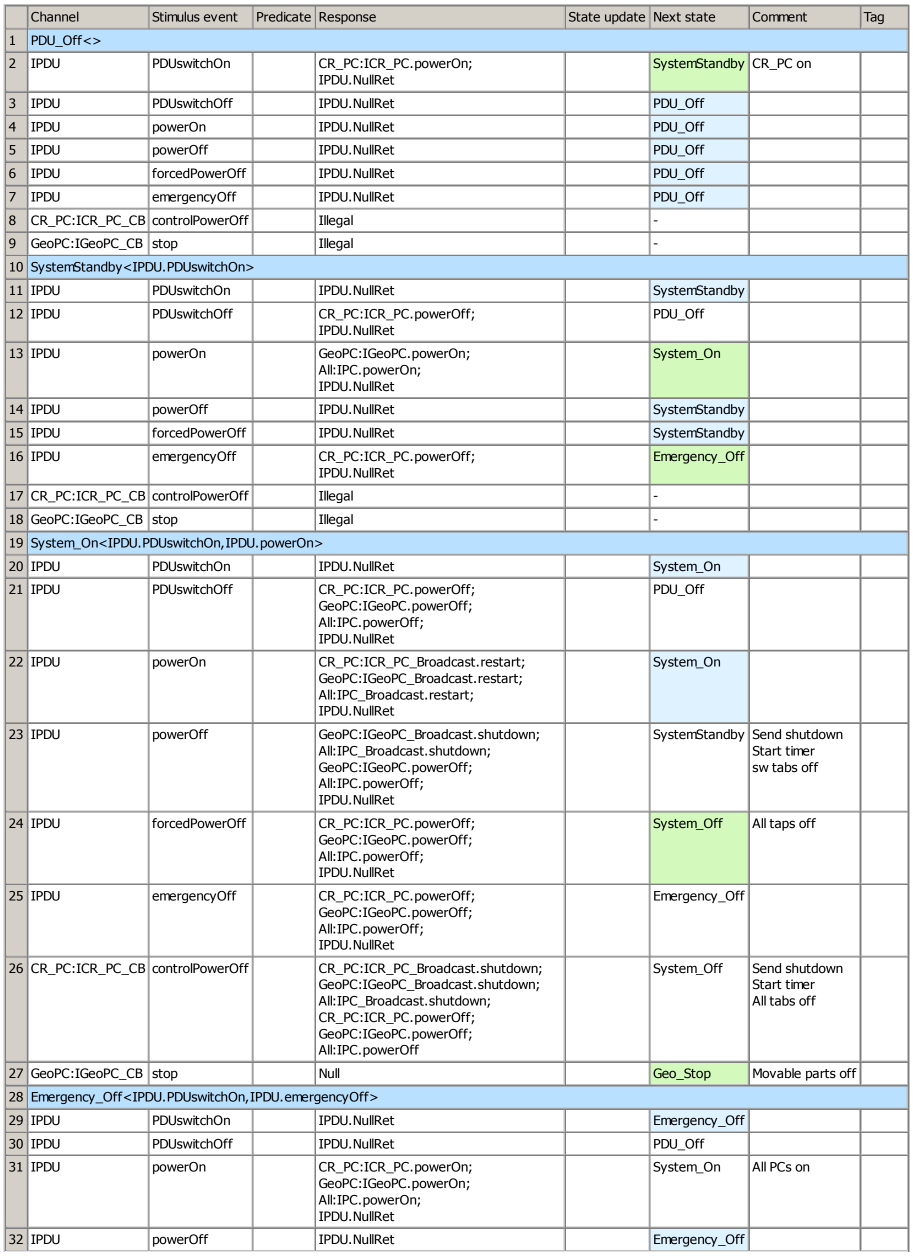


| | Channel | Stimulus event | Predicate | Response | State update | Next state | Comment | Tag |
|---|---|---|---|---|---|---|---|---|
| 1 | CR_PC_Off<> | | | | | | | |
| 2 | ICR_PC | powerOn | | ICR_PC.NullRet | | Operational | | |
| 3 | ICR_PC | powerOff | | Illegal | | - | | |
| 4 | ICR_PC_Broadcast | restart | | Illegal | | - | | |
| 5 | ICR_PC_Broadcast | shutdown | | Illegal | | - | | |
| 6 | ICR_PC_INT | controlPowerOff | | Blocked | | + | | |
| 7 | Operational<ICR_PC.powerOn> | | | | | | | |
| 8 | ICR_PC | powerOn | | Illegal | | - | | |
| 9 | ICR_PC | powerOff | | ICR_PC.NullRet | | CR_PC_Off | | |
| 10 | ICR_PC_Broadcast | restart | | ICR_PC_Broadcast.NullRet | | Operational | | |
| 11 | ICR_PC_Broadcast | shutdown | | ICR_PC_Broadcast.NullRet | | OS_Shutdown | | |
| 12 | ICR_PC_INT<Yoked> | controlPowerOff | | ICR_PC_CB.controlPowerOff | | Operational | | |
| 13 | OS_Shutdown<ICR_PC.powerOn,ICR_PC_Broadcast.shutdown> | | | | | | | |
| 14 | ICR_PC | powerOn | | Illegal | | - | | |
| 15 | ICR_PC | powerOff | | ICR_PC.NullRet | | CR_PC_Off | | |
| 16 | ICR_PC_Broadcast | restart | | Illegal | | - | | |
| 17 | ICR_PC_Broadcast | shutdown | | Illegal | | - | | |
| 18 | ICR_PC_INT | controlPowerOff | | Blocked | | + | | |

| | Channel | Stimulus event | Predicate | Response | State update | Next state | Comment | Tag |
|---|---|---|---|---|---|---|---|---|
| 1 | PC_Off<> | | | | | | | |
| 2 | IPC | powerOn | | IPC.NullRet | | Operational | | |
| 3 | IPC | powerOff | | Illegal | | - | | |
| 4 | IPC_Broadcast | shutdown | | Illegal | | - | | |
| 5 | IPC_Broadcast | restart | | Illegal | | - | | |
| 6 | Operational<IPC.powerOn> | | | | | | | |
| 7 | IPC | powerOn | | Illegal | | - | | |
| 8 | IPC | powerOff | | IPC.NullRet | | PC_Off | | |
| 9 | IPC_Broadcast | shutdown | | IPC_Broadcast.NullRet | | OS_Shutdown | | |
| 10 | IPC_Broadcast | restart | | IPC_Broadcast.NullRet | | Operational | | |
| 11 | OS_Shutdown<IPC.powerOn,IPC_Broadcast.shutdown> | | | | | | | |
| 12 | IPC | powerOn | | Illegal | | - | | |
| 13 | IPC | powerOff | | IPC.NullRet | | PC_Off | | |
| 14 | IPC_Broadcast | shutdown | | Illegal | | - | | |
| 15 | IPC_Broadcast | restart | | Illegal | | - | | |

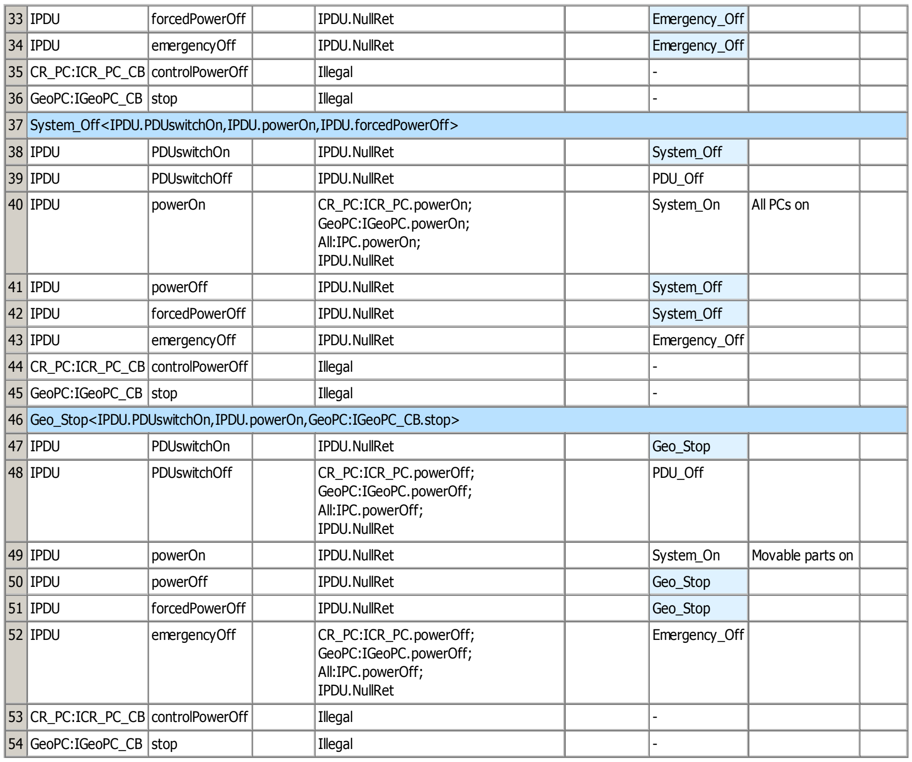
**Figure 3.7** External specification of the ControlPC and the normal PC

includes one internal event that models the internal behavior of powering off the entire system request. The internal event is ‘yoked’ in rule case 12 which means that sending the *ICR_PC_CB.powerOff* callback to the queue of the PDU is restricted (the yoking threshold



| | Channel | Stimulus event | Predicate | Response | State update | Next state | Comment | Tag |
|---|---|---|---|---|---|---|---|---|
| 1 | PDU_Off<> | | | | | | | |
| 2 | IPDU | PDUswitchOn | | CR_PC:ICR_PC.powerOn;<br>IPDU.NullRet | | SystemStandby | CR_PC on | |
| 3 | IPDU | PDUswitchOff | | IPDU.NullRet | | PDU_Off | | |
| 4 | IPDU | powerOn | | IPDU.NullRet | | PDU_Off | | |
| 5 | IPDU | powerOff | | IPDU.NullRet | | PDU_Off | | |
| 6 | IPDU | forcedPowerOff | | IPDU.NullRet | | PDU_Off | | |
| 7 | IPDU | emergencyOff | | IPDU.NullRet | | PDU_Off | | |
| 8 | CR_PC:ICR_PC_CB | controlPowerOff | | Illegal | | - | | |
| 9 | GeoPC:IGeoPC_CB | stop | | Illegal | | - | | |
| 10 | SystemStandby<IPDU.PDUswitchOn> | | | | | | | |
| 11 | IPDU | PDUswitchOn | | IPDU.NullRet | | SystemStandby | | |
| 12 | IPDU | PDUswitchOff | | CR_PC:ICR_PC.powerOff;<br>IPDU.NullRet | | PDU_Off | | |
| 13 | IPDU | powerOn | | GeoPC:IGeoPC.powerOn;<br>All:IPC.powerOn;<br>IPDU.NullRet | | System_On | | |
| 14 | IPDU | powerOff | | IPDU.NullRet | | SystemStandby | | |
| 15 | IPDU | forcedPowerOff | | IPDU.NullRet | | SystemStandby | | |
| 16 | IPDU | emergencyOff | | CR_PC:ICR_PC.powerOff;<br>IPDU.NullRet | | Emergency_Off | | |
| 17 | CR_PC:ICR_PC_CB | controlPowerOff | | Illegal | | - | | |
| 18 | GeoPC:IGeoPC_CB | stop | | Illegal | | - | | |
| 19 | System_On<IPDU.PDUswitchOn,IPDU.powerOn> | | | | | | | |
| 20 | IPDU | PDUswitchOn | | IPDU.NullRet | | System_On | | |
| 21 | IPDU | PDUswitchOff | | CR_PC:ICR_PC.powerOff;<br>GeoPC:IGeoPC.powerOff;<br>All:IPC.powerOff;<br>IPDU.NullRet | | PDU_Off | | |
| 22 | IPDU | powerOn | | CR_PC:ICR_PC_Broadcast.restart;<br>GeoPC:IGeoPC_Broadcast.restart;<br>All:IPC_Broadcast.restart;<br>IPDU.NullRet | | System_On | | |
| 23 | IPDU | powerOff | | GeoPC:IGeoPC_Broadcast.shutdown;<br>All:IPC_Broadcast.shutdown;<br>GeoPC:IGeoPC.powerOff;<br>All:IPC.powerOff;<br>IPDU.NullRet | | SystemStandby | Send shutdown<br>Start timer<br>sw tabs off | |
| 24 | IPDU | forcedPowerOff | | CR_PC:ICR_PC.powerOff;<br>GeoPC:IGeoPC.powerOff;<br>All:IPC.powerOff;<br>IPDU.NullRet | | System_Off | All taps off | |
| 25 | IPDU | emergencyOff | | CR_PC:ICR_PC.powerOff;<br>GeoPC:IGeoPC.powerOff;<br>All:IPC.powerOff;<br>IPDU.NullRet | | Emergency_Off | | |
| 26 | CR_PC:ICR_PC_CB | controlPowerOff | | CR_PC:ICR_PC_Broadcast.shutdown;<br>GeoPC:IGeoPC_Broadcast.shutdown;<br>All:IPC_Broadcast.shutdown;<br>CR_PC:ICR_PC.powerOff;<br>GeoPC:IGeoPC.powerOff;<br>All:IPC.powerOff | | System_Off | Send shutdown<br>Start timer<br>All tabs off | |
| 27 | GeoPC:IGeoPC_CB | stop | | Null | | Geo_Stop | Movable parts off | |
| 28 | Emergency_Off<IPDU.PDUswitchOn,IPDU.emergencyOff> | | | | | | | |
| 29 | IPDU | PDUswitchOn | | IPDU.NullRet | | Emergency_Off | | |
| 30 | IPDU | PDUswitchOff | | IPDU.NullRet | | PDU_Off | | |
| 31 | IPDU | powerOn | | CR_PC:ICR_PC.powerOn;<br>GeoPC:IGeoPC.powerOn;<br>All:IPC.powerOn;<br>IPDU.NullRet | | System_On | All PCs on | |
| 32 | IPDU | powerOff | | IPDU.NullRet | | Emergency_Off | | |

is 1 so that only one *ICR_PC_CB.powerOff* is allowed at a time in the queue of the PDU).



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 33 | IPDU | forcedPowerOff | | IPDU.NullRet | | Emergency_Off | | |
| 34 | IPDU | emergencyOff | | IPDU.NullRet | | Emergency_Off | | |
| 35 | CR_PC:ICR_PC_CB | controlPowerOff | | Illegal | | - | | |
| 36 | GeoPC:IGeoPC_CB | stop | | Illegal | | - | | |
| 37 | System_Off<IPDU.PDUswitchOn,IPDU.powerOn,IPDU.forcedPowerOff> | | | | | | | |
| 38 | IPDU | PDUswitchOn | | IPDU.NullRet | | System_Off | | |
| 39 | IPDU | PDUswitchOff | | IPDU.NullRet | | PDU_Off | | |
| 40 | IPDU | powerOn | | CR_PC:ICR_PC.powerOn;<br>GeoPC:IGeoPC.powerOn;<br>All:IPC.powerOn;<br>IPDU.NullRet | | System_On | All PCs on | |
| 41 | IPDU | powerOff | | IPDU.NullRet | | System_Off | | |
| 42 | IPDU | forcedPowerOff | | IPDU.NullRet | | System_Off | | |
| 43 | IPDU | emergencyOff | | IPDU.NullRet | | Emergency_Off | | |
| 44 | CR_PC:ICR_PC_CB | controlPowerOff | | Illegal | | - | | |
| 45 | GeoPC:IGeoPC_CB | stop | | Illegal | | - | | |
| 46 | Geo_Stop<IPDU.PDUswitchOn,IPDU.powerOn,GeoPC:IGeoPC_CB.stop> | | | | | | | |
| 47 | IPDU | PDUswitchOn | | IPDU.NullRet | | Geo_Stop | | |
| 48 | IPDU | PDUswitchOff | | CR_PC:ICR_PC.powerOff;<br>GeoPC:IGeoPC.powerOff;<br>All:IPC.powerOff;<br>IPDU.NullRet | | PDU_Off | | |
| 49 | IPDU | powerOn | | IPDU.NullRet | | System_On | Movable parts on | |
| 50 | IPDU | powerOff | | IPDU.NullRet | | Geo_Stop | | |
| 51 | IPDU | forcedPowerOff | | IPDU.NullRet | | Geo_Stop | | |
| 52 | IPDU | emergencyOff | | CR_PC:ICR_PC.powerOff;<br>GeoPC:IGeoPC.powerOff;<br>All:IPC.powerOff;<br>IPDU.NullRet | | Emergency_Off | | |
| 53 | CR_PC:ICR_PC_CB | controlPowerOff | | Illegal | | - | | |
| 54 | GeoPC:IGeoPC_CB | stop | | Illegal | | - | | |

**Figure 3.8** The specification of the PDU state machine

The specification of the PDU controller was described using an ASD design model, matching the original state machine introduced in Figure 3.2. The complete specification is depicted in Figure 3.8.

**Modeling the behavior of the PDU controller** Using the ASD:Suite, we included the used interface models of the PCs and explicitly specify the number of instances of each interface model before describing the behavior of the PDU design model. Obviously, our design model includes one instance of the ControlPC interface model, one instance of the GeoPC interface model and five instances of the PC interface models.

To give an example of the usage of these instances in the ASD:Suite consider rule case 26 of Figure 3.8. The rule case specifies that when the PDU controller receives the *controlPowerOff* asynchronous event from the *CR_PC:ICR_PC_CB* interface (via its queue), it executes a list of responses one by one until completion. For example, the response *CR_PC:ICR_PC_Broadcast.shutdown* denotes sending the *shutdown* message to the ControlPC instance via the *ICR_PC_Broadcast* channel synchronously (note that all calls from client to used components are synchronous in ASD). Similarly, the synchronous response *All:IPC.powerOff* in rule case 26 implicitly denotes powering off all five PCs sequentially one by one, i.e., *PC1.IPC.powerOff*, .., *PC5.IPC.powerOff*.

The *shutdown* message intended for all PCs has to be different than the *shutdown* message intended for all other PCs excluding the ControlPC. But, in our model we don't use distinct events. Instead, we synchronously send the message to the intended PCs depending on the state. For example, rule case 23 depicts sending the *shutdown* message to the GeoPC and the normal PCs, while rule case 26 depicts sending the message to all PCs.

During the specification process of the PDU design model, a number of key important decisions had been discussed early and considered carefully. These decisions had mainly been raised because the ASD specification process forces specification completeness, by filling-in and thinking about every possible stimulus in every table. Since the original state machine of the PDU is not complete, in the sense that not all external calls or internal callback stimuli events are depicted in every state, a decision was considered to initially assign the *Illegal* response to every internal callback stimulus received from the PCs if the stimulus does not appear in a state of the original state machine.

For example, we assign *Illegal* responses to the *controlPowerOff* and the *stop* callback stimuli in all states except *System_On* (see rule cases 26 and 27 in Figure 3.8). Similarly, all external user commands not present in a state are ignored, i.e., they make a self-transition in the state (see rule case 32 in Figure 3.8, for example). This includes switching on the PDU even if it is already switched on.

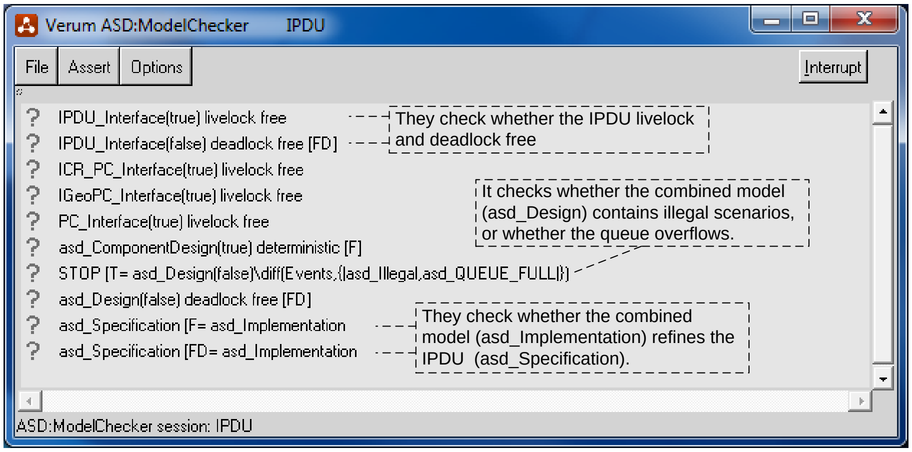
### Formal verification of the PDU controller

Upon the completion of all ASD models, the formal verification process using model checking was started. Figure 3.9 depicts a screenshot of the formal checks performed remotely by the FDR2 model checker using the ASD:Suite.

The first and the second properties check whether the IPDU interface model is livelock and deadlock free. The third, fourth and fifth properties verify that the interfaces of the PCs are livelock free. Verifying the deadlock freedom can be established for each interface model separately using the ASD:Suite.

The sixth property checks whether the combined model is a deterministic design. The purpose of this check is to prevent ambiguities in the generated source code when compiled with the rest of the product code.

The seventh property searches for illegal and queue overflow scenarios in the combined model (asd_Design). The eighth property verifies that the combined model is deadlock free. While the last two properties are used to check whether the combined model (asd_Implementation) refines the IPDU interface model (asd_Specification), under both the Failure and Failures-Divergence models.



```
? IPDU_Interface(true) livelock free
? IPDU_Interface(false) deadlock free [FD]
? ICR_PC_Interface(true) livelock free
? IGeoPC_Interface(true) livelock free
? PC_Interface(true) livelock free
? asd_ComponentDesign(true) deterministic [F]
? STOP [T= asd_Design(false)\diff(Events,{|asd_Illegal,asd_QUEUE_FULL|})
? asd_Design(false) deadlock free [FD]
? asd_Specification [F= asd_Implementation
? asd_Specification [FD= asd_Implementation
ASD:ModelChecker session: IPDU
```

They check whether the IPDU livelock and deadlock free

It checks whether the combined model (asd_Design) contains illegal scenarios, or whether the queue overflows.

They check whether the combined model (asd_Implementation) refines the IPDU (asd_Specification).

**Figure 3.9** Formal checks performed by FDR2 for the behavioral verification

We performed the behavioral verification of the PDU step by step. We first began by checking the existence of illegal scenarios in the combined model of the PDU (the seventh check). The FDR2 model checker detected a major error embedded in the design of the PDU. The FDR2 counterexample is visualized in the sequence diagram of Figure 3.10.

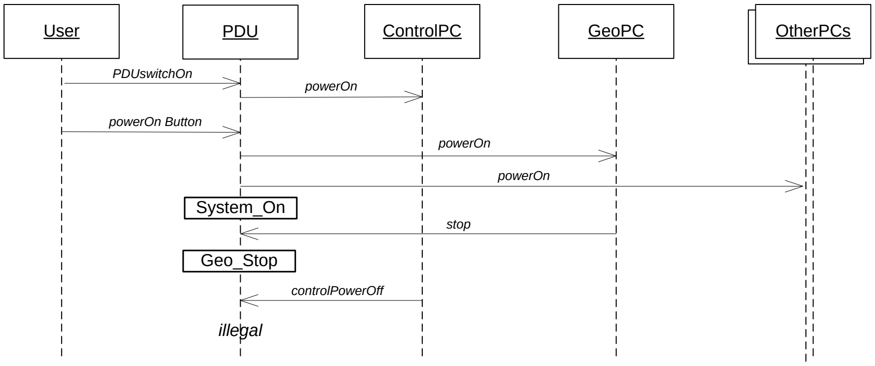


**Figure 3.10** The FDR2 counterexample

The practical scenario of the potential consequences of this error is as follows. During the regular execution of the system (the PDU is in the *System_On* state), the clinical users may experience some issues related the movable parts. Consequently, the clinical users might choose to press the *Stop* buttons attached to the body of the movable parts. After the GeoPC has sent the *Stop* signal, the PDU immediately switches off the power that supplies the movable parts, transits to the *Geo_Stop* state, and waits for the subsequent user input. But the user might desire to entirely power off the system for safety purposes. This appeared to be impossible in the current design.

More precisely the user of the ControlPC would not be able to power off the system via the *controlPowerOff* signal when the PDU is in the *Geo_Stop* state, and also both *powerOff* and *forcePowerOff* commands would have no effect on the PDU. Furthermore, if the user chooses to strictly cut the power down from the mains switch, the UPS would start automatically, if it is attached, and the erroneous situation would remain. Only pressing the *Emergency* button would rescue the user from this case since it entirely cuts down the power to the system.

The benefits of specification completeness plus the formal verification using model checking for detecting the error is obvious here. Assigning the *Illegal* response to the absent callback stimuli had effectively helped us detecting the veiled error.

Another design error was found during the specification review of the models by team members, due to a missing requirement. Consider the state machine of Figure 3.2 once more. The *forcedPowerOff* and the *controlPowerOff* transitions from the *System_Standby* state to *System_Off* state were found missing. This means that the clinical user would not be able to power off the ControlPC upon pressing the *PowerOff* button for 10 seconds or systematically power off the system using the *controlPowerOff* signal when the system is in the *System_Standby* state. Initially, this was a desired behavior since the ControlPC should always be operational, but lately a decision was made to also consider powering off the ControlPC in the *System_Standby* state.

### The improved PDU controller

After the design errors had been communicated to the PDU designers, the design had been adapted. The modified state machine of the PDU is depicted in Figure 3.11. It includes the missing *forcedPowerOff* and *controlPowerOff* transitions from *System_Standby* state to *System_Off* state. Additionally, the modified state machine allows the clinical users to power off the system when the PDU is in the *Geo_Stop* state.

Subsequently, the specification of the PDU design model had been adapted to the changes. All responses to internal callback stimuli received from PCs not specified in the original state machine were set to *Null*. Moreover, the ASD specification of the *System_Standby* and the *Geo_Stop* states had been adapted. The complete specification of the improved PDU model is introduced in [43], Appendix A. The corresponding improved specification of the external behavior is listed in [43], Appendix B.

All properties listed in Figure 3.9 succeeded except the last property, see Figure 3.12. The succeeded checks are preceded by green ticks signs, while the failed check is preceded by a cross sign. Upon clicking on the failed check, FDR2 shows another window to

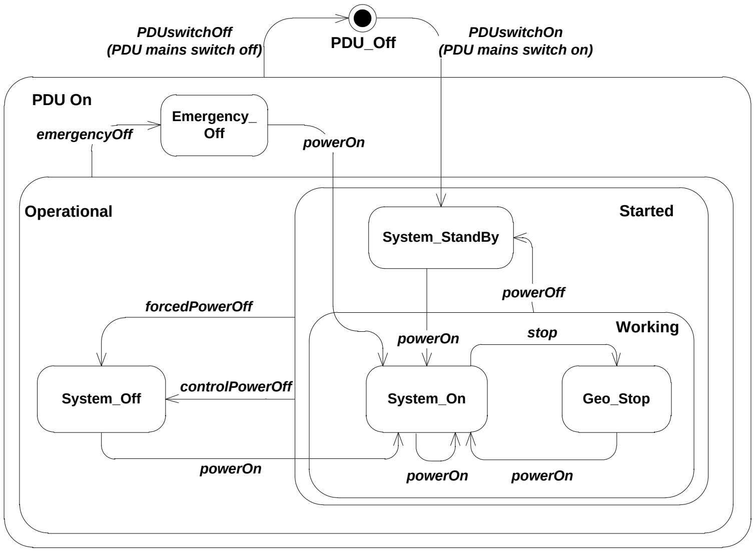
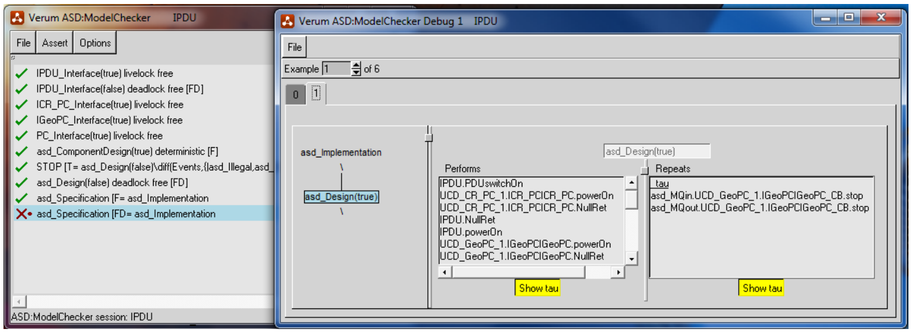


**Figure 3.11** The improved state machine after the formal verification and the specification review

visualize the counterexamples. For this property FDR2 reports six counterexamples in total where divergences might occur, which affect the external behavior. The analysis of these counterexamples reveals that the source of all divergence scenarios is basically the same. The sequence diagram that explains the erroneous scenario is visualized in Figure 3.13.


Verum ASD:ModelChecker IPDU
File
Assert
Options
Verum ASD:ModelChecker Debug 1 IPDU
File
0
1
Show tau
Show tau

✓ IPDU_Interface(true) livelock free
✓ IPDU_Interface(false) deadlock free [FD]
✓ ICR_PC_Interface(true) livelock free
✓ IGeoPC_Interface(true) livelock free
✓ PC_Interface(true) livelock free
✓ asd_ComponentDesign(true) deterministic [F]
✓ STOP [T= asd_Design(false)\diff(Events,{|asd_Illegal,asd_
✓ asd_Design(false) deadlock free [FD]
✓ asd_Specification [F= asd_Implementation
✗ asd_Specification [FD= asd_Implementation

ASD:ModelChecker session: IPDU

Example 1 of 6

asd_Implementation
asd_Design(true)

asd_Design(true)

Performs

```
IPDU.PDUswitchOn
UCD_CR_PC_1.ICR_PCICR_PC.powerOn
UCD_CR_PC_1.ICR_PCICR_PC.NullRet
IPDU.NullRet
IPDU.powerOn
UCD_GeoPC_1.IGeoPCIGeoPC.powerOn
UCD_GeoPC_1.IGeoPCIGeoPC.NullRet
```

Repeats

```
_tau
asd_MQin.UCD_GeoPC_1.IGeoPCIGeoPC_CB.stop
asd_MQout.UCD_GeoPC_1.IGeoPCIGeoPC_CB.stop
```

**Figure 3.12** All checks succeeded except refinement under the Failure-Divergences model

The counterexample shows that the GeoPC could continuously inform the PDU about a pressed *Stop* button on the body of a movable part. Then, the PDU endlessly treats the *stop* signals, with the possibility that the external user commands are not treated immediately.

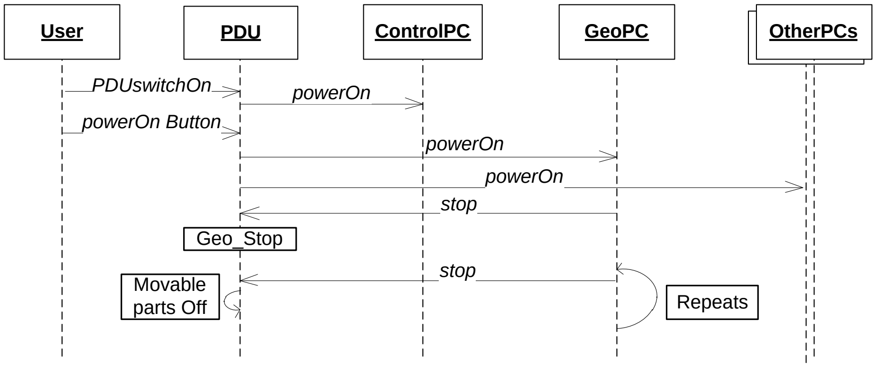


**Figure 3.13** Example of a divergence that affects the external behavior of the PDU

We don't really consider these divergences as critical errors. They are rather benign, but they can happen indeed.

### 3.2.4 Modeling and verification efforts

The activities of modeling and verifying the behavior of the PDU were conducted part time in parallel with other traditional activities devoted to the PDU development. Understanding the PDU domain plus studying various design documents [55, 54, 60] for the purpose of modeling the behavior of the PDU took approximately 35 hours.

The modeling and verification efforts of all ASD models took 32 hours in total. In general, the effort of creating the ASD models was not a time demanding process, because of the high-level description provided by the ASD:Suite. The team involved in the modeling, specification, verification and review processes were highly skilled in traditional development methods, but had limited knowledge in formal methods. But despite this limitation, team members were able to quickly understand and review the ASD specification, and to favorably provide their feedbacks and suggestions for improvements although no one of the reviewers had previously been exposed to any ASD training courses.

Table 3.2 depicts the statistical data of the ASD models. The first column lists the name of all ASD interface and design models, related to the PDU and the PCs. The second column contains the total number of rule cases, specified and reviewed by team members.

The third, fourth, and fifth columns include statistical data produced by the FDR2 model checker for checking deadlocks of each model independently. All models are deadlock free. Note that the data presented for the PDU (design) model is related to the combined model that includes the parallel composition of the PDU design plus the interface models of the PCs. The third column depicts the number of generated states, while the fourth column presents the number of generated transitions. The time in seconds spent for verification by FDR2 is depicted in the fifth column.

| Model | Rule cases | States | Transitions | Time |
|---|---|---|---|---|
| IPDU | 54 | 16 | 67 | <1 sec |
| PDU | 54 | 824 | 1,360 | <1 sec |
| ICRPC | 18 | 16 | 25 | < 1 sec |
| IGeoPC | 18 | 16 | 25 | < 1 sec |
| OtherPCs | 15 | 15 | 23 | < 1 sec |

**Table 3.2** Statistical data related to the ASD models of the PDU

The PDU team decided to continue the development of the PDU controller using the ASD technology and to further investigate other design alternatives. The development process of the PDU was continued by other team members newly introduced to the ASD method. The team investigated further other design alternatives, and applied the technology on the development of various parts of the X-ray system, especially on the services deployed on the PCs that communicate with the PDU.

Such a service is called the power control service (PCS). The following section provides details of developing the PCS. We show how both ASD and the test driven development were combined in the development process of the power control services.

Finally, the formal behavioral verification, team reviews and the specification completeness processes performed throughout this project provided a proper framework, for assisting the work, and decreasing potential efforts, devoted to error fixing at later stages of the project.

**The work performed after verifying the PDU controller.** The behavior of the PDU and the PCs were extended such that they include intermediate transiting states. The external specification and the design of the PDU were extended with extra callback events that reflect the internal states of the system: callbacks indicating that the system is on, off, starting up or in *Geo_Stop* state, for instance. The details can be found in Chapter 7.

When extending the design of the PDU we did not only consider the transiting states but also applied a number of specification techniques to avoid the state space explosion problem [75, 41]. We compared a number of design and specification styles, for example between a design of the PDU that uses a pushing strategy (where PCs notify the PDU about their states) and another alternative design that employs a polling mechanism (where the PDU queries the states of the PCs when needed).

Since not all specification guidelines of [75, 41] can be modeled using the current ASD tool, we instead used mCRL2, CSP/FDR2 and CADP and exploited some of their useful features of verification and formal refinement. The details are provided in Chapter 7.

## 3.3 The Power Control Service

### 3.3.1 Introduction

In the previous section we showed that employing the ASD techniques for the formal specification and verification was beneficial for the behavioral correctness of the PDU in the sense that errors were detected earlier during the design phase, and in a very short time. Furthermore, we elaborated more on how ASD components were specified and formally verified.

In this section we show how the ASD techniques were tightly integrated with the development processes in a real industrial development project. We present a workflow which combines test-driven development of components with the commercial formal approach of ASD and describe experiences with them at Philips Healthcare.

An analysis of the first usage of the ASD approach at Philips Healthcare shows that it leads to the development of components with fewer reported defects compared to components developed with more traditional development approaches [45, 42]. Therefore, formal methods are gradually becoming more and more credible in developing software within Philips Healthcare. 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:

- How can formal techniques be tightly integrated with standard development processes in industry? To which 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?
- What is the impact of the modeling and formal verification on the project planning? Is more time needed during the design phase? Can the test and integration phase be shortened?
- Which artifacts 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 to deal with changes; how flexible is the approach?
- How does the approach fit into the existing quality management system, e.g., concerning the required review procedures.

We report about the experiences with these issues during the development of components of the power control service (PCS) for the interventional X-ray system. Note that this is not a case study, but a real development project for a service that is used by different parts of the system which are developed at different sites.

This section is structured as follows. Section 3.3.2 introduces the PCS and its role in the interventional X-ray system. Section 3.3.3 describes the application of the proposed workflow of combining the test-driven development with ASD to develop the PCS. In Section 3.3.4 we discuss the results achieved in this project. Section 3.3.5 contains our main observations and current answers to the questions raised above.

## 3.3.2 Context of the Power Control Service

The embedded software of the interventional X-ray system is deployed on a cluster of PCs and devices that cooperate with one another to achieve various clinical procedures. As mentioned in the previous section, the control of power to these components is the responsibility of a central power distribution unit (PDU). Clinical users of any individual PC cannot control the power of the PC without using the PDU. The PDU also controls communication signals related to the startup and shutdown of the PCs. Figure 3.14 depicts the deployment of the PCS in the PCs and the relation to the PDU.

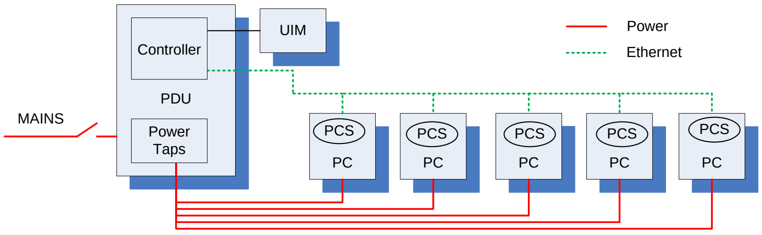


**Figure 3.14** The PCS in the context of power distribution

As can be seen in Figure 3.14, each PC includes a PCS which is 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 shutdown first the running applications and next the operating systems (OS), in an orderly fashion.

Figure 3.15 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 shutdown the PC. Moreover, the PCS can also send events to the PDU to enable or disable a number of buttons on the UIM (User Interface Module).

Another example of a concurrent component is the *InstallApplication* which is an external

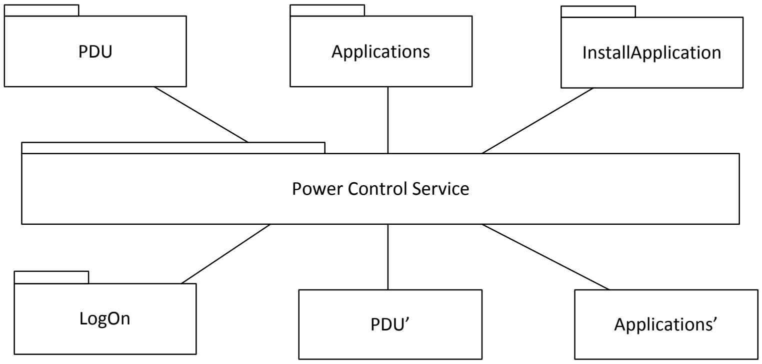


**Figure 3.15** The PCS as a black-box surrounded by concurrent components

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 parallel components. Due to the concurrent execution, controlling the flow of events among the components is rather complex, and since the PCS is deployed on every PC, any error is replicated on every PC and potentially leads to serious problems of the entire system.

Moreover, the PCS may lose connection with other components at any time due to a failure of other components (e.g., applications) or with the PDU (e.g., due to a network outage). The PCS has to be robust against such failures, especially when it 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 a malfunction has disappeared, the PCS ensures that all its internal components are synchronized back with other external components to a predefined state.

Due to the highly complex behavior 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 behavior 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.

### 3.3.3 Steps of developing components of PCS

In this section we report about the component-based development of the PCS from October 2010 till October 2011. The development process contained five increments, each implementing a 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 3.16.

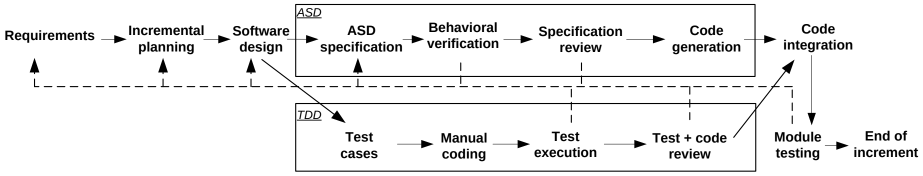


**Figure 3.16** Steps performed in a development increment

**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 3.17. In this decomposition, ASD components are depicted in a gray color, whereas light colored 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).

The ASD components of the PCS have been realized in a top-down order. Each ASD component is designed as a state machine that captures the global states of lower level components. Starting point is the *PduEventController* component which is modeled as a top-level state machine that captures overall global states (or modes) of a PC: normal mode, installing, starting/stopping applications, etc. Later, lower-level components are realized. For instance, the component *InstallTransitioning* implements detailed behavior of the installation mode of the top-level state machine and is responsible of safeguarding detailed transitions from normal mode to installation mode, and vice versa.

Experience shows that most novice ASD users tend to design rather large components leading to large ASD models [80, 42]. Although this might be acceptable in traditional development methods, it leads to serious problems when using formal techniques such as ASD:Suite. The key issues encountered with large models were as follows.

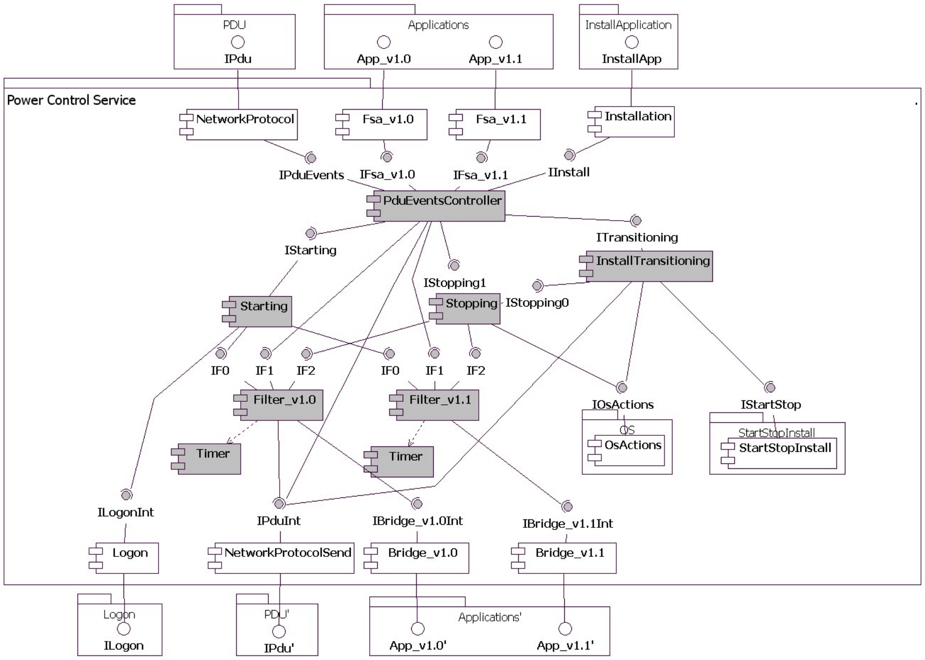


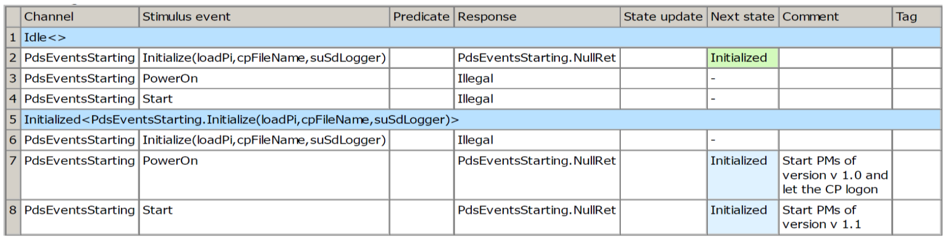
**Figure 3.17** Components of the PCS

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

During the development of the PCS, the first point was the main concern. Earlier experience showed that as soon 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 an unacceptable delay to the tight schedules of the project and its deliverables.

Therefore, the design of the PCS has been decomposed into rather small components, described using small models following the ASD recipe. Although the ASD approach shown in Figure 2.5 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.

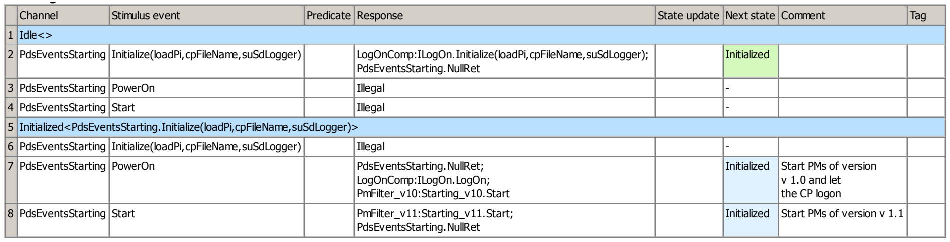
**ASD specification and formal verification.** The ASD models were specified using the ASD:Suite version 6.2.0. An example of a very small ASD interface model is shown in Figure 3.18. The model represents the interface of the *Starting* component and consists of two sub-tables, representing the states *Idle* and *Initialized*, each having three rule cases.



| | Channel | Stimulus event | Predicate | Response | State update | Next state | Comment | Tag |
|---|---|---|---|---|---|---|---|---|
| 1 | Idle<> | | | | | | | |
| 2 | PdsEventsStarting | Initialize(loadPi,cpFileName,suSdLogger) | | PdsEventsStarting.NullRet | | Initialized | | |
| 3 | PdsEventsStarting | PowerOn | | Illegal | | - | | |
| 4 | PdsEventsStarting | Start | | Illegal | | - | | |
| 5 | Initialized<PdsEventsStarting.Initialize(loadPi,cpFileName,suSdLogger)> | | | | | | | |
| 6 | PdsEventsStarting | Initialize(loadPi,cpFileName,suSdLogger) | | Illegal | | - | | |
| 7 | PdsEventsStarting | PowerOn | | PdsEventsStarting.NullRet | | Initialized | Start PMs of version v 1.0 and let the CP logon | |
| 8 | PdsEventsStarting | Start | | PdsEventsStarting.NullRet | | Initialized | Start PMs of version v 1.1 | |

**Figure 3.18** Interface model of the *Starting* component

The corresponding design model of the *Starting* component is depicted in Figure 3.19. It extends the interface model with calls to its used components *LogOn*, *Filter_v1.0*, and *Filter_v1.1*.



| | Channel | Stimulus event | Predicate | Response | State update | Next state | Comment | Tag |
|---|---|---|---|---|---|---|---|---|
| 1 | Idle<> | | | | | | | |
| 2 | PdsEventsStarting | Initialize(loadPi,cpFileName,suSdLogger) | | LogOnComp:ILogOn.Initialize(loadPi,cpFileName,suSdLogger);<br>PdsEventsStarting.NullRet | | Initialized | | |
| 3 | PdsEventsStarting | PowerOn | | Illegal | | - | | |
| 4 | PdsEventsStarting | Start | | Illegal | | - | | |
| 5 | Initialized<PdsEventsStarting.Initialize(loadPi,cpFileName,suSdLogger)> | | | | | | | |
| 6 | PdsEventsStarting | Initialize(loadPi,cpFileName,suSdLogger) | | Illegal | | - | | |
| 7 | PdsEventsStarting | PowerOn | | PdsEventsStarting.NullRet;<br>LogOnComp:ILogOn.LogOn;<br>PmFilter_v10:Starting_v10.Start | | Initialized | Start PMs of version v 1.0 and let the CP logon | |
| 8 | PdsEventsStarting | Start | | PmFilter_v11:Starting_v11.Start;<br>PdsEventsStarting.NullRet | | Initialized | Start PMs of version v 1.1 | |

**Figure 3.19** Design model of the *Starting* component

After the completion of the design model of a component, given interface models of a component and its used components, the ASD:Suite has been used to formally verify absence of deadlocks, livelocks, illegal calls, and conformance of the design model with respect to the interface model. Usually this revealed quite a number of errors, both in the design model and the interface models. Since changes in interface models affects other 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 behavior. 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 with 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.

The first error in the ASD components was caused by a wrong order in the response list of a stimulus event of a rule case. This error caused ASD components to log messages in a reverse order. The second error was due to the invocation of an illegal stimulus event in one of the *Filter* components, which unexpectedly received an initialize request from one of its client components although it was already initialized. Such a multi-client scenario is not checked by ASD:Suite.

The entire PCS code was exposed to further testing on module level at the end of all increments. After that, both production code and test code were carefully reviewed by team members. As a result of review, some 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 behavior of the service.

### 3.3.4 Results

Throughout all increments, no major redesign was needed. In general, the construction of all PCS components was rather smooth and gradually evolved along the development increments. The IBM ClearCase [51] code management system was used to store the code and the ASD models.

Since Philips quality management enforces developers to comply with coding standards provided by the TIOBE technology [81, 1], this created a problem, because the ASD generated code did not comply to the required coding standard. However, changes of ASD components will always be carried out on the level of the design models and changing the generated code directly is not allowed. Hence it was acceptable to exclude the generated code from the checks on the coding standard.

During the development of ASD components, we took care that the interface and design models remained small. Hence, the formal verification of interface and design models took less than a second. In this way, the approach is more efficient compared to traditional development.

Feedback from independent test teams was positive and the service runs stable and reliable. Team members of the PCS project appreciated the quality of the service, and decided to further incorporate the ASD technology to the development of other parts of the system. The behavioral 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.

The end quality result of the PCS service is remarkable, and the entire service exhibited only 0.17 defect per KLOC. This level of quality favorably compares to the industry standard defect rate of 1-25 defects per KLOC [61]. The PCS service was deployed on all PCs, and further tested by independent teams 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.

### 3.3.5 Concluding Remarks

We have described the experiences at Philips Healthcare with the ASD approach. 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 the ASD:Suite, statement and function 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.

Experience from other projects 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 analyze and often tough to reproduce due to the concurrent nature of components. Moreover, fixing an error in the code often causes others to emerge, and others to be unveiled with a great potential of causing unexpected failures in the field.

Our experience with ASD differs from the observations of the previous paragraph. Design errors were detected by the model checker early and automatically before any single line of code is being written or generated. The behavioral verification thoroughly checked the correctness behavior 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 clean 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 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 accept 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 model 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 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 ASD requires small components; as a guideline a design model should not contain more than 250 rule cases, a few asynchronous callbacks, leading to nearly 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.

# Chapter 4

# Formal Methods in the Backend Subsystem

[1] Experience Report on Developing the Front-end Client Unit under the Control of Formal Methods. In: Proceedings of the 27th ACM Symposium on Applied Computing, The Software Engineering Track (ACM SAC-SE 2012)

[2] Experience Report on Designing and Developing Control Components using Formal Methods. In: Proceedings of the 18th International Symposium on Formal Methods, 2012.

## 4.1 Introduction

The industrial project established for the Backend subsystem aims at developing a number of application services that realize the procedures, rules, algorithms and workflow steps required to accomplish clinical examinations. The Backend project involves the development of an architecture that consists of 12 concurrent software units that include multiple parallel components (i.e., multiple processes that include multiple threads). The project includes approximately 33 software engineers, designers and architects.

The purpose of this chapter is to provide an experience report on the application of formal methods to the development of two units of the Backend subsystem. We show that formal techniques could substantially influence the quality of the developed software of the two units. Furthermore, since formal methods enforce rigorous disciplined processes, the errors found after applying such methods tend to be simple errors, easily detected and fixed, and not profound design or interface errors.

The first unit is called the Frontend Client (FEClient) and is detailed in Section 4.3. The unit is used to safeguard interactions towards the Frontend subsystem. Using the FEClient we demonstrate steps that were followed to develop the unit and we report about the percentage of time spent for each software related deliverable such as requirements, specification, design, etc. We discuss typical errors found in the ASD components, and then show that such errors were easy to find and to fix.

In Section 4.4 we introduce the Orchestration module in the Backend Controller unit. This unit is the central part of the X-ray machine used for accomplishing the required clinical examinations through coordinating predefined workflow steps. Using the Orchestration case we elaborate more on the design steps followed to obtain verifiable components using model checking. More importantly, we illustrate the peculiarities of the resulting components demonstrating how they were easily verified following the ASD recipe. We further show that the components were easy to maintain and extend when the requirements evolved. Similar to the FEClient, we demonstrate typical errors encountered during the construction of the components.

Finally, we show that the ASD technology eliminated errors earlier in the design phase so that the resulting quality of ASD components was remarkable. We provide supporting statistical data for both the FEClient and the Orchestration components.

## 4.2 The Context of the BackEnd Subsystem

Figure 4.1 depicts the deployment of the three main subsystems of the X-ray machine. The subsystems communicate with one another via standardized, formally verified ASD interfaces (e.g., the BEFEInterface). These interfaces are made formal in order to ensure equal understanding of the intended behavior among separate teams developing the subsystems and to reduce communication overhead.

Below we briefly address the functionality of the subsystems from a high-level perspective to the extent required for introducing components of the FEClient and the Orchestration module.

The Backend subsystem houses graphical user interfaces (GUI), patients databases and a number of predefined X-ray settings, used to achieve required clinical examinations. Through the user interface clinical users can manage patients' data and exam details and can review related X-ray images. The Backend is also responsible of supporting different types of Frontends.

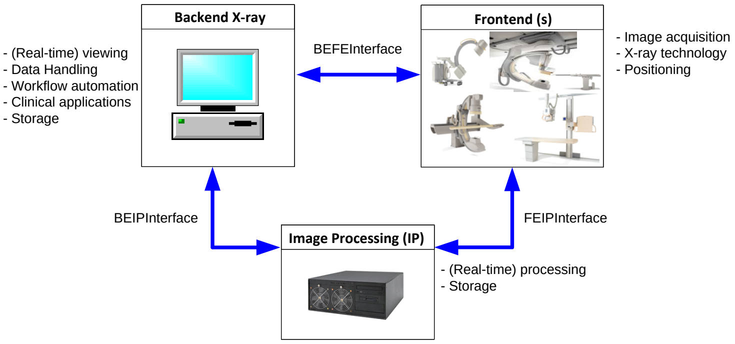


**Figure 4.1** Subsystems with distinct responsibilities and formal interfaces

The Frontend subsystem controls motorized movements of the table where patients can lay and the stands that hold the X-ray collimators and the image detectors. It is also in charge of calibrating these components upon requests sent remotely by the Backend, based on the predefined X-ray settings, selected by clinical users from the GUI.

When all components are calibrated and prepared, the Frontend demands the Backend to prepare its internal units before it asks for permission to start image acquisition. Upon obtaining permission, the Frontend starts acquiring X-ray images and sends related data to the IP subsystem for further processing. After that, the IP subsystem sends the processed images to the Backend for viewing on various screens and for local storage to facilitate future references.

The two units of the Backend incorporated the ASD technology are detailed in the subsequent sections, namely the FEClient and the Orchestration.

## 4.3 The Frontend Client

### 4.3.1 Introduction

The purpose of this section is to report on the experiences with developing the control part of the FEClient units. The control part was developed using the ASD approach in a pipeline of consecutive increments. As we will demonstrate shortly, the control part of this unit exhibits good quality results compared to the non-control part which was developed using traditional development methods.

This section is structured as follows. The context of the FEClient unit and its responsibilities are described in Section 4.3.2. The steps accomplished for developing the FEClient using ASD are demonstrated in Section 4.3.3, highlighting the time required by each step and the issues encountered. In Section 4.3.4 we discuss the nature of the errors found during the construction.

### 4.3.2 Context of the FEClient unit

Figure 4.2 depicts the deployment of the FEClient unit in the Backend subsystem and its position with respect to the Frontend subsystem. The FEClient mediates messages between various units of the Backend and the Frontend subsystem across a physical network. The interaction between the two subsystems is standardized by a predefined communication protocol, specified in the BEFE ASD interface model, which is used by the components of the FEClient and is implemented by other components developed by other teams in the Frontend (detailed in the subsequent chapter).

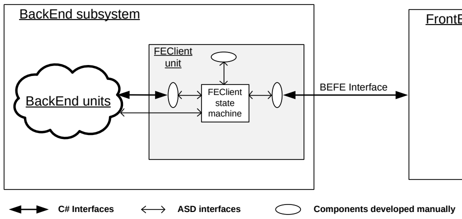


**Figure 4.2** Deployment of the FEClient in the Backend

The challenges imposed on the FEClient is that any of the following components may fail during the execution of the system: any unit of the Backend; the Frontend subsystem; the network connection between the Frontend and the Backend subsystems. For example, if the Backend loses its connection to the Frontend subsystem, the problem may be that the Frontend subsystem has a major failure, or that there is a network outage between the two subsystems.

In either case the failure may be temporarily or persistent. The FEClient knows whether the Frontend subsystem is at fault, and therefore must correctly respond to internal Backend requests, by providing proper handlings: viewing of patients' images and data, regardless of the presence or absence of the Frontend subsystem, for instance. Once the source of failure has disappeared, the FEClient must ensure that both subsystems are synchronized back to a predefined state. Briefly, the FEClient provides numerous benefits such as:

- manipulation of data in readable formats comprehensible for the communicating subsystems,
- guaranteeing the consistency of states between the external subsystems and internal Backend units,
- exchanging of patients' data and exams, constructed and originated by Backend units, and sent towards the Frontend subsystem,
- processing of X-ray settings, constructed and exchanged by the Backend and the Frontend subsystems,
- and handling the requests for acquiring X-ray images from the Frontend subsystem.

The FEClient includes a complex control part (a state machine), which maintains the current state of the system, and enables units on the Backend to correctly communicate with the FronEnd and vice versa. The control part comprises a total of 88 different stimuli, 211 responses and 26 distinct states. The impetus of the FEClient state machine was the need of safeguarding the flow of information between the fully concurrent subsystems, by preventing potential deadlocks, livelocks, race conditions, and illegal interactions. Due to the distributed nature of the system, each component has only partial information about the state of the system, raising the complexity of providing correct interactions among the concurrent, interacting parties.

### 4.3.3 The application of ASD for developing the FEClient

We report about the activities accomplished for developing the FEClient, starting from January 2008 till the end of 2010. The development of the FEClient involved 3 full-time and 1 part-time team members. All had sufficient programming skills, but limited background in formal methods. The team had been exposed to ASD training courses, to learn the fundamentals of the method and its technologies. The ASD method required a learning curve because it was new to the development team. Therefore, time, investments and experience were required before developers became skilled in the technology.

At the beginning of incorporating ASD to the development of FEClient, two ASD consultants were present, who devoted roughly half of their time to the project, helping developers to rapidly learn the technology and its practices. The unit was developed in a small team to afford greater quality and control, compared to larger teams or individuals.

The traditional development process was adapted to fit the ASD method. Table 4.1 briefly describes these processes plus the percentage of time conducted for developing the control part of the FEClient throughout all of its increments. Below, we briefly describe these processes and refer to [45] for more details.

| | | |
|---|---|---|
| 5% | Incremental planning | Formation of teams and their responsibilities; work breakdown estimation for each function including time, efforts, deadlines, risks, etc. for each planned function |
| 10% | Software design | Decomposition of the unit into ASD and manually developed components; assigning responsibilities to components; adapting design to new planned functions |
| 25% | Functional specification | Specifying the external behavior of the FEClient towards its clients; describing the ASD design model of FEClient; specifying the interface models of the used components |
| 24% | behavioral verification | Searching for deadlocks, livelocks, illegal calls; detecting race conditions and violation of protocol of interactions; formally check the refinement correctness of FEClient internal implementation against its external specification |
| 5% | Specification review | Team review of interface specifications for all specified rule-cases; checking traceability to informal requirements; checking naming consistencies |
| 10% | Code generation and integration | Generating code in C# language; integrating the generated code to the system; implementing glue code |
| 20% | Testing | Unit test of generated and manually written code; function and coverage test for manually written code |
| 1% | End of increment | Problems solving; bug fixing |

**Table 4.1** Time and activities of developing FEClient

*Requirements and incremental planning.* To ensure that the development team clearly understands the essential functions of the system before development activities begin, Philips chose to formally express the requirements of the system in tags using CaliberRM, a software requirements management tool. To increase their awareness even more, the development team is required to reference the tags in the specification of ASD models. As soon as the requirements of the FEClient had been clarified, the planning and the work breakdown estimations were prepared.

*FEClient design.* On completion of incremental planning, the design of FEClient components started as working drafts, reviewed by team members in a number of sessions. Feedback from each team review session was incorporated to further improve the informal designs. Team reviews provided opportunities for code economy by identifying reusable modules (or common services) such as tracing and logging. At the end of the design step the distribution of components was accomplished, with well-defined interfaces and responsibilities.

*Functional specification.* After the design step was accomplished, fifteen ASD interface models that capture the external behavior of the FEClient state machine component and the internally used components were described using the ASD:Suite. The structure of the ASD models is depicted in Figure 4.3. The FEClient state machine was described using an ASD design model, implementing both the external behavior towards the clients and the internal behavior of the FEClient towards the internal components and towards the Frontend subsystem. The development of the manually coded modules was done in parallel to the ASD modeling.

The first column of Table 4.2 lists the ASD models, from which code was generated and correctness verification was checked. The second column reports the total number of rule cases, specified and reviewed by team members.

The IFECSM (Interface FEClient State Machine) model is the interface model that captures the external behavior of the FEClient unit. This model is used by other ASD clients located in the Orchestration module (we detail them in the subsequent section). As can be seen from the table the model comprises a substantial number of specified rule cases. This directly affected the description of the FEClient design model (FECSM), which is clearly the most complex model among all others. The reason of this complexity is that ASD allows only one design model for refining any interface model, which means that a decomposition of an interface model to a number of simple design models is not supported at the moment of writing this thesis.

*Behavioral verification.* After all models were specified, the formal behavioral verification process using model checking was started. Race conditions, deadlocks, livelocks, and illegal scenarios violating the communication protocols were discovered early, causing either to adapt the specification or to redesign the components. The verification of the FEClient included the interface model of the BEFE interface. The communication protocol specified in the early BEFE interface model contained some errors discovered during the behavioral verification and specification reviews. Such errors were communicated to the team involved in the development of the interface.

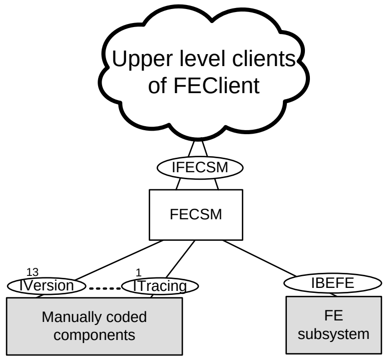


**Figure 4.3** The structure of the FEClient

The third, fourth and fifth columns of Table 4.2 demonstrate statistical outputs for checking deadlock freedom: the generated states, the generated transitions and the time in seconds required for verification respectively. All models are deadlock free.

The verification of all interface models were accomplished separately, except for the state machine, which was verified as a combined model that includes the state machine design plus all interface models of used components. As can be inferred from the table roughly 8.5 thousand states can be generated per second by FDR2, but this number was not constant for all remote verification sessions of the state machine. In some circumstances the remote verification process was slow, and the ASD users were forced to wait longer until verification results appeared on their screens.

*Specification review.* When the behavioral verification process had been completed, specification reviews of ASD models were conducted to verify completeness, correctness, and traceability to the original tagged requirements. Participants in review sessions varied, but always included the owner of the specification and one or more persons who had previously been trained in ASD. In general it was difficult for non ASD users to follow the review process. The early lack of systematic compare and merge of models in the ASD:Suite complicate the review process further. Reviews were performed in a number of sessions of roughly half an hour each, and documented in dedicated separate review sheets.

*Code generation and code integration.* Once specification reviews were accomplished, the models were automatically translated into the target language, in this case, C#. The last column sketches the generated lines of code (LOC), in the C# programming language, excluding blank and comment lines. The generated code was integrated with the manually written code by implementing glues of appropriate adapter and wrapper code. The integration process of the FEClient generated code with the generated code of other ASD components was remarkably smooth.

The interface model of the FEClient is used by 5 fully concurrent ASD client components (located in the Orchestration module), which use the interface model for the behavioral verification, according to the ASD recipe. Consequently only one error had been reported (detailed in the subsequent section) during the integration of the FEClient generated code with the code of the ASD client components.

*FEClient test.* Unit testing was started after the generated code was integrated with the manually written code. The FEClient state machine always passed its unit test, and only few errors were discovered during system test. During the construction of the FEClient few errors had been committed; fixing these errors often commenced at the end of each increment. We discuss these errors in subsequent section.

The total number of hours spent for specifying and verifying ASD models plus generating and integrating code of all the FEClient increments was nearly 700 hours. Table 4.2 depicts statistical data of the FEClient state machine and the interface models of the used components.

| Model | Rule cases | States | Tran. | Time (sec) | Exe. LOC |
|---|---|---|---|---|---|
| FECSM | 2376 | 1996830 | 5249538 | 230 | 11121 |
| IFECSM | 1068 | 3028 | 7,112 | 0 | 173 |
| IBEFE | 2,183 | 931 | 8537 | 0 | 129 |
| ICTFFacade | 51 | 6 | 28 | 0 | 70 |
| IConfigRepository | 8 | 2 | 7 | 0 | 53 |
| IServiceFactory | 8 | 4 | 4 | 0 | 49 |
| IEnumConversions | 16 | 2 | 15 | 0 | 61 |
| IParameters | 18 | 7 | 22 | 0 | 88 |
| IParameterCache | 4 | 2 | 3 | 0 | 49 |
| IPerformance | 5 | 2 | 4 | 0 | 50 |
| IReportLogging | 3 | 2 | 2 | 0 | 48 |
| IRunTag | 12 | 15 | 33 | 0 | 53 |
| ISystemType | 4 | 3 | 4 | 0 | 48 |
| ITracing | 3 | 2 | 2 | 0 | 48 |
| IUserGuidance | 16 | 18 | 64 | 0 | 65 |
| IVersionExchange | 4 | 3 | 4 | 0 | 48 |
| ASD run-time | - | - | - | - | 701 |

**Table 4.2** The ASD models of the FEClient

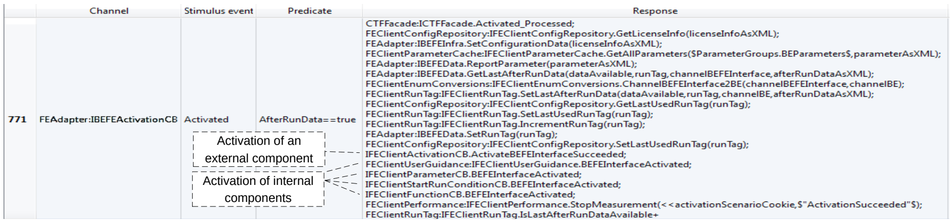
### 4.3.4 Type of errors found during developing the FEClient

The FEClient development team prepared careful reports of all errors found during the construction of the unit. These defects were submitted to a defect tracking system, which is part of a sophisticated code management system. For scrutiny purposes, Philips opened their error reports, and we carefully investigated them trying to determine the impact of formal methods on the quality of the code. Our analysis reveals the followings.

A total of eleven errors related to both ASD and the manually written code were reported along the construction of the unit. Four errors were found during implementation, five during integration, and two during system testing.

Three of the eleven errors were caused by design, e.g., missing a response to a test component in the ASD state machine; and eight errors introduced during implementation, e.g., a redundant “WARNING” word in some traces, which complicated the analysis of other traces.

Of the eleven errors, four would have caused failures during system execution. One of the four errors is severe and most likely to occur, e.g., misspellings in data could cause a crash at the Frontend subsystem. One error is average with low probability of occurrence, e.g., a race condition between a request to acquire images from the Frontend subsystem and a request to exchange X-ray settings from the Backend clients. The remaining two are minor errors, e.g., a crash due to a failure to load a missing dll file during version exchange with the Frontend subsystem.



| | Channel | Stimulus event | Predicate | Response |
|---|---|---|---|---|
| 771 | FEAdapter:IBEFEActivationCB | Activated | AfterRunData==true | CTFFacade:ICTFFacade.Activated_Processed;<br>FEClientConfigRepository:IFEClientConfigRepository.GetLicenseInfo(licenseInfoAsXML);<br>FEAdapter:IBEFEInfra.SetConfigurationData(licenseInfoAsXML);<br>FEClientParameterCache:IFEClientParameterCache.GetAllParameters($ParameterGroups.BEParameters$,parameterAsXML);<br>FEAdapter:IBEFEData.ReportParameter(parameterAsXML);<br>FEAdapter:IBEFEData.GetLastAfterRunData(dataAvailable,runTag,channelBEFEInterface,afterRunDataAsXML);<br>FEClientEnumConversions:IFEClientEnumConversions.ChannelBEFEInterface2BE(channelBEFEInterface,channelBE);<br>FEClientRunTag:IFEClientRunTag.SetLastAfterRunData(dataAvailable,runTag,channelBE,afterRunDataAsXML);<br>FEClientConfigRepository:IFEClientConfigRepository.GetLastUsedRunTag(runTag);<br>FEClientRunTag:IFEClientRunTag.SetLastUsedRunTag(runTag);<br>FEClientRunTag:IFEClientRunTag.IncrementRunTag(runTag);<br>FEAdapter:IBEFEData.SetRunTag(runTag);<br>FEClientConfigRepository:IFEClientConfigRepository.SetLastUsedRunTag(runTag);<br>IFEClientActivationCB.ActivateBEFEInterfaceSucceeded;<br>FEClientUserGuidance:IFEClientUserGuidance.BEFEInterfaceActivated;<br>IFEClientParameterCB.BEFEInterfaceActivated;<br>IFEClientStartRunConditionCB.BEFEInterfaceActivated;<br>IFEClientFunctionCB.BEFEInterfaceActivated;<br>FEClientPerformance:IFEClientPerformance.StopMeasurement(<<activationScenarioCookie,$"ActivationSucceeded"$);<br>FEClientRunTag:IFEClientRunTag.IsLastAfterRunDataAvailable+ |

Activation of an external component
Activation of internal components

**Figure 4.4** The order of activating boundary components caused an error

Figure 4.4 depicts a specification of a rule case that caused an unintended behavior of the system, during the integration phase in one of the project increments, due to a wrong order in the responses list. The rule case specifies that when the Frontend subsystem is activated, it sends the *Activated* event to the Backend via the BEFE callback interface; this is indicated by the channel and the stimulus event of the rule case. Upon receiving the *Activated* event, the FEClient sequentially executes a list of responses, each until completion.

The order of the depicted responses was not correct since one concurrent external client component (called the Activation controller) was activated before the internal components of the FEClient. The order was initially made this way to shorten the time required for activating the units of the Backend. But, if the client component was quick, it could send events to the internal components which were not activated yet. The consequence was that the user interface, on the screen of the Backend, shows an indication that acquiring images is not possible, while it should be.

Due to the concurrent nature of the client component, the error was hard to reproduce manually, but locating the source of the error was easy. Correcting the activation order, such that the external client is activated after the internal components, was straightforward, and indeed solved the issue.

A summary of all discovered errors is given in Table 4.3. We use the error categories defined by Basili and Selby in [10]. The error severity codes are as follows.

| | |
|---|---|
| M | Major error, |
| N | Minor error, |
| V | Average error, |
| H | High probability of occurrence, |
| L | Low probability of occurrence, |
| F | Error would have caused a failure during system execution. |

Four extra codes specify whether the error was caused/found during design (“D”), implementation (“I”), integration (“G”) or system testing (“T”).

| Category | Error severity | Caused /Found | ASD | Description of error |
|---|---|---|---|---|
| Control | V/L/F | D/I | Yes | Race condition between exchanging X-ray settings and a request to acquiring X-ray images |
| | V | D/T | Yes | Missing a response to test component in a rule case |
| | M/L | D/G | Yes | Not possible to generate images although it should be possible |
| Data | V | I/G | No | Test interface is exposed in deployment environment |
| | M/H/F | I/G | No | Misspelling in DataDictionary caused exception at the Frontend subsystem |
| | V/L | I/G | No | Image Acquisition indicator is not enabled |
| Initialization | N/L/F | I/T | No | Missing configuration file caused exception in test system at startup |
| | V/L/F | I/G | No | Exception when version exchange assembly loading fails |
| External | V | I/I | No | Tracing shows up in logging database |
| | V | I/I | No | Coding standard violation |
| Cosmetic | N | I/I | No | Redundant WARNING word in tracing |
| Computation | - | - | - | No errors |
| Interface | | - | - | No errors |

**Table 4.3** Summary of errors found during the construction of the FEClient unit

In general, the errors reported during the development of the FEClient are simple goofs, easily found and fixed, and not critical design or interface errors. Fixing these errors was generally quick and rarely caused other errors to appear as a consequence of the fix.

## 4.4 The Orchestration Module

### 4.4.1 Introduction

The purpose of this section is to report on the steps followed to design components of the Orchestration module. These steps preceded the steps of modeling and developing the components using the ASD technology. We discuss how these design steps effectively helped us constructing verifiable components. We illustrate the peculiarities that facilitate verifying the components compositionally without facing the state space explosion problem of model checking.

Finally, we investigate the effectiveness of using ASD to the quality of the module, demonstrating the defects reported along the development. In this project we also show that the errors that escaped the ASD formal verification were easy to locate and to fix, not deep design or interface errors.

This section is structured as follows. In Section 4.4.2 the context of the Orchestration module within the X-Ray system is introduced. Section 4.4.3 details steps accomplished for designing components of the Orchestration module, and the peculiarities that facilitate verifying them easily using model checking. In Section 4.4.4 we detail typical errors left behind by the ASD formal techniques.

### 4.4.2 The context of the Orchestration module

One of the key units of the Backend is the Backend controller (BEC), which includes the Orchestration module as one of its control modules. Figure 4.5 depicts the deployment of the Orchestration module in the Backend surrounded by a number of concurrent units (i.e., multiple processes include multiple threads) on the boundary.

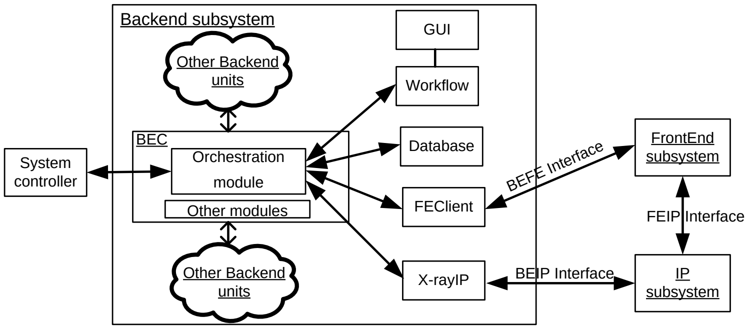


**Figure 4.5** Relation of Orchestration as a black-box with other units

The impetus of introducing the Orchestration module was the result of migrating from decentralized architecture, where units were working on their own, observing changes in the system through a shared blackboard and then react upon them, to a more centralized one. The main challenge imposed on the decentralized architecture was the need to know the overall state of the entire system and whether all units are synchronized with one another in predefined states. Further, extensibility and maintainability were complex to achieve and utterly challenging. The Orchestration module is mainly responsible of coordinating a number of phases required to achieve the clinical examinations and harmonizing the flow of events between the concurrent interacted subsystems. These phases are depicted in Figure 4.6 and summarized below.

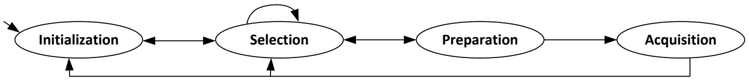


**Figure 4.6** Global system phases

*The Initialization phase.* At the start up of the system, the system controller instructs the Orchestration module to start the initialization phase of the system. Consequently, the Orchestration module initializes and activates a number of internal units of the Backend and the external subsystems through boundary units (e.g., the Frontend subsystem via the FEClient unit). This includes ensuring that all required services and configurations are loaded, proper messages and indicators are displayed on user terminals and further that the Backend is connected to compatible, supported subsystems.

*The Selection phase.* After the Orchestration module ensures that all components of the system are fully activated, the Orchestration accepts selection requests related to patients and to clinical examinations and subsequently enters the Selection mode. In this mode patient's data can be selected and sent by the GUI to the Orchestration module through the workflow controller. At the moment of receiving a selection request, the Orchestration checks whether it is allowed to start the selection procedures (e.g., there is no active image acquisition) and then distributes the data to internal units of the Backend and to the external subsystems.

The data includes information about a patient and is applied throughout the system in steps. This briefly starts by distributing personal data of the patient followed by the predefined exam and then the X-ray settings (called also X-Ray protocols) to internal units of the Backend and to the external subsystems. Based on these settings various software and hardware components are calibrated and prepared such as the X-ray collimators, image detectors and performing proper automatic positioning of the motorized movable parts such as the tables and the stands.

*The Preparation and Image Acquisition phases.* When the selection procedures are successfully accomplished, the Orchestration module can enter the preparation phase. This starts when the Frontend sends corresponding settings back to the Backend in order to properly prepare and program the IP subsystem. After that the Frontend asks permission to start the generation of X-ray for image acquisition.

When the Orchestration module ensures that all internal units of the Backend and the IP subsystem are prepared for receiving incoming images, the Orchestration module gives permission to the Frontend subsystem to start image acquisition. After that, the Frontend acquires image data and sends them to the IP subsystem for further processing. The processed images are sent to the Backend for viewing on different terminals synchronized and controlled by the Backend.

### 4.4.3 Developing and designing the Orchestration module

The control part of the Orchestration module was developed following the workflow illustrated previously in Section 2.4. The development activities went through a total of six consecutive increments, involving 2 full-time and 2 part-time team members.

Each increment included two members who were involved not only in developing the Orchestration module but also in building other modules of the BEC unit. All members attended ASD training courses, to learn the fundamentals of the ASD approach and its accompanying technologies. Team members of the Orchestration had sufficient programming skills, but limited background in formal mathematical methods.

The Orchestration module was one of the first modules which were built using ASD. Although the ASD approach hides all formal details from end-users, the team members were confronted with the steep learning curve. As a result the first version of the Orchestration module suffered from some problems. For example, some models were over-specified, too complex to understand and model checking took a substantial amount of time for verification.

Below we discuss the steps we took to get to a better (ASD) design. After that, we demonstrate peculiarities of design components that facilitate verifying them compositionally following the previously addressed ASD recipe (see Section 2.2).

**Design Steps** Designing software is a creative process and typically requires several iterations to come to a final design. So although there is no fixed recipe there are steps that can guide this process. Below we address the steps applied to design the Orchestration module. Consider that although the steps are described in a linear fashion the process is rather iterative. Even the requirements phase might be revisited because of questions arise during design.

*Setting the stage: the context diagram.* As a first step we defined the context diagram of the Orchestration module as a black-box. The context diagram depicts the module and its external environment i.e., all other components it interacts with. Using the requirement documents we constructed the list of messages/stimuli that the module exchanges with the external environment, in other words its input and outputs. The context diagram was used to draw the main sequence diagrams (between the module and its external environment) including the sequence diagrams for the non-happy flow.

*Divide and concur: decomposition.* As a second step we decomposed the black box from step 1 into smaller components. The decomposition was done by identifying different aspects of the problem domain. As Orchestration is about controlling and coordinating changes in the overall system state (e.g., selecting a new patient or starting image acquisition) we decided to use one overall controller controlling the system state and separate controllers which control details of the state transition when moving from one state to another.

*Defining responsibilities.* We then re-iterated the list of requirements allocated to the Orchestration module and allocated each requirement to one (if possible) or more of its components. While doing so new components were identified, e.g., the one guarding the connection to the frontend subsystem.

*Repeat the process.* For each of the individual components the process was repeated. We defined the context diagram, input and output messages/stimuli and the main sequence diagrams for each individual component.

*Define Interfaces.* Based on the previous step we identified the provided and used interfaces of each component. After that we prepared initial drafts of state machines for each component.

*Identify hand written components and their interfaces.* As we are using ASD which does not deal very well with data it is important to factor out code that is responsible for data related operations or code that interfaces to legacy code. In the case of Orchestration the module distributes information which is needed for the state transition (e.g., a reference to the patient to be selected for acquisition). This requires retrieving data from a repository which has to be written by hand.

*Constructing ASD models.* After all these steps the ASD models (interface and design) were constructed based on draft state machine for each component. In parallel, the code of handwritten components was written.

### The resulting ASD components

The final structure of the Orchestration components is depicted in Figure 4.7. Below we detail their peculiarities that effectively had helped verifying them compositionally in a reasonable time using the ASD:Suite.

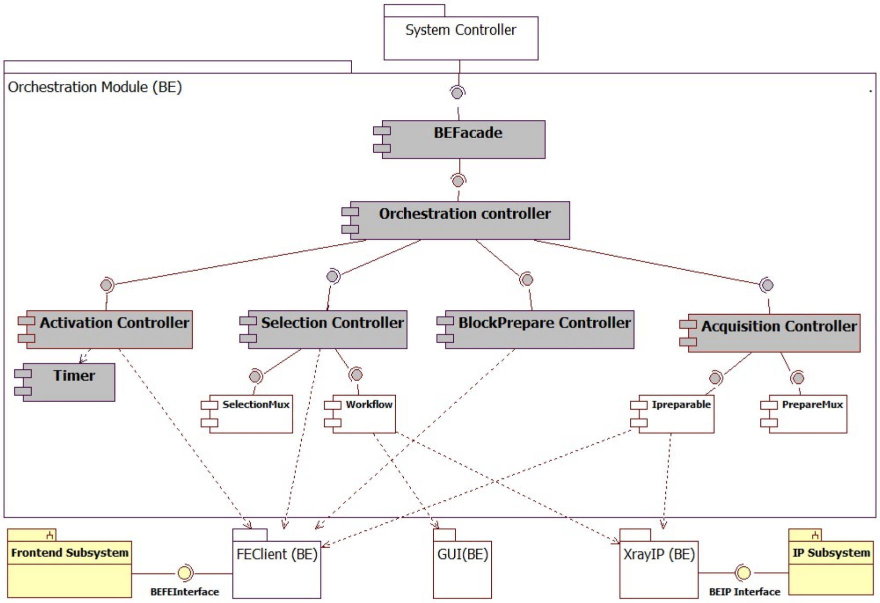


**Figure 4.7** Decomposition of the Orchestration components

The *BEFacade* component includes a high abstract state machine that captures the overall system states, seen at that level. This state machine knows only whether the system is initialized, activated or deactivated. It includes events that only affect these global states. The detailed behavior that refines these states is pushed down to the Orchestration controller component.

The *Orchestration* controller state machine includes states that capture the overall modes of the system. That is, whether the system is busy activating, performing selection procedures, or performing image acquisition. The Orchestration controller, for instance, does not know which particular type of selection is performed but it knows that the selection procedure is active or has finally succeeded or failed. Detailed procedures of these phases are the responsibility of lower-level components. The same concept applies to all other modes, e.g., activation and acquisition. The Orchestration controller component mainly coordinates the behavior of the used components positioned at the lower-level, give permissions to start certain phase and ensures that certain procedures are mutually exclusive and run to completion. It also ensures that units are synchronized back to a predefined state when a connection with other subsystems is re-established (e.g., reselecting previously selected patient).

The *Activation* controller is responsible of handling detailed initialization behavior, including ensuring that connection to subsystems is established and periodically checking if there is network outage between the Backend and other subsystems. The Activation controller retries to establish the connection with other subsystems and informs Orchestration when this is done. When activation is succeeded, the Backend knows that compatible, supported subsystems are connected, and thus accepts requests to proceed to the following phase.

The *Selection* controller is in charge of performing detailed selection procedures with other subsystems after getting permission from the Orchestration controller. The selection controller knows which part of the system has succeeded with the selection. It includes internal components (e.g., the *SelectionMux*) used to distribute selection related signals to other units, gather their responses and reports back the end result to the selection controller. The selection controller informs the Orchestration controller about the end result of the selection, i.e., whether succeeded or failed.

The *BlockPrepare* controller prevents any possible race conditions between selection procedures and image acquisition procedures to prevent mixing of patients' cases.

The *Acquisition* controller is responsible of preparing all internal components of the Backend and other subsystems for image acquisition and reports the end result back to the Orchestration controller. The controller includes also internal components (e.g., PrepareMux) to distribute preparation related signals to various units, gather related results, and sends back the end result to the Orchestration controller.

Each ASD component uses common modules (or reusable components) such as those used for tracing (to allow in-house diagnostics by developers) and logging (to facilitate diagnostics by field service engineers in the field) and displaying user guidance to indicate the progress of certain procedures. The ASD components may include a timer or a queue to store incoming callback events sent by lower-level components.

### Constructing the ASD components following the ASD recipe

Components of the Orchestration module were realized in a mixture of top-down and bottom-up fashions. Each ASD design model is verified in isolation with the direct interface models of lower-level components, providing that these interface models are refined by corresponding design and other interface models. The compositional construction and verification is visualized in Figure 4.8 and is self-explainable. Both Orchestration and FEClient units were constructed concurrently. The FEClient team provided the IFECSM ASD interface model to the Orchestration team as a formal external specification describing the protocol of interaction between the two units, and the allowable and forbidden sequence of events crossing the boundary.

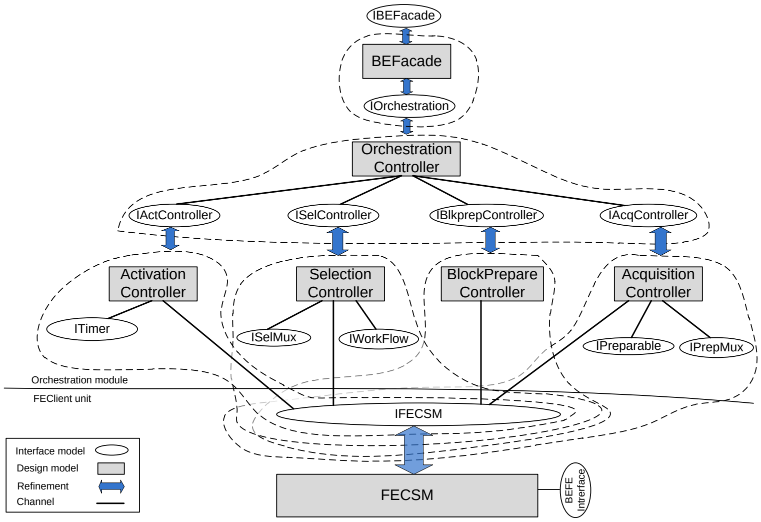


**Figure 4.8** Compositional construction and verification of components. Handwritten components are hidden.

Table 4.4 depicts the final statistical data of components of the Orchestration module. It mainly shows that components were verified in a reasonable time using model checking. The first column lists the names of the components. The second column represents the total number of ASD models of each component, presenting the sum of one design model plus the interface models of the boundary ASD and non ASD components.

The third column demonstrates the total number of rule cases, specified and thoroughly reviewed by team members. The fourth, fifth and sixth columns reports statistical outputs for merely checking deadlock freedom using the model checker FDR2: the generated states, the generated transitions and the time required for verification in seconds respectively. All models are deadlock free.

The last two columns present the automatically generated lines of code (LOC), in C#. The total LOC represents all source lines of code, including blank and comment lines. The executable LOC includes all executable source lines excluding comments and blanks.

| Component | ASD models | Rule cases | States | Transitions | Time (sec) | Total LOC | Exec. LOC |
|---|---|---|---|---|---|---|---|
| AcquisitionController | 9 | 458 | 576296 | 2173572 | 30 | 4151 | 3891 |
| ActivationController | 5 | 622 | 351776 | 1512204 | 28 | 2188 | 2062 |
| BECFacadeICC | 2 | 85 | 28 | 33 | 1 | 590 | 502 |
| BlockPrepareController | 2 | 33 | 16484 | 55298 | 1 | 838 | 784 |
| OrchestrationController | 8 | 448 | 9948256 | 42841904 | 1111 | 2940 | 2580 |
| SelectionController | 8 | 807 | 2257180 | 9657242 | 110 | 3450 | 3190 |
| SelectionState | 2 | 42 | 665 | 2393 | 1 | 622 | 566 |
| ASD runtime | - | - | - | - | - | 852 | 746 |
| Total | 36 | 2495 | - | - | - | 15631 | 14321 |

**Table 4.4** The ASD models of the Orchestration

The total sum of hours spent for designing, specifying and verifying ASD models plus generating and integrating code was nearly 1290 hours.

One drawback of the ASD compositional verification is that it is difficult to known whether all components work together as intended. This tends to be hard or even impossible to establish using model checking because of the limitation of the state space explosion. Therefore, the entire unit hosting the Orchestration module was exposed to a model-based testing technology, supported by ASD, called the compliance test framework.

Using this technology, the entire unit was systematically tested under a statistical quality control, based on usage models that specify the usage scenarios as state machines. The description of usage models is similar to the ASD tabular specification but extended with probabilities of usage. The use of this technology revealed a few errors but most were not directly related to ASD code. For example, incorrect handling of data in the manually written code and cases were usage models specify additional behavior not implemented yet by the Orchestration.

### 4.4.4 Type of errors found during developing the Orchestration

The development team of the Orchestration also found a number of errors during the construction of the Orchestration module. This includes errors discovered not only during testing but also during implementation and integration in case such errors hinder other teams developing other units. Similar to the FEClient case, we detail the nature of these errors that escaped the formal techniques of ASD.

Eight errors related to both ASD and the manually written components were reported along the construction of the module. Six of these errors are related to ASD while two related to the manually coded components. Two errors were found during implementation, five during integration, and one during subsystem testing.

Five of the eight were design defects, e.g., the GUI loses connection with Orchestration after it prematurely restarts; and three errors introduced during implementation, e.g., sending a wrong user guidance to the GUI.

Of the eight errors, two would have caused failures during system execution. The two errors are severe and most likely to occur in the field, e.g., an exception raised while selecting X-ray protocol that causes the process hosting the Orchestration module to unexpectedly terminate.

One error is minor, e.g., the GUI shows that two patients are on the table due to a wrong response sent from Orchestration to the GUI.

After carefully analyzing the detailed reports of these defects we found that the errors were easy to find and to fix, not profound design or interface errors. Table 4.5 depicts a summary of these errors. We use the error severity codes mentioned in Section 4.3.4.

| No. | Description of error | Error severity | Caused /Found | ASD |
|---|---|---|---|---|
| 1 | Orchestration logging: invalid user guidance sent to GUI. | V/H | I/G | N |
| 2 | When GUI restarts connection is lost with Orchestration. | M/H/F | D/G | N |
| 3 | Assertion during selection of X-ray protocol. | V/H/F | I/G | Y |
| 4 | Failing protocol selection not correctly handled. | V/L | I/G | Y |
| 5 | Incorrect state update in ASD Selection Controller model. | M/H | D/I | Y |
| 6 | Possible to get two patients on the table. | N/L | D/T | Y |
| 7 | Case selection request received before reselection. | M/H | D/G | Y |
| 8 | When connection re-established, old case was not reselected. | M/L | D/I | Y |

**Table 4.5** Summary of errors found during the construction of the Orchestration module

## 4.5 Quality results of the FEClient and the Orchestration

The code developed for the FEClient and the Orchestration through the first three increments is part of the X-ray machines released to the market. The other three increments were devoted to extending the module with additional functionalities and new features for a more sophisticated X-ray machine planned to be released in the future.

It is notable that ASD components were easy to maintain and to extend due to the high-level description of ASD specification, and the high abstract behavior of the components. In general, it was easy to adapt the models and generate new verified code.

For example, in the fifth increment there were serious changes in the standard interface between the Backend and the Frontend subsystems (the IBEFEInterface), due to evolution of requirements. The changes propagated to a number of units including the FEClient and the Orchestration module. These changes caused substantially adapting existing components and introducing new components (e.g., the BlockPrepare controller).

At the end of that increment it was of a surprise to the FEClient team and the Orchestration team that all units worked together correctly after integration, from the first run, without any visible errors in the execution of the system. They spend a substantial effort to bring units together based on their experience with more conventional development approaches.

The feedbacks and comments from the project and team leaders were very positive, and the units appeared to be stable and reliable. But the project leaders wanted to know more about whether the use of formal methods affected the quality of the code.

| Parameter | FEClient | Orchestration |
|---|---|---|
| ASD ELOC | 12,854 | 15,631 |
| Handwritten ELOC | 15,462 | 3,970 |
| Total ELOC | 28,316 | 19,601 |
| ASD defects | 3 | 6 |
| Handwritten defects | 8 | 2 |
| Total defects | 11 | 8 |
| ASD defect/KLOC | 0.23 | 0.38 |
| Handwritten defect/KLOC | 0.52 | 0.5 |
| Total defect/KLOC | 0.4 | 0.41 |
| Test ELOC | 10,943 | 3,966 |

**Table 4.6** Summary of the end quality of the units

Table 4.6 summarizes the end quality of the units. When comparing the quality of ASD code with the manually written code it appears that the ASD code is better although the manually written code is very simple. Nevertheless, the entire developed code exhibits high quality figures and favorably compares to the standard of 1-25 defects per KLOC for software developed in industrial settings [61].

The quality of the ASD developed code depends on many factors, including thorough specification reviews and behavioral verification. The model checking technology covered all potential execution scenarios, so that defects were found early and quickly with the click of a button. It further took the place of manual coverage testing which is typically time consuming and uncertain.

The quality of the manually coded components depends on many other factors such as external specification of components, code reviews, automatic code standard checks and coverage testing. For the FEClient case, it appeared to us that code review was far more effective than coverage testing, and that more issues had been found during review than in testing. But, unit testing had provided key benefits of preparing coverage reports, detecting potential memory leaks and optimizing memory usage.

Developers of the unit were committed to 100 percent function coverage and 80 percent statement coverage. The team had tracked coverage testing using the NCover tool, which reports the percentage of functions and statements covered by sets of tests. Such reports are mandatory for other test teams before they can start the subsystem and system tests.

Although more effort and time were spent to obtain the ASD code compared to other manually coded components, project and team leaders were positive about the end result since there were only few errors submitted along the construction of the module.

Although there were some delays on the deliverable of the ASD components due to spending more time in learning ASD and obtaining verifiable designs, there was less time spent in testing compared to the other manually coded modules. This at the end led to less time spent to resolve problems found in testing at later stages compared to manually coded modules [33].

Finally, the ASD components were robust against the increasing evolution and the frequent changes of requirements. Team members appreciated the end quality of the software, relating that to the firm specification and formal verification technologies provided by the ASD approach.

Team members appreciated the ultimate quality of the developed software. The behavioral 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.

# Chapter 5

# Evaluating the Use of Formal Techniques

[1] Evaluating the Use of Formal Techniques in Industry (To appear as a technical report in 2012)

[2] Analyzing the effects of formal methods on the development of industrial control software. In: Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM 2011)

## 5.1 Introduction

In this chapter we evaluate the effectiveness of applying the formal techniques of ASD to the quality and the productivity of the developed units of the Frontend and the Backend subsystems. We compare the units that incorporate ASD with others that were developed in more traditional development methods, highlighting the main challenges and the key issues encountered during the application.

In Section 5.2 we detail the steps accomplished to empirically evaluate the ASD techniques in the Frontend subsystem. Furthermore, we compare the resulting quality and productivity against the industry standards used in Philips Healthcare and also those others reported worldwide in the literature. In Section 5.3 we detail the resulting units in the Backend subsystem, comparing the quality of the units incorporating ASD with the units that do not.

## 5.2 Evaluating the Formal Techniques in the Frontend

### 5.2.1 Introduction

The focus of this section is to investigate whether the use of the ASD formal techniques actually resulted in visible improvements for the software units of the Frontend subsystem, demonstrating key issues encountered when incorporating the techniques with the industrial development processes.

As discussed in the previous chapters, the ASD technology was successful in a number of projects at Philips [42, 70, 74]. In this chapter, we investigate whether the approach is also successful in other projects with other environments, circumstances and backgrounds. Moreover, we provide a more detailed quantitative analysis of the effectiveness of these techniques.

In order to determine any major improvement to the software developed by these formal techniques, we first need to collect quantifiable evidences and analyze these techniques empirically and rigorously. Therefore, this requires answering the following research questions:

- Can these formal techniques deliver product code? And if so, is the code of high or low quality?
- Do these formal techniques require more time in development compared to traditional development?
- What about the productivity using these techniques?
- Do the techniques require specialized mathematicians for a successful application?

- Do these techniques always produce zero-defect software? If not, which type of errors is expected and how many compared to industrial standards?

- Which artifacts should we consider when evaluating these techniques empirically? Should we include the formal models or the related code?

This section is structured as follows. Section 5.2.2 briefly sketches the industrial context. Section 5.2.3 details the phases of developing the ASD components and the main issues and limitations encountered. We analyze the data related to the developed code to evaluate the ASD approach in Section 5.2.4. In Section 5.2.5 we extend our analysis to study the cause and the type of errors that could escape the formal techniques. Section 5.2.6 details the end results comparing the ASD code with other code developed at Philips and also with the industry standards reported worldwide. Finally, Section 5.2.7 contains our conclusions by answering the research questions mentioned earlier.

### 5.2.2 Description of the project

Due to migrating to the component-based centralized architecture, some of the units of the Frontend subsystem were redesigned in order to adapt to the required changes. Other units were kept intact and were entirely reused in the new architecture (e.g., the units that control the hardware devices).

The Frontend subsystem includes nearly 1030K of effective lines of code (excluding blanks and comments). It consists of 22 units in total, two of which are the target of this study: the Application State Controller (ASC) and the Frontend Adapter (FEA). Both units comprise a number of modules that includes concurrent components with well-defined interfaces and responsibilities.

One of the key responsibilities of the ASC is managing the external X-ray requests, sent by the clinical operators via dedicated X-ray pedals and hand-switches. The unit counts, filters, and ensures priorities of such requests. It is also responsible for maintaining the overall system state and coordinating interactions with units surrounding the Frontend subsystem (e.g., the Backend subsystem).

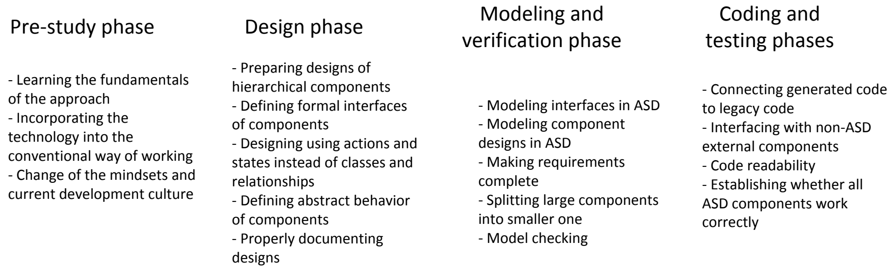
The FEA unit is mainly responsible for the interfaces with the Backend external subsystem, through a network. Its functionality is to some extent similar to the FEClient unit presented earlier but it is located in the Frontend subsystem (FEClient is located in the Backend subsystem). The unit exchanges information related to patients and their exam details, issued by the Backend subsystem. Furthermore, it is responsible for monitoring the presence of other remote subsystems and converting incoming information to readable xml and string formats.

The ASD technology was used to formally develop the internal components of the units, expecting that this formal technique detects and prevents any potential errors at early stages of the project. For example, clinical users can press the pedals or the switches at any time, even if the internal components or the hardware are not prepared or configured yet for image acquisition (i.e., before the Backend exchanges all patient related details).

### 5.2.3 The development process of ASD components

In this section we report about the effort spent during three increments for developing ASD components of both the ASC and the FEA units, starting from January 2011 till August 2011. The effort was accomplished by 5 full-time team members, who had sufficient programming knowledge, but limited skills in formal methods.

Along the three increments, the ASD formal technology was smoothly and tightly integrated with the traditional development processes, but the related details are outside the scope of this section. We instead refer to [74, 70] or to the previous chapters for more information on how the technology was incorporated and exploited in other projects of iXR.



Pre-study phase

- Learning the fundamentals of the approach
- Incorporating the technology into the conventional way of working
- Change of the mindsets and current development culture

Design phase

- Preparing designs of hierarchical components
- Defining formal interfaces of components
- Designing using actions and states instead of classes and relationships
- Defining abstract behavior of components
- Properly documenting designs

Modeling and verification phase

- Modeling interfaces in ASD
- Modeling component designs in ASD
- Making requirements complete
- Splitting large components into smaller one
- Model checking

Coding and testing phases

- Connecting generated code to legacy code
- Interfacing with non-ASD external components
- Code readability
- Establishing whether all ASD components work correctly

**Figure 5.1** Summary of main challenges and issues when incorporating ASD in industry

This section concentrates more on the main challenges and the key issues encountered when incorporating the ASD approach for the first time to the projects of the Frontend. Figure 5.1 visualizes the phases of the ASD development processes and the main challenges encountered during each phase. Below they are briefly described.

**Pre-study phase.** The three increments were preceded by a pre-study period, during which the team attended a one week ASD course to get familiar with the approach and its related technologies. The course was limited to learning how to use the ASD:Suite (e.g., how to fill-in the tables and verify them compositionally using model checking and how to generate and integrate the code).

Furthermore, during the pre-study period the entire reference architecture of the Frontend subsystem has been discussed, to get consensus about the functionality concerns among all teams.

After the responsibility of the units that incorporate ASD was clarified, the ASD team explored various design alternatives and approaches to compose suitable ASD components. During that time, the team was still confronted with the steep learning curve of how to make a design that fits ASD, so that ample time was required before the team became skilled in the technology. The problem was not in the ASD tooling itself but in the design philosophy behind ASD which required certain architectural patterns to enable efficient model checking.

As there was a lack of ASD design guides, cookbooks or patterns that could aid the team to incorporate the technology in the way of working and to prepare formally verifiable components, team members initially tried to adapt the ASD technology to the existing way of developing software at Philips Healthcare.

Since the object-oriented method was the dominating approach of design and implementation, team members started investigating the suitability of ASD for developing Object-Oriented designs. For this purpose, the team reviewed and thoroughly studied the well-known object-oriented design patterns [35], trying to model them alternatively using ASD.

As a result, the team realized that developing object-oriented designs with using ASD is not productive since ASD is an action-oriented, component-based technology. Hence, after quite some time the team understood that the successful application of the technology requires changing the development culture and the mind-sets.

**Design phase.** Based on the knowledge gained, team members prepared initial design drafts containing hierarchal components with well-defined interfaces and responsibilities, without using object-oriented patterns. The designs were iteratively reviewed and refactored until they were approved by team members.

In order to document and increase the awareness of these designs among the team and other important stakeholders (e.g., external testers), components and their interrelationships were extensively described in informal documents, using Microsoft Word and Visio. Such documents detailed not only the decomposition of components and their responsibilities but also the happy flow of information crossing their boundaries using a number of sequence diagrams. Later, the documents were extended with the unhappy flow of messages. Subsequently, the state machines representing the behaviors of each component were clearly identified and understood among all members.

However, one drawback encountered was that the informal documents quickly became substantially big, due to the intensive level of details of (un)happy flow of information they include. This eventually caused difficulty of reading, understanding, maintaining and adapting such lengthy documents, especially at later stages were components had to be redesigned from scratch due to numerous design snags, encountered during formal verification using model checking.

| Component | Models | Rule | States | Transitions | Time | ELOC |
|---|---|---|---|---|---|---|
| | | cases | | | (Sec) | |
| **ASC unit** | | | | | | |
| AcquisitionController | 6 | 432 | 16912 | 79840 | 2 | 1685 |
| AcquisitionRequests | 5 | 837 | 192320 | 1027032 | 20 | 2821 |
| ASCExamEpxManager | 4 | 165 | 160 | 339 | < 1 | 928 |
| ASCMisc | 4 | 58 | 2633 | 4963 | < 1 | 963 |
| ASCMiscDecoupler | 2 | 13 | 7 | 9 | < 1 | 356 |
| RequestCounter | 5 | 113 | 45560 | 76233 | 3 | 1133 |
| RequestFilter | 3 | 381 | 51790 | 226910 | 1 | 994 |
| RunController | 4 | 842 | 8058224 | 41150288 | 444 | 5126 |
| **FEA unit** | | | | | | |
| AcqCtrlAcqRequests | 3 | 353 | 2802 | 7805 | < 1 | 465 |
| BETStateless | 3 | 404 | 8640 | 38704 | < 1 | 518 |
| CmdStateless | 3 | 62 | 12 | 24 | < 1 | 704 |
| CxaAdpMain | 7 | 2794 | 139824 | 461292 | 50 | 5944 |
| DAcqCtrl | 4 | 1300 | 117412 | 364066 | 13 | 3206 |
| DActivation | 3 | 103 | 4480 | 14688 | < 1 | 585 |
| Decoupler | 2 | 11 | 3 | 3 | < 1 | 239 |
| DWrapper | 4 | 408 | 8928 | 46736 | < 1 | 674 |
| FEProxyVE | 6 | 5270 | 597096 | 1764366 | 289 | 9645 |
| FEProxyVEStateless | 6 | 202 | 4976 | 19320 | < 1 | 1753 |
| FSStateless | 3 | 31 | 220 | 636 | < 1 | 554 |
| UGStateless | 3 | 88 | 44388432 | 76939995 | 6853 | 951 |

**Table 5.1** Modeling and verification statistics of the ASD components

**Modeling and verification phase.** Next we describe the modeling and the verification activities.

**Modeling the ASD components.** When the informal documents had been reviewed and approved by team members, the team started specifying the state machines of each component, using the ASD:Suite version 6.2.0. Following the ASD recipe, the models of the components were specified stepwise, in a top-down fashion, starting with interface models and refining them by detailed design models and other interfaces.

In general, filling-in the ASD tables was a straightforward task. The team carefully filled-in and thought about every stimulus in every state, asking questions of what must be done as responses to stimulus events not addressed by the incomplete informal documents and the state machines.

Indeed, the technology consequently helped the team finding omissions and gaps in the initial set of requirements and the designs and hence initiated early discussions with various stakeholders. Subsequently, this increased the quality of requirements and designs at early phases of development, and saved development time, comparing to addressing these issues at later stages of the project, using a more traditional development method.

However, due to the specification completeness some ASD interface models were too big, hard to review and to maintain since the specified protocols included too detailed behavior, although the ASD:Suite was lately extended with a nice filtering feature. This resulted in decomposing the components further into smaller components, to increase readability and maintainability.

Moreover, specification completeness forces having complete requirements at early stages of development, and this is relatively challenging especially for a complex system like the Frontend (lead architects often tend to concentrate on important, high abstract aspects of the system and to leave the details to later stages of the project). Although one could choose to work on a subset of the interface, still the corresponding specification must be complete.

Table 5.1 lists the developed ASD components for the ASC and the FEA units, demonstrating the number of ASD design and interface models for each component (column 3), and the sum of specified rule-cases (column 4). Each component includes one implemented interface model, one design model, and a number of used and implemented interface models.

**Formal verification.** Formal verification started with reviewing the ASD specifications, which were checked row-by-row for correctness and traceability to the informal requirements. After specification reviews, the models were verified using model checking.

With a click of a button, the model checker detected various deadlocks, livelocks, illegal scenarios, and race conditions, which required immediate fixes in the models. In some cases, solving these errors caused redesigning the ASD components, especially when the fix made model checking impossible.

Another important factor of redesigning was the lack of abstraction in the behavior of the components. For example, the use of substantial number of stateless callback events entering a queue in any order may take FDR2 hours or days to calculate the state space that captures all possible execution scenarios. Adding a new event, due to the evolution of requirements for instance, may make verification virtually impossible. Hence, components were re-factored to remedy this shortcoming and to eventually accomplish verification in shorter time.

During the formal verification, the state space explosion problem was frequently encountered although FDR2 could favorably handle billions of states. Note that, within our industrial context, waiting for a very long time is usually not acceptable due to the tight deadlines of the incremental planning. Worst, before an error is discovered, the model checker may have already taken a substantial time. Hence, developers are forced to wait for the model checker repeatedly during the process of removing errors.

During formal verification, the team realized that different design styles could substantially influence the verifiability of components using formal techniques [75]. Hence, they avoided a number of design styles that may needlessly increase the state space and the time required for verification [50].

Based on the knowledge gained, the team could eventually obtain a set of formally verified components. Each ASD component was verified using the predefined set of ASD properties. Table 5.1, columns 5-7, includes the output data from FDR2 related to the refinement check (since it is most time consuming) for the design model of each ASD component, showing the states, transitions, and time in seconds. The data indicates that most of the components were verified within acceptable range of states and transitions, calculated in a reasonable time by FDR2. Note that, some components took less than a second for verification, covering all possible execution circumstances of the component.

**Coding and testing phases.** Finally we describe the code generation, integration and testing.

**Code generation and integration.** As soon as ASD components had been formally verified and further reviewed to ensure that fixing the model checking errors did not break the intended behavior, the code was generated automatically and integrated with the code of other components via glue code. The last column of Table 5.1 quantifies the number of the effective lines of code, automatically generated in the C++ programming language, excluding blank and comment lines.

Experience shows that, in a more conventional development method, integrating components is a nightmare, due to the substantial effort required to bring all components to correctly work together. Therefore, it was surprising that integrating ASD components with one another was always smooth, did not require any glue code, and often was accomplished without any visible errors. However, integrating ASD code with the surrounding components that did not undergo formal verification (e.g., manually developed or legacy code) caused some errors and delays.

In some cases, we needed to review the generated code when integrating the code or when analyzing and debugging some error traces. In general, the generated code was comprehendible. The main advantages of the generated code over the handwritten code are:

- The code of all components has the same coding shape and structure, following similar coding standards;
- All generated code is readable and is constructed using well-known patterns, such as the object-oriented *State* and *Proxy* patterns;
- The code does not contain ad-hoc solutions, workarounds or tricks;
- The structure of the code allows systematic translation to other languages or other type of models, if needed;

- Changes are done at the model level, not at the low-level code;
- The code is thoroughly verified using model checking; previous and current experiences [74, 70] indicate that errors left behind are simple to find and to fix;
- The support for different programming languages makes models more platform-independent than hand-written code.

**Testing.** An apparent limitation of the ASD compositional verification is that it is impossible to formally establish whether the combination of ASD components yields the required behavior. It is not possible to express domain specific properties or to relate events in implemented and used interfaces. However, at the end of each increment, the Frontend units including the ASD components were thoroughly and extensively tested by specialized test team, using various types of testing such as model-based statistical test, smoke test, regression test, performance test, etc, of which details are outside the scope of this thesis. As a result of testing, a few errors were detected; details will be given in subsequent sections.

### 5.2.4 Data analysis

In this section, we analyze the project data in order to compare the end quality of the units which incorporate ASD with the other units of the Frontend. The purpose is to establish whether the use of ASD formal techniques had a positive or negative impact on the quality of the developed software, within the organization.

To accomplish this goal, we took several steps. We started with collecting the total number of effective lines of code that had been newly introduced plus the changed legacy code, for every unit separately. We restricted ourselves to the period bounded by two baselines representing the start and the end of the three increments.

After that, we carefully investigated the reports of 202 submitted defects, and partitioned them in order to individually analyze the coding defects and others arising due to, for instance, documents, requirements or designs issues. We then considered a total of 104 reports related to coding, and distributed them to the respective units. These errors were unveiled during the in-house subsystem tests and are not post-release defects or found after delivery.

Table 5.2 depicts the results of our data collection, showing only a subset of the Frontend units. The units that exhibit similar results or were not changed during the increments have been excluded for readability purposes. Hidden from the table is also the amount of reused or legacy code, developed and verified during previous projects.

Given the obtained data, we could estimate the overall defect density of each unit separately, as depicted in the last column of Table 5.2. Although the ASD units appeared to be slightly better than some other units, at that stage of the analysis process, the effectiveness of the ASD formal techniques were controversial so we felt that more refined analysis is needed to obtain more insight. Note that some of the manually coded units exhibit zero defects, but the reason was that most of the changes were on the level of interfaces and not on the core internal behavior of the units.

| Unit | ASD ELOC | HW ELOC | ASD Defects | HW Defects | ASD D/ KELOC | HW D/ KELOC | D/KELOC |
|---|---|---|---|---|---|---|---|
| ASC | 14006 | 5784 | 13 | 10 | 0.92817 | 1.72891 | 1.16 |
| FEA | 25238 | 9489 | 1 | 18 | 0.03962 | 1.89693 | 0.55 |
| IGC | 0 | 6,326 | 0 | 35 | N/A | 5.53272 | 5.53 |
| SC | 0 | 3,340 | 0 | 0 | N/A | 0 | 0 |
| SIModel | 0 | 6,202 | 0 | 0 | N/A | 0 | 0 |
| IDS | 0 | 2,650 | 0 | 7 | N/A | 2.64151 | 2.64 |
| NGUI | 0 | 2,848 | 0 | 0 | N/A | 0 | 0 |
| PandB | 0 | 3,161 | 0 | 1 | N/A | 0.31636 | 0.31 |

**Table 5.2** Statistical data of some units of the Frontend

Therefore, we decided to separate the ASD code and the handwritten (HW) code and analyze them in isolation. The ASD code and the manually written code are quantified in the second and third column of Table 5.2. Then, we studied the defects of ASD units once more, distinguishing ASD defects from those related to the handwritten code, as listed in the fourth and fifth columns. Consequently, we could estimate the defect rate of ASD and non ASD code, as depicted in columns six and seven.

As can be inferred from the table, for the two units that incorporate ASD, the quality of ASD code seems to be better than the corresponding manually written code, especially for the FEA unit, which received only one defect. After studying the corresponding defect report we found that the error was not only related to ASD components but rather to a chain of ASD and non ASD components, due to a missing parameter in a method. Nevertheless, developers of the FEA unit, clearly, could deliver close to zero defects per thousand ASD lines of code.

However, the FEA unit received more errors related to the handwritten code. More than half of these errors were caused by a component responsible of string and xml manipulations. This consequently was the reason of degrading the quality of the entire unit.

This is different for the ASD unit, which included numerous errors in the ASD code, but still the end quality was slightly better than the manually written code. The amount of ASD errors in the ASC unit motivated us to further investigate the behavior of the components in depth and also to study the nature and the type of the detected errors, which were left behind by the ASD formal technologies. We detail this in the subsequent section.

The reason of why these errors were not detected using ASD is that the ASD technology does not support any means to define and verify system-specific properties. Although the ASD:Suite allows uploading CSP code, by which verifiers can specify additional properties, this is very impractical since the internal structure of the CSP model is totally hidden from verifiers and requires a CSP expert to be present at the Verum company.

External testers have different mindsets and attitudes of looking at the behavior of the components than the development team and hence they could detect some error scenarios not addressed by the development team. Most of the detected errors were due to experiencing unintended, unexpected behaviors (e.g., after a user presses and releases a number of pedals and switches it was expected that a particular type of X-ray resumes but it did stop). But, as a result of the fixed set of properties the ASD technology currently supports, none of these errors was due to deadlocks or illegal interactions (e.g., there were no crashes due to null reference exceptions or illegal invocation of methods at some states).

The reason that the FEA unit included fewer defects is that the unit implements a protocol of interaction between the Frontend and the other subsystems and does not include a complex functional behavior compared to the ASC unit. Hence, the main specification of the FEA unit is represented by the protocol specified in its ASD interface model. Moreover, this interface model has been reviewed frequently because it is used by other subsystems.

### 5.2.5 Analyzing the cause of ASD errors

The purpose of this section is to figure out the root cause of the errors in the ASD components that escaped the ASD formal techniques. To do so, we began with analyzing the ASD components individually, especially those related to the ASC unit, trying to identify the responsible component that contributed much to the defects and why. Moreover, we investigated whether there is a correlation between the complexity of ASD components and the volume of received errors.

Initially, this appeared to be challenging since we did not possess any systematic means to measure the complexity of components at the model level. Therefore, we alternatively assessed the components using two other means. First, we evaluated the models concerning understandability and the review easiness, based on our “common sense”. Second, we chose to systematically analyze the generated code, using available code analysis tools and techniques. The two steps are detailed below.

In the first step, we evaluated the readability of the design model of each component, and assigned review codes based on the degree of complexity in reviewing and comprehending the models: VE= Very easy, E=Easy, M= Moderate, C= Complex and VC= Very Complex. Table 5.3 column 3 includes the result of the assessment.

For example, the *RunController* component is considered to be complex since it includes 23 input stimuli, for which a response is required to be defined in 16 states, and 10 state variables being used as predicates in almost all rule-cases. Often there are several rule-cases for a certain stimulus in a state, to distinguish combinations of values of variables.

A sample of a complex specification of rule-cases in the *RunController* design model is depicted in Figure 5.2. Visible in the figure are only 5 rule-cases related to the *FailedSC* stimulus event, which had been duplicated 31 times with different combinations of predi-

| Component | Review | Avg | Avg | Max | Avg | Avg | Defects |
|---|---|---|---|---|---|---|---|
| | | M/C | S/M | CC | depth | CC | |
| **ASC unit** | | | | | | | |
| AcquisitionController | E | 3.42 | 3.3 | 16 | 1.03 | 1.41 | 1 |
| AcquisitionRequests | M | 5 | 6.3 | 18 | 1.26 | 2.26 | 3 |
| ASCExamEpxManager | E | 3.17 | 2.8 | 4 | 0.88 | 1.25 | 0 |
| ASCMisc | VE | 3.62 | 2.2 | 5 | 0.79 | 1.08 | 0 |
| ASCMiscDecoupler | VE | 3.5 | 1.7 | 3 | 0.73 | 1.14 | 0 |
| RequestCounter | M | 3.57 | 5 | 13 | 1.17 | 1.90 | 1 |
| RequestFilter | M | 4.38 | 7 | 18 | 1.33 | 2.73 | 1 |
| RunController | C | 7.61 | 10.5 | 157 | 1.42 | 5.71 | 7 |
| **FEA unit** | | | | | | | |
| AcqCtrlAcqRequests | VE | 4.08 | 2.3 | 3 | 0.99 | 1.17 | 0 |
| BETStateless | VE | 2.57 | 1.5 | 3 | 0.84 | 1.13 | 0 |
| CmdStateless | VE | 4.05 | 2.2 | 3 | 0.8 | 1.09 | 0 |
| CxaAdpMain | C | 7.44 | 5.5 | 13 | 1.09 | 2.08 | 0 |
| DAcqCtrl | M | 7.95 | 3.7 | 16 | 0.95 | 1.27 | 1 |
| DActivation | VE | 3.1 | 2.1 | 5 | 0.84 | 1.17 | 0 |
| Decoupler | VE | 3 | 1.4 | 3 | 0.7 | 1.14 | 0 |
| DWrapper | VE | 2.46 | 1.6 | 4 | 0.84 | 1.16 | 0 |
| FEProxyVE | M | 16 | 3.9 | 13 | 0.9 | 1.12 | 0 |
| FEProxyVEStateless | VE | 4.4 | 2.5 | 9 | 0.85 | 1.12 | 0 |
| FSStateless | VE | 2.82 | 1.6 | 3 | 0.8 | 1.11 | 0 |
| UGStateless | VE | 4.58 | 2.3 | 4 | 0.84 | 1.07 | 0 |

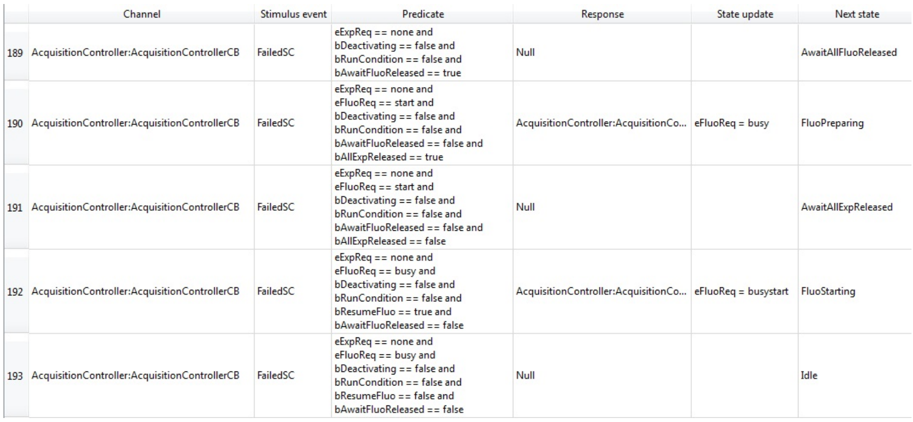
**Table 5.3** Statistical data of ASD components

cate values in the original model.

On the other hand, the user-guidance *UGStateless* component of the FEA unit is considered to be very easy since it contains only two states without the use of any predicates. The component is enabled or disabled to allow the flow of information traffic to other components. Although the component is easy to read and to understand, it was the most time-consuming component when verified using model checking, as can be seen in Table 5.1. The reason is that the component receives a large number of callback events stored in a queue. Therefore, FDR2 calculates all possible orders these callbacks may take in the queue.

We distributed the errors to the respective components, as depicted in Table 5.3 column 9. As can be seen, most of the ASD errors reside in the *RunController* component, unveiling an apparent correlation between the complexity of the component and the errors found.

In the second step, we performed a static analysis on the generated code, seeking similar correlations between complex code and the error density. The motivation was that the complexity of the models can also be reflected in the corresponding generated code. We used the *SourceMonitor* tool Version 3.2 [48] to analyze the generated code since the features catered by the tool seemed to be a good fit to our aim.



| | Channel | Stimulus event | Predicate | Response | State update | Next state |
|---|---|---|---|---|---|---|
| 189 | AcquisitionController:AcquisitionControllerCB | FailedSC | eExpReq == none and<br>bDeactivating == false and<br>bRunCondition == false and<br>bAwaitFluoReleased == true | Null | | AwaitAllFluoReleased |
| 190 | AcquisitionController:AcquisitionControllerCB | FailedSC | eExpReq == none and<br>eFluoReq == start and<br>bDeactivating == false and<br>bRunCondition == false and<br>bAwaitFluoReleased == false and<br>bAllExpReleased == true | AcquisitionController:AcquisitionCo... | eFluoReq = busy | FluoPreparing |
| 191 | AcquisitionController:AcquisitionControllerCB | FailedSC | eExpReq == none and<br>eFluoReq == start and<br>bDeactivating == false and<br>bRunCondition == false and<br>bAwaitFluoReleased == false and<br>bAllExpReleased == false | Null | | AwaitAllExpReleased |
| 192 | AcquisitionController:AcquisitionControllerCB | FailedSC | eExpReq == none and<br>eFluoReq == busy and<br>bDeactivating == false and<br>bRunCondition == false and<br>bResumeFluo == true and<br>bAwaitFluoReleased == false | AcquisitionController:AcquisitionCo... | eFluoReq = busystart | FluoStarting |
| 193 | AcquisitionController:AcquisitionControllerCB | FailedSC | eExpReq == none and<br>eFluoReq == busy and<br>bDeactivating == false and<br>bRunCondition == false and<br>bResumeFluo == false and<br>bAwaitFluoReleased == false | Null | | Idle |

**Figure 5.2** An example of complex rule-cases

Table 5.3 includes some selected code metrics produced by the tool: the average number of methods per class (Avg M/C), the average statements per method (Avg S/M), the maximum cyclomatic complexity (Max CC), the average block depth, and the average cyclomatic complexity (Avg CC).

As can be seen in the table, the *RunController* component also appears to be very complex compared to other generated code of other components. Notable is that the 157 max complexity of the *RunController* component resides in the corresponding code of the rule-cases of the *FailedSC* stimulus event presented earlier in Figure 5.2. In the code, the rule-cases are represented by a single method (called *FailedSC*) containing 30 related if-else statements.

The amount of errors of the *RunController* component motivated us to study the type of these errors and their evolution. Four of the seven errors had a similar cause, namely missing updates of state variables before a state transition. The team solved these errors by adding more rule-cases with different predicates and also additional state variables, which increased the complexity even more.

Another error was caused by missing storing values in the data part of the component. Two errors were caused due to missing requirements, where external verifiers tested some behavior not yet implemented in the units.

## 5.2.6 The quality and performance results

In this section, we evaluate the end quality and productivity of the developed ASD units, by comparing them against the industry standards reported worldwide in the literature. The best sources we could find are [53, 61, 66, 62], where interesting statistics related to a number of projects of different types and sizes are thoroughly described. We concentrate more on those statistics revealed for software systems analogous to the Frontend.

In [76], Linger and Spangler compared the quality of code developed under the Cleanroom software engineering formal method to the industry standard of 30-50 defects per KLOC. Jones in [53] presents an average of 1.7 coding errors per function point (p. 102, Table 3.11), which roughly corresponds to a range of 14-58 defects per C++ KLOC (after consulting Table 3.5 on p. 78 of [53]).

Furthermore, McConnell presents in [62] (page 242, Table 21-11) a breakdown of industry average defect rate based on software size, where our type of software is estimated to include 4-100 defects per KLOC. In [61] McConnell explicitly states an industry average of 1-20 defects per KLOC during the construction of software (p. 521), and also mentioned a range of 10-20 defects per KLOC, in the Microsoft Applications Division, during in-house testing. McConnell classifies the expected error density based on the project size, where our system is expected to include 4-100 errors per KLOC (p. 652, Table 27-1).

At Philips Healthcare, project and team leaders are often concerned with delivering the features and function planned at the start of the incremental development. The corresponding delivered code should exhibit 6 allowable defects per KLOC with an average productivity of 2 LOC per staff-hour, at the end of each increment. Any code that includes more errors, during the in-house construction, can be rejected and sent back to the developers, but this rarely happened.

From the data presented earlier in Table 5.2, we conclude that the ASD technology could deliver quality code, averaging the ASD code of the ASC and FEA units to only 0.36 defects per KLOC. The entire code of the two units reveals an average of 0.86 defects per KLOC. Hence, if we consider the range of 4-10 defects per KLOC presented by McConnell in [61], the ASD code appears to be nearly 10 times better in quality.

Similar to comparing the quality of the units, we compare the productivity in terms of the number of lines of code per staff-hour (hours per month = 132, based on 22 days, 6 hours a day). McConnell in [61] (p. 522) confirms that it is cheaper and better to develop high-quality software than it is to develop and mend low-quality software, so that it was of no surprise that a formal Cleanroom project could deliver nearly 5.61 LOC per staff-hour [77]. He also mentioned an industry average of 250-300 LOC per work-month (1.9-2.3 LOC per staff-hour), including all non-coding overhead.

Furthermore, McConnell in [61] (p. 653 Table 27-2) lists the expected productivity based on the size of the software product. Given these statistics, the productivity of software similar to the Frontend subsystem ranges between 700 to 10,000 LOC per staff-year with a nominal value of 2,000 LOC per staff-year (i.e., 0.4 to 6.3 with a nominal value of 1.3 LOC per staff-hour).

In [53], Jones presents a productivity figure of 435 ELOC for C++ per staff-month (page 73, Table 3.4), which is equal to 3.3 ELOC per staff-hour. Furthermore, he provides figures for the average and best practices for systems software (p. 339, Table 9.7). There, Jones presents a 4.13 and 8.76 as an average and best-in-class function points per staff-month (which is equal to 1.7 and 3.5 as an average and best-in-class LOC per staff-hour, after consulting Table 3.5 on p. 78).

Cusumano et al., in [28] studied the data of a number of worldwide projects, and found a median of 450 LOC per staff-month (3.41 LOC per staff-hour) for the data sample related to the Japanese and European projects. The projects include roughly 48 percent generated code.

Consequently, we can use the above measures to compare the productivity of ASD developed units. The total time spent for developing the ASD components is 2378 hours, affording an average of 16 ELOC per staff-hour. The total time spent for developing the two units, including the time spent for non-coding overhead, is 5701 hours, which favorably yields 9.6 ELOC per staff-hour. Therefore, if we consider the range of 0.4 to 6.3 LOC per staff-hour, provided by McConnell in [61], the productivity of ASD appears to be nearly 3 times more.

The ASD productivity is increased due to many factors, we briefly address them below.

- The ASD:Suite is a model-based tool that provides easy to use graphical user interface. Specification of models is done by filling the ASD tables with actions selected form an ordered list and using clicks. Compared to traditional text editors, the graphical interface of ASD is more productive.
- The code is obtained automatically by a click of a button so manual coding overhead is reduced.
- Integrating ASD components is automatic and requires no overhead at all.
- The internal structure of ASD generated code is not tested since it is formally verified (but black-box testing is required).
- Bugs (if any) are easy to locate and to fix.

However, the ASD generated code is made more general so that it includes some code that can be removed in some cases. As estimation, nearly one third of the code size can be removed in case the code is manually tuned but changing the generated code manually is not allowed.

Finally, the developed units appeared to be stable and reliable against the frequent changes of requirements. Team and project leaders were satisfied with the results and decided to exploit the ASD technology for developing other parts of the system.

### 5.2.7 Conclusions of applying ASD in the Frontend

We demonstrated a formal component-based approach called ASD and how its formal techniques were exploited for developing control components of two software units of the

Frontend subsystem. We elaborated more on the issues encountered during its application. The result of our investigation shows that the ASD technology could effectively deliver quality software with high productivity. Below, we answer the questions raised in the introduction.

*Can these formal techniques deliver product code? And if so, is the code of high or low quality?* Compared to the industry standards of Philips and worldwide, the ASD technology could clearly deliver product code that exhibits good quality figures. But, obtaining this level of quality depended on many factors like the experience of users and the level of abstractions in designs, for instance.

Notable is that a widespread view in industry is that formal methods are immature and impractical and may not scale to industrial applications. But, as we previously showed, the ASD technology could scale due to the use of the compositional development and verification of components. Although this allows parallel developments of components, establishing formally whether the entire system is behaving as expected is still challenging and currently requires other informal means such as testing.

*Do they require more time in development compared to traditional development?* Another common view in industry is that formal methods consume plenty of the development time and often cause delays to software delivery. But, on the contrary, the ASD technology could save the development time, although a lot of time was spent in the pre-study period to learn the fundamentals of the technology. Experiences show that after acquiring the learning curve, experienced ASD teams with sufficient knowledge of the technology and the context can achieve considerably shorter development cycles. This is because the ASD technology systematically allows preventing problems earlier rather than detecting and fixing problems at later stages, which is time-consuming and very costly.

*What about the productivity using these techniques?* As can be inferred from the presented data, developers were 3-5 times more productive compared to industrial standards. This resulted from the fact that developers were only concerned with models, from which verified code is generated automatically with the click of a button, and hence reducing implementation overhead. Another important fact is that less or even no time was spent for integration and manual testing, which are usually time consuming and uncertain.

*Do the techniques require specialized mathematicians for a successful application?* Some of the team members had formal methods skills limited to few courses at the university level, but others had no previous knowledge in formal methods at all. The ASD technology was very utilizable since all formal details were hidden from end-users. Our experiences also show that software engineers in industry are often very skilled and proficient in programming but not as well at constructing abstract designs and formal specification and verification.

*Do these techniques always produce zero-defect software? If not, which type of errors is expected and how many compared to industrial standards?* As we saw before, although the components were formally specified and verified, still errors were found during testing. Hence, the used techniques do not always lead to defect-free software. However, our study shows that the formally developed software contains very few defects.

*Which artifacts should we consider when evaluating these techniques empirically? Should we include the formal models or the related code?* Evaluation requires a product in hand to be analyzed so that a challenging task was analyzing the product and the complexity at the models level. As there was no systematic means to analyze the models, we analyzed the corresponding code. An interesting future direction is to develop tools and techniques for establishing static analysis of models.

## 5.3 Evaluating the Formal Techniques in the Backend

Below we report about the end quality results of the developed units during two consecutive projects of the Backend subsystem, starting from January 2008 till the end of 2010. We compare the quality of the ASD code and the manually written code for each unit and also compare the units that incorporate ASD with the others that were developed using the traditional development methods. The data presented in this section represents statistical data of the developed units including the Orchestration and the FEClient units, detailed in the previous chapter. The data is related to only the first four increments since this work was accomplished before the work of the previous chapter.

| | | Effective lines of code | | | Defects | | Defects/ KELOC | | |
|---|---|---|---|---|---|---|---|---|---|
| ASD used | Unit | HW | ASD | ASD% | HW | ASD | HW | ASD | Total |
| Yes | FEClient | 15,462 | 12,153 | 44.01% | 9 | 2 | 0.582 | 0.1646 | 0.398 |
| Yes | Orchestration | 3,970 | 8,892 | 69.13% | 3 | 4 | 0.757 | 0.45 | 0.544 |
| Yes | XRayIP | 14,270 | 2,188 | 13.29% | 27 | 0 | 1.892 | 0 | 1.641 |
| No | Acquisition | 6,140 | 0 | 00.00% | 33 | 0 | 5.375 | NA | 5.375 |
| No | BEC | 7,007 | 0 | 00.00% | 44 | 0 | 6.279 | NA | 6.279 |
| No | EPX | 7138 | 0 | 00.00% | 7 | 0 | 0.981 | NA | 0.981 |
| No | FEAdapter | 13,190 | 0 | 00.00% | 18 | 0 | 1.365 | NA | 1.365 |
| No | QA | 23,303 | 0 | 00.00% | 90 | 0 | 3.862 | NA | 3.862 |
| No | Status Area | 8,969 | 0 | 00.00% | 52 | 0 | 5.798 | NA | 5.798 |
| No | TSM | 6,681 | 0 | 00.00% | 7 | 0 | 1.048 | NA | 1.048 |
| No | UIGuidance | 20,458 | 0 | 00.00% | 23 | 0 | 1.124 | NA | 1.124 |
| No | Viewing | 19,684 | 0 | 00.00% | 294 | 0 | 14.936 | NA | 14.936 |

**Table 5.4** Statistical data during in-house construction of the Backend units

For the Backend subsystem we also analyzed every defect submitted along the development process of the units and excluded those related to documentation (e.g., specification or requirement documents) from the calculations.

Table 5.4 summarizes the accomplished work and reports about the quality results of the Backend software units. For each unit the number of effective (logical) lines of code (ELOC) written manually, and those generated automatically from ASD models are reported. The total number of submitted defects against the ASD code and the handwritten code of each unit is depicted in the table. These numbers represent the errors captured during in-house design, implementation, integration, and testing phases (i.e., not post-release errors). The last three columns contain defect rates for the ASD and the non-ASD code. The lase column represents the defect rate for the entire unit, e.g., the rate for the Orchestration unit is 0.5 errors per KLOC, and for the Viewing unit is 14.9 errors per KELOC.

To compare the quality of ASD code with the handwritten code we select the FEClient, the Orchestration and the XRayIP units as representative cases in order to establish a pairwise comparison. Each unit was developed by an independent team who was responsible for constructing the ASD components and the other components of the unit. The teams had attended ASD courses and had the same domain knowledge and expertise. For each unit the extent of applying ASD varies and hence the units differ in the percentage of ASD code. So, we consider not only one unit developed by one team but we perform multiple comparisons. This avoids selection bias and reduces variations. Hence, it gives a more accurate view of the outcomes.

As can be seen from Table 5.4 the size of ASD code and the handwritten code in the FEClient unit are nearly the same but there were more errors found in the handwritten code. The quality of ASD code appears to be roughly 4 times better than the handwritten code in this unit.

The Orchestration includes more ASD code but still the error density in the handwritten code is relatively higher. It is notable that most of the handwritten components can be implemented using ASD but the team decided to rather develop them conventionally. Both the ASD components and the handwritten components include state behavior but clearly the formally developed part appears to be better.

The XRayIP unit includes more handwritten code. The unit represents a case with a significant reduction in the amount of errors in the control part. It exhibits zero defects in the ASD components while there were substantial errors discovered in the handwritten code.

Thus, we can conclude that the ASD technology improved the quality of the developed units. It is also clear that the units that incorporate ASD are of better quality compare to the others. Therefore, it seems that applying ASD to develop the control part of the other units that do not undergo formal techniques may improve their quality.

The members of teams attribute the ultimate quality of the developed units to the rigor and disciplines enforced by the ASD technology. The conventionally developed units did not undergo formal correctness verification. However, the units were strictly examined at different levels of code and design reviews, unit test, integration test, and system test. Traditionally developed units of the Backend are already of good quality.

Other factors besides software errors can play a key role for defects to emerge. For example, some defects of the Viewing unit appeared due to migrating to new services supplied by external suppliers. Over 40% of the depicted defects of this unit are cosmetic errors (e.g., “Annotation text: font size not changed”), which do not cause potential failures during the execution of the system.

Evaluating ASD in the Backend is reported in detail in [42]. At the time of writing the paper, calculating the productivity figure was not of our main concern as we were more interested in comparing the quality of the ASD code. However, we can calculate the productivity of the ASD code given the reported hours in the paper.

| Unit | DM | IM | Rule cases | States | Time (sec) | Hours |
|---|---|---|---|---|---|---|
| Orchestration | 8 | 26 | 2,857 | 15,954,291 | 1,847 | 1288 |
| FEClient | 1 | 15 | 5779 | 1,996,830 | 230 | 696 |
| XrayIp | 1 | 6 | 1,051 | 2,874 | 0 | 268 |

**Table 5.5** Statistical data of ASD models in the Backend

Table 5.5 presents statistical data related to ASD models and the hours spent in developing the ASD components. The productivity of the ASD code of FEClient is 17.5 ELOC per staff-hour while it is 6.9 and 8.2 for the Orchestration and the X-rayIP respectively, averaging the productivity to 10.9 ELOC per staff-hour. Comparing the productivity of these representative units of the Backend to the average standard of 3.4 presented previously in this chapter we can conclude that the productivity of ASD is nearly 3 times better than the average standard.

## 5.4 Other formal techniques used in other projects

In this section we present a number of worldwide industrial projects that incorporated formal techniques in software development and report about their achieved quality and productivity. We considered the work accomplished in [86] and its references (over 70 publications) as a starting point to seek these projects (the work includes a survey and a comprehensive review of formal methods application in industry). Furthermore, we searched other projects using web search engines and by visiting a number of home pages hosting the formal techniques.

We classified all publications based on the year of publication and reviewed them from 2012 backwards until 2002 (10 years). Through this period we found relatively very few publications reporting quantitative evidences that demonstrate the impact of formal techniques in industry (most detailing case studies of applying formal methods at different stages of software development plus the performance of the formal method tools and not the performance of the projects). This motivated us to search even backwards until late 80s’.

Table 5.6 summarizes the results by listing 14 projects that fit our goal. The projects are listed in a chronological order, highlighting the used formal technique, the size of the developed software, the programming language used for implementation, the defect density, the productivity in terms of the lines of code produced per staff-hour, and the phase where the errors were counted.

| Year | Project | Technology | Size (KLOC) | Prog. Language | D/KLOC | LOC/ man-hour | Phase |
|---|---|---|---|---|---|---|---|
| 1988 | IBM COBOL Structuring Facility | Cleanroom | 85 | PL/I | 3.4 | 5.6 | Certification test |
| 1989 | NASA Satellite Control | Cleanroom | 40 | FORTRAN | 4.5 | 5.9 | Certification test |
| 1991 | IBM System Product | Cleanroom (partial) | 107 | Mixed | 2.6 | 3.7 | All Testing |
| 1996 | MaFMeth | VDM + B | 3.5 | C | 0.9 | 13.6 | Unit testing |
| 1998 | Line 14, Paris metro | B method | 86 | Ada | Zero | - | Testing + after release |
| 1999 | DUST-EXPERT | VDM | 17.5 and 15.8 | C++ and Prolog | $\leq$1 | - | Testing + after release |
| 1999 | Siemens FALKO | ASM | 11.9 | C++ | 0.17 | 2.2 | After release |
| 2000 | VDMTools | VDM | 23.3 | C++ | - | 12.4 | - |
| 2000 | TradeOne, Tax Exem. | VDM | 18.4 | C++ | 0.7 | 10 | Integration test |
| 2000 | TradeOne, Option | VDM | 64.4 | C++ | 0.67 | 7 | Integration test |
| 2006 | Tokeneer ID Station | SPARK | 10 | Ada | Zero | 6.3 | Reliability test after delivery |
| 2007 | Shuttle, Paris airport | B Method | 158 | Ada | - | - | - |
| 2011 | Philips, Backend | ASD | 23.2 | C# | 0.26 | 10.9 | Along development |
| 2012 | Philips, Frontend | ASD | 39.2 | C++ | 0.36 | 16.5 | Subsystem Test |

**Table 5.6** List of projects incorporated formal techniques in software development

Linger in [77] listed 15 projects where the Cleanroom formal engineering method was used, summarizing the results achieved for each project. All developed systems exhibit quality figures that range between 0 to 5 errors per KLOC with an average of 3.3 errors per KLOC. Compared to the mentioned range of 30 to 50 errors/KLOC in traditional development, Linger concluded that the developed systems present remarkable quality.

From the 15 projects, three projects that reveal quality and productivity figures are depicted in Table 5.6. The other remaining projects do not include productivity figures so they were excluded from our consideration.

The first Cleanroom project, the IBM COBOL Structuring Facility, included a team of 6 developers (it was their first development project). The product exhibits 3.4 errors per KLOC and several major components were certified without experiencing any error. The average productivity was 5.6 LOC per man-hour.

The second Cleanroom project was concerned with the development of a Satellite controller carried out by the Software Engineering Laboratory at NASA. The system included 40 KLOC of FORTRAN and certified with 4.5 errors/KLOC. The productivity was 5.9 LOC/person-hour, resulting in an 80% improvement over previous averages known in the laboratory.

The third Cleanroom project included 50 people, developed complex system software at IBM using various programming languages. The system exhibited 2.6 errors/KLOC, where five of its eight components experienced no errors during testing. The team used the Cleanroom method for the first time [46].

The MaFMeth project [13] incorporated VDM and the B method in software development. The project included 3500 lines of C code, estimated by the developers from 8000 lines of generated code. The reported errors were found during unit testing. Errors found during validation testing or errors found after releasing the system were not available. Productivity was 13.6 LOC per hour with an error density of 0.9 error per KLOC.

The B Method was used to develop safety critical components of the automatic train operating system, the metro line 14 in Paris [4, 11]. Members of the development and validation teams were newcomers in formal methods, but were supported by B experts, when needed. The developed components included 86,000 of mathematically verified Ada code and the system did not experience any error during independent testing or after release. However, the numbers regarding the effort spent for the entire development were missing except for the correctness proofs. Nevertheless, the project was completed successfully and went off according to the schedule [11].

The DUST-EXPERT project incorporated VDM to software development and was successfully released with 15.8 KLOC of prolog and 17.5 KLOC of C++ [25]. The system exhibited less than one error per KLOC. The errors were found during coverage testing and after product release. Productivity was above industry norms but there were no figures provided in the paper. Productivity of Prolog was less than C++ due to the high-abstract level and the rigorous way the core Prolog was generated. Developers involved were skilled in formal methods.

The Abstract State Machines (ASM) were used in the development of a software package, developed at Siemens, called FALKO [17]. The package was redesigned from scratch due its complexity. The newly developed package included roughly 11900 generated and manually written C++ LOC, developed in nearly 66 man-weeks effort. Two errors were found after product release and were fixed directly in the generated code. The end quality was 0.17 and the productivity was 2.2 LOC per hour.

In [57], VDM was used to formally develop some components of the VDM toolset itself. Table 5.6 includes some metrics related to the VDM-C++ code generator component, which was formally specified using VDM but manually implemented using C++. The productivity was 12.4 LOC per hour but compared with other components the productivity was less due to its complexity and the involvement of new employees. No figures related to the errors found were reported.

The VDM toolset was used for developing some components of a business application, called TradeOne [32]. Two subsystems of the application were developed under the control of VDM++ where the first exhibits a productivity figure of nearly 10 lines of C++ and Java per staff-hour while the second subsystem 6.1 lines of C++ per staff-hour. The error rates of both subsystems are less than one error per KLOC. The errors were reported during integration testing and there were no errors discovered after releasing the product.

The Tokeneer ID Station (TIS) project was carried out by Praxis High Integrity Systems and accomplished by three part-time members over one year using SPARK [9]. The overall productivity of the TIS core system was 6.3 LOC of Ada per man-hour. The system did not exhibit any error whatsoever during reliability testing and also since delivery.

The B Method was successful in developing software components of a driverless shuttle at Paris Roissy Airport [4, 7]. The developed software included 158 KLOC of generated code. The generated code includes lots of duplications due to the lack of sharing in the code and the intermediate steps performed by the code generator. The code is estimated to be 60 KLOC in size after tuning. However, there was no data available about the total time spent in development or the number or type of errors encountered along the construction of the software.

To this end, we found it rather difficult to compare the quality and productivity of these projects since they were developed in different programming languages and represent distinct software domains. Furthermore, the reported errors were counted at different stages of each project.

But as a general conclusion we can say that the formal techniques used in these projects had favorably increased the productivity and the quality of the developed systems although there were no discussions regarding the weaknesses and the main difficulties encountered when applying the techniques.

Nevertheless, given the level of the gained quality and productivity it is worth investigating why most organizations do not incorporate formal engineering methods in their development processes. Since the 80's, it is still difficult to see whether the use of these techniques in industry is increasing, decreasing or remaining constant over time.

# Chapter 6

# Proposed Design Guidelines

Specification guidelines to avoid the state space explosion problem. In: Proceedings of the 4th IPM International Conference, Fundamentals of Software Engineering, 2011.

Specification guidelines to avoid the state space explosion problem. CS-Report 10-14, Eindhoven University of Technology, 2010. (extended version)

## 6.1 Introduction

Components of the early presented projects were mainly verified using model checking but this often led to the state explosion. In order to avoid this based on the experiences gained from these projects and others, we propose a number of specification guidelines to obtain verifiable, model-checkable components.

As commonly known, the model checking tools require a model or specification that precisely describes the behavior of components to be verified. The model is thoroughly checked to prove that the components always satisfy certain requirements. The tools perform enumerative, systematic exploration of all (or part of) possible execution scenarios of the modeled system. The set of the execution scenarios are often characterized by an LTS (label transition system or state space) which contains states and transitions labeled by actions performed by the components.

But the behavioral verification is limited by the state space explosion problem, which arises when the verified components include a huge number of states that cannot fit into memory, despite the use of clever verification algorithms and powerful computers. Although model checking technologies nowadays available can potentially handle billions of states, they still suffer from this problem. For some practical cases developers have to wait hours or days for outcomes resulting from the tools when verifying even a single property of their systems.

We believe that the state space explosion problem must also be dealt with in another way, namely by designing models such that their behavior can be verified. We call this *design for verifiability* or *modeling for verifiability*.

What we propose is that systems are designed such that the state spaces of their behavioral models are small. This does impose certain restrictions on how systems can behave. For instance, maintaining local copies of data throughout a system blows up the state space, and is therefore not recommended. When modeling existing systems, we advocate that sometimes the models are shaped such that the state space does not grow too much.

Compared to the development of state space reduction techniques, design for verifiability is a barely addressed issue. The best we could find is [58], but it primarily addresses improvements in verification technology, too. Specification styles from the perspective of expressiveness have been addressed [85], but verifiability is also not really an issue here.

In this chapter we provide seven specification guidelines that we learned by specifying complex realistic systems (e.g. traffic control systems, medical equipment, domestic appliances, communication protocols). For each specification guideline we provide an application from the domain of traffic light controllers.

For each guideline we give two examples. The first one does not take the guideline into account and the second does. Generally, the first specification is very natural, but leads to a large state space. Then we provide a second specification that uses the guideline. We show by a transition system or a table that the state space that is using the guideline is much smaller. The 'bad' and the 'good' specification are in general not behaviorally equivalent (for instance in the sense of branching bisimulation) but as we will see, they both capture the application's intent. Throughout this chapter we use mCRL2 for formal specification and state space generation.

In hindsight, we can say that it is quite self evident why the guidelines have a beneficial effect on the size of the state spaces. Some of the guidelines are already quite commonly used, such as reordering information in buffers, if the ordering is not important. The use of synchronous communication, although less commonly used, also falls in this category. Other guidelines such as information polling are not really surprising, but specifiers appear to have a natural tendency to use information pushing instead. The use of confluence and determinacy, and external specifications may be foreign to most specifiers.

Although we provide a number of guidelines that we believe are really important for the behavioral modellist, we do not claim completeness. Without doubt we have overlooked a number of specification strategies that are helpful in keeping state spaces small. Hopefully this chapter will be an inspiration to investigate state space reduction from this perspective, which ultimately can be accumulated in effective teaching material, helping both students and working practitioners to avoid the pitfalls of state space explosion.

## 6.2 Overview of design guidelines

In this section we give a short description of the seven guidelines that we present in this chapter. Each guideline is elaborated in its own section with an example where the guideline is not used, and an intuitively equivalent description where the guideline is used. We provide information on the resulting state spaces, showing why the use of the guideline is advantageous.

I **Information polling**. This guideline advises to let processes ask for information, whenever it is required. The alternative is to share information with other components, whenever the information becomes available. Although, this latter strategy clearly increases the number of states of a system, it appears to prevail over information polling in most specifications that we have seen.

II **Global synchronous communication**. If more parties communicate with each other, it can be that a component 1 communicates with a component 2, and subsequently, component 2 informs a component 3. This requires two consecutive communications and therefore two state transitions. By using multi-actions it is possible to let component 1 communicate with component 2 that synchronously communicates with a component 3. This only requires one transition. By synchronising communication over different components, the number of states of the overall system can be substantially reduced.

III **Avoid parallelism among components**. If components operate in parallel, the state space grows exponentially in the number of components. By sequentialising the behavior of these components, the size of the total state space is only the sum of the sizes of the state spaces of the individual components. In this latter case state spaces are small and easy to analyse, whereas in the former case analysis might be quite hard. Sequentialising the behavior can for instance be done by introducing an arbiter, or by letting a process higher up in the process hierarchy to allow only one sub-process to operate at any time.

IV **Confluence and determinacy**. When parallel behavior cannot be avoided, it is useful to model such that the behavior is $\tau$-confluent. In this case $\tau$-prioritisation can be applied when generating the state space, substantially reducing the size of the state space. Modelling a system such that it is $\tau$-confluent is not easy. A good strategy is to strive for determinacy of behavior. This means that the 'output' behavior of a system must completely be determined by the 'input'. This is guaranteed whenever an internal action (e.g. receiving or sending a message from/to another component) can be done in a state of a single component, then no other action can be done in that state.

V **Restrict the use of data**. The use of data in a specification is a main cause for state-space explosion. Therefore, it is advisable to avoid using data whenever possible. If data is essential, try to categories it, and only store the categories. For example, instead of storing a height in millimeters, store *too_low*, *right_height* and *too_high*. Avoid buffers and queues getting filled, and if not avoidable try to apply confluence and $\tau$-prioritization. Finally, take care that data is only stored in one way. E.g., storing the names of the files that are open in an unordered buffer is a waste. The buffer can be ordered without losing information, substantially reducing the state footprint.

VI **Compositional design and reduction**. If a system is composed out of more components, it can be fruitful to combine them in a stepwise manner, and reduce each set of composed components using an appropriate behavioral equivalence. This works well if the composed components do not have different interfaces that communicate via not yet composed components. So typically, this method does not work when the components communicate in a ring topology, but it works very nicely when the components are organized as a tree.

VII **Specify the external behavior of sets of sub-components**. If the behavior of sets of components are composed, the external behavior tends to be overly complex. In particular the state space is often larger than needed. A technique to keep this behavior small is to separately specify the expected external behavior first. Subsequently, the behaviors of the components are designed such that they meet this external behavior.

## 6.3 Guideline I: Information polling

One of the primary sources of many states is the occurrence of data in a system. A good strategy is to only read data when it is needed and to decide upon this data, after which the data is directly forgotten. In this strategy data is polled when required, instead of pushed to those that might potentially need it. An obvious disadvantage of polling is that much more communication is needed. This might be problematic for a real system, but for verification purposes it is attractive, as the number of states in a system becomes smaller when using polling.

Currently, it appears that most behavioral specifications use information pushing, rather than information polling. E.g., whenever some event happens, this information is immediately shared with neighboring processes.

Furthermore, we note that there is also a discussion of information pulling versus information pushing in distributed system design from a completely different perspective [5]. Here, the goal is to minimize response times of distributed systems. If information when needed must be requested (=pulled) from other processes in a system, the system can become sluggish. But on the other hand, if all processes inform all other processes about every potentially interesting event, communication networks can be overloaded, also leading to insufficient responsiveness. Note that we prefer the verb ‘to poll’ over ‘to pull’, because it describes better that information is repeatedly requested.

In order to illustrate the advantage of information polling, we provide two specifications. The first one is ‘bad’ in the sense that there are more states than in the second specification. We are now interested in a system that can be triggered by two sensors $trig_1$ and $trig_2$. After both sensors fire a trigger, a traffic light must switch from red to green, from green to yellow, and subsequently back to red again. For setting the aspect of the traffic light, the action $set$ is used. One can imagine that the sensors are proximity sensors that measure whether cars are waiting for the traffic light. Note that it can be that a car activates the sensors, while the traffic light shows another color than red. In Figure 6.1 this system is drawn.

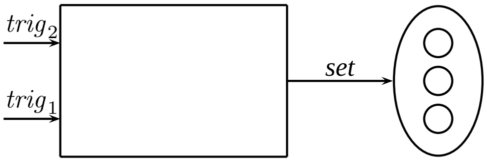


**Figure 6.1** A simple traffic light with two sensors

First, we define a data type $Aspect$ which contains the three aspects of a traffic light.

**sort** $Aspect = \mathbf{struct}\ green \mid yellow \mid red;$

The pushing controller is very straightforward. The occurrence of $trig_1$ and $trig_2$ indicate that the respective sensors have been triggered. In the pushing strategy, the controller must be able to always deal with incoming signals, and store their occurrence for later use. Below, the pushing process has two booleans $b_1$ and $b_2$ for this purpose. Initially, these booleans are false, and the traffic light is assumed to be red. The booleans become *true* if a trigger is received, and are set to *false*, when the traffic light starts with a $green$, $yellow$ and $red$ cycle.

**proc** $Push(b_1, b_2{:}\mathbb{B}, c{:}Aspect)$
$\quad = \underline{trig}_1 \cdot Push(true, b_2, c)$
$\quad + \underline{trig}_2 \cdot Push(b_1, true, c)$
$\quad + (b_1 \wedge b_2 \wedge c \approx red) \rightarrow \underline{set}(green) \cdot Push(false, false, green)$
$\quad + (c \approx green) \rightarrow \underline{set}(yellow) \cdot Push(b_1, b_2, yellow)$
$\quad + (c \approx yellow) \rightarrow \underline{set}(red) \cdot Push(b_1, b_2, red);$
**init** $Push(false, false, red);$

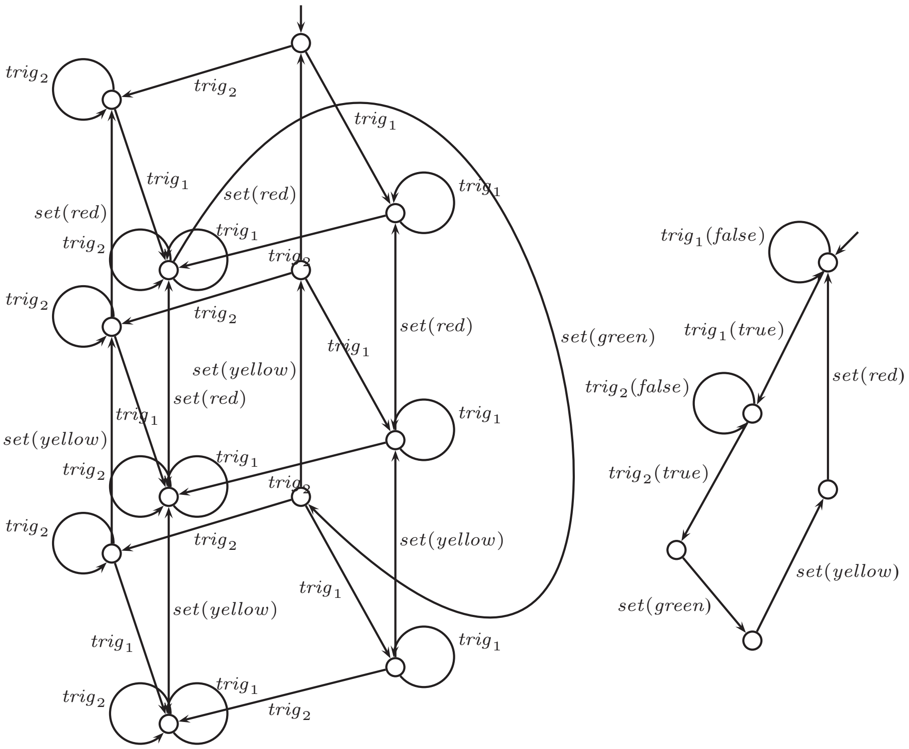


**Figure 6.2** Transition systems of push/poll processes

The polling controller differs from the pushing controller in the sense that the actions $trig_1$ and $trig_2$ now have a parameter. It checks whether the sensors have been triggered using the actions $trig_1(b)$ and $trig_2(b)$. The boolean $b$ indicates whether the sensor has been triggered ($true$: triggered, $false$: not triggered). In $Poll$, sensor $trig_1$ is repeatedly polled, and when it indicates by a $true$ that it has been triggered, the process goes to $Poll_1$. In $Poll_1$ sensor $trig_2$ is polled, and when both sensors have been triggered $Poll_2$ is invoked. In $Poll_2$ the traffic light goes through a colour cycle and back to $Poll$.

**proc** $Poll = \underline{trig}_1(false)\cdot Poll + \underline{trig}_1(true)\cdot Poll_1;$
$Poll_1 = \underline{trig}_2(false)\cdot Poll_1 + \underline{trig}_2(true)\cdot Poll_2;$
$Poll_2 = \underline{set}(green)\cdot\underline{set}(yellow)\cdot\underline{set}(red)\cdot Poll;$
**init** $Poll;$

The transition systems of both systems are drawn in Figure 6.2. At the left the diagram for the pushing system is drawn, and at the right the behavior of the polling traffic light controller is depicted. The diagram at the left has 12 states while the diagram at the right has 5, showing that even for this very simple system polling leads to a smaller state space.

## 6.4 Guideline II: Use global synchronous communication

Communication along different components can sometimes be modeled by synchronizing the communication over all these components. For instance, instead of modeling that a message is forwarded in a stepwise manner through a number of components, all components engage in one big action that says that the message travels through all components at once. In the first case there is a new state for every time the message is forwarded. In the second case the total communication only requires one extra state. The use of global synchronous communication can be justified if passing this message is much faster than the other activities of the components, or if passing such a message is insignificant relative to the other activities.

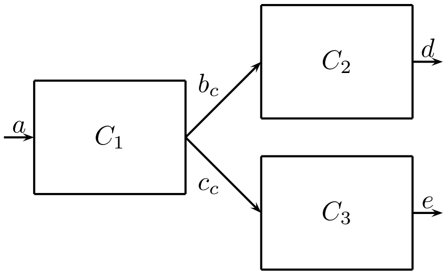


**Figure 6.3** Synchronous/asynchronous message passing

Several formalisms use global synchronous interactions as a way to keep the state space of a system small. The co-ordination language REO uses the concept very explicitly [6]. A derived form can be found in Uppaal, which uses committed locations [37].

To illustrate the effectiveness of global synchronous communication, we provide the system in Figure 6.3. A trigger signal enters at $a$, and is non-deterministically forwarded via $b_c$ or $c_c$ to one of the two components at the right. Non-deterministic forwarding is used, to make the application of confluence impossible (see guideline IV). One might for instance think that there is a complex algorithm that determines whether the information is forwarded via $b_c$ or $c_c$, but we do not want to model the details of this algorithm. After being passed via $b_c$ or $c_c$, the message is forwarded to the outside world via $d$ or $e$. To illustrate the effect on state spaces, it is not necessary that we pass an actual message, and therefore it is left out.

The asynchronous variant is described below. Process $C_1$ performs $a$, and subsequently performs $b_s$ or $c_s$, i.e. sending via $b$ or $c$. The process $C_2$ reads via $b$ by $b_r$, and then performs a $d$. The behavior of $C_3$ is similar. The whole system consists of the processes $C_1$, $C_2$ and $C_3$ where $b_r$ and $b_s$ synchronize to become $b_c$, and $c_r$ and $c_s$ become $c_c$. The behavior of this system contains 8 states and is depicted in Figure 6.4 at the left.

**proc** $C_1 = \underline{a}\cdot(b_s + c_s)\cdot C_1;$
$C_2 = b_r\cdot\underline{d}\cdot C_2;$
$C_3 = c_r\cdot\underline{e}\cdot C_3;$

**init** $\nabla_{\{\underline{a},b_c,c_c,\underline{d},\underline{e}\}}(\Gamma_{\{b_r|b_s\rightarrow b_c,c_r|c_s\rightarrow c_c\}}(C_1||C_2||C_3));$

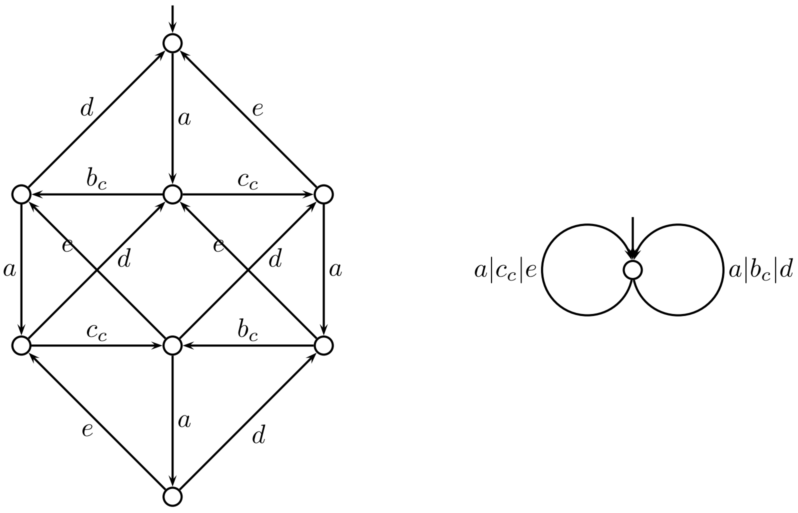


**Figure 6.4** Transition systems of a synchronous and an asynchronous process

The synchronous behavior of this system can be characterized by the following mCRL2 specification. Process $C_1$ can perform a multi-action $a|b_s$ (i.e. action $a$ and $b_s$ happen exactly at the same time) or a multi-action $a|c_s$. This represents the instantaneous receiving and forwarding of a message. Similarly, $C_2$ and $C_3$ read and forward the message instantaneously. The effect is that the state space only consists of one state as depicted in Figure 6.4 at the right.

**proc** $C_1 = \underline{a}|b_s{\cdot}C_1 + \underline{a}|c_s{\cdot}C_1;$
$C_2 = b_r|\underline{d}{\cdot}C_2;$
$C_3 = c_r|\underline{e}{\cdot}C_3;$
**init** $\nabla_{\{\underline{a}|c_c|\underline{e},\underline{a}|b_c|\underline{d}\}}(\Gamma_{\{b_r|b_s\rightarrow b_c, c_r|c_s\rightarrow c_c\}}(C_1||C_2||C_3));$

The operator $\nabla_{\{a|c_c|e,a|b_c|d\}}$ allows the two multi-actions $a|c_c|e$ and $a|b_c|d$, enforcing in this way that in both cases these three actions must happen simultaneously.

## 6.5 Guideline III: Avoid parallelism among components

When models have many concurrent components that can independently perform an action, then the state space of the given model can be reduced by limiting the number of components that can simultaneously perform activity. Ideally, only one component can perform activity at any time. This can for instance be achieved by one central component that allows the other components to do an action in a round robin fashion.

It very much depends on the nature of the system whether this kind of modeling is allowed. If the primary purpose of a system is the calculation of values, sequentializing appears to be defendable. If on the other hand the sub-components are controlling all kinds of devices, then the parallel behavior of the sub-components might be the primary purpose of the system and sequentialization can not be used.

In some specification languages explicit avoidance of parallel behavior between components has been used. For instance Esterel [12] uses micro steps which can be calculated per component. In Promela there is an explicit atomicity command, grouping behavior in one component that is executed without interleaving of actions of other components [47].

As an example we consider $M$ traffic lights guarding the same number of entrances of a parking lot. See Figure 6.5 for a diagrammatic representation where $M{=}3$. A sensor detects that a car arrives at an entrance. If there is space in the garage, the traffic light shows green for some time interval. There is a detector at the exit, which indicates that a car is leaving. The number of cars in the garage cannot exceed $N$.

The first model is very simple, but has a large state space. Each traffic light controller ($TLC$) waits for a trigger of its sensor, indicating that a car is waiting. Using the $enter_s$ action it asks the $Coordinator$ for admission to the garage. If a car can enter, this action is allowed by the co-ordinator and a traffic light cycle starts. Otherwise the $enter_s$ action is blocked. The $Coordinator$ has an internal counter, counting the number of cars. When a $leave$ action takes place, the counter is decreased. When a car is allowed to enter (via $enter_r$), the counter is increased.

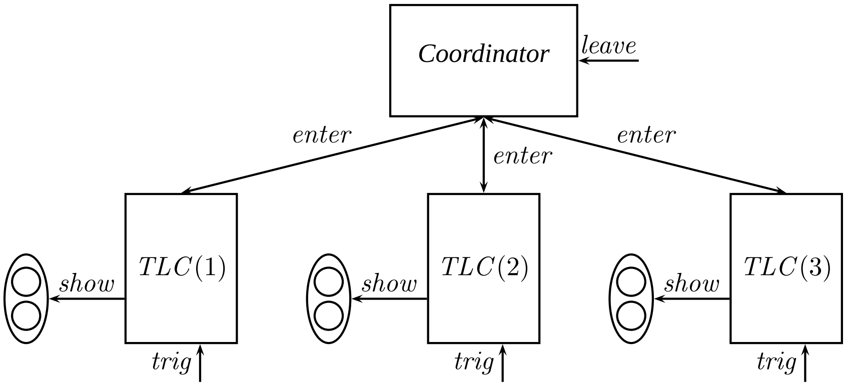


**Figure 6.5** A parking lot with three entrances

**proc** $Coordinator(count{:}\mathbb{N})$
$\quad = (count{>}0)\rightarrow\underline{leave}\cdot Coordinator(count-1)$
$\quad + (count{<}N)\rightarrow enter_r\cdot Coordinator(count+1);$

$TLC(id{:}\mathbb{N}^+)$
$\quad = \underline{trig}(id)\cdot enter_s\cdot\underline{show}(id, green)\cdot\underline{show}(id, red)\cdot TLC(id);$

**init** $\nabla_{\{\underline{trig},\underline{show},enter_c,\underline{leave}\}}(\Gamma_{\{enter_s|enter_r\rightarrow enter_c\}}$
$\quad (Coordinator(0)\|TLC(1)\|TLC(2)\|TLC(3)));$

The state space of this control system grows exponentially with the number of traffic light controllers. In columns 2 and 4 of Table 6.1 the sizes of the state spaces for different $M$ are shown. It is also clear that the number of parking places $N$ only contributes linearly to the state space.

Following the guideline, we try to limit the amount of parallel behavior in the traffic light controllers. So, we put the initiative in the hands of the co-ordinator in the second model. It assigns the task of monitoring a sensor to one of the traffic light controllers at a time. The traffic controller will poll the sensor, and only if it has been triggered, switch the traffic light to green. After it has done its task, the traffic light controller will return control to the co-ordinator. Of course if the parking lot is full, the traffic light controllers are not activated. Note that in this second example, only one traffic light can show green at any time, which might not be desirable.

$$
\begin{array}{ll}
\textbf{proc} & \mathit{Coordinator}(\mathit{count}{:}\mathbb{N},\ \mathit{active\_id}{:}\mathbb{N}^{+}) \\
 & \quad = (\mathit{count}{>}0) {\rightarrow} \underline{\mathit{leave}} {\cdot} \mathit{Coordinator}(\mathit{count}{-}1,\ \mathit{active\_id}) \\
 & \quad + (\mathit{count}{<}N) {\rightarrow} \mathit{enter}_s(\mathit{active\_id}) {\cdot} \sum_{b:\mathbb{B}} \mathit{enter}_r(b) {\cdot} \\
 & \qquad \mathit{Coordinator}(\mathit{count}{+}\mathrm{if}(b,1,0), \mathrm{if}(\mathit{active\_id}{\approx}M, 1, \mathit{active\_id}{+}1)); \\
 & \\
 & \mathit{TLC}(\mathit{id}{:}\mathbb{N}^{+}) \\
 & \quad = \mathit{enter}_r(\mathit{id}) {\cdot} \\
 & \qquad (\ \underline{\mathit{trig}}(\mathit{id}, \mathit{true}) {\cdot} \underline{\mathit{show}}(\mathit{id}, \mathit{green}) {\cdot} \underline{\mathit{show}}(\mathit{id}, \mathit{red}) {\cdot} \mathit{enter}_s(\mathit{true}) + \\
 & \qquad\quad \underline{\mathit{trig}}(\mathit{id}, \mathit{false}) {\cdot} \mathit{enter}_s(\mathit{false}) \\
 & \qquad ) {\cdot} \\
 & \qquad \mathit{TLC}(\mathit{id}); \\
 & \\
\textbf{init} & \nabla_{\{\underline{\mathit{trig}}, \underline{\mathit{show}}, \mathit{enter}_c, \underline{\mathit{leave}}\}} (\Gamma_{\{\mathit{enter}_s | \mathit{enter}_r \rightarrow \mathit{enter}_c\}} \\
 & \qquad\qquad\qquad (\mathit{Coordinator}(0,1) || \mathit{TLC}(1) || \mathit{TLC}(2) || \mathit{TLC}(3)));
\end{array}
$$

As can be seen in Table 6.1 the state space of the second model only grows linearly with the number of traffic lights.

| $M$ | parallel | restricted | parallel | restricted |
|---|---|---|---|---|
| | $(N = 10)$ | $(N = 10)$ | $(N = 100)$ | $(N = 100)$ |
| 1 | 44 | 61 | 404 | 601 |
| 2 | 176 | 122 | $1,616$ | $1,202$ |
| 3 | 704 | 183 | $6,464$ | $1,803$ |
| 4 | $2,816$ | 244 | $25,856$ | $2,404$ |
| 5 | $11,264$ | 305 | $103,424$ | $3,005$ |
| 6 | $45,056$ | 366 | $413,696$ | $3,606$ |
| 10 | $11.5\,10^6$ | 610 | $106\,10^6$ | $6,010$ |

**Table 6.1** State space sizes of parking lot controllers ($M$: no. of traffic lights, $N$: no. of parking places)

## 6.6 Guideline IV: Confluence and determinacy

In [39, 67] it is described how $\tau$-confluence and determinacy can be used to assist process verification. By modelling such that a system is $\tau$-confluent, verification can become substantially easier. The formulations in [39, 67] are slightly different; we use the formulation from [39] because it is more suitable for verification purposes.

A transition system is $\tau$-*confluent* iff for every state $s$ that has an outgoing $\tau$ and an outgoing $a$-transition, $s \xrightarrow{\tau} s'$ and $s \xrightarrow{a} s''$, respectively, there is a state $s'''$ such that $s' \xrightarrow{a} s'''$ and $s'' \xrightarrow{\tau} s'''$. This is depicted in Figure 6.6. Note that $a$ can also be a $\tau$, but then the states $s'$ and $s''$ must be different.

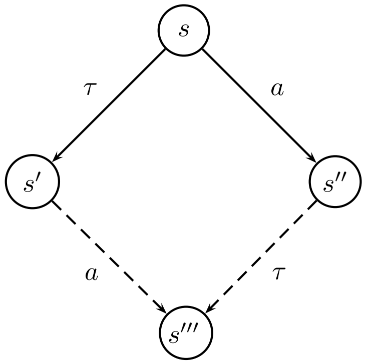


**Figure 6.6** Confluent case

When traversing a state space of a $\tau$-confluent transition system, it is allowed to ignore all outgoing transitions from a state that has at least one outgoing $\tau$-transition, except one outgoing $\tau$-transition. This operation is called *$\tau$-prioritization*. It preserves branching bisimulation equivalence [82] and therefore almost all behaviorally interesting properties of the state space. There is one snag, namely that if the resulting $\tau$ transitions form a loop, then the outgoing transitions of one of the states on the loop must be preserved. The first algorithm to generate a state space while applying $\tau$-prioritization is described in [15]. When $\tau$-prioritization has been applied to a transition system, large parts of the 'full' state space have become unreachable. Of the remaining reachable states, many have a single outgoing $\tau$-transition $s \xrightarrow{\tau} s'$. The states $s$ and $s'$ are branching bisimilar and can be mapped onto each other, effectively removing one more state. Furthermore, all states on a $\tau$-loop are branching bisimilar and can therefore be merged into one state, too.

If a state space is $\tau$-confluent, then $\tau$-prioritization can have a quite dramatic reduction of the size of the state space. This technique allows to generate the prioritized state space of highly parallel systems with thousands of components. In Figure 6.7 a $\tau$-confluent transition system is depicted before and after application of $\tau$-prioritization, and the subsequent merging of branching bisimilar states.

To employ $\tau$-prioritization, a system must be defined such that it is $\tau$-confluent. The main rule of thumb is to take care that if an internal action can be performed in a state of a component, no other action can be done in that state. These internal actions include sending information to other components. If data is received, it must be received from only one component. A selection among different components offering data is virtually always non confluent. Note that in particular pushing information generally destroys confluence. Pushed information must always be received, so, in particular it must be received while internal choices are made and information is forwarded.

We model a simple crossing system that contains two traffic lights. First, we are not

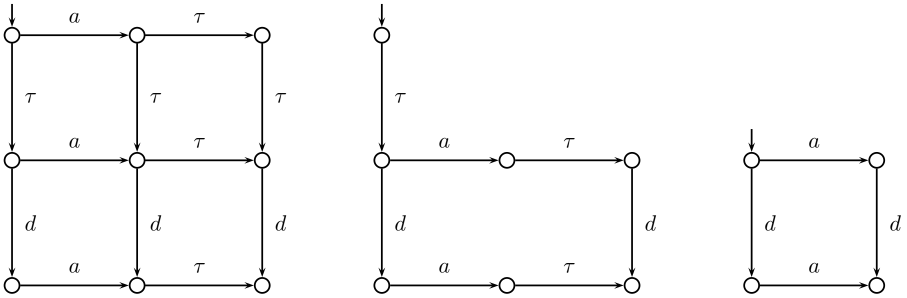


**Figure 6.7** The effect of $\tau$-prioritization and branching bisimulation compression

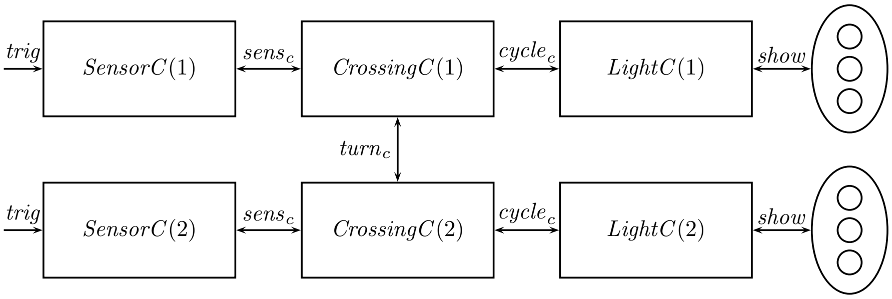


**Figure 6.8** A simple traffic light with two triggers

bothered about confluence. Each traffic light has a sensor indicating that traffic is waiting. We use a control system with 6 components (see Figure 6.8).

For each traffic light we have a sensor controller $SensorC$, a crossing controller $CrossingC$, and a traffic light controller $LightC$. The responsibility of the first is to detect whether the sensor is triggered, using the action $trig$, and forward its occurrence using the action $sens$ to *CrossingC*. The crossing controller takes care that after receiving a $sens$ message, it negotiates with the other crossing controller whether it can turn the traffic light to green (using the $turn$ action), and informs $LightC$ using the action $cycle$ to set the traffic light to green. The light controller will switch the traffic light to green, yellow and red, and subsequently informs the crossing controller that it has finished (by sending a $cycle$ message back).

Below a straightforward model of this system is provided. The sensor controller $SensorC$ gets a trigger via the action $trig$ and forwards it using $sens_s$. The traffic light controller is equally simple. After a trigger (via $cycle_r$), it cycles through the colours, and indicates through a $cycle_s$ message that it finished.

The crossing controller $\mathit{CrossingC}$ is a little more involved. It has four parameters. The first one is $\mathit{id}$ which holds an identifier for this controller (i.e. $1$ or $2$). The second parameter $\mathit{my\_turn}$ indicates whether this controller has the right to set the traffic light to green. The third parameter is $\mathit{sensor\_triggered}$ which stores whether a sensor trigger has arrived. The fourth one is $\mathit{cycle}$ indicating whether the traffic light controller is currently going through a traffic light cycle. The most critical actions are allowing the traffic light to become green ($\mathit{cycle}_s$) and giving 'my turn' to the other crossing controller ($\mathit{turn}_s$). Both can only be done if no traffic light cycle is going on and it is 'my turn'.

Note that at the init clause all components are put in parallel, and using the communication operator $\Gamma$ and allow operator $\nabla$ it is indicated how these components must communicate.

$$
\begin{array}{ll}
\textbf{proc} & \mathit{SensorC}(\mathit{id}{:}\mathbb{N}^{+}) = \underline{\mathit{trig}}(\mathit{id})\cdot \mathit{sens}_s(\mathit{id})\cdot \mathit{SensorC}(\mathit{id}); \\
\\
 & \mathit{LightC}(\mathit{id}{:}\mathbb{N}^{+}) \\
 & \quad = \ \mathit{cycle}_r(\mathit{id})\cdot \\
 & \qquad \underline{\mathit{show}}(\mathit{id}, \mathit{green})\cdot \\
 & \qquad \underline{\mathit{show}}(\mathit{id}, \mathit{yellow})\cdot \\
 & \qquad \underline{\mathit{show}}(\mathit{id}, \mathit{red})\cdot \\
 & \qquad \mathit{cycle}_s(\mathit{id})\cdot \\
 & \qquad \mathit{LightC}(\mathit{id}); \\
\\
 & \mathit{CrossingC}(\mathit{id}{:}\mathbb{N}^{+}, \mathit{my\_turn}, \mathit{sensor\_triggered}, \mathit{cycle}{:}\mathbb{B}) \\
 & \quad = \ \mathit{sens}_r(\mathit{id})\cdot \mathit{CrossingC}(\mathit{id}, \mathit{my\_turn}, \mathit{true}, \mathit{cycle}) \\
 & \quad + (\mathit{sensor\_triggered} \wedge \mathit{my\_turn} \wedge \neg \mathit{cycle}) \rightarrow \mathit{cycle}_s(\mathit{id})\cdot \\
 & \qquad\qquad\qquad \mathit{CrossingC}(\mathit{id}, \mathit{my\_turn}, \mathit{false}, \mathit{true}) \\
 & \quad + \mathit{cycle}_r(\mathit{id})\cdot \mathit{CrossingC}(\mathit{id}, \mathit{my\_turn}, \mathit{sensor\_triggered}, \mathit{false}) \\
 & \quad + \mathit{turn}_r\cdot \mathit{CrossingC}(\mathit{id}, \mathit{true}, \mathit{sensor\_triggered}, \mathit{cycle}) \\
 & \quad + (\neg \mathit{sensor\_triggered} \wedge \mathit{my\_turn} \wedge \neg \mathit{cycle}) \rightarrow \mathit{turn}_s\cdot \\
 & \qquad\qquad\qquad\qquad\qquad \mathit{CrossingC}(\mathit{id}, \mathit{false}, \mathit{sensor\_triggered}, \mathit{cycle});
\end{array}
$$

$$
\begin{array}{ll}
\textbf{init} & \nabla_{\{\underline{\mathit{trig}}, \underline{\mathit{show}}, \mathit{sens}_c, \mathit{cycle}_c, \mathit{turn}_c\}} \\
 & \quad (\Gamma_{\{\mathit{sens}_r | \mathit{sens}_s \rightarrow \mathit{sens}_c, \mathit{cycle}_r | \mathit{cycle}_s \rightarrow \mathit{cycle}_c, \mathit{turn}_r | \mathit{turn}_s \rightarrow \mathit{turn}_c\}} \\
 & \qquad (\mathit{SensorC}(1) || \mathit{SensorC}(2) || \\
 & \qquad\ \mathit{CrossingC}(1, \mathit{true}, \mathit{false}, \mathit{false}) || \mathit{CrossingC}(2, \mathit{false}, \mathit{false}, \mathit{false}) || \\
 & \qquad\ \mathit{LightC}(1) || \mathit{LightC}(2)));
\end{array}
$$

This straightforward system description has a state space of 160 states. We are interested in the behavior of the system where $\mathit{trig}$ and $\mathit{show}$ are visible, and the other actions are hidden. We can do this by applying the hiding operator $\tau_{\{\mathit{sens}_c, \mathit{cycle}_c, \mathit{turn}_c\}}$ to the process. The system is confluent with respect to the hidden $\mathit{cycle}_c$ action. The hidden $\mathit{sens}_c$ and $\mathit{turn}_c$ actions are not contributing to the confluence of the system.

In the uppermost row of Table 6.2 the sizes of the state space are given: of the full state space, after applying tau-prioritization and after applying branching bisimulation reduction.

In order to employ the effect of confluence, we must make the hidden actions $turn_c$ and $sense_c$ confluent, too. The reason that these actions are not confluent is that handing over a turn and triggering a sensor are possible in the same state, and they can take place in any order. But the exact order in which they happen causes a different traffic light go to green.

We can prevent this by making the behavior of the crossing controller $CrossingC$ deterministic. A very simple way of doing this is given below. We only provide the definition of $SensorC$ and $CrossingC$ as $LightC$ remains the same and the init line is almost identical. The idea of the specification below is that the controllers $CrossingC$ are in charge of the sensor and light controllers. When the crossing controller has the turn, it polls the sensor. And only if it has been triggered, it initiates a traffic light cycle. In both cases it gives the turn to the other crossing controller.

$$
\begin{array}{ll}
\textbf{proc} & \mathit{SensorC}(\mathit{id}{:}\mathbb{N}^+) = \mathit{sens}_r(\mathit{id})\cdot \sum_{b:\mathbb{B}} \underline{\mathit{trig}}(\mathit{id},b)\cdot \mathit{sens}_s(\mathit{id},b)\cdot \mathit{SensorC}(\mathit{id});\\
 & \\
 & \mathit{CrossingC}(\mathit{id}{:}\mathbb{N}^+, \mathit{my\_turn}{:}\mathbb{B}) =\\
 & \qquad \mathit{my\_turn}\\
 & \qquad \rightarrow \mathit{sens}_s(\mathit{id})\cdot\\
 & \qquad\quad (\mathit{sens}_r(\mathit{id},\mathit{true})\cdot\\
 & \qquad\qquad \mathit{cycle}_s(\mathit{id})\cdot\\
 & \qquad\qquad \mathit{cycle}_r(\mathit{id})\\
 & \qquad\qquad +\\
 & \qquad\qquad \mathit{sens}_r(\mathit{id},\mathit{false})\\
 & \qquad\quad )\cdot\\
 & \qquad\quad \mathit{turn}_s\cdot\\
 & \qquad\quad \mathit{CrossingC}(\mathit{id},\mathit{false})\\
 & \qquad \diamond\ \mathit{turn}_r\cdot\\
 & \qquad\quad \mathit{CrossingC}(\mathit{id},\mathit{true});
\end{array}
$$

The state space of this system turns out to be small, namely 20 states (see Table 6.2, second row). It is even smaller after applying $\tau$-prioritization, namely 8 states. Remarkably, this is also the size of the state space after branching bisimulation minimization.

As the state space is small, it is possible to inspect the state space in full (see Figure 6.9). An important property of this system is that the relative ordering in which the triggers at sensor 1 and sensor 2 are polled does not influence the ordering in which the traffic lights go to green. This sequence is only determined by the booleans that indicate whether the sensor is triggered or not. This effect is not very clear here, because the sensors are polled in strict alternation. But in the next example we see that this property also holds for more complex controllers, where the polling order is not strictly predetermined.

The previous solution can be too simple for certain purposes. We show that the deterministic specification style can still be used for more complex systems, and that the state space that is generated using $\tau$-prioritisation is still much smaller than state spaces generated without the use of confluence.

So, for the sake of the example we assume that it is desired to check the sensors while a

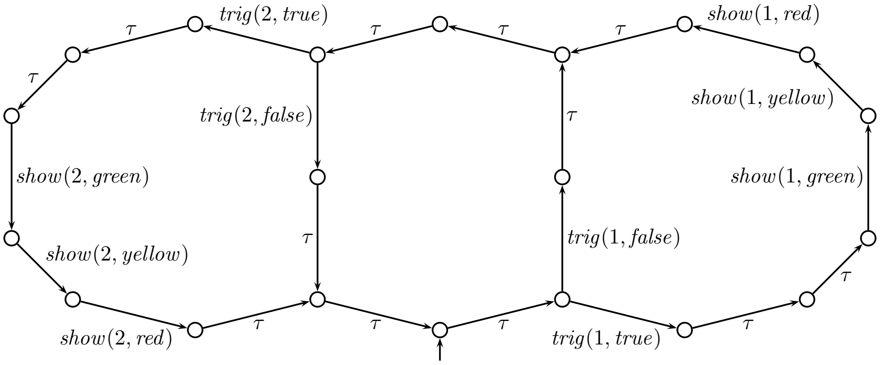


**Figure 6.9** The state space of a simple confluent traffic light controller

traffic light cycle is in progress. Both crossing controllers continuously request the sensors to find out whether they have been triggered. If none is triggered the traffic light controllers inform each other that the turn does not have to switch side. If the crossing controller whose turn it is, gets the signal that its sensor has been triggered, it awaits the end of the current traffic light cycle ($cycle_r(id)$), and simply starts a new cycle ($cycle_s(id)$). If the sensor of the crossing controller that does not have the turn is triggered, this controller indicates using $turn_s(true)$ that it wants to get the turn. It receives the turn by $turn_r$. Subsequently, it starts its own traffic light cycle.

The structure of the system is the same as in the non-confluent traffic light cycle, and therefore the init part is not provided in the specification below.

```
proc   $SensorC(id{:}\mathbb{N}^+) = sens_r(id)\cdot \sum_{b:\mathbb{B}} \underline{trig}(id, b)\cdot sens_s(id, b)\cdot SensorC(id);$

       $CrossingC(id{:}\mathbb{N}^+, my\_turn{:}\mathbb{B}) =$
              $sens_s(id)\cdot$
              $(\ sens_r(id, true)\cdot$
                $(my\_turn \rightarrow cycle_r(id) \diamond turn_s(true)\cdot turn_r)\cdot$
                $cycle_s(id)\cdot$
                $CrossingC(id, true)$
                $+$
                $sens_r(id, false)\cdot$
                $(\ my\_turn$
                  $\rightarrow (\ turn_r(true)\cdot$
                         $cycle_r(id)\cdot$
                         $turn_s\cdot$
                         $CrossingC(id, false)$
```

```
                    $+$
                    $turn_r(false)\cdot$
                    $CrossingC(id, true)$
                  $)$
               $\diamond$ $turn_s(false)\cdot$
                  $CrossingC(id, false)$
            $)$
         $);$

$LightC(id{:}\mathbb{N}^+, active{:}\mathbb{B}) =$
      $active$
      $\rightarrow$ $cycle_s(id)\cdot LightC(id, false)$
      $\diamond$ $cycle_r(id)\cdot\underline{show}(id, green)\cdot\underline{show}(id, yellow)\cdot\underline{show}(id, red)\cdot$
                                             $LightC(id,true)$;
```

This more complex traffic light controller has a substantially larger state space of 310 states. However, when the state space is generated with $\tau$-prioritisation, it has shrunk to 56 states, which is also its minimal size modulo branching bisimulation or even weak trace equivalence.

| | no<br>reduction | after<br>$\tau$-prioritisation | mod<br>branch bis |
|---|---:|---:|---:|
| Non-confluent controller | 160 | 128 | 124 |
| Simple confluent controller | 20 | 8 | 8 |
| Complex confluent controller | 310 | 56 | 56 |

**Table 6.2** The number of states of the transitions systems for a simple crossing

The complexity of the system is in the way the sensors are polled. Figure 6.10 depicts the behavior where showing the aspects of the traffic lights is hidden. As in the simple confluent controller, the relative ordering of the incoming triggers does not matter for the state the system ends up in. E.g., executing sequences $trig(2, false)\ trig(1, true)$ and $trig(1, true)\ trig(2, false)$ from the initial state lead to the lowest state in the diagram. This holds in general. Any allowed reordering of the triggers from sensor 1 and 2 with respect to each other will bring one to the same state.

## 6.7 Guideline V: Restrict the use of data

The use of data in behavioral models can quickly blow up a state space. Therefore, data should always be looked at with extra care, and if its use can be avoided, this should be done. If data is essential (and it almost always is), then there are several methods to reduce its footprint. Below we give three examples, one where data is categorized, one where the content of queues is reduced and one where buffers are ordered.

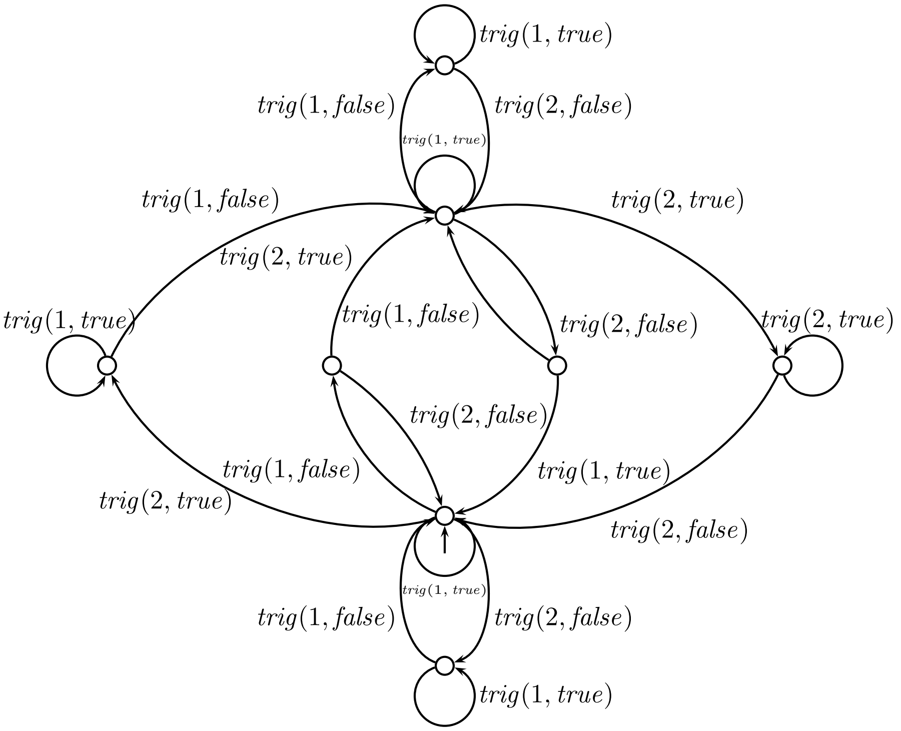


**Figure 6.10** The sensor polling pattern of a more complex confluent controller

In order to reduce the state space of a behavioral model, it sometimes helps to categorize the data in categories, and formulate the model in terms of these categories, instead of individual values. From the perspective of verification, this technique is called abstract interpretation [29]. Using this technique, a given data domain is interpreted in categories, in order to assist the verification process. Here, we advice that the modeler uses the categories in the model, instead of letting the values be interpreted in categories during the verification process. As the modeler generally knows his model best, he also has a good intuition about the appropriate categories.

Consider for example an intelligent approach controller which measures the distance of an approaching car as depicted in Figure 6.11. If the car is expected to pass distance 0 before the next measurement, a trigger signal is forwarded. The farthest distance the approach controller can observe is $M$. A quite straightforward description of this system is given below. Using the action $dist$ the distance to a car is measured, and the action $trig$ models the trigger signal.

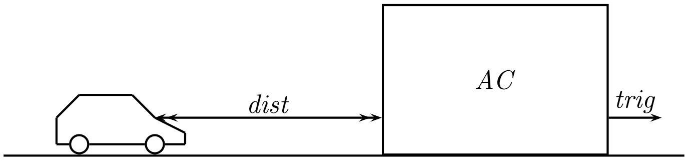


**Figure 6.11** An advanced approach controller

**map** $M : \mathbb{N};$
**eqn** $M = 100;$
**proc** $AC(d_{prev}{:}\mathbb{N}) = \sum_{d:\mathbb{N}}(d{<}M){\rightarrow}(\underline{dist}(d){\cdot}(2d{<}d_{prev}){\rightarrow}\underline{trig}{\cdot}AC(M){\diamond}AC(d));$
**init** $AC(M);$

The state space of this system is a staggering $M^2{+}1$ states big, or more concretely 10001 states. This is of course due to the fact that the values of $d$ and $d_{prev}$ must be stored in the state space to enable the evaluation of the condition $2d{<}d_{prev}$. But only the information needs to be recalled whether this condition holds, instead of both values of $d$ and $d_{prev}$. So, a first improvement is to move the condition backward as is done below, leading to a required $M{+}1$ states, or 101 in this concrete case.

**proc** $AC_1(d_{prev}{:}\mathbb{N}) = \sum_{d:\mathbb{N}}(d{<}M){\rightarrow}((2d{<}d_{prev}){\rightarrow}\underline{dist}(d){\cdot}\underline{trig}{\cdot}AC_1(M)$
$\qquad\qquad\qquad\qquad{\diamond}\underline{dist}(d){\cdot}AC_1(d));$
**init** $AC_1(M);$

But we can go much further, provided it is possible to abstract from the concrete distances. Let us assume that the only relevant information that we obtain from the individual distances is whether the car is far from the sensor or nearby. Note that we abstract from the concrete speed of the car which was used above. The specification of this abstract approach controller $AAC$ is given by:

**sort** $\mathit{Distance} = \mathbf{struct}\ \mathit{near} \mid \mathit{far};$
**proc** $AAC = \sum_{d:\mathit{Distance}} \underline{dist}(d){\cdot}((d{\approx}\mathit{near}){\rightarrow}\underline{trig}{\cdot}AAC{\diamond}AAC);$
**init** $AAC;$

Note that $M$ does not occur anymore in this specification. The state space is now reduced to two states.

We now provide an example showing how to reduce the usage of buffers and queues. Polling and $\tau$-confluence are used, to achieve the reduction. We model a system with autonomous traffic light controllers. Each controller has one sensor and controls one traffic light that can be red or green. If a sensor is triggered, the traffic light must show green. At most one traffic light can show green at any time. The controllers are organised in a ring, where each controller can send messages to its right neighbour, and receive messages from its left neighbour. For reasons of efficiency we desire that there are unbounded queues

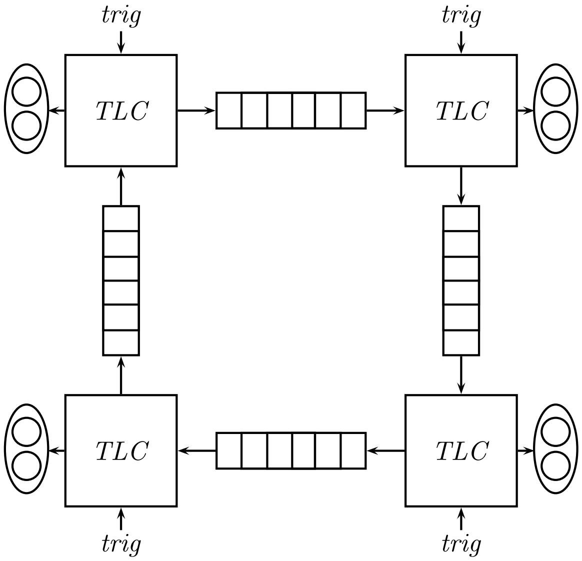


**Figure 6.12** Process communication via unbounded queues

between the controllers, such that no controller is ever hampered in forwarding messages to its neighbour. The situation is depicted in Figure 6.12.

We make a straightforward protocol, where we do not look into efficiency. Whenever a traffic light controller receives a trigger, it wants to know from the other controllers that they are not showing green. For this reason it sends its sequence number with an '*active*' tag around. If it makes a full round without altering the '*active*' tag, it switches its own traffic light to green. Otherwise, if the tag is switched to '*passive*', it retries sending the message around. A formal description is given by the following specification. The process $Queue(id, q)$ describes an infinite queue between the processes with identifiers $id$ and $id{+}1$ (modulo the number of processes). The parameter $q$ contains the content of the queue. The process $TLC(id, triggered, started)$ is the process with id $id$ where $triggered$ indicates that it has been triggered to show green, and $started$ indicates that it has started with the protocol sketched above. In the initialisation we describe the situation where there are two processes and two queues, but the protocol is suited for any number of processes and an equal number of queues.

| $N$ | Non confluent | After branching bis | Confluent | With $\tau$-prioritisation | After branching bis |
|---|---|---|---|---|---|
| 2 | 116 | 58 | 10 | 6 | 6 |
| 3 | $3.2\ 10^3$ | 434 | 15 | 9 | 9 |
| 4 | $122\ 10^3$ | $3\ 10^3$ | 20 | 12 | 12 |
| 5 | $5.9\ 10^6$ | $21\ 10^3$ | 25 | 15 | 15 |
| 6 | $357\ 10^6$ | - | 30 | 18 | 18 |
| 20 | - | - | 100 | 60 | 60 |

**Table 6.3** Traffic lights connected with queues

```
sort  $Aspect = \mathbf{struct}\ green \mid red;$
      $Message = \mathbf{struct}\ active(get\_number : \mathbb{N})?is\_active \mid$
                                                  $passive(get\_number : \mathbb{N});$
map   $N : \mathbb{N}^+;$
eqn   $N = 2;$
proc  $Queue(id{:}\mathbb{N}, q{:}List(Message)) =$
        $\sum_{m:Message} q_{in_r}(id, m)\cdot Queue(id, m \triangleright q)+$
        $(\#q>0) \rightarrow q_{out_s}((id+1) \bmod N, rhead(q))\cdot Queue(id, rtail(q));$

      $TLC(id{:}\mathbb{N}, triggered, started{:}\mathbb{B}) =$
        $\underline{trig}(id)\cdot TLC(id, true, started)+$
        $(triggered \land \neg started)$
           $\rightarrow q_{in_s}(id, active(id))\cdot TLC(id, false, true)+$
        $\sum_{m:Message} q_{out_r}(id, m)\cdot$
             $((started \land is\_active(m) \land get\_number(m) \not\approx id)$
                $\rightarrow q_{in_s}(id, passive(get\_number(m)))\cdot$
                                                  $TLC(id, triggered, started)$
                $\diamond((started \land get\_number(m) \approx id)$
                   $\rightarrow (is\_active(m) \rightarrow \underline{show}(id, green)\cdot \underline{show}(id, red)\cdot$
                                                  $TLC(id, triggered, false)$
                                  $\diamond TLC(id, true, false)$
                   $)$
                   $\diamond q_{in_s}(id, m)\cdot TLC(id, triggered, started)$
             $));$

init  $\tau_{\{q_{in_c}, q_{out_c}\}}(\nabla_{\{\underline{trig}, \underline{show}, q_{in_c}, q_{out_c}\}}(\Gamma_{\{q_{in_r}|q_{in_s} \rightarrow q_{in_c}, q_{out_r}|q_{out_s} \rightarrow q_{out_c}\}}($
        $TLC(0, false, false) || TLC(1, false, false) || Queue(0, []) || Queue(1, []))));$
```

Note that the state space of this system is growing very dramatically with the number of processes. See the second column in Table 6.3. In the third column the state space is given after a branching bisimulation reduction, where only the actions *show* and *trig* are visible. Even the state space after branching bisimulation reduction is quite large. A dash indicates that the mCRL2 toolset failed to calculate the state space or the reduction thereof (running out of space on a 1Tbyte main memory linux machine).

We will reduce the number of states by making the system confluent. We replace data pushing by polling. The structure of the protocol becomes quite different. Each process must first obtain a mutually exclusive '*token*', then polls whether a trigger has arrived, and if so, switches the traffic light to green. Subsequently, it hands the token over to the next process. The specification is given below for two processes. The specification of the queue is omitted, as it is exactly the same as the one of the previous specification.

**sort** $Aspect = \mathbf{struct}\ green \mid red;$
$Message = \mathbf{struct}\ token;$
**map** $N : \mathbb{N}^+;$
**eqn** $N = 2;$

**proc** $TLC(id{:}\mathbb{N}, active{:}\mathbb{B}) =$
$\quad active \rightarrow (\underline{trig}(id, true) \cdot \underline{show}(id, green) \cdot \underline{show}(id, red) + \underline{trig}(id, false)) \cdot$
$\qquad\qquad q_{in_s}(id, token) \cdot TLC(id, false)$
$\qquad \diamond\, q_{out_r}(id, token) \cdot TLC(id, true);$

**init** $\tau_{\{q_{in_c}, q_{out_c}\}}(\nabla_{\{\underline{trig}, \underline{show}, q_{in_c}, q_{out_c}\}}(\Gamma_{\{q_{in_r} | q_{in_s} \rightarrow q_{in_c}, q_{out_r} | q_{out_s} \rightarrow q_{out_c}\}}($
$\quad TLC(0, true) || TLC(1, false) || Queue(0, []) || Queue(1, []))));$

The number of states of the state space for different number of processes are given in the fourth column of Table 6.3. In the fifth and sixth columns the number of states after $\tau$-prioritisation and branching bisimulation reduction are given. Note that the number of states after $\tau$-prioritisation is equal to the number of states after application of branching bisimulation. Note also that the differences in the sizes of the state spaces is quite striking.

As a last example we show the effect of ordering buffers. With queues and buffers different contents can represent the same data. If a buffer is used as a set, the ordering in which the elements are put into the buffer is irrelevant. In such cases it helps to maintain an order on the data structure. As an example we provide a simple process that reads arbitrary natural numbers smaller than $N$ and puts them in a set. The process doing so is given below.

**map** $N : \mathbb{N};$
$insert, ordered\_insert : \mathbb{N} \times List(\mathbb{N}) \rightarrow List(\mathbb{N});$

**var** $n, n' : \mathbb{N}; b : List(\mathbb{N});$
**eqn** $insert(n, b) = if(n \in b, b, n \triangleright b);$
$ordered\_insert(n, []) = [n];$
$ordered\_insert(n, n' \triangleright b) = if(n < n', n \triangleright n' \triangleright b,$
$\qquad\qquad if(n \approx n', n' \triangleright b, n' \triangleright ordered\_insert(n, b)));$
$N = 10;$
**proc** $B(buffer{:}List(\mathbb{N})) = \sum_{n:\mathbb{N}} (n < N) \rightarrow \underline{read}(n) \cdot B(insert(n, buffer));$

**init** $B([]);$

If the function $insert$ is used, the elements are put into a set in an arbitrary order (more precisely, the elements are prepended). If the function $ordered\_insert$ is used instead of

| $N$ | non ordered | ordered |
|---|---|---|
| 1 | 2 | 2 |
| 2 | 5 | 4 |
| 3 | 16 | 8 |
| 4 | 65 | 16 |
| 5 | 326 | 32 |
| 6 | $2.0\ 10^3$ | 64 |
| 7 | $14\ 10^3$ | 128 |
| 8 | $110\ 10^3$ | 256 |
| 9 | $986\ 10^3$ | 512 |
| 10 | $9.9\ 10^6$ | $1.02\ 10^3$ |
| 11 | $109\ 10^6$ | $2.05\ 10^3$ |
| 12 | $1.30\ 10^9$ | $4.10\ 10^3$ |

**Table 6.4** Number of states of an non ordered/ordered buffer with max. $N$ elements

$insert$, the elements occur in ascending order in the buffer. In Table 6.4 the effect of ordering is shown. Although the state spaces with ordering also grow exponentially, the beneficial effect of ordering does not need further discussion.

## 6.8 Guideline VI: Compositional design and reduction

When a system that must be designed consists of several components, it can be wise to organize these components in such a way that stepwise composition and reduction are possible. The idea is depicted in Figure 6.13. At the left hand side of Figure 6.13 a set of communicating components $C_1, \ldots, C_5$ is depicted. In the middle, the interfaces $I_1, \ldots, I_7$ are also shown. At the right the system has a tree structure.

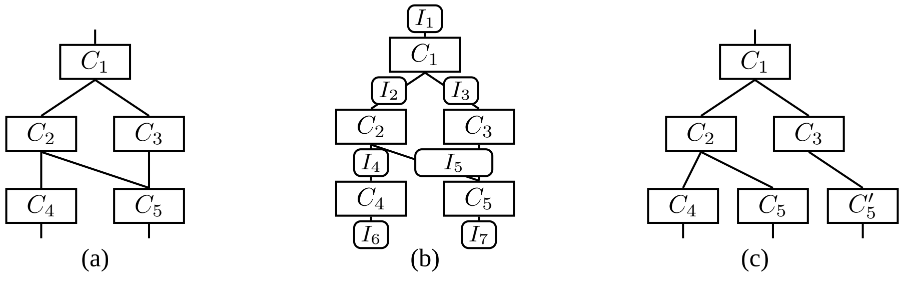


**Figure 6.13** The compositional design and verification steps

When calculating the behavior of the whole system, a characterization of the simultaneous behavior at the interfaces $I_1$, $I_6$ and $I_7$ is required where all communication at the other interfaces is hidden. Unfortunately, calculating the whole behavior before hiding internal communication may not work, because the whole behavior has too many states. An alternative is to combine and hide in an alternating fashion. After each hiding step a behavioral reduction is applied, which results in a reduced transition system.

For instance, the interface behavior at $I_2$, $I_5$ and $I_6$ can be calculated from the behavior of $C_2$ and $C_4$ by hiding the behavior at $I_4$. Subsequently, $C_3$ and $C_5$ can be added, after which the communication at $I_5$ can be hidden. At last adding $C_1$ and hiding the actions at the interfaces $I_2$ and $I_3$ finishes the calculation of the behavior. This alternation of composing behavior and hiding actions is quite commonly known and some toolsets even developed a script language to allow for an optimal stepwise composition of the whole state space [36].

In order to optimally employ this stepwise sequence of composition, hiding and reduction, it is desired that as much communication as possible can be hidden to allow for a maximal reduction of behavior. But there is something even more important. If a subset of components has more interfaces that will be closed off by adding more components later, it is very likely that there is some relationship between the interactions at these interfaces. As long as the set of components has not been closed, the interactions at these interfaces are unrelated, often leading to a severe growth in the state space of the behavior of this set of sub-components. When closing the dependent interfaces, the state space is brought to its expected size. If such dependent but unrestricted interfaces occur, the use of stepwise composition and reduction is generally ineffective.

As an example consider Figure 6.13 again. If $C_2$, $C_3$, $C_4$ and $C_5$ have been composed, the system has interactions at interfaces $I_2$ and $I_3$ that can happen independently. Adding $C_1$ restricts the behavior at these interfaces. For instance, $C_1$ can strictly alternate between sending data via $I_2$ and $I_3$, but without $C_1$ any conceivable order must be present in the behavior of $C_2$, $C_3$, $C_4$ and $C_5$.

Dependent but unrestricted interfaces can be avoided by using a tree topology. See Figure 6.13 (c) where the dependency at interfaces $I_2$ and $I_3$ has been removed by duplicating component $C_5$. If a tree topology is not possible, then it is advisable to restrict behavior at dependent but unrestricted interfaces as much as possible from inside sets of components.

As an example we provide yet another distributed traffic controller (see Figure 6.14). There are a certain number $N$ of traffic lights. At the central $TopController$ component requests arrive using a $set(m)$ action to switch traffic light $m$ to green. This request is forwarded via intermediate components (called $Controller$s) to traffic light controllers ($TLC$s). If a traffic light has been set to green and subsequently to red again, an action $ready(n)$ indicates that the task has been accomplished. The system must guarantee that one traffic light can be green at any time but the order in which this happens is not prescribed.

We start presenting a solution that does not have a tree topology. Using the principle of

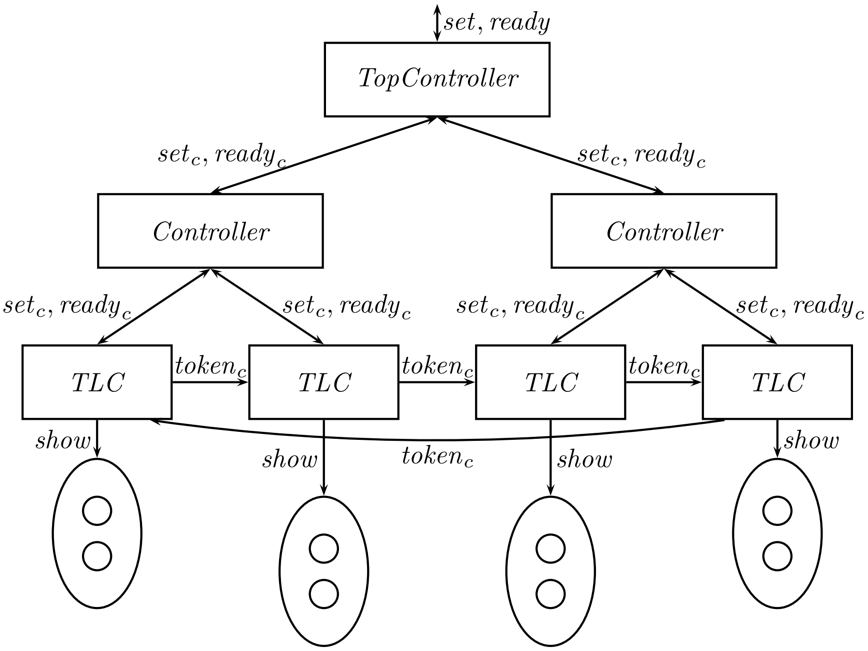


**Figure 6.14** Distribution of system components

separation of concerns, we let the traffic light controllers be responsible for taking care that no two traffic lights are showing green at the same time. The top- and other controllers have as task to inform the traffic light controllers that they must set the light to green, and they transport the ready messages back to the central controller.

The traffic light controllers use a simple protocol as described in the queue example in section 8. They continuously exchange a token. The owner of the token is allowed to set the traffic light to green. The parameter $id$ is the identifier of the traffic light. The parameter $level$ indicates the level of the traffic light controllers. The top controller has level 0. In Figure 6.14 the level of the traffic light controllers is 2. Furthermore, $has\_token$ indicates that this traffic light controller owns the token, and $busy$ indicates that it must let the traffic light go through a green-red cycle.

The controllers and the top controller are more straightforward. They pass set commands from top to bottom, and send ready signals from bottom to top. The parameters $id_{low}$ and $id_{high}$ indicate the range of traffic lights over which this controller has control. The description below describes a system with four traffic light controllers.

**sort** $Aspect = \mathbf{struct}\ green \mid red;$

| | bottom control | | bottom + top control | | top control | |
|---|---|---|---|---|---|---|
| | 4 nodes | 8 nodes | 4 nodes | 8 nodes | 4 nodes | 8 nodes |
| Total system | $10.0\ 10^3$ | $236\ 10^6$ | $1.09\ 10^3$ | $96.3\ 10^3$ | 368 | $15.6\ 10^3$ |
| Mod branch. bis. | $3.84\ 10^3$ | $39.8\ 10^6$ | 236 | $7.42\ 10^3$ | 236 | $7.42\ 10^3$ |
| Without top controller | $1.80\ 10^3$ | $25.3\ 10^6$ | $1.80\ 10^3$ | $25.3\ 10^6$ | - | - |
| Mod branch. bis. | 983 | $5.9\ 10^6$ | 983 | $5.9\ 10^6$ | - | - |
| Half system | 131 | $93.9\ 10^3$ | 131 | $93.9\ 10^3$ | 56 | $16.8\ 10^3$ |
| Mod branch. bis. | 107 | $44.1\ 10^3$ | 107 | $44.1\ 10^3$ | 33 | $3.06\ 10^3$ |

**Table 6.5** State space sizes for a hierarchical traffic light controller

**proc**

$$\begin{array}{l}
\mathit{ControllerTop}(\mathit{id}_{low}, \mathit{id}_{high}{:}\mathbb{N}) = \\
\quad \sum_{n:\mathbb{N}}(\mathit{id}_{low}{\leq}n \wedge n{\leq}\mathit{id}_{high}){\rightarrow}(\underline{\mathit{set}}(n){\cdot}\mathit{set}_s(n,1){+}\mathit{ready}_r(n,1){\cdot}\underline{\mathit{ready}}(n)){\cdot} \\
\qquad\qquad\qquad \mathit{ControllerTop}(\mathit{id}_{low}, \mathit{id}_{high});
\end{array}$$

$$\begin{array}{l}
\mathit{Controller}(\mathit{id}_{low}, \mathit{id}_{high}, \mathit{level}{:}\mathbb{N}) = \\
\quad \sum_{n:\mathbb{N}}(\mathit{id}_{low}{\leq}n \wedge n{\leq}\mathit{id}_{high}){\rightarrow} \\
\qquad (\mathit{set}_r(n, \mathit{level}){\cdot}\mathit{set}_s(n, \mathit{level}{+}1){\cdot}\mathit{Controller}(\mathit{id}_{low}, \mathit{id}_{high}, \mathit{level}){+} \\
\qquad\; \mathit{ready}_r(n, \mathit{level}{+}1){\cdot}\mathit{ready}_s(n, \mathit{level})){\cdot}\mathit{Controller}(\mathit{id}_{low}, \mathit{id}_{high}, \mathit{level});
\end{array}$$

$$\begin{array}{l}
\mathit{TLC}(\mathit{id}, \mathit{level}{:}\mathbb{N}, \mathit{has\_token}, \mathit{busy}{:}\mathbb{B}) = \\
\quad \mathit{set}_r(\mathit{id}, \mathit{level}){\cdot}\mathit{TLC}(\mathit{id}, \mathit{level}, \mathit{has\_token}, \mathit{true}){+} \\
\quad (\mathit{has\_token}{\wedge}\mathit{busy}){\rightarrow}\underline{\mathit{show}}(\mathit{id}, \mathit{green}){\cdot}\underline{\mathit{show}}(\mathit{id}, \mathit{red}){\cdot}\mathit{ready}_s(\mathit{id}, \mathit{level}){\cdot} \\
\qquad\qquad\qquad \mathit{TLC}(\mathit{id}, \mathit{level}, \mathit{has\_token}, \mathit{false}){+} \\
\quad (\mathit{has\_token}{\wedge}\neg\mathit{busy}){\rightarrow}\mathit{token}_s((\mathit{id}{+}1) \bmod 4){\cdot}\mathit{TLC}(\mathit{id}, \mathit{level}, \mathit{false}, \mathit{busy}){+} \\
\quad (\neg\mathit{has\_token}){\rightarrow}\mathit{token}_r(\mathit{id}){\cdot}\mathit{TLC}(\mathit{id}, \mathit{level}, \mathit{true}, \mathit{busy});
\end{array}$$

**init**

$$\begin{array}{l}
\nabla_{\{\mathit{set}_c, \mathit{ready}_c, \mathit{token}_c, \underline{\mathit{show}}, \underline{\mathit{set}}, \underline{\mathit{ready}}\}}( \\
\qquad\qquad \Gamma_{\{\mathit{set}_r|\mathit{set}_s\rightarrow\mathit{set}_c, \mathit{ready}_r|\mathit{ready}_s\rightarrow\mathit{ready}_c, \mathit{token}_r|\mathit{token}_s\rightarrow\mathit{token}_c\}}( \\
\qquad \mathit{ControllerTop}(0,3)||\mathit{Controller}(0,1,1)||\mathit{Controller}(2,3,1)|| \\
\qquad \mathit{TLC}(0,2,\mathit{true},\mathit{false})||\mathit{TLC}(1,2,\mathit{false},\mathit{false})|| \\
\qquad \mathit{TLC}(2,2,\mathit{false},\mathit{false})||\mathit{TLC}(3,2,\mathit{false},\mathit{false})));
\end{array}$$

In order to understand the state space of components and sets of sub-components, we look at the size of the whole state space, the size of the state space without the top controller, and the size of half the system with one controller and two TLCs. The results are listed in Table 6.5 for a system with four and eight traffic light controllers. In case of four traffic lights, a half system has two traffic lights and one controller. In case of eight traffic lights, a half system has four traffic lights and three controllers. The results of the sizes of the state spaces are given in the columns under the header ‘bottom control’. In all cases the size of the state space modulo branching bisimulation is also given. Here all internal actions are hidden and the external actions $\mathit{show}$, $\mathit{set}$ and $\mathit{ready}$ are visible.

What we note is that the sizes of the state spaces are large. In particular the size of the state space modulo branching bisimulation of the system without the top controller multiplied with the size of the top controller is almost as large as the size of the total state space. The state space of the top controller for four traffic lights has 9 states and the one for eight traffic lights has 17 states. It makes little sense to use compositional verification in this case, but the fact that the top controller hardly restricts the behavior of the rest of the system saves the day. If the top controller is more restrictive compositional verification makes no sense at all.

If we analyse the large state space of this system, we see that the independent behavior of the controllers substantially adds to the size of the state space. We can restrict this by giving more control to the top controller. Whenever it receives a request to $set$ a traffic light to green, it stores it in a set called $requests$. Whenever a traffic light is allowed to go to green, indicated by $busy$ equals false, the top controller non-deterministically selects an index of a traffic light from $requests$ and instruct it to go to green. The specification of the new top controller is given below.

$$
\begin{array}{ll}
\textbf{proc} & \mathit{ControllerTop}(id_{low}, id_{high}{:}\mathbb{N}) = \mathit{ControllerTop}(id_{low}, id_{high}, \emptyset, \mathit{false}); \\
\\
& \mathit{ControllerTop}(id_{low}, id_{high}{:}\mathbb{N}, \mathit{requests}{:}\mathit{Set}(\mathbb{N}), \mathit{busy}{:}\mathit{Bool}) = \\
& \quad \sum_{n:\mathbb{N}}(id_{low}{\leq}n \wedge n{\leq}id_{high} \wedge n{\notin}\mathit{requests}){\to} \\
& \qquad\qquad \underline{set}(n){\cdot}\mathit{ControllerTop}(id_{low}, id_{high}, \mathit{requests}{\cup}\{n\}, \mathit{busy}){+} \\
& \quad \sum_{n:\mathbb{N}}(id_{low}{\leq}n \wedge n{\leq}id_{high} \wedge n{\in}\mathit{requests} \wedge \neg\mathit{busy}){\to} \\
& \qquad\qquad set_s(n,1){\cdot}\mathit{ControllerTop}(id_{low}, id_{high}, \mathit{requests} \setminus \{n\}, \mathit{true}){+} \\
& \quad \sum_{n:\mathbb{N}}(id_{low}{\leq}n \wedge n{\leq}id_{high} \wedge n{\in}\mathit{requests}){\to} \\
& \qquad\qquad \mathit{ready}_r(n,1){\cdot}\underline{\mathit{ready}}(n){\cdot} \\
& \qquad\qquad\qquad\qquad \mathit{ControllerTop}(id_{low}, id_{high}, \mathit{requests}, \mathit{false});
\end{array}
$$

The resulting state spaces are given in Table 6.5 under the header ‘bottom and top control’. The first observation is that the sizes of the state spaces without top control and of a half system have not changed. This is self evident, as only the top controller has been replaced. It is important to note that the sizes of the state space modulo branching bisimulation of the system without top controller is almost as large as the unreduced state space of the full system for four traffic lights. For eight traffic lights the intermediate reduced state space is much larger than the unreduced system of the full state space.

We can remove the low level control via the exchange of the token. This is possible because the top controller now guarantees that at most one traffic light shows green. This is done by replacing the specification of the traffic light controller by the simple specification below. Note that the communication topology of the system now has a tree structure.

$$
\begin{array}{ll}
\textbf{proc} & \mathit{TLC}(id, \mathit{level}{:}\mathbb{N}) = \\
& \qquad set_r(id, \mathit{level}){\cdot}\underline{\mathit{show}}(id, \mathit{green}){\cdot}\underline{\mathit{show}}(id, \mathit{red}){\cdot} \\
& \qquad\qquad\qquad\qquad \mathit{ready}_s(id, \mathit{level}){\cdot}\,\mathit{TLC}(id, \mathit{level});
\end{array}
$$

We are not interested anymore in the behavior of the system with all the traffic light controllers and no top controller. We only need to look at the sizes of the half systems which can be reduced and both half systems can directly be combined with the top controller. Note that in this way we circumvent the blow-up of intermediate processes. Note also that the resulting state spaces modulo branching bisimulation for the system with ‘top control’ are the same as those for ‘bottom and top control’. This shows that the token exchange is really immaterial when the top controller guarantees that at most one traffic light goes to green. Finally, note that the half systems with bottom control are only slightly bigger than the half systems with top control. From this we can conclude that token exchange by itself does not contribute substantially to the size of the state space.

## 6.9 Guideline VII: Specify external behavior of sets of sub-components

In the previous section we mentioned that stepwise composition and reduction might be a way to avoid a blow-up of the state space. But we observed that sometimes the composed behavior of sets of components is overly complex, and contains far too many states, even after applying a behavioral reduction.

In order to keep the behavior of such sets of components small, it is useful to first design the desired external behavior of this set of components, and to subsequently design the behavior of the components such that they meet this external behavior. The situation is quite comparable to the implementation of software. If the behavior is governed by the implementation, a system is often far less understandable and usable, than when a precise specification of the software has been provided first, and the software has been designed to implement exactly the specified behavior.

The use of external behavior for various purposes was most notably defended in the realm of protocol specification [84], although keeping the state space small was not one of these purposes. The word service was commonly used in this setting for the external behavior. More recently, the ASD development method has been proposed, where a system is to be defined by first specifying the external behavior of a system, which is subsequently implemented [20]. The purpose here is primarily to allow a designer to keep control over his system.

In order to illustrate how specifications can be used to keep external behavior small, we provide a simple example, and show how a small difference in the behavior of the components has a distinctive effect on the complexity in terms of states. From the perspective of the task that the components must perform, the difference in the description looks relatively minor. The example is inspired by the third sliding window protocol in [73] which is a fine example of a set of components that provides the intended task but has a virtually incomprehensible external behavior.

Our system is depicted in Figure 6.15. The first specification has a complex external behavior whereas the external behavior of the second is straightforward. The system consists of a device-monitor and a controller that can be started ($start$) or stopped ($stop$) by an external source. The device-monitor observes the status of a number of devices and sends the defected device number to the controller via the action $broken$. The controller

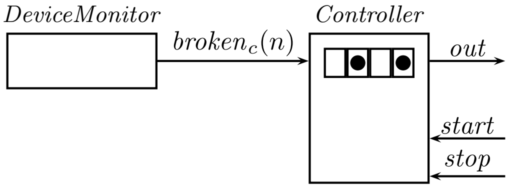


**Figure 6.15** A system comprises a controller and a device-monitor

comprises a buffer that stores the status of the devices.

The first specification can be described as follows. The device monitor is straightforward in the sense that it continuously performs actions $broken_s(n)$ for numbers $n{<}M$. The parameter *buff* represents the buffer by a function from natural numbers to booleans. If $\mathit{buff}(i)$ is true, it indicates that a fault report has been received for device $i$. The boolean parameter *bool* indicates whether the controller is switched on or off and the natural number $i$ is the current position in the buffer, which the controller uses to cycle through the buffer elements. It sends an action $out$ whenever it encounters an element that is set to $true$. The internal action $int$ takes place when the controller moves to investigate the next buffer place.

$$\begin{array}{ll}
\textbf{map} & M{:}\mathbb{N}^{+};\\
\textbf{eqn} & M{=}2;\\
\textbf{map} & \mathit{buff}_0{:}\mathbb{N}{\to}\mathbb{B};\\
\textbf{eqn} & \mathit{buff}_0 = \lambda n{:}\mathbb{N}.\mathit{false};\\
\textbf{proc} & \mathit{DeviceMonitor} = \sum_{n:\mathbb{N}}(n{<}M){\to}broken_s(n).\mathit{DeviceMonitor};\\
 & \mathit{Controller}(\mathit{buff}{:}\mathbb{N}{\to}\mathbb{B}, \mathit{bool}{:}\mathbb{B}, i{:}\mathbb{N})\\
 & \quad = \ \sum_{n:\mathbb{N}} broken_r(n){\cdot}\mathit{Controller}(\mathit{buff}[n{\to}\mathit{true}], \mathit{bool}, i)\\
 & \quad + \ (\neg\mathit{buff}(i){\wedge}\mathit{bool}){\to}\underline{stop}{\cdot}\mathit{Controller}(\mathit{buff}, \mathit{false}, i)\\
 & \quad + \ (\neg\mathit{bool}){\to}\underline{start}{\cdot}\mathit{Controller}(\mathit{buff}, \mathit{true}, i)\\
 & \quad + \ (\mathit{buff}(i){\wedge}\mathit{bool}){\to}\underline{out}{\cdot}\mathit{Controller}(\mathit{buff}[i{\to}\mathit{false}], \mathit{bool}, (i{+}1) \bmod M)\\
 & \quad + \ (\neg\mathit{buff}(i){\wedge}\mathit{bool}){\to}int{\cdot}\mathit{Controller}(\mathit{buff}, \mathit{bool}, (i{+}1) \bmod M)\\
\textbf{init} & \tau_{\{broken_c, int\}}(\nabla_{\{broken_c, \underline{out}, \underline{start}, \underline{stop}, int\}}(\Gamma_{\{broken_r | broken_s \to broken_c\}}(\\
 & \qquad\qquad\qquad \mathit{Controller}(\mathit{buff}_0, \mathit{false}, 0) || \mathit{DeviceMonitor})));
\end{array}$$

The total number of devices is denoted by $M$. All positions of *buff* are initially set to $false$ as indicated by the lambda expression $\lambda n{:}\mathbb{N}.\mathit{false}$. In this specification the controller blocks the $stop$ request if there is a defected device at index $i$ of the buffer forming a dependency between external and internal behavior. If we calculate the state space of the external behavior of this system with $M = 2$ and apply a branching bisimulation reduction [82], we obtain the state space depicted in Figure 6.16 at the left. Note that the behavior is remarkably complex. In particular a number of $\tau$-transitions complicate the transition system. But they cannot be removed as they are essential for the perceived external behavior of the system. Table 6.6 provides the number of states produced as a function of the number of devices monitored in the system. The table shows that the state

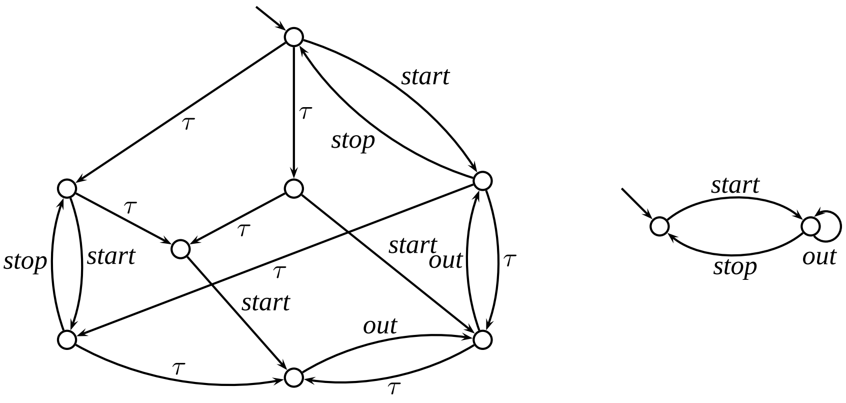


**Figure 6.16** The system external behavior

space of the original system and the state space capturing the external behavior are comparable. This indicates a complex external behavior that might complicate verification with external parties and makes understanding the behavior quite difficult. It might be

| M | No. of original states | | No. of external states | |
|---|---|---|---|---|
| | 1st spec | 2nd spec | 1st spec | 2nd spec |
| 1 | 4 | 4 | 2 | 2 |
| 2 | 16 | 16 | 8 | 2 |
| 3 | 48 | 48 | 16 | 2 |
| 4 | 128 | 128 | 32 | 2 |
| 5 | 320 | 320 | 64 | 2 |
| 6 | 768 | 768 | 128 | 2 |
| 10 | $20.5\ 10^3$ | $20.5\ 10^3$ | $2.48\ 10^3$ | 2 |

**Table 6.6** Sizes of the original and external state space of the monitor controllers

amazing that the external state space of the system is large. Actual expectation is that it should be small, matching the specification below, depicted in the transition system in Figure 6.16 at the right.

**proc** $\mathit{Stopped} = \underline{\mathit{start}}\cdot\mathit{Started};$
$\mathit{Started} = \underline{\mathit{out}}\cdot\mathit{Started} + \underline{\mathit{stop}}\cdot\mathit{Stopped};$
**init** $\mathit{Stopped};$

Investigation of the cause of the difference between the actual and the expected sizes of the transition systems leads to the conclusion that blocking the $\mathit{stop}$ action when $\mathit{buff}(i)$

is true is the cause of the problem. If we remove this from the condition of the stop action, we obtain the mCRL2 specification of the *DeviceMonitor* process below. In this specification the $stop$ request is processed independently from the rest of the behavior.

$$
\begin{array}{ll}
\textbf{proc} & \mathit{DeviceMonitor} = \sum_{n:\mathbb{N}}(n<M)\rightarrow \mathit{broken}_s(n).\mathit{DeviceMonitor};\\
 & \mathit{Controller}(\mathit{buff}{:}\mathbb{N}\rightarrow\mathbb{B}, \mathit{bool}{:}\mathbb{B}, i{:}\mathbb{N})\\
 & \quad = \ \sum_{n:\mathbb{N}} \mathit{broken}_r(n)\cdot \mathit{Controller}(\mathit{buff}[n\rightarrow \mathit{true}], \mathit{bool}, i)\\
 & \quad + \ \mathit{bool}\rightarrow \underline{\mathit{stop}}\cdot \mathit{Controller}(\mathit{buff}, \mathit{false}, i)\\
 & \quad + \ (\neg \mathit{bool})\rightarrow \underline{\mathit{start}}\cdot \mathit{Controller}(\mathit{buff}, \mathit{true}, i)\\
 & \quad + \ (\mathit{buff}(i)\wedge \mathit{bool})\rightarrow \underline{\mathit{out}}\cdot \mathit{Controller}(\mathit{buff}[i\rightarrow \mathit{false}], \mathit{bool}, (i{+}1) \bmod M)\\
 & \quad + \ (\neg \mathit{buff}(i)\wedge \mathit{bool})\rightarrow \mathit{int}\cdot \mathit{Controller}(\mathit{buff}, \mathit{bool}, (i{+}1) \bmod M)
\end{array}
$$

As can be seen from Table 6.6, the number of states of the non-reduced model remains the same. However, the reduced behavior is exactly the one depicted in Figure 6.16 at the right for any constant $M$.

## 6.10 Conclusion

We have shown that different specification styles can substantially influence the number of states of a system. We believe that an essential skill of a behavioral modellist is to make models such that the insight that is required can be obtained. If a system is to be designed such that it provably satisfies a number of behavioral requirements, then the behavior must be sufficiently small to be verified. If an existing system is modeled to obtain insight in its behavior, then on the one hand the model should reflect the existing system sufficiently well, but on the other hand the model of the system should be sufficiently simple to allow to answer relevant questions about the behavior of the system.

As far as we can see hardly any attention has been paid to the question how to make behavioral models such that they can be analyzed. All attention appears to be directed to the question of how to analyse given models better. But it is noteworthy that it is very common in other modeling disciplines to let models be simpler than reality. For instance in electrical engineering models are as much as possible reduced to sets of linear differential equations. In queueing theory, only a few queueing models can be studied analytically, and therefore, it is necessary to reduce systems to these standard models if analytical results are to be obtained.

We provided seven guidelines, based on our experience with building models of various systems. There is no claim that this set is complete, or even that these seven guidelines are the most important model reduction techniques. What we hope is that this chapter will induce research such that more reduction techniques will be uncovered, described, classified and subsequently become a standard ingredient in teaching behavioral modeling.

# Chapter 7

# Applying the Guidelines to the PDU Controller

Investigating the effects of designing industrial control software using push and poll strategies. CS-Report 11-16, Eindhoven University of Technology, 2011.

## 7.1 Introduction

In the previous chapter we proposed a number of guidelines to tackle the state space explosion problem, namely by designing software components such that they can easily be verified. In this chapter we apply a number of these guidelines to the design and the formal verification of a real industrial system, namely the controller of the power distribution unit (PDU), we introduced earlier in Chapter 4. Through this work we propose a number of alternative designs to achieve the required functionality of the controller. We compare designs that uses the guidelines from those that do not. As a result, we found that the designs that do not use the guidelines have substantially more states and may easily hit the limit of state space explosion in case of potential future extensions.

Following the ASD approach, we start by describing a single desired external behavior of the controller. Then, we provide two main designs, where the first uses a pushing strategy and the second uses a polling strategy. As explained in previous chapter, by pushing we mean that components of a system share their information with others when the information is available, while polling means that components poll (or ask) information from others only when it is needed. As will be demonstrated shortly, other guidelines such as the restricted use of data and the use of global synchronous communication have been applied further and substantially helped reducing the state space. All design alternatives refine the external behavior of the controller so that they all provide the intended external behavior of the system.

Throughout this chapter we use mCRL2 for formal specification and state space generation. Additionally, we use the refinement concept to prove formal refinement of designs against the external behavior. For this we use mCRL2, CADP [2] and CSP/FDR2.

The results of this work confirm that different design styles can influence and reduce the number of the generated states of the modeled systems and that the guidelines are effective in practical applications.

This chapter is organized as follows. Section 7.2 gives an overview of the context of the PDU controller. The strategies and tactics used to accomplish the tasks of modeling and verifying the controller are described in Section 7.3. The external behavior of the controller is detailed in Section 7.4. The designs of the controller using the pushing strategy are demonstrated in Section 7.5, while the designs implementing the poll strategy are described in Section 7.6. In Section 7.7 we give some statistical data, comparing the push and poll variants and the tools used throughout this work.

## 7.2 The PDU controller

We start by extending the state machines of both the PCs and the PDU depicted in Figure 3.3 and Figure 3.11 with the transiting states.

**PCs and devices** The state machines of the PCs are extended with transiting state to model the start-up progress of the PC, see the state machines in Figure 7.1.

Initially, a PC is in the *Off* state. When it is supplied with power, it transits to the *StartingUp* state where the Operating System (OS) boots up and then the clinical applications are started. After the OS and the applications are up-and-running, the PC transits to the *Operational* state.

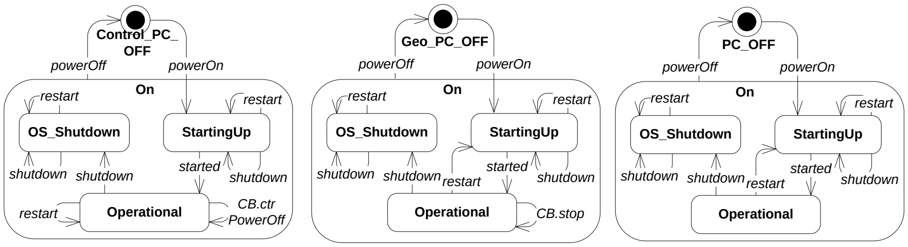


**Figure 7.1** The external behavior of all PCs

The clinical applications of a PC are restarted upon receiving a *restart* message from the PDU in *Operational* state. Additionally, when a *shutdown* message is received from the PDU, the PC stops all running applications and shuts down the OS.

**Behavior of the Power Distribution Controller** The PDU controller implements the extended state machine of Figure 7.2. The state machine distinguishes the following six stable states as described in Table 7.1, and the two transiting stated as described in Table 7.2.

The state machine includes eight distinct events in total. The events *PDUswitchOn* and *PDUswitchOff* indicate switching the mains disconnector switch on and off, respectively. The *powerOff* event indicates that the user presses the *PowerOff* button for less than 3 seconds, while the *forcedPowerOff* event denotes that the user presses the same button more than 10 seconds. Both *powerOn* and *emergencyOff* represent pressing the *PowerOn* and *EmergencyOff* buttons, respectively. *ControlPowerOff* and *stop* events indicate receiving callback signals from both the ControlPC and the GeoPC, where the first requests the PDU to power off the complete system and the second demands the PDU to immediately cut down the power to the movable segments.

Table 7.3 summarizes the required tasks for each transition of the state machine. For example, when the system is in the *System_Off* state and the user presses the *PowerOn* button, all permanent and switchable taps are switched on, and therefore all PCs and devices start-up. Eventually, all PCs and devices are started-up and the system can potentially move to the *System_On* state.

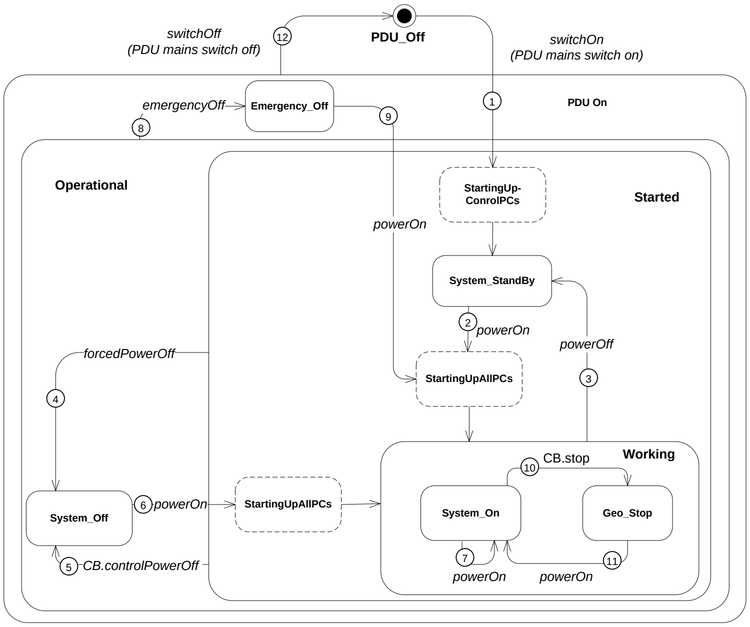


**Figure 7.2** The high-level behavior of the PDU [60]

| State | Property |
|---|---|
| PDU_Off | The mains disconnector switch is open which means that the PDU is powerless. All PCs and devices are off. |
| System_Standby | The permanent power taps are powered. All switchable taps are powerless. ControlPC is on. |
| System_On | All permanent and switchable taps are powered. All PCs and devices are on. |
| GEO_Stop | Similar to the *System_On* state, only the movable parts are powerless. All PCs and devices are on. Motorized movements are disabled. |
| System_Off | All permanent and switchable taps are powerless. All PCs and devices are off. |
| Emergency_Off | All permanent and switchable taps are powerless. All PCs and devices are off. |

**Table 7.1** The stable states of the PDU state machine [60]

| State | Property |
|---|---|
| StartingUpControlPC | The permanent power taps are powered. The ControlPC is starting up. |
| StartingUpAllPCs | All permanent and switchable taps are powered. Not all PCs or devices are fully operational. |

**Table 7.2** The transiting states of the PDU state machine [60]

| Transition | Activity |
|---|---|
| 1 | Boot PDU; the PDU switches on all permanent power taps; the ControlPC is starting up. |
| 2 | The PDU switches on all switchable taps, one by one to avoid a big inrush current; all devices are starting up. |
| 3 | The PDU broadcasts a "shutdown" message to shutdown all control devices except the ControlPC; the PDU switches off all switchable taps when power load is below a threshold or when the timer expires. |
| 4 | The PDU immediately switches off all power taps. |
| 5 | The PDU broadcasts a "shutdown" message to shutdown all control devices including the ControlPC; the PDU switches off all taps when power load is below threshold or when the timer expires. |
| 6 | The PDU switches on all taps, one by one to avoid a big inrush current; all devices are starting up. |
| 7 | The PDU broadcasts a "restart" message; the applications of all control devices are restarted. |
| 8 | Disconnect the PDU internal power bus. |
| 9 | The PDU switches on all taps, one by one to avoid a big inrush current; all devices are starting up. |
| 10 | The PDU switches off the power taps that supply motor drives of movable parts. |
| 11 | The PDU switch on the power taps that supply motor drives of movable parts. |
| 12 | The PDU is switched off; all taps are switched off. |

**Table 7.3** The activities required for each transition of the PDU state machine [60]

In the *System_On* state, if the user again presses the *PowerOn* button, the PDU broadcasts a *restart* message over the Ethernet network. Consequently, the PCs and devices shall restart their applications. But, if the user presses the *PowerOff* button for less than 3 seconds, the PDU broadcasts a *shutdown* message over the Ethernet network. Upon receiving the message by the PCs, they gradually shutdown their applications and then their OS. When all PCs and devices are shutdown, the taps will be made powerless by the PDU.

Beside the above mentioned events we introduce a number of indication callback events that reflect the status (or modes) of the system:

- the *startingUp* event informs external users that the system is in the process of starting up its components,
- the *systemStandby* event notifies users that the system is in the *System_StandBy* state,
- the *off* event tells users that the entire system is off,
- the *systemOn* event informs users that the system is up-and-running and fully operational,
- and the *geoStop* event indicates users that all motorized movements are disabled.

## 7.3 Strategy and tactics

The conceptual structure of the specification of the PDU controller is depicted in Figure 7.3. The external behavior of the PDU and the PCs are depicted as ovals. The design of the PDU controller is shown as a square shape. The communication channels with the direction of information flow are depicted using arrows.

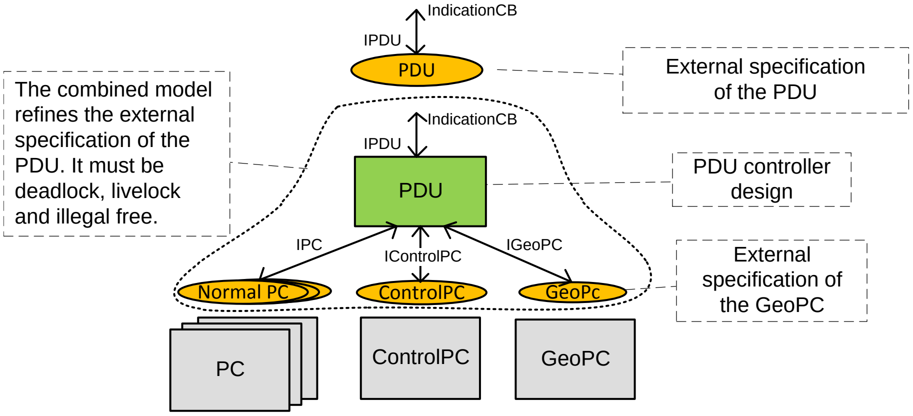

IndicationCB
IPDU
PDU
External specification of the PDU
The combined model refines the external specification of the PDU. It must be deadlock, livelock and illegal free.
IndicationCB
IPDU
PDU
PDU controller design
IPC
IControlPC
IGeoPC
Normal PC
ControlPC
GeoPc
External specification of the GeoPC
PC
ControlPC
GeoPC

**Figure 7.3** Conceptual structure of the specification of the PDU controller

The figure shows the structure of a combined model that includes the parallel composition of the PDU controller and the external specification of the PCs, highlighting the communication channels used for exchanging information among the components. Each design alternative of the PDU controller has a different combined model. To construct these models we have followed a number of steps, summarized below.

**Modeling the external behavior of the PDU** First, we modeled the desired external behavior of the PDU with respect to the external users of the system. This specification includes all external commands issued by the user console plus all indication callback signals sent to the user. The specification is identical for all design alternatives, and is used as a guide for implementing the alternative designs we are comparing. This external behavior excludes any internal interaction with the PCs.

**Describing the external behavior of the PCs** The external behavior of each PC is described with respect to the interaction required with the PDU. The description excludes any activities performed internally by the PC.

**Constructing alternative designs for the PDU controller** We design the PDU controller in two manners, namely a design where PCs 'push' their information to the PDU when the information is available, and another design where the PDU 'polls' information from the PCs whenever it is required.

For each design manner there are a number of alternatives that assist further reducing the state space. All design alternatives adhere to the external specification, and provide the external users of the system with the expected behavior.

**Modeling conventions** In the specification of all models any action pre-fixed by the letter '*r*' denotes the receiving party of a communication whereas actions pre-fixed by '*s*' denote the sending party. The result of a communication is denoted by an action without any pre-fixed letter.

**Specification completeness** In every state of the external behavior of the PCs we assign illegal responses to the stimuli if they are not expected in a state. The same response is assigned to callbacks received from the PCs in the specification of the PDU design for detecting unexpected callbacks. During the behavioral verification we search for the occurrences of such an event plus deadlock and livelock scenarios.

**Refining the external behavior** Each design alternative is checked against the external specification using a number of refinement models: weak-trace [23], Failures [78], Failures-Divergence [78], observational [68], safety [18], branching-bisimulation [82] and Tau* [31]. The reason of choosing refinement over equivalence check is that checking equivalence may tend to be overly complex. It may require that both the implementation and the external behavior to strictly have the same structure, so the external specification might be forced to be adjusted to satisfy the structure of the design. This is what we are trying to avoid here.

Instead of using equivalence checks we prove refinement of designs by means of inclusion (or preorder) checks. Precisely, we prove that the behavior of a design is included in the behavior of the external specification. Upon the success of the check we know that the design always exposes expected behavior to the external world under the refinement model being used, i.e., no extra unexpected behavior would result from the concrete implementation of the design crossing the external boundary.

We believe that specifying the external behavior of a system prior to its implementation assists constructing the system better, but does not guarantee building the internal behavior of the system correctly. Checking correctness of internal behavior of systems can be accomplished by other means such as searching for deadlocks, livelocks, illegal interactions and verifying properties on systems.

The details of the steps performed throughout this case are addressed in the subsequent sections.

## 7.4 The external specification of the PDU controller

We started our modeling activities by considering the fifth guideline. The external specification of the PDU controller in the mCRL2 language is listed below. It precisely describes the external behavior of the PDU, with respect to the external users, reflecting the internal modes of the system using states and visible indication callbacks, matching the state machine of Figure 7.2. It includes all user commands as input stimuli, and all user indication callbacks as responses to the external world. It excludes all internal interactions such as internal system messages and powering on/off the PCs.

```
proc   ExtSpec(s:State) = (
          (s≈PDU_Off)→(
            IPDU(PDUswitchOn)·IndicationCB(startingUp)·ExtSpec(StartingUpCrPC)
                          )+
          (s≈System_StandBy)→(
            IPDU(PDUswitchOff)·ExtSpec(PDU_Off)
          + IPDU(powerOn)·IndicationCB(startingUp)·ExtSpec(StartingUpAllPCs)
          + IPDU(powerOff)·ExtSpec(System_StandBy)
          + IPDU(forcedPowerOff)·IndicationCB(off)·ExtSpec(System_Off)
          + IPDU(emergencyOff)·IndicationCB(off)·ExtSpec(Emergency_Off)
          + int·IndicationCB(off)·ExtSpec(System_Off)
                                 )+
          (s≈System_On)→(
            IPDU(PDUswitchOff)·ExtSpec(PDU_Off)
          + IPDU(powerOn)·IndicationCB(startingUp)·ExtSpec(StartingUpAllPCs)
          + IPDU(powerOff)·IndicationCB(systemStandby)·ExtSpec(System_StandBy)
          + IPDU(forcedPowerOff)·IndicationCB(off)·ExtSpec(System_Off)
          + IPDU(emergencyOff)·IndicationCB(off)·ExtSpec(Emergency_Off)
```

$$
\begin{array}{l}
+\ \mathit{int}\cdot \mathit{IndicationCB}(\mathit{off})\cdot \mathit{ExtSpec}(\mathit{System\_Off})\\
+\ \mathit{int}\cdot \mathit{IndicationCB}(\mathit{geoStop})\cdot \mathit{ExtSpec}(\mathit{Geo\_Stop})\\
\qquad\qquad\qquad )+\\
(s{\approx}\mathit{StartingUpAllPCs})\rightarrow(\\
\quad \mathit{IPDU}(\mathit{PDUswitchOff})\cdot \mathit{ExtSpec}(\mathit{PDU\_Off})\\
+\ \mathit{IPDU}(\mathit{powerOn})\cdot \mathit{ExtSpec}(\mathit{StartingUpAllPCs})\\
+\ \mathit{IPDU}(\mathit{powerOff})\cdot \mathit{IndicationCB}(\mathit{systemStandby})\cdot \mathit{ExtSpec}(\mathit{System\_StandBy})\\
+\ \mathit{IPDU}(\mathit{powerOff})\cdot \mathit{ExtSpec}(\mathit{StartingUpCrPC})\\
+\ \mathit{IPDU}(\mathit{forcedPowerOff})\cdot \mathit{IndicationCB}(\mathit{off})\cdot \mathit{ExtSpec}(\mathit{System\_Off})\\
+\ \mathit{IPDU}(\mathit{emergencyOff})\cdot \mathit{IndicationCB}(\mathit{off})\cdot \mathit{ExtSpec}(\mathit{Emergency\_Off})\\
+\ \mathit{int}\cdot \mathit{IndicationCB}(\mathit{off})\cdot \mathit{ExtSpec}(\mathit{System\_Off})\\
+\ \mathit{int}\cdot \mathit{IndicationCB}(\mathit{systemOn})\cdot \mathit{ExtSpec}(\mathit{System\_On})\\
+\ \mathit{int}\cdot \mathit{ExtSpec}(\mathit{StartingUpAllPCs})\\
+\ \mathit{int}\cdot \mathit{IndicationCB}(\mathit{geoStop})\cdot \mathit{ExtSpec}(\mathit{Geo\_Stop})\\
\qquad\qquad\qquad\qquad\qquad )+\\
(s{\approx}\mathit{Geo\_Stop})\rightarrow(\\
\quad \mathit{IPDU}(\mathit{PDUswitchOff})\cdot \mathit{ExtSpec}(\mathit{PDU\_Off})\\
+\ \mathit{IPDU}(\mathit{powerOn})\cdot \mathit{IndicationCB}(\mathit{systemOn})\cdot \mathit{ExtSpec}(\mathit{System\_On})\\
+\ \mathit{IPDU}(\mathit{powerOff})\cdot \mathit{IndicationCB}(\mathit{systemStandby})\cdot \mathit{ExtSpec}(\mathit{System\_StandBy})\\
+\ \mathit{IPDU}(\mathit{forcedPowerOff})\cdot \mathit{IndicationCB}(\mathit{off})\cdot \mathit{ExtSpec}(\mathit{System\_Off})\\
+\ \mathit{IPDU}(\mathit{emergencyOff})\cdot \mathit{IndicationCB}(\mathit{off})\cdot \mathit{ExtSpec}(\mathit{Emergency\_Off})\\
+\ \mathit{int}\cdot \mathit{IndicationCB}(\mathit{off})\cdot \mathit{ExtSpec}(\mathit{System\_Off})\\
\qquad\qquad\qquad )+\\
(s{\approx}\mathit{System\_Off})\rightarrow(\\
\quad \mathit{IPDU}(\mathit{PDUswitchOff})\cdot \mathit{ExtSpec}(\mathit{PDU\_Off})\\
+\ \mathit{IPDU}(\mathit{powerOn})\cdot \mathit{IndicationCB}(\mathit{startingUp})\cdot \mathit{ExtSpec}(\mathit{StartingUpAllPCs})\\
+\ \mathit{IPDU}(\mathit{powerOff})\cdot \mathit{ExtSpec}(\mathit{System\_Off})\\
+\ \mathit{IPDU}(\mathit{forcedPowerOff})\cdot \mathit{ExtSpec}(\mathit{System\_Off})\\
+\ \mathit{IPDU}(\mathit{emergencyOff})\cdot \mathit{ExtSpec}(\mathit{Emergency\_Off})\\
\qquad\qquad\qquad )+\\
(s{\approx}\mathit{Emergency\_Off})\rightarrow(\\
\quad \mathit{IPDU}(\mathit{PDUswitchOff})\cdot \mathit{ExtSpec}(\mathit{PDU\_Off})\\
+\ \mathit{IPDU}(\mathit{powerOn})\cdot \mathit{IndicationCB}(\mathit{startingUp})\cdot \mathit{ExtSpec}(\mathit{StartingUpAllPCs})\\
+\ \mathit{IPDU}(\mathit{powerOff})\cdot \mathit{ExtSpec}(\mathit{Emergency\_Off})\\
+\ \mathit{IPDU}(\mathit{forcedPowerOff})\cdot \mathit{ExtSpec}(\mathit{Emergency\_Off})\\
+\ \mathit{IPDU}(\mathit{emergencyOff})\cdot \mathit{ExtSpec}(\mathit{Emergency\_Off})\\
\qquad\qquad\qquad\qquad\qquad )+\\
(s{\approx}\mathit{StartingUpCrPC})\rightarrow(\\
+\ \mathit{IPDU}(\mathit{PDUswitchOff})\cdot \mathit{ExtSpec}(\mathit{PDU\_Off})\\
+\ \mathit{IPDU}(\mathit{powerOn})\cdot \mathit{ExtSpec}(\mathit{StartingUpCrPC})\\
+\ \mathit{IPDU}(\mathit{powerOff})\cdot \mathit{ExtSpec}(\mathit{StartingUpCrPC})\\
+\ \mathit{IPDU}(\mathit{forcedPowerOff})\cdot \mathit{IndicationCB}(\mathit{off})\cdot \mathit{ExtSpec}(\mathit{System\_Off})\\
+\ \mathit{IPDU}(\mathit{emergencyOff})\cdot \mathit{IndicationCB}(\mathit{off})\cdot \mathit{ExtSpec}(\mathit{Emergency\_Off})\\
+\ \mathit{int}\cdot \mathit{IndicationCB}(\mathit{systemStandby})\cdot \mathit{ExtSpec}(\mathit{System\_StandBy})\\
\qquad\qquad\qquad\qquad\qquad ));
\end{array}
$$

To briefly explain the model we choose the *System_On* state as an example. The state includes seven summands in total. It precisely describes that when the PDU is in the *System_On* state, it can receive any external command from the users. This is indicated by the first five summands. Upon receiving an external command the PDU may send indication callback signals and then transits to a next state. For example, when the PDU receives the *powerOff* command, it sends the *systemStandby* indication to the external users and then transits to the *System_StandBy* state.

The last two summands of the state represent the cases where external users can receive indications that the system is off or transiting to the *Geo_Stop* state, due to some internal interactions with the PDU. Both *int* events represent detailed activities performed by the concrete implementation of the PDU. For example, *int.IndicationCB(off)* represents the following internal activities:

1. The user of the ControlPC has requested the PDU to power off the entire system via the internal *controlPowerOff* callback event.
2. The PDU treats the signal by sending the *shutdown* message around to all devices.
3. The PDU switches all taps off.
4. The PDU sends the *IndicationCB(off)* signal to the external world.
5. The PDU transits to the *System_Off* state.

The same technique had been applied to all states of the PDU, matching the original state machine of Figure 7.2. The complete specification of the model can be found in [44], Appendix A.

When the specification of the model was completed, it was checked for absence of deadlocks and livelocks. The corresponding LTS had been generated, and used at later stages for the refinement check against the concrete designs of the PDU using mCRL2 and CADP.

## 7.5 Implementing the PDU controller using the push strategy

In this variant, the design of the controller utilizes a pushing strategy, in the sense that all PCs share information with the PDU controller upon changes in their internal states. This is illustrated in the sequence diagram in Figure 7.4. For instance, when the PDU is in the *System_On* state and the *Stop* button is pressed, the GeoPC notifies the PDU controller by sending the *stop* callback event. The same applies to the *controlPowerOff* callback from the ControlPC. Furthermore, when the PCs are powered on by the PDU, the PDU waits for callbacks from the PCs indicating that they are ready and fully operational.

### 7.5.1 The external behavior of the PCs

In this section we introduce the external specification of the ControlPC that describes the external behavior that is related to the PDU controller. Similarly, the specification of the remaining PCs is straightforward and therefore omitted from the text, but can be found in [44], Appendix B. The specification of the PCs are identical for all push design variants.

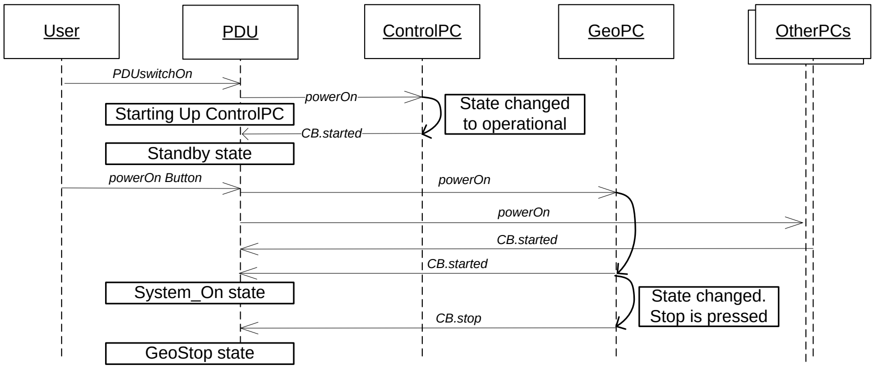


**Figure 7.4** Example of a scenario where pushing is used

```
proc    ControlPC(s:PCState) = (
          (s≈PC_Off)→(
            rICR_PC(powerOn)·ControlPC(StartingUp)
          + rICR_PC(powerOff)·Illegal·delta
          + rICR_PC_Broadcast(restart)·Illegal·delta
          + rICR_PC_Broadcast(shutdown)·Illegal·delta
                           )+
          (s≈Operational)→(
            rICR_PC(powerOn)·Illegal·delta
          + rICR_PC(powerOff)·ControlPC(PC_Off)
          + sICR_PC_CB(controlPowerOff)·ControlPC(WaitingShutdown)
          + rICR_PC_Broadcast(restart)·ControlPC(StartingUp)
          + rICR_PC_Broadcast(shutdown)·ControlPC(OS_Shutdown)
                             )+
          (s≈WaitingShutdown)→(
            rICR_PC(powerOn)·Illegal·delta
          + rICR_PC(powerOff)·ControlPC(PC_Off)
          + rICR_PC_Broadcast(restart)·ControlPC(StartingUp)
          + rICR_PC_Broadcast(shutdown)·ControlPC(OS_Shutdown)
                                 )+
          (s≈OS_Shutdown)→(
            rICR_PC(powerOn)·Illegal·delta
          + rICR_PC(powerOff)·ControlPC(PC_Off)
          + rICR_PC_Broadcast(restart)·Illegal·delta
          + rICR_PC_Broadcast(shutdown)·Illegal·delta
                              )+
          (s≈StartingUp)→(
            rICR_PC(powerOn)·Illegal·delta
          + rICR_PC(powerOff)·ControlPC(PC_Off)
          + rICR_PC_Broadcast(restart)·Illegal·delta
          + rICR_PC_Broadcast(shutdown)·Illegal·delta
          + sICR_PC_CB(started)·ControlPC(Operational)
                             )
                                );
```

The specification of the ControlPC is straightforward. It includes five states. In any state the ControlPC can receive a number of legal and illegal stimuli events.

Note that, when the ControlPC is in the *StartingUp* state, it can send (or push) the *sICR_PC_CB(started)* callback event to the PDU and then transits to the *Operational* state. Similarly, when the ControlPC is in the *Operational* state, it can send the callback event *sICR_PC_CB(controlPowerOff)* to the PDU, as a request to power off the entire system.

### 7.5.2 The design of the PDU controller

There are mainly four alternative models for the PDU designs that incorporate the push strategy. The details of each of them are introduced below.

**The asynchronous PDU controller** In this variant the PDU controller communicates with the PCs synchronously and sequentially one-by-one, but the PCs communicate with the PDU asynchronously. The PDU includes a queue to store incoming callback events from the PCs.

The first issue we encountered when verifying this variant was the queue size and the large number of interleaving caused by the queue and the external commands. The PCs can quickly send callback events to the queue leading to filling-up a queue of any arbitrary size. External commands can arrive while there are still unprocessed callbacks in the queue, hence verification was initially not doable.

Therefore, we had to limit the behavior of the PCs such that having more than one similar callback at a time in the queue is prohibited. Furthermore, we give any callback event a priority to be processed by the PDU over any external user command, so the queue has to be emptied first.

Below we introduce a part of the design specification, demonstrating only the *PDU_Off* stable state and the *StartingUp_CR_PC* transiting state. The entire specification can be found in [44], Appendix B.

```
proc  PDU_State_Machine(s:PDUState,
                        geopcOn, crpcOn, geoPressed:Bool, startedPc:Nat) = (
      (s≈PDU_Off)→(
        rIPDU(PDUswitchOn)·sICR_PC(powerOn)·IndicationCB(startingUp)·
            PDU_State_Machine(StartingUp_CR_PC, geopcOn, crpcOn,
                                                      geoPressed, startedPc)
      + rICR_PC_CB(controlPowerOff)·
            PDU_State_Machine(PDU_Off, geopcOn, crpcOn, geoPressed, startedPc)
      + rIGeoPC_CB(stop)·
            PDU_State_Machine(PDU_Off, geopcOn, crpcOn, geoPressed, startedPc)
      + rICR_PC_CB(started)·
            PDU_State_Machine(PDU_Off, geopcOn, crpcOn, geoPressed, startedPc)
      + rIGeoPC_CB(started)·
            PDU_State_Machine(PDU_Off, geopcOn, crpcOn, geoPressed, startedPc)
```

$$
\begin{array}{l}
+\ \sum_{id:\mathbb{N}+} \cdot rIPC\_CB(id, started)\cdot\\
\qquad PDU\_State\_Machine(PDU\_Off, geopcOn, crpcOn, geoPressed, startedPc)\\
\qquad\qquad\qquad\qquad )+\\
(s \approx StartingUp\_CR\_PC) \rightarrow (\\
\quad rIPDU(PDUswitchOff)\cdot sICR\_PC(powerOff)\cdot\\
\qquad PDU\_State\_Machine(PDU\_Off, geopcOn, crpcOn, geoPressed, startedPc)\\
+\ rIPDU(powerOn)\cdot\\
\qquad PDU\_State\_Machine(StartingUp\_CR\_PC, geopcOn, crpcOn,\\
\qquad\qquad\qquad\qquad\qquad\qquad geoPressed, startedPc)\\
+\ rIPDU(powerOff)\cdot\\
\qquad PDU\_State\_Machine(StartingUp\_CR\_PC, geopcOn, crpcOn,\\
\qquad\qquad\qquad\qquad\qquad\qquad geoPressed, startedPc)\\
+\ rIPDU(forcedPowerOff)\cdot sICR\_PC(powerOff)\cdot IndicationCB(off)\cdot\\
\qquad PDU\_State\_Machine(System\_Off, geopcOn, crpcOn,\\
\qquad\qquad\qquad\qquad\qquad\qquad geoPressed, startedPc)\\
+\ rIPDU(emergencyOff)\cdot sICR\_PC(powerOff)\cdot IndicationCB(off)\cdot\\
\qquad PDU\_State\_Machine(Emergency\_Off, geopcOn, crpcOn,\\
\qquad\qquad\qquad\qquad\qquad\qquad geoPressed, startedPc)\\
+\ rICR\_PC\_CB(controlPowerOff)\cdot Illegal\cdot delta\\
+\ rICR\_PC\_CB(started)\cdot IndicationCB(systemStandby)\cdot\\
\qquad PDU\_State\_Machine(SystemStandby, geopcOn, crpcOn,\\
\qquad\qquad\qquad\qquad\qquad\qquad geoPressed, startedPc)\\
+\ rIGeoPC\_CB(stop)\cdot\\
\qquad PDU\_State\_Machine(StartingUp\_CR\_PC, geopcOn, crpcOn,\\
\qquad\qquad\qquad\qquad\qquad\qquad geoPressed, startedPc)\\
+\ rIGeoPC\_CB(started)\cdot\\
\qquad PDU\_State\_Machine(StartingUp\_CR\_PC, geopcOn, crpcOn,\\
\qquad\qquad\qquad\qquad\qquad\qquad geoPressed, startedPc)\\
+\ \sum_{id:\mathbb{N}+} \cdot rIPC\_CB(id, started)\cdot\\
\qquad PDU\_State\_Machine(StartingUp\_CR\_PC, geopcOn, crpcOn,\\
\qquad\qquad\qquad\qquad\qquad\qquad geoPressed, startedPc)\\
\qquad\qquad\qquad\qquad\qquad )+\\
\\
....(the\ specification\ of\ the\ rest\ of\ states\ follows)\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad );
\end{array}
$$

The first summand of the *PDU_Off* state specifies that when the PDU is switched on, it powers on the ControlPC, sends an indication to the user that the system is starting-up and transits to the *StartingUp_CR_PC* state. The other remaining summands of the state specify that any (late) callback events received in the state are consumed.

When the PDU is in the *StartingUp_CR_PC*, it can process a number of external commands and internal callbacks. All callbacks are ignored except those originating from the ControlPC. When the PDU knows that the ControlPC is ready and fully operational by receiving the *rICR_PC_CB(started)* from the queue, it sends an indication to the user that the system is in the *SystemStandby* state. Receiving a request to power off the system from the ControlPC is illegal since the ControlPC has to start first, see the summand that corresponds to the *rICR_PC_CB(controlPowerOff)* stimulus event.

The specification of the PDU controller includes a number of data parameters used for remembering the status of the PCs. For example, the *startedPc* data parameter is of type natural number and is used to count how many normal PCs have started.

**The asynchronous PDU controller with global synchronous communication** The model of this variant is almost identical to the previous model, except that the fourth guideline is used. We noticed that powering on/off the PCs can be modeled using multi-actions. That is, instead of modeling this behavior by sending the *powerOn* or *powerOff* events to the PCs sequentially, all PCs engage into one big action, denoting that the event occurs at the same time for all PCs.

To clarify the concept, consider the following examples. The following Handler process communicates with the PDU (via the *rcommandhandler* and srelease actions) and the PCs (via the *sIPC* action), where all communications are done sequentially until completion. This process is used in the specification of the asynchronous push model addressed earlier.

$$\begin{aligned} \textbf{proc}\quad & Handler = \\ & \quad \textstyle\sum_{c:Command} \cdot rcommandhandler(c)|sIPC(1,c)\cdot \\ & \qquad\quad sIPC(2,c)\cdot sIPC(3,c)\cdot sIPC(4,c)\cdot srelease|sIPC(5,c)\cdot Handler \end{aligned}$$

Obviously, this process results in five successive states, with the possibility of interleaving with other processes.

On the other hand, the following Handler process describes the use of multi-actions, used for this design variant. All communications with PCs are done in one step.

$$\begin{aligned} \textbf{proc}\quad & Handler = \\ & \quad \textstyle\sum_{c:Command} \cdot rcommandhandler(c)|sIPC(1,c)|sIPC(2,c)| \\ & \qquad\quad sIPC(3,c)|sIPC(4,c)|sIPC(5,c)\cdot Handler \end{aligned}$$

Clearly, this process results in a single state.

Since the number of states are reduced to a single state, the entire state space can also be reduced, taking into account the reduced interleaving. The complete specification of this model is listed in [44], Appendix C.

**The synchronous PDU controller** In this variant all interactions between the PDU and the PCs are synchronous. In contrast with the previous variants, the PDU does not include any queue, and all received callbacks from the PCs are processed synchronously. Still, all PCs inform (or push) the PDU upon the changes of their states, but in a synchronous manner.

The specification of this variant is listed in [44], Appendix D. The specification is similar to the asynchronous variant except that the queue placed between the PDU and the PCs is removed.

**The synchronous PDU controller with global synchronous communication** Here, the model of synchronous PDU controller above is adapted, such that powering on/off PCs is accomplished by multi-actions. The detail of using multi-actions is previously described for the asynchronous controller with global synchronous communication variant, and hence is omitted here. The complete specification of this variant is introduced in [44], Appendix E.

## 7.6 Implementing the PDU controller using the poll strategy

In this section we present a model that describes the implementation of the PDU controller using a polling strategy. We used the first guideline to accomplish this model. Using this style, the PDU controller polls the PCs to acquire their states. Figure 7.5 visualizes an example of polling used for designing the controller.

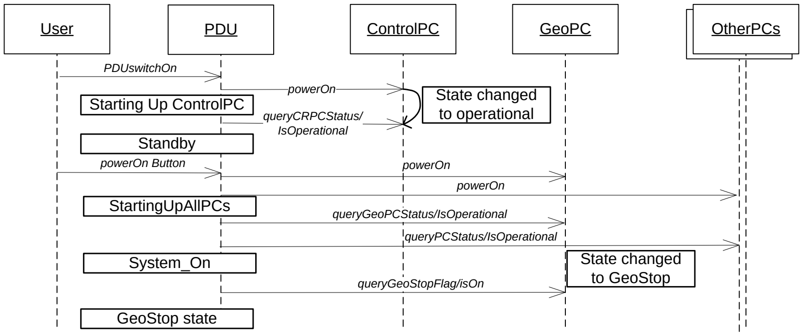


**Figure 7.5** Example of a scenario where polling is used

### 7.6.1 The external behavior of the PCs

Before describing the design of the controller, we first need to describe the external behavior of the PCs. Below, a fragment of the mCRL2 specification related to the external behavior of the ControlPC is described. The specification of the GeoPC and the normal PCs are straightforward and almost identical to this specification.

```
proc  ControlPC(s:PCState) = (
      (s≈PC_Off)→(
        rICR_PC(powerOn)·ControlPC(PC_On)
      + rICR_PC(powerOff)·Illegal·delta
      + rICR_PC(queryCRPCStatus)·Illegal·delta
      + rICR_PC(queryCRPCPowerOffFlag)·Illegal·delta
      + rICR_PC_Broadcast(restart)·Illegal·delta
      + rICR_PC_Broadcast(shutdown)·Illegal·delta
                      )+

      (s≈PC_On)→(
        rICR_PC(powerOn)·Illegal·delta
      + rICR_PC(powerOff)·ControlPC(PC_Off)
      + rICR_PC(queryCRPCStatus)·sICR_PCrVal(IsOperational)·
                                                        ControlPC(PC_On)
      + rICR_PC(queryCRPCStatus)·sICR_PCrVal(IsNotOperational)·
                                                        ControlPC(PC_On)
      + rICR_PC(queryCRPCPowerOffFlag)·sICR_PCrVal(IsOn)·
                                                        ControlPC(PC_On)
      + rICR_PC(queryCRPCPowerOffFlag)·sICR_PCrVal(IsOff)·
                                                        ControlPC(PC_On)
      + rICR_PC_Broadcast(restart)·ControlPC(PC_On)
      + rICR_PC_Broadcast(shutdown)·ControlPC(OS_Shutdown)
                    )+
      (s≈OS_Shutdown)→(
        rICR_PC(powerOn)·Illegal·delta
      + rICR_PC(powerOff)·ControlPC(PC_Off)
      + rICR_PC(queryCRPCStatus)·Illegal·delta
      + rICR_PC(queryCRPCPowerOffFlag)·sICR_PCrVal(IsOff)·
                                                   ControlPC(OS_Shutdown)
      + rICR_PC_Broadcast(restart)·ControlPC(OS_Shutdown)
      + rICR_PC_Broadcast(shutdown)·ControlPC(OS_Shutdown)
                           )       ) ;
```

As can be seen from the specification, the ControlPC has three main states:

- *PC_Off*: the PC is off, which means that the tap is switched off;
- *PC_On*: the PC is on, which means that the tap is switched on, but the applications on the PC can be operational or not; and
- *OS_Shutdown*: the tap is on but the OS and the applications are shut down.

Per state it is defined which calls are allowed to be issued by the design of the PDU controller, and which of them are illegal. Every query (or poll) method has a return value that is immediately sent back to the PDU.

When the ControlPC is powered on, it can receive a number of signals by polling. The ControlPC non-deterministically replies to these signals indicating its current state: for example, observe the summands with *queryCRPCStatus* and *queryCRPCPowerOffFlag* calls which non-deterministically return a value in the *PC_On* state.

### 7.6.2 The design of the PDU controller

The PDU controller design has to adhere to the external specification of the PDU on the one hand, and to correctly use the specifications of the PCs on the other hand. To implement a polling mechanism, the PDU utilizes internal timers to stimulate the PDU to poll status of the PCs in certain states. As we will see shortly, the fourth guideline is employed to abstract from concrete data values of the timer. For example, we abstract from the progress of timer values in milliseconds by a single event denoting the expiration of the time.

Moreover, the third guideline is used for modeling the start-up behavior of the system. Compared to the push model the PDU sequentially polls information about the state of the PCs, when it is needed. The PDU does not expect any spontaneous information to be pushed by the PCs. The complete specification of this variant can be found in [44], Appendix F.

Below we introduce a fragment of the controller design specification, related to *PDU_Off*, *StartingUpCrPC* and *WaitingCRPCReply* states.

```
proc    PDU_State_Machine(s : PDUState, cRPCstarted, geoPCstarted,
                                        geoPressed : Bool, state : PDUState) = (
        (s≈PDU_Off)→(
          IPDU(PDUswitchOn)·sICR_PC(powerOn)·IndicationCB(startingUp)·
              PDU_State_Machine(StartingUpCrPC, false, false, false, none)
                        )+
        (s≈StartingUpCrPC)→(
          IPDU(PDUswitchOff)·sICR_PC(powerOff)·
              PDU_State_Machine(PDU_Off, cRPCstarted,
                                        geoPCstarted, geoPressed, state)
        + IPDU(powerOn)·
              PDU_State_Machine(StartingUpCrPC, cRPCstarted,
                                        geoPCstarted, geoPressed, state)
        + IPDU(powerOff)·
              PDU_State_Machine(StartingUpCrPC, cRPCstarted,
                                        geoPCstarted, geoPressed, state)
        + IPDU(forcedPowerOff)·sICR_PC(powerOff)·IndicationCB(off)·
              PDU_State_Machine(System_Off, cRPCstarted,
                                        geoPCstarted, geoPressed, state)
        + IPDU(emergencyOff)·sICR_PC(powerOff)·IndicationCB(off)·
              PDU_State_Machine(Emergency_Off, cRPCstarted,
                                        geoPCstarted, geoPressed, state)
        + IPDUTimer(pollPC)·sICR_PC(queryCRPCStatus)·
              PDU_State_Machine(WaitingCRPCReply, cRPCstarted,
                                        geoPCstarted, geoPressed, state)
                              )+
        ...
(s≈WaitingCRPCReply)→(
  rICR_PCrVal(IsOperational)·IndicationCB(systemStandby)·
        PDU_State_Machine(System_Standby, true,
                                    geoPCstarted, geoPressed, state)
+ rICR_PCrVal(IsNotOperational)·
        PDU_State_Machine(StartingUpCrPC, false,
                                    geoPCstarted, geoPressed, state)
                          )+
...
                                                                ) ;
```

The fragment describes that when the system is switched on in the *PDU_Off* state, the ControlPC is powered on, the user gets an indication that the system is starting up, and the PDU transits to the *StartingUpCrPC* state. As can be inferred from the specification, the *StartingUpCrPC* state is used to not only monitor the progress of starting up the ControlPC, but also to react upon the external requests from users.

Then, when the PDU is stimulated by the timer via the *pollPC* signal, the PDU requests the state of the ControlPC by sending the *queryCRPCStatus* signal and transits to the *WaitingCRPCReply* state, waiting a response from the ControlPC. As specified in the external behavior of the ControlPC, either *IsOperational* or *IsNotOperational* signals are returned to the PDU. Depending on the return value, the PDU transits back to *StartingUpCrPC* (and hence can query the status of the ControlPC again), or gives an indication that the system is in standby and transits to the *System_Standby* state.

Similarly, when the system is in the *System_Standby* stable state and the *PowerOn* button is pressed, the PDU transits to the *StartingUpAllPCs* state where all other PCs are checked, in the same manner of checking the status of the ControlPC described above.

```
...
(s≈WaitingPC1statusReply)→(
  rIPCrVal(1, IsOperational)·sIPC(2, queryPCstatus)·
        PDU_State_Machine(WaitingPC2statusReply, cRPCstarted,
                                    geoPCstarted, geoPressed, state)
+ rIPCrVal(1, IsNotOperational)·
        PDU_State_Machine(StartingUpAllPCs, cRPCstarted,
                                    geoPCstarted, geoPressed, state)
                                )+
(s≈WaitingPC5statusReply)→(
  (!geoPressed)→rIPCrVal(5, IsOperational)·IndicationCB(systemOn)·
        PDU_State_Machine(System_On, cRPCstarted,
                                    geoPCstarted, geoPressed, state)
+ (geoPressed)→rIPCrVal(5, IsOperational)·IndicationCB(geoStop)·
        PDU_State_Machine(Geo_Stop, cRPCstarted,
                                    geoPCstarted, geoPressed, state)
+ rIPCrVal(5, IsNotOperational)·
        PDU_State_Machine(StartingUpAllPCs, cRPCstarted,
                                    geoPCstarted, geoPressed, state)
                                )+
...
```

That is, the first PC is checked if it is operational or not. If the first PC is not operational, the system can transit back to the *StartingUpAllPCs* state; see for example the specification of the *WaitingPC1statusReply* state above. If the first PC is operational, then the second PC is checked, and so on until all PCs are operational. When the last PC is operational, an indication is sent to the user, and then the PDU moves to the *System_On* state, see the *WaitingPC5statusReply* state.

During starting up of all PCs, the PDU queries the GeoPC and the ControlPC to check the status of whether any of the *Stop* buttons has been pressed or if the user needs to power off the entire system. If these flags are on, on the respective PCs, the PDU immediately switches off the taps supply the movable part or starts to power off the entire system. The PDU remembers the status of the *Stop* button, and therefore, when the last PC is operational, the PDU transits to *System_On* or *Geo_Stop* stable states.

**The poll controller with global synchronous communication** In combination with the first guideline, we use guideline 2 to model the instantaneous powering on or off the PCs. The same global synchronous communication concept used for the *Handler* process of the push model is also used here. We refer to Appendix G in [44] for the entire specification of this model.

## 7.7 Results of the experiments

After the specification of all models were created using the mCRL2 description language, we started the verification tasks. We used the mCRL2 tool set (July 2011 release) for performing verification and state space generation on a Unix-based server machine (4 × 2.5 Ghz processor and 46 GB RAM). The generated state spaces of all models were further analyzed using CADP (June 2011 beta release) for checking deadlocks, livelocks, illegals and proving refinements of designs against the external specification.

Table 7.4 depicts the activities performed throughout this work together with the tools used to accomplish each of them. The '✓' mark indicates a feature supported by the tool and being used in this work, '$-$' denotes that the feature is supported by the tool but is not being used in this work, and '×' indicates that the tool does not support the feature. As can be seen from the table, the formal specification using CADP is skipped since we used the mCRL2 for state space generation. The state space was analyzed later using both mCRL2 and CADP. We also translated the mCRL2 models to CSP and used FDR2 (FDR2 2.91 academic use release) for state space generation. FDR2 was used to verify refinements under traces, failures and Failures-Divergence models, of which the last two are not supported by both mCRL2 and CADP. When the state space of each model has been generated, branching bisimulation reduction was applied after all internal events not visible on the external specification are hidden, to facilitate the verification and refinement tasks.

The three tools were used for searching for occurrences of deadlocks and illegals. All

| Activity | mCRL2 | CADP | CSP/FDR2 |
|---|---|---|---|
| Formal specification | ✓ | – | ✓ |
| State space generation | ✓ | – | ✓ |
| Branching Bisimulation Reduction | ✓ | – | × |
| Checking deadlocks and illegals | ✓ | ✓ | ✓ |
| Checking livelocks | – | – | ✓ |
| Checking Weak-traces | ✓ | ✓ | ✓ |
| Checking Failures (-Divergences) | × | × | ✓ |
| Checking Observational | × | ✓ | × |
| Checking Safety | × | ✓ | × |
| Checking Tau* | × | ✓ | × |
| Checking Branching | × | ✓ | × |

**Table 7.4** List of performed tasks plus the tools used to realize them.

tools provided the same result, namely all models are deadlock and illegal free. After we hid all events except those exposed in the external specification, we checked for the occurrences of livelocks. Checking livelocks merely was accomplished using FDR2. The reason of choosing FDR2 over other tools is that FDR2 provides readable, easy to analyze, counterexamples in case livelocks exist. The mCRL2 for example can report a sequence that leads to a cycle of *tau* events, but one can hardly deduce the corresponding original actions that form the cycle. The same applies for CADP.

We encountered a similar issue when trying to prove refinement of designs against the external behavior using both mCRL2 and CADP. The tools can easily find counterexamples when a refinement check is violated, under the refinement models they support. But, the generated counterexamples were hard to read since all original internal actions were permanently replaced by the hidden action *tau*. By using mCRL2 and CADP, we spent extra time analyzing the counterexamples and to 'guess' the correct original events correspond to the hidden events by matching the sequence of *tau's* on the original system. This indeed caused more efforts and time to be spent for modeling and verification since we did not efficiently know whether the design or the external specification was incorrect. Notable is that knowing the original actions correspond to the hidden action *tau* when checking refinements was straightforward in FDR2.

However, when we attempted to verify an initial model of the push design using FDR2, the tool quickly crashed during the compilation phase. The reason is that the model initially implements a *list* to store the started PCs, see the *startedPc* data parameter in the push model introduced earlier. The controller needs this list during the start-up of the system in order to know that all PCs are fully operational before moving to the *System_On* state. It seems that having such a list in our model caused FDR2 to crash, and thus when replacing the list by a counter, the issue was solved indeed. Notable is that mCRL2 dealt with both types of push models that include either a list or a counter of started PCs effectively. The last four refinement checks were performed using CADP, which was the only tool supporting them.

In Table 7.5 we summarize the end result of checking refinements of designs, under a number of refinement models. The table is self-explainable. All designs refine the external specification under all refinement models, which means that all designs provide the expected behavior to external users of the system according to the predefined external specification. The only exception is the refinement of the poll models under the Failures-Divergence model, which fails due to the presence of a livelock.

| Model | Weak-traces | Failures | Fail. Diverg. | observational | safety | Tau* | branching |
|---|---|---|---|---|---|---|---|
| Async. Push | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Async. Push Global Sync. | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Sync Push | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Sync Push Global sync. | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Poll | ✓ | ✓ | × | ✓ | ✓ | ✓ | ✓ |
| Poll Global sync. | ✓ | ✓ | × | ✓ | ✓ | ✓ | ✓ |

**Table 7.5** Results of checking refinements of designs against the external behavior

The livelock exists in the poll models since internal *tau* loops can easily be formed, see Figure 7.6 for a livelock scenario. For example, in case the ControlPC is not operational, the PDU controller will query it again. This can continue forever, unless the ControlPC becomes operational. But, since the external users can still issue external commands even if the ControlPC is not operational, we consider this livelock to be rather benign, and indeed the livelock represents a desired behavior.[1]

Note that all designs are deadlock, livelock and illegal free except the poll designs, which are not livelock free due to the above mentioned reason.

| Model | States | Transitions | BB | | BBDP | |
|---|---|---|---|---|---|---|
| External specification | 15 | 53 | 13 | 46 | 13 | 47 |
| Async. Push | 78,088,550 | 122,354,296 | 47 | 173 | 47 | 173 |
| Async. Push Global Sync. | 44,866,381 | 75,945,810 | 47 | 173 | 47 | 173 |
| Push sync | 6,318 | 8,486 | 23 | 111 | 23 | 111 |
| Push sync global sync | 3,832 | 6,000 | 23 | 111 | 23 | 111 |
| Poll | 953 | 1,367 | 14 | 54 | 14 | 60 |
| Poll global sync | 608 | 1,022 | 14 | 54 | 14 | 60 |

**Table 7.6** State spaces of all models

[1] In fact there are additional services deployed on a number of PCs for monitoring the status of PCs. If they detect that there is some PC has failed to start, they try to start it again using its baseboard management control (BMC) via its intelligent platform management interface (IPMI), through the Ethernet network. But, the PDU team is not responsible of implementing these services.

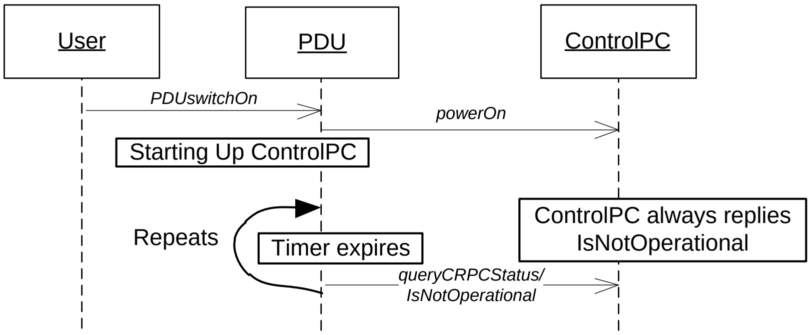


**Figure 7.6** A divergence example

The last table sums up the statistical data related to the size of generated state spaces. The second and third columns shows the number of generated states and transitions for the entire state spaces. The branching-bisimulation (BB) columns depict the number of resulting states and transitions after the branching-bisimulation reduction was applied on the original state space, while those resulting from branching-bisimulation compression with divergence preserving (BBDP) are depicted in the last columns.

The difference between the number of transitions of the poll models after compression using BB and BBDP indicates that the poll model includes divergences. A divergence scenario of the poll model was discussed earlier.

As can be seen from the table, the poll variants appear to be better than others, with only 953 and 608 states. They show also fewer states after compression. This favorably compares to the asynchronous push model which includes 78,088,550 states. Therefore, it seems that extending the asynchronous push model further with extra details may limit the verification process, unless the design of the PDU controller is decomposed into a number of smaller components verified in isolation, or on-the-fly reduction techniques are used for circumventing a foreseen state space explosion.

Finally, the above results indicate that different design styles can substantially influence the number of states of the modeled systems. This confirms that these design styles are effective in practice. Although more experiments need to be done with these design styles, we are strengthened in our believe that these styles are very important and designers should be actively aware of such strategies if they want to design verifiable systems.

# Chapter 8

# Conclusions

## 8.1 Introduction

In this chapter we summarize the achieved results of this work and the future directions. The general theme of this thesis is to demonstrate how formal techniques were applied to the development of industrial control components of an X-ray machine, developed at Philips Healthcare. The thesis discusses different aspects on how the techniques were tightly integrated into the development life cycle, investigating in depth the main encountered issues and their practical solutions. Furthermore, the thesis evaluates the effectiveness of these techniques to the quality and the productivity of a number of developed software units. The concept of quality refers here to the error density in terms of the number of reported defects per thousand lines of code while productivity denotes the number of lines of code produced per hour.

The general conclusion of this work is that, in the context of Philips Healthcare, formal techniques could deliver better quality code compared to software developed using the traditional development methods. In some cases, e.g., the PCS, the techniques could deliver zero-defects software, especially when they were combined with the test-driven development method and when the components were small and manageable. Our findings indicate the possibility of a 10 fold reduction in number of errors and also a threefold increase in the productivity. We reached this conclusion by thoroughly investigating the way formal techniques were applied, the peculiarities of ASD models, the related code of the units and by carefully analyzing over 1400 defect reports. Below we summarize the main results and the key observations that we obtained during this work.

## 8.2 Summary of achieved results and observations

In Chapter 3, we detailed the application of the ASD formal techniques in the BasiX project. The technology was used for developing two software systems, namely the controller of the power distribution unit (PDU) and the power control services (PCS).

First, as a result of applying ASD to formally specify and verify the controller of the PDU, we detected and corrected two hidden errors in the design of the controller before the actual implementation was commenced. This clearly leads to reduce the overhead and the cost of detecting and correcting the errors at later stages of the project or after the release of the product to the market. Using this case we found that ASD specification completeness was in particular very effective in assisting the detection of the errors using both manual inspections and formal verification using model checking.

Second, for the PCS a workflow that combines the ASD technology and the test-driven development method for developing the control and non-control components of the PCS was introduced. The main result of combining the two techniques is that the entire PCS software exhibits zero-defects and the PCS runs correctly after deploying it on every PC from the first execution. The reason is that the control components were developed under the control of ASD formal techniques. Another important reason is that the components of the PCS were kept small and manageable. However, due to the encountered tensions regarding the compatibility of the ASD generated code with other used tools and techniques, the ASD generated code was excluded from the coding standards check and all ASD artifacts including the ASD models were stored in the version management system.

A common view when applying ASD was that there is no need to test the generated code since it is formally verified. But, this was not totally correct. We found that only statement and function coverage tests can be excluded but not the black-box test. An advantage of using the test-driven development method is that few errors in the ASD generated code were detected by the developer during the black-box test of the PCS.

In chapter 4, we detailed the application of the ASD technology to the development of sizable software units, namely the FEClient and the Orchestration of the Backend subsystem. We investigated how the ASD approach was used and the percentage of time consumed for each development process. We found that the design and formal specification and verification processes took a longer time than other development processes such as testing and integration which were very smooth due to the use of formal techniques.

To calculate the quality figures of the developed code, we analyzed the code and investigated the submitted defects. After that, we could estimate the quality level of the units. By comparing the quality of the ASD code with other handwritten code we found that ASD code was better in the sense that fewer defects were submitted along the construction of the units although the handwritten code was simple. Despite that there were few errors found, the ASD technology could deliver close to zero defects per thousand lines of code.

Table 8.1 classifies the typical types of errors left behind by the ASD formal verification, summarizing the potential reasons of why they were not detected using the ASD formal techniques. In general, we found that nearly 20% of the errors in the ASD code were related to the data part while 80% were related to the control part.

In Chapter 5, we evaluated the use of ASD formal techniques to the quality and the productivity of the developed units, comparing them with the statistics reported for worldwide projects. We started by considering each subsystem in isolation, starting with the Frontend subsystem and then the Backend subsystem. By comparing the quality and the productivity of the units incorporating formal methods with others, we found that formal methods could deliver nearly 10 times better quality code and led to better productivity.

Despite that the units exhibit high quality figures, the incorporation of ASD in the Frontend initially was not very smooth and developers ran into a number of issues, which we detailed in Chapter 6. When developers became skilled in the technology, they could eventually compose verifiable components.

However, we noticed that when the specification of ASD models was too complex and an error was found, developers tend to ignore the complexity and fix these errors by adding more details to the models. This increased the complexity of the models and made comprehensibility and maintainability even worse.

A potential solution was to pull out some of the details from the complex components to

| | Defect Type | Reason of overlooking |
|---|---|---|
| 1 | Misspellings in data parameters causes the generation of independent variables in the code. | ASD does not check consistency of the variables declared in the data parameters passed in calls and callbacks. |
| 2 | Incorrect data values were passed in the parameters of the calls. | Correctness of the data values in parameters is not checked in ASD. |
| 3 | Waiting in a state for a timeout while overlooking to start the timer. | Models in this case can be deadlock and illegal free since in the state where a timeout is expected it is still possible to react to external stimuli. |
| 4 | Wrong order in the response list. | Compositional verification does not include the design models of other components. |
| 5 | Incorrect behavior at some states. | Lack of formulating and verifying system specific properties. |
| 6 | Missing responses in rule cases. | Lack of formulating and verifying system specific properties. |
| 7 | Missing requirements. | Behavior is not implemented. |
| 8 | Incorrect invocations at some states. | Incorrect used interfaces. |

**Table 8.1** Summary of the type of errors that escaped formal techniques

other new or existing components as early as possible, but this rarely happened. Consequently, as we show in Chapter 6, complex models contributed more to the errors in the control part of the units and decreased the quality.

Another example is that the BEFE interface model specifying the interaction between the Fronend and the Backend subsystems was overly complex. The advantage of using formal techniques here is that they give an early warning about the complexity of the proposed architecture. The amount of details the BEFE interface had indicated very early in the development cycle that the reference architecture has to be considered again to find the sources of complexity and the potential solutions.

But similarly, developers ignored this, continued the development of the subsystems and realized the complexity at very late stages of the project. Although the units that incorporate formal methods were complex, the end quality was good and they were very robust against the frequent changes of the requirements. This was not the case for other units developed manually since they were redesigned again from scratch, for example the Viewing and the X-ray IP units in the Backend.

All team members involved in developing the ASD components in the BasiX, the Frontend and the Backend had nearly the same experience and skills in developing software systems as well as the same level of domain knowledge. The knowledge of formal methods was limited to some courses at the university level and none of the development team had substantial mathematical skills. Despite this limitation the team could deliver good quality software regardless of the issues encountered at the start of applying the formal techniques, and hence the successful use of these techniques did not require expert mathematicians.

Chapters 3,4, and 5 of this thesis treated different aspects of the application of ASD formal techniques to the development of software components in three different projects. From the perspectives detailed in the chapters, the pros and the cons of these techniques were highlighted. Table 8.2 summarizes some of the strengths and the weaknesses of the ASD technology, ordered by the development phases.

| Pros | Cons |
|---|---|
| Design | |
| - Designs are constructed as highly cohesive, low-coupled components with well-defined interfaces.<br>- Step-wise refinement of systems from external behavior to concrete behavior allows the technology to scale to industrial applications.<br>- Components are structured in strict layers. This decreases dependencies among the components.<br>- Less freedom in designing components makes designs more easy to understand.<br>- Suitable to design event-based reactive systems.<br>- Action-oriented, state-based approach. | - Hard to obtain verifiable designs due to the lack of design guidelines.<br>- Time is needed for paradigm shifting from object-orientation to component-based, action-oriented approach.<br>- Not every developer can compose designs with verifiable components.<br>- Not suitable for designing low-level real-time controllers.<br>- Not suitable for designing algorithms or systems require data computations such as image processing, construction of compilers or databases...etc. |
| Formal Specification | |
| - Formal specification provides a shared understanding among all involved stakeholders.<br>- Specification completeness forces thinking of every possible scenario.<br>- Completeness of specification leads to find omissions and gaps in requirements early in development.<br>- The quality of requirements increases.<br>- Critical design decisions are reflected in the specification of the models and not in the minds of the developers or in the code. | - Models may become over-specified (big tables) due to completeness.<br>- Requirements evolve, hard to obtain a complete set at early stages.<br>- No means to calculate the complexity of components from the specified models.<br>- Currently, not possible to refine a single interface model by multiple design models.<br>- Big models are hard to review, adapt, change, or understand. |
| Formal Verification | |
| - No manual intervention is required for verification and verification is automatic performed with the click of a button.<br>- All formal details are hidden from normal developers, facilitating industrial usages.<br>- FDR2 checks all possible execution scenarios of a component, searching for deadlocks, livelocks and illegal interactions. | - When preparing work breakdown estimations it is hard to estimate the time required for verification.<br>- Model checking may take hours or may even be impossible causing delays to deliverables.<br>- Verification completeness by model checking may cause fixing race conditions which hardly occur in practice. |

| | |
|---|---|
| - Counterexamples are traced back to the ASD models. | - Verification of system-specific and timing properties are not supported. |
| Implementation | |
| - The generated code completely represents what was specified and verified without abstracting or excluding any behavioral details.<br>- No tricks, workarounds, or cleaver solutions are in the generated code.<br>- Code has the same shape and structure, facilitating debugging and navigation through the code.<br>- All code follows the same coding style and standards, enabling systematic translation to other formats. | - Names of methods, variables, parameters and the lines of code are long.<br>- Since the behavior of run-time creation or disposing of ASD components is not formally checked, some related, unforeseen issues may appear. For example, the system may hang since components instances are not disposed due to the presence of active ASD timers (canceling the timer is required first). |
| Integration | |
| - Integrating ASD generated code is often smooth, quick and does not require glue code.<br>- No errors during integrating ASD components.<br>- Integration time and cost reduces significantly. | - If handwritten code or legacy code is not formally checked, integration may introduce some unforeseen errors.<br>- Some errors may appear when combining all ASD components together, especially if components do not take a strict hierarchical shape. |
| Testing | |
| - No white-box testing is required for the generated code (development time is shortened) | - Testing ASD components as a black-box is still required to guarantee correctness.<br>-Testing the data part using traditional testing methods is required. |

**Table 8.2** Summary of the strengths and the weaknesses of the ASD technology

In Chapter 6 we introduced a number of specification and design guidelines to circumvent the state space explosion problem using the mCRL2 toolset. In brief, for each design guideline we introduce two different designs, both are correct in the sense that they maintain the same application intent. But, the first design overlooks the guideline so it subsequently produces a large state space. The second design uses the guideline so the resulting state space is substantially less compared to the first design.

The reason of choosing mCRL2 over ASD is that some of the guidelines may not be easily realized using the current ASD:Suite. For example, the guideline global synchronous communication may require a manual intervention to the underlying CSP code. The same applies to the compositional design and reduction since there is currently no means to compress a set of ASD components and compose the reduced system with other components.

All the guidelines abstract away any implementation details and focus mainly on constructing verifiable components. Therefore, generating high-level code from the specification was not our main concern but we may consider this as a future step.

We felt the need to establish a framework to design verifiable systems based on the experiences gained from industry and academia. We do not claim that the guidelines are complete and would provide solutions to all design cases but, hopefully, they will provide an inspiration to further investigate state space reduction, by academia and industry, from this perspective, which ultimately can help software practitioners to avoid the pitfalls of state space explosion.

In fact, our experience shows that most developers tend to design software components that usually overlook the guidelines, so that their initial models usually suffer from the state space explosion. As a next logical step, they gradually and iteratively modify their designs, migrating to other alternatives where usually our proposed guidelines can be inferred.

However, such a transition is done in a heuristic way and also by a manner of trial and error that consume plenty of the development time. Consequently, this may cause tensions with team and project leaders concerned with meeting their tight deadlines of the incremental planning. Therefore, we believe that considering the guidelines before the actual design of components may reduce the time and the overhead that will be devoted to obtain verifiable components.

In Chapter 7 we applied some of the above mentioned guidelines to the controller of the power distribution unit. As a result we found that the design alternative of the controller that overlooks the guidelines produces over 70 million states while the design that uses the guidelines produces only 608 states. Both designs are correct in the sense that they will provide the external users the expected behavior described in a single external specification. Note that, the original design of the PDU controller provided by the responsible team did not undergo the guidelines especially the first guideline related to pushing versus polling. Indeed, this is in line with our observation that designers often tend to use pushing of information instead of polling.

An important result achieved in Chapter 7 is that we found that the guidelines are effective for designing verifiable components in industrial settings, and hence could provide a suitable framework to design verifiable components of real industrial cases.

## 8.3 Future work

We introduce the future directions for each work reported in the previous chapters. We start by the work accomplished in the BasiX project. As a future work of the PDU controller case introduced in Chapter 3, we consider extending the specification of the controller and the PCs with the emergency and error recovery modes. Since the ASD:Suite does not support verifying properties of systems (at the time of writing this thesis), we may translate the ASD specification to a corresponding mCRL2 model and verify the system properties.

Considering the PCS project, a disadvantage of having many small components is that it is less clear whether together they realize the desired functionality. In future work we would like to investigate whether additional formal techniques can help to check the overall functionality of a set of components. Another relevant direction that will be explored is the use of formal interface models for conformance testing, using model-based testing techniques.

An issue we encountered when developing the FEClient was that the design model was substantially big. In future work we will investigate how a single interface model can be refined by not only one design model but by a number of design models. Furthermore, the activation error caused by the wrong ordering in the responses list is motivating us to find additional means to formally verify a group of design models using model checking. Since the state space may occur in this case, we would like to investigate the possibility to iteratively compress a design model with its used interface on the fly and use the compressed system to verify the design of other components automatically.

Considering the Orchestration project, we introduced a number of steps we followed to design the components and their responsibilities. An interesting future direction is to extend and define concrete steps to be followed, especially for novice developers, which would lead to decompose highly-abstract, component-based verifiable software.

ASD is suitable for control components that include discrete behavior. During the development process accomplished along this work, it was not required to consider performance or real time aspects. In fact there were some performance targets that have been set for certain scenarios (e.g., X-ray settings must be applied within a specified time). In general, ASD was not used to model performance aspects but there were some performance issues in the handwritten part especially when accessing databases. As a future work triggered by the above, we would like to investigate extending the ASD specification and verification to include timing details, so that we can verify real time requirements.

With respect to the application of the ASD technology in the Frontend subsystem, we could not find a systematic means to compute the complexity of components at the models level. Additionally, we found that big models are not necessarily complex models as there were a few smaller models which were higher in complexity. An interesting future direction is to find means to perform static analysis on the models. For example, we could define limits of number of stimuli events in a component (similar to methods per class in the code), a bound for the number of replicated rule cases caused by state variables (similar to the control structure in the code), maximum number of responses in a stimulus (similar to number of statements per method), etc.

As a further future work, we would like to perform an empirical evaluation of various verification techniques used to verify the ASD models and figure out which technique is more successful of detecting the veiled errors and under which circumstances. For example, we would like to compare the following techniques: specification completeness, thorough inspection and review, model checking and verification of properties, statistical model-based testing supported by ASD and random testing by means of the Input Output Conformance Testing technology. The early results in this area show that thorough inspection was far more effective in detecting design issues than verifying formal properties, especially when the models are small, but more work need to be done before drawing any general statements or conclusions.

With respect to the specification guidelines that we introduced in Chapter 6, we would like to investigate the possibility of adding more guidelines to the list. Furthermore, we would like to find a means to relate the designs of each guideline by proofing some equivalence relations. Since applying the guidelines to the PDU controller in Chapter 7 was beneficial, we would like to apply them to other industrial design cases. Hopefully, we could provide a suitable framework to design verifiable components of real industrial cases and to enhance the quality of future industrial software products.

# Bibliography

[1] *Philips Healthcare - C# Coding Standard, Version 2.0*. http://www.tiobe.com/content/paperinfo/gemrcsharpcs.pdf, 2011.

[2] P.A. Abdulla and K. Rustan M. Leino, editors. *Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, volume 6605 of *Lecture Notes in Computer Science*. Springer, 2011.

[3] J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996.

[4] J.-R. Abrial. Formal methods: Theory becoming practice. 13(5):619–628, may 2007. http://www.jucs.org/jucs_13_5/formal_methods_theory_becoming.

[5] S. Acharya, M. Franklin, and S. Zdonik. Balancing push and pull for data broadcast. *SIGMOD Rec.*, 26:183–194, June 1997.

[6] F. Arbab. Reo: a channel-based coordination model for component composition. *Mathematical. Structures in Comp. Sci.*, 14:329–366, June 2004.

[7] F. Badeau and A. Amelot. Using b as a high level programming language in an industrial project: roissy val. In *Proceedings of the 4th international conference on Formal Specification and Development in Z and B*, ZB'05, pages 334–354, Berlin, Heidelberg, 2005. Springer-Verlag.

[8] J.C.M. Baeten and W.P. Weijland. *Process algebra*. Cambridge Tracts in Theoretical Computer Science 18, 1990.

[9] J. BARNES, R. CHAPMAN, R. JOHNSON, J. WIDMAIER, D. COOPER, and B. EVERETT. Engineering the tokeneer enclave protection system. *In Proceedings of the 1st International Symposium on Secure Software Engineering*, 2006.

[10] V.R. Basili and R.W. Selby. Comparing the effectiveness of software testing strategies. *IEEE Trans. Softw. Eng.*, 13:1278–1296, December 1987.

[11] P. Behm, P. Benoit, A. Faivre, and J.-M. Meynadier. Meteor: A successful application of b in a large project. In *Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems-Volume I - Volume I*, FM '99, pages 369–387, London, UK, 1999. Springer-Verlag.

[12] G. Berry and G. Gonthier. The esterel synchronous programming language: design, semantics, implementation. *Sci. Comput. Program.*, 19:87–152, November 1992.

[13] J. Bicarregui, J. Dick, and E. Woods. Quantitative analysis of an application of formal methods. In *Proceedings of the Third International Symposium of Formal Methods Europe on Industrial Benefit and Advances in Formal Methods*, FME '96, pages 60–73, London, UK, 1996. Springer-Verlag.

[14] J. Bicarregui, J. Fitzgerald, P.G. Larsen, and J. Woodcock. Industrial practice in formal methods: A review. In A. Cavalcanti and D. Dams, editors, *FM 2009: Formal Methods. Second World Congress*, volume 5850 of *Lecture Notes in Computer Science*, pages 810–813. Springer-Verlag, 2009.

[15] S. Blom and J.V.D. Pol. State space reduction by proving confluence. In *Proceedings of the 14th International Conference on Computer Aided Verification*, CAV '02, pages 596–609, London, UK, 2002. Springer-Verlag.

[16] G. Booch, J.E. 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.

[17] E. Börger, P. Päppinghaus, and J. Schmid. Report on a practical application of asms in software design. In *Proceedings of the International Workshop on Abstract State Machines, Theory and Applications*, ASM '00, pages 361–366, London, UK, 2000. Springer-Verlag.

[18] A. Bouajjani, J.C Fernandez, S. Graf, C. Rodriguez, and J. Sifakis. Safety for branching time semantics. In *Proceedings of the 18th international colloquium on Automata, languages and programming*, pages 76–92, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[19] G. Broadfoot. Introducing formal methods into industry using cleanroom and csp. *Dedicated Systems Magazine*, 2005.

[20] G.H. Broadfoot. ASD case notes: Costs and benefits of applying formal methods to industrial control software. In *FM 2005: Formal Methods*, volume 3582 of LNCS, pages 548–551. Springer (2005), 2005.

[21] G.H. Broadfoot and P.J. Broadfoot. Academia and industry meet: Some experiences of formal methods in practice. In *APSEC '03: Proceedings of the Tenth Asia-Pacific Software Engineering Conference Software Engineering Conference*, pages 49–58. IEEE Computer Society, 2003.

[22] J.H. Broadfoot and P.J. Hopcroft. An analytical software design system, EP 1749264, 06-2009, 06 2009.

[23] S.D Brookes, C.A.R Hoare, and A.W Roscoe. A theory of communicating sequential processes. *J. ACM*, 31:560–599, June 1984.

[24] ClearSy. Atelier B, 2011. Industrial tool supporting the B method. *http://www.atelierb.eu/en/*.

[25] T. Clement, I. Cottam, P.K.D. Froome, and C. Jones. The development of a commercial "shrink-wrapped application" to safety integrity level 2: The dust-experttm story. In *Proceedings of the 18th International Conference on Computer Computer Safety, Reliability and Security*, SAFECOMP '99, pages 216–225, London, UK, 1999. Springer-Verlag.

[26] I. Crnkovic. *Building Reliable Component-Based Software Systems*. Artech House, Inc., 2002.

[27] CSK Systems Corporation. VDMTools, 2011. Industrial tool supporting VDM++. *http://www.vdmtools.jp/en/*.

[28] M. Cusumano, A. MacCormack, C.F. Kemerer, and B. Crandall. Software development worldwide: The state of the practice. *IEEE Softw.*, 20(6):28–34, November 2003.

[29] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Trans. Program. Lang. Syst.*, 19:253–291, March 1997.

[30] Esterel Technologies. SCADE Suite, 2011. Model based development environment dedicated to critical embedded software. *http://www.esterel-technologies.com/products/scade-suite/*.

[31] J.C Fernandez and L. Mounier. On the fly verification of behavioural equivalences and preorders. In *Proceedings of the 3rd International Workshop on Computer Aided Verification*, CAV '91, pages 181–191, London, UK, 1992. Springer-Verlag.

[32] J. Fitzgerald, P.G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef. *Validated Designs For Object-oriented Systems*. Springer-Verlag TELOS, Santa Clara, CA, USA, 2005.

[33] B. Folmer. Personal communication. 2010.

[34] Formal Systems (Europe) Ltd. *FDR2 model checker*, 2011. `http://www.fsel.com/`.

[35] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[36] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. Cadp 2010: A toolbox for the construction and analysis of distributed processes. In *TACAS*, pages 372–387, 2011.

[37] G.Behrmann, A. David, and K.G. Larsen. A tutorial on uppaal. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004, Bertinoro, Italy, September 13-18, 2004, Revised Lectures*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer, 2004.

[38] J. F. Groote, J. J. A. Keiren, Aad H. J. Mathijssen, S. C. W. Ploeger, F. P. M. Stappers, C. Tankink, Y. S. Usenko, Muck J. Weerdenburg, W. Wesselink, T. A. C. Willemse, and J. van der Wulp. The MCRL2 Toolset. In *Proceedings of the International Workshop on Advanced Software Development Tools and Techniques (WASDeTT, 2008)*, Proc. International Workshop on Advanced Software Development Tools and Techniques, July 2008.

[39] J. F. Groote and M. P. A. Sellink. Confluence for process verification. *Theor. Comput. Sci.*, 170:47–81, December 1996.

[40] Jan Groote, Aad Mathijssen, Michel Reniers, Yaroslav Usenko, and Muck Van Weerdenburg. Analysis of distributed systems with mcrl2. *Process Algebra for Parallel and Distributed Processing*, pages 99–128, 2008.

[41] J.F Groote, T.W.D.M Kouters, and A.A.H Osaiweran. Specification guidelines to avoid the state space explosion problem. CS-Report 10-14, Eindhoven University of Technology, 2010.

[42] J.F Groote, A Osaiweran, and J.H Wesselius. Analyzing the effects of formal methods on the development of industrial control software. In *proceedings of the IEEE ICSM 2011*, Williamsburg, VA, USA, September 25-30, 2011. to appear.

[43] J.F Groote, A.A.H Osaiweran, and J.H Wesselius. Analyzing a controller of a power distribution unit using formal methods. CS-Report 11-14, Eindhoven University of Technology, 2011.

[44] J.F Groote, A.A.H Osaiweran, and J.H Wesselius. Investigating the effects of designing industrial control software using push and poll strategies. CS-Report 11-16, Eindhoven University of Technology, 2011.

[45] J.F. Groote, A.A.H. Osaiweran, and J.H. Wesselius. Experience report on developing the front-end client unit under the control of formal methods. In *Proceedings of the 27th ACM Symposium on Applied Computing*. ACM Press, in print., 2012.

[46] P. A. Hausler. A recent cleanroom success story: The redwing project. In *Seventeenth Annual Software Engineering Workshop*, NASA Goddard Space Flight Center, Greenbelt, MD, December 1992.

[47] G.J. Holzmann. *The SPIN Model Checker : Primer and Reference Manual*. Addison-Wesley Professional, September 2003.

[48] SourceMonitor homepage. *http://www.campwoodsw.com/sourcemonitor.html*.

[49] J. Hooman. *Specification and Compositional Verification of Real-Time Systems*, volume 558 of *Lecture Notes in Computer Science*. Springer, 1991.

[50] 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 (FHIES 2011), Pre-symposium Proceedings*, pages 92–109. UNU-IIST Report 454, McSCert Report 5. http://www.iist.unu.edu/ICTAC/FHIES2011/Files/fhies2011_8_17.pdf.

[51] IBM ClearCase. *http://www-01.ibm.com/software/awdtools/clearcase/, 2011.*

[52] J.M.Carter and J.H.Poore. Sequence-based specification of feedback control systems in Simulink®. In *CASCON '07: Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*, pages 332–345, ACM, New York, NY, USA, 2007.

[53] C. Jones. *Software assessments, benchmarks, and best practices*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[54] R. Kleihorst. Feature design - foundation power distribution subsystem, internal Philips document. 2010.

[55] R. Kleihorst. Power distribution unit - concept specification, internal Philips document, xdy036-080190. 2010.

[56] D. Kozen. Results on the propositional mu-calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.

[57] P.G. Larsen. Ten years of historical development "bootstrapping" VDMTools. *J. UCS*, pages 692–709, 2001.

[58] F. J. Lin, P. M. Chu, and M. T. Liu. Protocol verification using reachability analysis: the state space explosion problem and relief strategies. *SIGCOMM Comput. Commun. Rev.*, 17:126–135, August 1987.

[59] R.C. Linger. Cleanroom process model. *IEEE Software*, 11(2):50–58, 1994.

[60] M. Loos. Feature design - startup/shutdown, internal Philips document, v0.6. 2010.

[61] S. McConnell. *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA, 2004.

[62] S. McConnell. *Software Estimation: Demystifying the Black Art*. Microsoft Press, Redmond, WA, USA, 2006.

[63] mCRL2 toolset homepage. *http://www.mcrl2.org/, 2011.*

[64] B. Meyer. Applying ”design by contract”. *Computer*, 25(10):40–51, October 1992.

[65] H.D. Mills. Stepwise refinement and verification in box-structured systems. *Computer*, 21(6):23–36, 1988.

[66] H.D. Mills. Certifying the correctness of software. In *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, pages 373 – 381, Kauai, HI, 1992.

[67] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag, 1980.

[68] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[69] A. Osaiweran, T. Fransen, J.F. Groote, and B.J. van Rijnsoever. Experience report on designing and developing control components using formal methods. In *Proceedings of the 18th international Symposium of formal methods*, page (In press), Cnam, Paris, France, 27-31 August, 2012. Springer.

[70] A. Osaiweran, M. Schuts, J. Hooman, and J.H. Wesselius. Incorporating formal techniques into industrial practice: an experience report. In *Proceedings of the 9th International Workshop on Formal Engineering Approaches to Software Components and Architectures*, page (In press), Tallinn, Estonia, March 31, 2012. Electronic Proceedings in Theoretical Computer Science.

[71] S. Prowell, C. Trammell, R. Linger, and J. Poore. *Cleanroom Software Engineering: Technology and Process*. Addison-Wesley, 1999.

[72] S. J. Prowell and J. H. Poore. Foundations of sequence-based software specification. *IEEE Transactions on Software Engineering*, 29(5):417–429, 2003.

[73] A.S. Tanenbaum. *Computer networks. Second edition*. Prentice Hall, 1988.

[74] J.F. Groote, A. Osaiweran, and J.H. Wesselius. Experience report on developing the front-end client unit under the control of formal methods. In *Proceedings of the 27th ACM SAC-SE*, page (In press), Riva del Garda, Italy, March 25-29, 2012. ACM.

[75] J.F. Groote, T.W.D.M. Kouters, and Ammar Osaiweran. Specification guidelines to avoid the state space explosion problem. In *FSEN*, pages 112–127, 2011.

[76] R.A. Sprangler and R.C. Linger. The ibm cleanroom software engineering technology transfer program. In *Proceedings of the SEI Conference on Software Engineering Education*, pages 380–394, London, UK, 1992. Springer-Verlag.

[77] R.C. Linger. Cleanroom software engineering for zero-defect software. In *Proceedings of the 15th international conference on Software Engineering*, ICSE ’93, pages 2–13, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[78] A.W Roscoe. *The theory and practice of concurrency*. Prentice Hall, 1998.

[79] A.W. Roscoe. *Understanding Concurrent Systems*. Springer, 2010.

[80] M.T.W. Schuts. Improving software development. *Masters thesis*, Radboud University Nijmegen, The Netherlands, 2010.

[81] TIOBE homepage. *http://www.tiobe.com, 2011.*

[82] R.J van Glabbeek and P.W Weijland. Branching time and abstraction in bisimulation semantics. *J. ACM*, 43:555–600, May 1996.

[83] Verum. *ASD:Suite*, 2011. `http://www.verum.com/`.

[84] C.A. Vissers and L. Logrippo. The importance of the service concept in the design of data communications protocols. In *Proceedings of the IFIP WG6.1 Fifth International Conference on Protocol Specification, Testing and Verification V*, pages 3–17, Amsterdam, The Netherlands, The Netherlands, 1985. North-Holland Publishing Co.

[85] C.A. Vissers, G. Scollo, M.V. Sinderen, and E. Brinksma. Specification styles in distributed systems design and verification. *Theor. Comput. Sci.*, 89:179–206, August 1991.

[86] J. Woodcock, P.G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys*, 41(4):1–36, 2009.

# Summary

### Formal Development of Control Software in the Medical Systems Domain

In this thesis we describe the effectiveness of applying a number of formal techniques to the development of industrial control software at Philips Healthcare. We demonstrate how these techniques were tightly incorporated to the industrial workflow and the issues encountered during the application.

The work was established in an industrial context, dealing with real industrial projects and a real product concerning the development of interventional X-ray systems.

The results are very conclusive in the sense that the used formal techniques could deliver better quality code compared to the code developed in conventional development methods. Also, the results show that the productivity of the formally developed code is better than the productivity of code developed by projects at Philips Healthcare or projects reported worldwide.

The thesis also includes a number of design and specification guidelines that assist constructing verifiable components using model checking. The guidelines were successful in designing and verifying a controller component developed at Philips Healthcare. Hence, the guidelines can provide an effective framework to design verifiable control components in industrial settings.

# Curriculum Vitae

Ammar Osaiweran was born on the $14^{th}$ of March 1980 in Wadi Al-Madam, Taiz, Yemen. In August 2002 he obtained his bachelor degree in computer science from Thamar University, Thamar, Yemen. In 2003 he worked as a student assistant at the same university for three years. He obtained his master degree in computer science and engineering from the Technical University of Eindhoven in October 2008. His master graduation project was completed at Philips Healthcare (previously Philips Medical Systems) under supervision of prof.dr.ir. Jan Friso Groote. In November 2008 he continued his postgraduate study at the Technical University of Eindhoven as a Ph.D student being sponsored by Philips Healthcare and supervised by prof.dr.ir. Jan Friso Groote and prof.dr. Jozef Hooman.

# Index

## Titles in the IPA Dissertation Series since 2006

**E. Dolstra**. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

**R.J. Corin**. *Analysis Models for Security Protocols*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

**P.R.A. Verbaan**. *The Computational Complexity of Evolving Systems*. Faculty of Science, UU. 2006-03

**K.L. Man and R.R.H. Schiffelers**. *Formal Specification and Analysis of Hybrid Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

**M. Kyas**. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Faculty of Mathematics and Natural Sciences, UL. 2006-05

**M. Hendriks**. *Model Checking Timed Automata - Techniques and Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

**J. Ketema**. *Böhm-Like Trees for Rewriting*. Faculty of Sciences, VUA. 2006-07

**C.-B. Breunesse**. *On JML: topics in tool-assisted verification of JML programs*. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

**B. Markvoort**. *Towards Hybrid Molecular Simulations*. Faculty of Biomedical Engineering, TU/e. 2006-09

**S.G.R. Nijssen**. *Mining Structured Data*. Faculty of Mathematics and Natural Sciences, UL. 2006-10

**G. Russello**. *Separation and Adaptation of Concerns in a Shared Data Space*. Faculty of Mathematics and Computer Science, TU/e. 2006-11

**L. Cheung**. *Reconciling Nondeterministic and Probabilistic Choices*. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

**B. Badban**. *Verification techniques for Extensions of Equality Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

**A.J. Mooij**. *Constructive formal methods and protocol standardization*. Faculty of Mathematics and Computer Science, TU/e. 2006-14

**T. Krilavicius**. *Hybrid Techniques for Hybrid Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

**M.E. Warnier**. *Language Based Security for Java and JML*. Faculty of Science, Mathematics and Computer Science, RU. 2006-16

**V. Sundramoorthy**. *At Home In Service Discovery*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

**B. Gebremichael**. *Expressivity of Timed Automata Models*. Faculty of Science, Mathematics and Computer Science, RU. 2006-18

**L.C.M. van Gool**. *Formalising Interface Specifications*. Faculty of Mathematics and Computer Science, TU/e. 2006-19

**C.J.F. Cremers**. *Scyther - Semantics and Verification of Security Protocols*. Faculty of Mathematics and Computer Science, TU/e. 2006-20

**J.V. Guillen Scholten**. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition*. Faculty of Mathematics and Natural Sciences, UL. 2006-21

**H.A. de Jong**. *Flexible Heterogeneous Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

**N.K. Kavaldjiev**. *A run-time reconfigurable Network-on-Chip for streaming DSP applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

**M. van Veelen**. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems*. Faculty of Mathematics and Computing Sciences, RUG. 2007-03

**T.D. Vu**. *Semantics and Applications of Process and Program Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

**L. Brandán Briones**. *Theories for Model-based Testing: Real-time and Coverage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

**I. Loeb**. *Natural Deduction: Sharing by Presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2007-06

**M.W.A. Streppel**. *Multifunctional Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2007-07

**N. Trčka**. *Silent Steps in Transition Systems and Markov Chains*. Faculty of Mathematics and Computer Science, TU/e. 2007-08

**R. Brinkman**. *Searching in encrypted data*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

**A. van Weelden**. *Putting types to good use*. Faculty of Science, Mathematics and Computer Science, RU. 2007-10

**J.A.R. Noppen**. *Imperfect Information in Software Development Processes*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

**R. Boumen**. *Integration and Test plans for Complex Manufacturing Systems*. Faculty of Mechanical Engineering, TU/e. 2007-12

**A.J. Wijs**. *What to do Next?: Analysing and Optimising System Behaviour in Time*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

**C.F.J. Lange**. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML*. Faculty of Mathematics and Computer Science, TU/e. 2007-14

**T. van der Storm**. *Component-based Configuration, Integration and Delivery*. Faculty of Natural Sciences, Mathematics, and Computer Science,UvA. 2007-15

**B.S. Graaf**. *Model-Driven Evolution of*

*Software Architectures*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

**A.H.J. Mathijssen**. *Logical Calculi for Reasoning with Binding*. Faculty of Mathematics and Computer Science, TU/e. 2007-17

**D. Jarnikov**. *QoS framework for Video Streaming in Home Networks*. Faculty of Mathematics and Computer Science, TU/e. 2007-18

**M. A. Abam**. *New Data Structures and Algorithms for Mobile Data*. Faculty of Mathematics and Computer Science, TU/e. 2007-19

**W. Pieters**. *La Volonté Machinale: Understanding the Electronic Voting Controversy*. Faculty of Science, Mathematics and Computer Science, RU. 2008-01

**A.L. de Groot**. *Practical Automaton Proofs in PVS*. Faculty of Science, Mathematics and Computer Science, RU. 2008-02

**M. Bruntink**. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

**A.M. Marin**. *An Integrated System to Manage Crosscutting Concerns in Source Code*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

**N.C.W.M. Braspenning**. *Model-based Integration and Testing of High-tech Multi-disciplinary Systems*. Faculty of Mechanical Engineering, TU/e. 2008-05

**M. Bravenboer**. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. Faculty of Science, UU. 2008-06

**M. Torabi Dashti**. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

**I.S.M. de Jong**. *Integration and Test Strategies for Complex Manufacturing Machines*. Faculty of Mechanical Engineering, TU/e. 2008-08

**I. Hasuo**. *Tracing Anonymity with Coalgebras*. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

**L.G.W.A. Cleophas**. *Tree Algorithms: Two Taxonomies and a Toolkit*. Faculty of Mathematics and Computer Science, TU/e. 2008-10

**I.S. Zapreev**. *Model Checking Markov Chains: Techniques and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

**M. Farshi**. *A Theoretical and Experimental Study of Geometric Networks*. Faculty of Mathematics and Computer Science, TU/e. 2008-12

**G. Gulesir**. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

**F.D. Garcia**. *Formal and Computational Cryptography: Protocols, Hashes and Commitments*. Faculty of Science, Mathematics and Computer Science, RU. 2008-14

**P. E. A. Dürr**. *Resource-based Verification for Robust Composition of Aspects*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

**E.M. Bortnik**. *Formal Methods in Support of SMC Design*. Faculty of Mechanical Engineering, TU/e. 2008-16

**R.H. Mak**. *Design and Performance Analysis of Data-Independent Stream Processing Systems*. Faculty of Mathematics and Computer Science, TU/e. 2008-17

**M. van der Horst**. *Scalable Block Processing Algorithms*. Faculty of Mathematics and Computer Science, TU/e. 2008-18

**C.M. Gray**. *Algorithms for Fat Objects: Decompositions and Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-19

**J.R. Calamé**. *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20

**E. Mumford**. *Drawing Graphs for Cartographic Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-21

**E.H. de Graaf**. *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation*. Faculty of Mathematics and Natural Sciences, UL. 2008-22

**R. Brijder**. *Models of Natural Computation: Gene Assembly and Membrane Systems*. Faculty of Mathematics and Natural Sciences, UL. 2008-23

**A. Koprowski**. *Termination of Rewriting and Its Certification*. Faculty of Mathematics and Computer Science, TU/e. 2008-24

**U. Khadim**. *Process Algebras for Hybrid Systems: Comparison and Development*. Faculty of Mathematics and Computer Science, TU/e. 2008-25

**J. Markovski**. *Real and Stochastic Time in Process Algebras for Performance Evaluation*. Faculty of Mathematics and Computer Science, TU/e. 2008-26

**H. Kastenberg**. *Graph-Based Software Specification and Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27

**I.R. Buhan**. *Cryptographic Keys from Noisy Data Theory and Applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-28

**R.S. Marin-Perianu**. *Wireless Sensor Networks in Motion: Clustering Algorithms for Service Discovery and Provisioning*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-29

**M.H.G. Verhoef**. *Modeling and Validating Distributed Embedded Real-Time Control Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2009-01

**M. de Mol**. *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean*. Faculty of Science, Mathematics and Computer Science, RU. 2009-02

**M. Lormans**. *Managing Requirements Evolution*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

**M.P.W.J. van Osch**. *Automated Model-based Testing of Hybrid Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-04

**H. Sozer**. *Architecting Fault-Tolerant Software Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

**M.J. van Weerdenburg**. *Efficient Rewriting Techniques*. Faculty of Mathematics and Computer Science, TU/e. 2009-06

**H.H. Hansen**. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

**A. Mesbah**. *Analysis and Testing of Ajax-based Single-page Web Applications*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

**A.L. Rodriguez Yakushev**. *Towards Getting Generic Programming Ready for Prime Time*. Faculty of Science, UU. 2009-9

**K.R. Olmos Joffré**. *Strategies for Context Sensitive Program Transformation*. Faculty of Science, UU. 2009-10

**J.A.G.M. van den Berg**. *Reasoning about Java programs in PVS using JML*. Faculty of Science, Mathematics and Computer Science, RU. 2009-11

**M.G. Khatib**. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

**S.G.M. Cornelissen**. *Evaluating Dynamic Analysis Techniques for Program Comprehension*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

**D. Bolzoni**. *Revisiting Anomaly-based Network Intrusion Detection Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

**H.L. Jonker**. *Security Matters: Privacy in Voting and Fairness in Digital Exchange*. Faculty of Mathematics and Computer Science, TU/e. 2009-15

**M.R. Czenko**. *TuLiP - Reshaping Trust Management*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

**T. Chen**. *Clocks, Dice and Processes*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

**C. Kaliszyk**. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web*. Faculty of Science, Mathematics and Computer Science, RU. 2009-18

**R.S.S. O'Connor**. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory*. Faculty of Science, Mathematics and Computer Science, RU. 2009-19

**B. Ploeger**. *Improved Verification Methods for Concurrent Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-20

**T. Han**. *Diagnosis, Synthesis and Analysis of Probabilistic Models*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21

**R. Li**. *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis*. Faculty of Mathematics and Natural Sciences, UL. 2009-22

**J.H.P. Kwisthout**. *The Computational Complexity of Probabilistic Networks*. Faculty of Science, UU. 2009-23

**T.K. Cocx**. *Algorithmic Tools for Data-Oriented Law Enforcement*. Faculty of Mathematics and Natural Sciences, UL. 2009-24

**A.I. Baars**. *Embedded Compilers*. Faculty of Science, UU. 2009-25

**M.A.C. Dekker**. *Flexible Access Control for Dynamic Collaborative Environments*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26

**J.F.J. Laros**. *Metrics and Visualisation for Crime Analysis and Genomics*. Faculty of Mathematics and Natural Sciences, UL. 2009-27

**C.J. Boogerd**. *Focusing Automatic Code Inspections*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01

**M.R. Neuhäußer**. *Model Checking Nondeterministic and Randomly Timed Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02

**J. Endrullis**. *Termination and Productivity*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03

**T. Staijen**. *Graph-Based Specification and Verification for Aspect-Oriented Languages*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04

**Y. Wang**. *Epistemic Modelling and Protocol Dynamics*. Faculty of Science, UvA. 2010-05

**J.K. Berendsen**. *Abstraction, Prices and Probability in Model Checking Timed Automata*. Faculty of Science, Mathematics and Computer Science, RU. 2010-06

**A. Nugroho**. *The Effects of UML Modeling on the Quality of Software*. Faculty of Mathematics and Natural Sciences, UL. 2010-07

**A. Silva**. *Kleene Coalgebra*. Faculty of Science, Mathematics and Computer Science, RU. 2010-08

**J.S. de Bruin**. *Service-Oriented Discovery of Knowledge - Foundations, Implementations and Applications*. Faculty of Mathematics and Natural Sciences, UL. 2010-09

**D. Costa**. *Formal Models for Component Connectors*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10

**M.M. Jaghoori**. *Time at Your Service: Schedulability Analysis of Real-Time and Distributed Services*. Faculty of Mathematics and Natural Sciences, UL. 2010-11

**R. Bakhshi**. *Gossiping Models: Formal Analysis of Epidemic Protocols*. Faculty of Sciences, Department of Computer Science, VUA. 2011-01

**B.J. Arnoldus**. *An Illumination of the Template Enigma: Software Code Generation with Templates*. Faculty of Mathematics and Computer Science, TU/e. 2011-02

**E. Zambon**. *Towards Optimal IT Availability Planning: Methods and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-03

**L. Astefanoaei**. *An Executable Theory of Multi-Agent Systems Refinement*. Faculty of Mathematics and Natural Sciences, UL. 2011-04

**J. Proença**. *Synchronous coordination of distributed components*. Faculty of Mathematics and Natural Sciences, UL. 2011-05

**A. Moralı**. *IT Architecture-Based Confidentiality Risk Assessment in Networks of Organizations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-06

**M. van der Bijl**. *On changing models in Model-Based Testing*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-07

**C. Krause**. *Reconfigurable Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-08

**M.E. Andrés**. *Quantitative Analysis of Information Leakage in Probabilistic and Nondeterministic Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2011-09

**M. Atif**. *Formal Modeling and Verification of Distributed Failure Detectors*. Faculty of Mathematics and Computer Science, TU/e. 2011-10

**P.J.A. van Tilburg**. *From Computability to Executability – A process-theoretic view on automata theory*. Faculty of Mathematics and Computer Science, TU/e. 2011-11

**Z. Protic**. *Configuration management for models: Generic methods for model comparison and model co-evolution*. Faculty of Mathematics and Computer Science, TU/e. 2011-12

**S. Georgievska**. *Probability and Hiding in Concurrent Processes*. Faculty of Mathematics and Computer Science, TU/e. 2011-13

**S. Malakuti**. *Event Composition Model: Achieving Naturalness in Runtime Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-14

**M. Raffelsieper**. *Cell Libraries and Verification*. Faculty of Mathematics and Computer Science, TU/e. 2011-15

**C.P. Tsirogiannis**. *Analysis of Flow and Visibility on Triangulated Terrains*. Faculty of Mathematics and Computer Science, TU/e. 2011-16

**Y.-J. Moon**. *Stochastic Models for Quality of Service of Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-17

**R. Middelkoop**. *Capturing and Exploiting Abstract Views of States in OO Verification*. Faculty of Mathematics and Computer Science, TU/e. 2011-18

**M.F. van Amstel**. *Assessing and Improving the Quality of Model Transformations*. Faculty of Mathematics and Computer Science, TU/e. 2011-19

**A.N. Tamalet**. *Towards Correct Programs in Practice*. Faculty of Science, Mathematics and Computer Science, RU. 2011-20

**H.J.S. Basten**. *Ambiguity Detection for Programming Language Grammars*. Faculty of Science, UvA. 2011-21

**M. Izadi**. *Model Checking of Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-22

**L.C.L. Kats**. *Building Blocks for Language Workbenches*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2011-23

**S. Kemper**. *Modelling and Analysis of Real-Time Coordination Patterns*. Faculty of Mathematics and Natural Sciences, UL. 2011-24

**J. Wang**. *Spiking Neural P Systems*. Faculty of Mathematics and Natural Sciences, UL. 2011-25

**A. Khosravi**. *Optimal Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2012-01

**A. Middelkoop**. *Inference of Program Properties with Attribute Grammars, Revisited*. Faculty of Science, UU. 2012-02

**Z. Hemel**. *Methods and Techniques for the Design and Implementation of Domain-Specific Languages*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-03

**T. Dimkov**. *Alignment of Organizational Security Policies: Theory and Practice*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-04

**S. Sedghi**. *Towards Provably Secure Efficiently Searchable Encryption*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-05

**F. Heidarian Dehkordi**. *Studies on Verification of Wireless Sensor Networks and Abstraction Learning for System Inference*. Faculty of Science, Mathematics and Computer Science, RU. 2012-06

**K. Verbeek**. *Algorithms for Cartographic Visualization*. Faculty of Mathematics and Computer Science, TU/e. 2012-07

**D.E. Nadales Agut**. *A Compositional Interchange Format for Hybrid Systems: Design and Implementation*. Faculty of Mechanical Engineering, TU/e. 2012-08

**H. Rahmani**. *Analysis of Protein-Protein Interaction Networks by Means of Annotated Graph Mining Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2012-09

**S.D. Vermolen**. *Software Language Evolution*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-10

**L.J.P. Engelen**. *From Napkin Sketches to Reliable Software*. Faculty of Mathematics and Computer Science, TU/e. 2012-11

**F.P.M. Stappers**. *Bridging Formal Models – An Engineering Perspective*. Faculty of Mathematics and Computer Science, TU/e. 2012-12

**W. Heijstek**. *Software Architecture Design in Global and Model-Centric Software Development*. Faculty of Mathematics and Natural Sciences, UL. 2012-13

**C. Kop**. *Higher Order Termination*. Faculty of Sciences, Department of Computer Science, VUA. 2012-14

**A. Osaiweran**. *Formal Development of Control Software in the Medical Systems Domain*. Faculty of Mathematics and Computer Science, TU/e. 2012-15