Incorporating Formal Techniques into Industrial Practice: an Experience Report

Ammar Osaiweran[a], Mathijs Schuts[b], Jozef Hooman[c,1] and Jacco Wesselius[b]

[a] *Eindhoven University of Technology, Eindhoven, the Netherlands*

[b] *Philips Healthcare, Best, the Netherlands*

[c] *Embedded Systems Institute Eindhoven & Radboud University Nijmegen, the Netherlands*

Abstract

We report about experiences at Philips Healthcare with component-based development supported by formal techniques. The formal Analytical Software Design (ASD) approach of the company Verum has been incorporated into the industrial workflow. The commercial tool ASD:Suite supports both compositional verification and code generation for control components. For other components test-driven development has been used. We discuss the results of these combined techniques in a project which developed the power control service of an interventional X-ray system.

Keywords: component-based development, formal methods, analytical software design, test-driven development, software quality

1 Introduction

We describe our experiences with the use of a formal method during an industrial component-based development project. Our focus is the embedding of the method in the industrial workflow. As observed in [4,28], there are quite a number of reports about industrial case studies with formal methods, but very few publications describe second or subsequent use. Similarly, the literature about the incorporation of formal methods in the standard industrial development process is very limited.

We present a workflow which combines test-driven development of components with a commercial formal approach and describe experiences with it

[1] Supported by ITEA project Care4Me and COMMIT project Allegio.

at Philips Healthcare. In this introduction, we describe the motivation behind these approaches, the main characteristics of the formal techniques used, related work, and the main research questions.

This work has been carried out at the business unit interventional X-Ray (iXR) of Philips Healthcare, for developing components of a power control service (PCS) of an X-ray machine. The developed components are part of innovative X-ray systems that are used for minimally-invasive surgery where catheters are used to improve, for instance, a patient's blood vessels. This requires only a very small incision and physicians are guided by X-ray images. In this way, often open heart surgery can be avoided.

To support a fast realization of the quickly increasing amount of medical procedures that use this type of image guided surgery, a component-based development approach is introduced. New components are developed according to this paradigm and existing parts are gradually replaced by components with well-defined formal interfaces. The definition of formal interfaces supports parallel, multi-site development and improves the integration with the increasing amount of 3rd party components.

At Philips Healthcare, the component-based development approach is based on a formal approach called Analytical Software Design (ASD). This approach is supported by the commercial tool ASD:Suite of the company Verum [27]. ASD [6,17] enables the application of formal methods into industrial practice by a combination of the Box Structure Development Method [21] and CSP [14]. The ASD approach contains two types of models which are both based on state machines and described by a similar tabular notation: *interface models* and *design models*. At Philips, these models are exploited as follows:

- The *interface* models are used to define the interaction protocol between important system components in a formal way. ASD uses a Sequence-Based Specification Method [23] to obtain complete and consistent specifications. This means that the response to all possible sequences of input stimuli has to be defined. Sequences that cannot happen must be declared illegal explicitly. The tool ASD:Suite translates the sequence-based specifications into CSP. The FDR2 model checker [11] is used to verify a predefined set of properties such as absence of deadlock and livelock.
- Given an interface model of a control component, its internal behaviour can be described by means of a *design* model which typically uses the interface models of other components. By means of ASD:Suite it can be verified formally whether the design model refines the interface model. Very important in our industrial context is that ASD:Suite supports complete code generation from design models to a number of programming languages (C, C++, C#, Java). Hence, design models provide a platform-independent description of internal component behaviour.

ASD:Suite hides the CSP and FDR2 details, which is important to enable industrial usage. The tool applies a fixed set of correctness checks and error traces are visualized by means of sequence diagrams. To enable automated refinement checks, the use of design models is restricted to components with data-independent control decisions. Components that involve data manipulations or algorithms are implemented by other techniques. Hence, it is important that the ASD approach is compositional [16]; the formal verification uses only the interfaces of the used components, without knowing their implementation. A small example illustrating ASD can be found in [15].

The ASD approach has been inspired by the formal Cleanroom software engineering method [19,22] which is based on systematic stepwise refinement from formal specification to implementation. As observed in [5], 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++ [10] and the code generator of the industrial tool VDMTools [8]. Similarly, the B-method [2], which has been used to develop a number of safety-critical systems, is supported by the commercial Atelier B tool [7]. The SCADE Suite [9] provides a formal industry-proven method for 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.

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 [12,13]. 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 formal techniques can 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 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 paper is structured as follows. Section 2 presents the workflow that has been used to combine formal and traditional approaches for developing software components. Section 3 introduces the PCS and its role in the interventional X-ray system. Section 4 describes the application of the presented workflow to the PCS. In Section 5 we discuss the results achieved in this project. Section 6 contains our main observations and current answers to the questions raised above.

2 Integrating formal techniques in industrial workflow

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 1 outlines the flow of activities in a development increment, highlighting the steps to incorporate both the ASD and the test-driven development (TDD) [3] approaches.

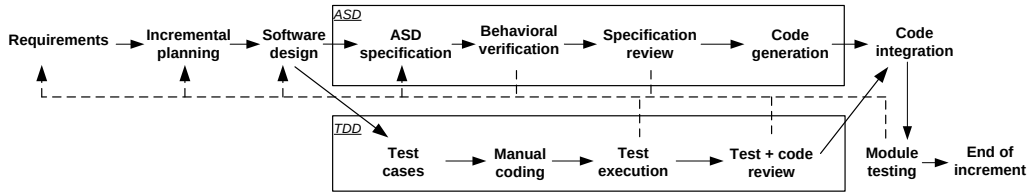

Fig. 1. Steps performed in a development increment

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

For planning and tracking a Work Breakdown Structure (WBS) is created. A WBS consists of tasks that need to be completed in a certain order to obtain a finished product. At the beginning of each increment a new WBS for that

increment is created. For each task, the time needed to complete the task is estimated with the Wideband Delphi estimation method [25]; this means that the effort needed for every task is estimated by two or more experienced software designers in the first phase. In the second phase, the software designers need to get consensus on the estimate. The outcome of the estimate is used in the planning. Not all tasks of the WBS are estimated; some are derived from historical data. Examples are overhead and average time needed to solve a Problem Report (PR).

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

The construction of software components starts with an accepted design, i.e., a decomposition into components with clear interfaces and well-defined responsibilities. Usually such a design is the result of iterative design sessions and approved by all team members. When the aim is to use ASD, a common design practice is to organize components in a hierarchical control structure. Typically, there is a main component on the top which is responsible for high-level, abstract behaviour, e.g., dealing with the main modes and the transitions between these modes. More detailed behaviour is delegated to lower-level components which deal with a particular mode or part of the functionality.

The control components are developed using ASD, whereas TDD is used for the other components. These two approaches are explained below, describing the well-known TDD approach only briefly.

2.1 The Test-Driven Development approach

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

2.2 The Analytical Software Design approach

An overview of the activities in the ASD approach is depicted in Figure 2. Starting point is a structure of the components as described above.

Given a structure of control components, each control component is developed using ASD according to the steps 1 through 6 of Figure 2:

1. Specification of externally visible behaviour. An ASD interface model of the component being developed is created. Such a model specifies not only the

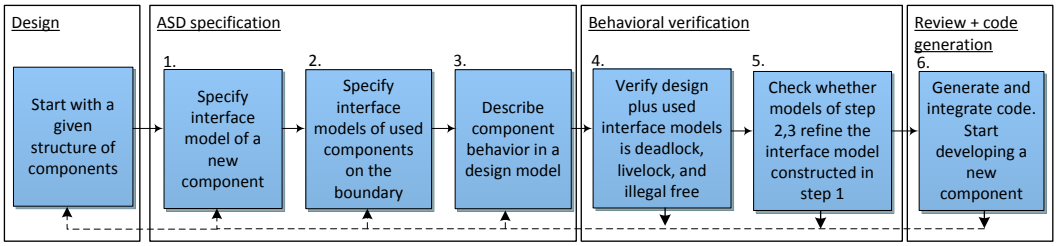

Fig. 2. The ASD approach to develop components

set of functions that can be called by its clients, but also the allowed order of these calls and the allowed responses. It can be seen as a formal specification of the interaction protocol between the component and its clients. 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.

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

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

4. Formal verification of the design model. The ASD:Suite tool systematically translates all ASD models to corresponding CSP processes for verification using the FDR2 model checker. Verification of the design model includes an exhaustive check on the absence of deadlocks, livelocks, and illegal interactions with the used interface models. When an error is detected by FDR2, ASD:Suite presents a nice sequence diagram and allows users to trace the source of the error in the models.

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

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

3 Context of the Power Control Service

The embedded software of an interventional X-ray system is deployed on a cluster of PCs and devices that cooperate with one another to achieve various clinical procedures. The control of power to these components is the responsibility of a central power distribution unit (PDU). Clinical users of an individual PC cannot control the power of the PC without using the PDU, as

depicted in Figure 3. The PDU also controls communication signals related to the startup and shutdown of the PCs.

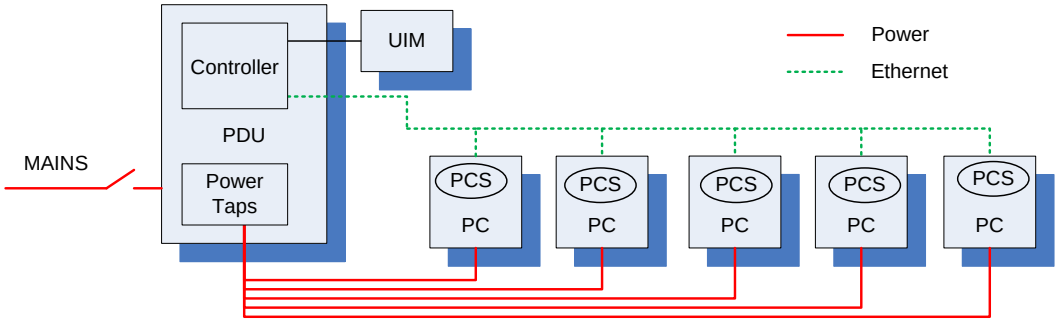

Fig. 3. The PCS in the context of power distribution

As can be seen in Figure 3, 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. The PDU is connected to a User Interface Module (UIM).

Figure 4 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.

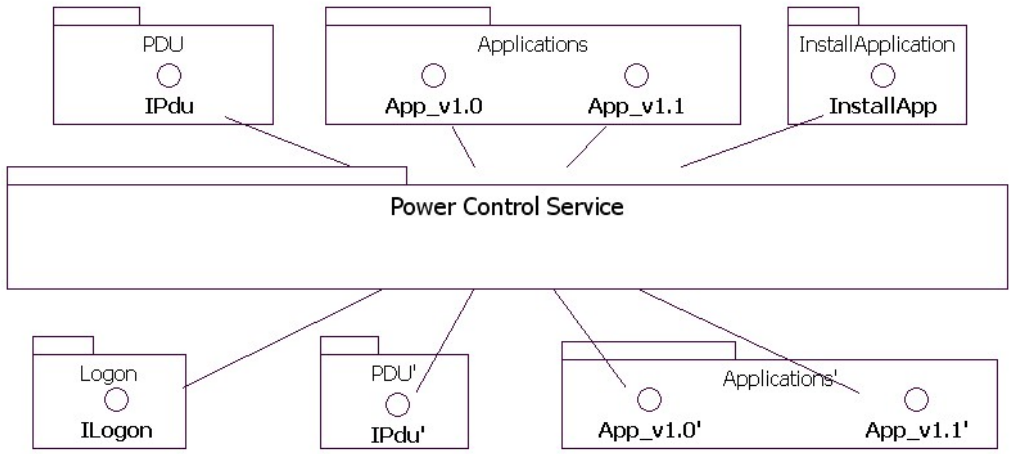

Fig. 4. The PCS as a black-box surrounded by concurrent components

Another example of a concurrent component is the *InstallApplication* which is an external component used to install and upgrade software on the

PC. During the installation of software on a PC, the PCS instructs the running applications to stop, start or restart.

The main function of the PCS is to coordinate all requests to and from these parallel components. Due to the concurrent execution, controlling the flow of events among the components is rather complex, and the architecture sketched in Figure 4 is prone to deadlocks, livelocks, race conditions and illegal interactions. Since the PCS is deployed on every PC, any error is replicated on every PC and potentially leads to serious problems 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 the PCS is in the middle of executing a particular scenario. When the PCS detects that the system is in a faulty state, it should take appropriate actions and log the events for further diagnostics by the field service engineer. As soon as the cause of malfunctions has disappeared, the PCS ensures that all its internal components are synchronized back with other external components to a predefined state.

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

4 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 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 1. Below we describe the development process in more detail, concentrating on the ASD part, since the TDD approach is more conventional.

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

Hence, the development process initially took place in a context where

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

Software design. The design of the PCS consists of a hierarchy of components, as depicted in Figure 5. 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 in the field).

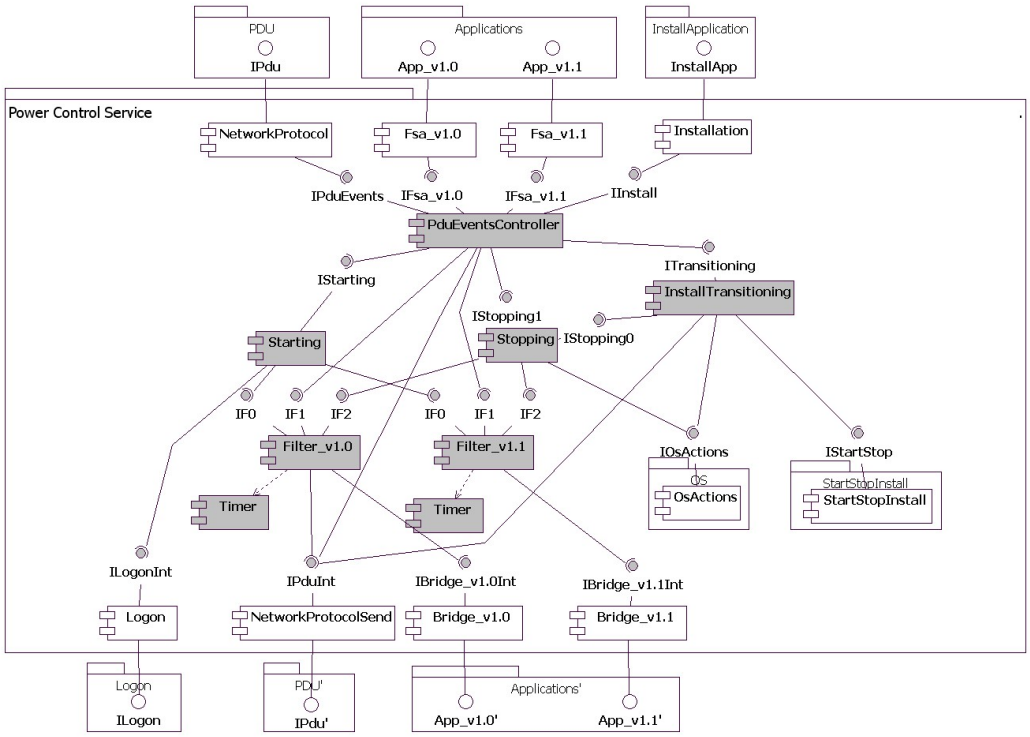

Fig. 5. Components of the PCS

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 behaviour of the instal-

lation 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 [24,13]. 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.

- 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 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 6, using a screenshot of ASD:Suite. The model represents the interface of the *Starting* component and consists of two sub-tables, representing states *Idle* and *Initialized*, each having three rule cases (rows in the table). In order to force developers to be complete, all rule cases must be filled in. That is, in all states the response to all stimuli must be specified. Events that are forbidden in a certain state are declared *Illegal* in the response field.

The corresponding design model of the *Starting* component is depicted in Figure 7. It extends the interface model with calls to its used components *LogOn*, *Filter_v1.0*, and *Filter_v1.1*.

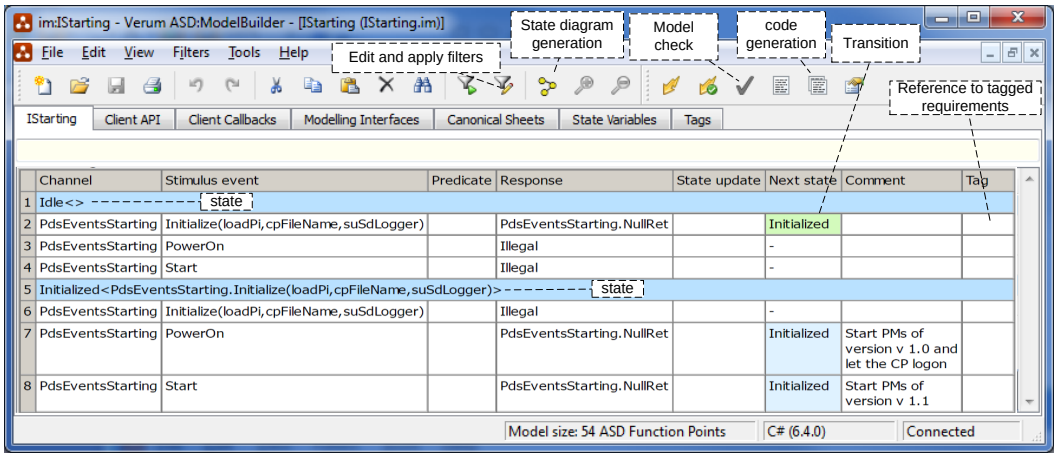

Fig. 6. Interface model of the *Starting* component in ASD:Suite

	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); 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; LogOnComp:ILogOn.LogOn; 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; PdsEventsStarting.NullRet		Initialized	Start PMs of version v 1.1	

Fig. 7. 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, 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 behaviour. For instance, the correct relation between client calls and calls to used components is not checked. Also the value of parameters is not verified. Hence, when all formal checks succeed, the ASD models were reviewed by the project team. The review process performed for the ASD models was similar to the review process of any normal source code developed manually. After the team review, including corrections and a re-check of the formal verification, C# source code was generated automatically using ASD:Suite. This code is then integrated with the manually coded components.

Testing. At the end of each increment the ASD generated code plus the manually coded components were exposed to black-box testing. Corresponding test cases were specified and implemented before and in parallel 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 manually written code and test code were carefully reviewed by team members. As a result of review, minor issues were identified and immediately resolved. Test cases were rerun in order to assure that the rework after review did not break the intended behaviour of the service.

5 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. Both code and ASD models are stored in a code management system, called IBM ClearCase [18].

Philips quality management enforces developers to comply with coding standards provided by the TIOBE technology [26,1]. 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. In our project, these models never consist of more than 300 rule cases and include at most 2 asynchronous stimuli. Hence, the formal verification of interface and design models took less than a second. Small models are also easy to inspect and to maintain. Keeping the components small, however, increases the number of models. Since

the verification is compositional, this does not increase the complexity of the formal verification; each component is verified in isolation with respect to its interfaces.

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 behavioural verification and the firm specification and code reviews provided a suitable framework for increasing the quality, assisting the work, and decreasing potential efforts devoted to bug fixing at later stages of the project.

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 [20]. 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.

6 Concluding Remarks

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

Test and integration. Concerning the code generated by ASD:Suite, 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, but unpredictably 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 behavioural verification thoroughly checked the correctness behaviour of com-

ponents 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 accepts models from previous versions.

Workflow. In the PCS project a lot of time was needed to clarify the requirements, since there were many stakeholders at different sites. We believe that in such a situation the formal ASD interface 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 not more than approximately 3000 lines of code. With these restrictions, the formal technique is rather easy to use without much training and models are easy to understand and to modify.

Future Work. 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.

Acknowledgement

We would like to thank the anonymous reviewers and Tom Fransen for useful comments on the text of this paper.

References

[1] Philips Healthcare - C# Coding Standard, Version 2.0. http://www.tiobe.com/content/paperinfo/gemrcsharpcs.pdf, 2011.

[2] J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996.

[3] K Beck. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[4] 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.

[5] G. Broadfoot. Introducing formal methods into industry using Cleanroom and CSP. *Dedicated Systems Magazine*, 2005.

[6] G.H Broadfoot and P.J Broadfoot. Academia and industry meet: Some experiences of formal methods in practice. In *10th Asia-Pacific Software Engineering Conferenc (APSEC 2003)*, pages 49–58, 2003.

[7] ClearSy. *Atelier B*, 2011. Industrial tool supporting the B method, http://www.atelierb.eu/en/.

[8] CSK Systems Corporation. *VDMTools*, 2011. Industrial tool supporting VDM++, http://www.vdmtools.jp/en/.

[9] Esterel Technologies. *SCADE Suite*, 2011. Model based development environment dedicated to critical embedded software, http://www.esterel-technologies.com/products/scade-suite/.

[10] J. Fitzgerald, P. G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef. *Validated Designs for Object-oriented Systems*. Springer, New York, 2005. Examples are available at http://www.vdmbook.com.

[11] Formal Systems (Europe) Ltd. *FDR2 model checker*, 2011. http://www.fsel.com/.

[12] Jan Friso Groote, Ammar Osaiweran, and Jacco 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.

[13] 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 27th IEEE ICSM 2011*, pages 467–472, Williamsburg, VA, USA, September 25-30, 2011.

[14] C.A.R Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[15] 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.

[16] Jozef Hooman. *Specification and Compositional Verification of Real-Time Systems*, volume 558 of *Lecture Notes in Computer Science*. Springer, 1991.

[17] P.J Hopcroft and G.H Broadfoot. Combining the box structure development method and CSP for software development. *Electronic Notes in Theoretical Computer Science*, 128(6):127–144, 2005.

[18] IBM ClearCase. http://www-01.ibm.com/software/awdtools/clearcase/, 2011.

[19] R.C Linger. Cleanroom process model. *IEEE Software*, 11(2):50–58, 1994.

[20] Steve McConnell. *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA, 2004.

[21] H. D. Mills. Stepwise refinement and verification in box-structured systems. *Computer*, 21:23–36, 1988.

[22] S. Prowell, C. Trammell, R. Linger, and J. Poore. *Cleanroom Software Engineering: Technology and Process*. Addison-Wesley, 1999.

[23] Stacy J. Prowell and Jesse H. Poore. Foundations of sequence-based software specification. *IEEE Transactions on Software Engineering*, 29:417–429, 2003.

[24] M.T.W. Schuts. Improving software development. *Masters thesis*, Radboud University Nijmegen, The Netherlands, 2010.

[25] A. Stellman and J. Greene. *Applied Software Project Management*. O'Reilly Media, 2005.

[26] TIOBE homepage. http://www.tiobe.com, 2011.

[27] Verum homepage. http://www.verum.com, 2011.

[28] J. Woodcock, P.G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys*, 41(4):1–36, 2009.