Incorporating formal techniques into industrial practice

Technische Universiteit Eindhoven
Department of Mathematics and Computer Science

Incorporating Formal Techniques into Industrial Practice

Ammar Osaiweran, Mathijs Schuts, Jozef Hooman

12/14

ISSN 0926-4515

editors: prof.dr. P.M.E. De Bra
prof.dr.ir. J.J. van Wijk

Reports are available at:
http://library.tue.nl/catalog/TUEPublication.csp?Language=dut&Type=ComputerScienceReports&Sort=Author&level=1 and
http://library.tue.nl/catalog/TUEPublication.csp?Language=dut&Type=ComputerScienceReports&Sort=Year&Level=1

Computer Science Reports 12-14
Eindhoven, October 2012

Incorporating Formal Techniques into Industrial Practice

Ammar Osaiweran[a], Mathijs Schuts[b], Jozef Hooman[c⋆]

[a] Eindhoven University of Technology, Eindhoven, The Netherlands
[b] Philips Healthcare, BU Interventional X-ray, Best, The Netherlands
[c] Radboud University Nijmegen, Nijmegen, and Embedded Systems Institute, Eindhoven, The Netherlands
a.a.h.osaiweran@tue.nl,mathijs.schuts@philips.com,jozef.hooman@esi.nl

Abstract. We report about experiences with component-based development supported by formal techniques at Philips Healthcare. 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.

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 [5, 30], 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 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 the X-ray machine depicted in Figure 1. 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.

⋆ Supported by ITEA project Care4Me and COMMIT project Allegio.

Fig. 1. Interventional X-Ray system

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 [29]. ASD [7, 19] enables the application of formal methods into industrial practice by a combination of the Box Structure Development Method [23] and CSP [16].

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 [14, 15]. 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 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 extends [1] with relevant details. It provides more explanation on the compositional construction and verification approach of ASD and shows by an example how components are built in isolation, considering only interfaces of boundary components. We also explain the formal checks that can be carried out by ASD:Suite. Additionally, more detailed information is given about the modeling and verification of the PCS at Philips Healthcare and the issues encountered. In particular, we describe two typical errors that were not discovered by the formal checks. Although these errors escaped both specification review and formal verification using model checking, they were easy to detect and to fix. Another extension concerns information about the evolution of the code of the PCS and the effort spent for developing its components.

This paper is structured as follows. Section 2 describes other work which is related to the ASD approach. Section 3 introduces the ASD approach as far as needed to understand the remainder of this paper. Section 4 presents the workflow that has been used to combine formal and traditional approaches for developing software components. Section 5 introduces the PCS and its role in the interventional X-ray system. Section 6 describes the application of the presented workflow to the PCS. In Section 7 we present two errors which were found after completing the formal verification using ASD:Suite. In Section 8 we discuss the results achieved in this project. Section 9 contains our main observations and current answers to the questions raised above.

2 Related work

The ASD approach has been inspired by the formal Cleanroom software engineering method [21, 24] which is based on systematic stepwise refinement from formal specification to implementation. As observed in [6], 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++ [11] and the code generator of the industrial tool VDMTools [9]. 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 [8]. The SCADE Suite [10] 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.

3 Fundamentals of Analytical Software Design

ASD is a component-based, model-driven approach that combines formal mathematical methods with industrial software development methods. The approach is supported by the commercial tool ASD:Suite of the company Verum. The tool supports two types of models which are both based on state machines and described by a similar tabular notation: *interface models* and *design models*. 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. An interface model describes not only signatures of methods to be invoked by other components but also the external behavior exposed to client components. Internal interactions with lower-level components are not present in this model.
- The *design* model describes the internal behavior of a component given its interface model and typically uses the interface models of other components. By means of the ASD:Suite it can be verified formally whether the design model refines the interface model. Very important in our industrial context is that ASD:Suite supports complete code generation from design models to a number of programming languages (C, C++, C#, Java). Hence, design models provide a platform-independent description of internal component behaviour.

ASD uses a Sequence-Based Specification Method [25] 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 [12] is used to verify a predefined fixed set of properties such as refinement and absence of deadlock and livelock. Error traces are visualized by means of sequence diagrams.

ASD:Suite hides the CSP and FDR2 details, which is important to enable industrial usage. To enable automated refinement checks, the use of design models is restricted to components with data-independent control decisions. Components that involve data manipulations or algorithms are implemented by other techniques. Hence, it is important that the ASD approach is compositional [18]; the formal verification uses only the interfaces of the used components, without knowing their implementation.

To illustrate the above concepts we introduce a small example, focusing more on the specification and verification of ASD models. Figure 2 depicts an example

system that includes a controller component (Ctr) and a sensor device. The sensor is assumed to monitor the status of a door of the X-ray examination room. When the sensor detects that the door is open, it notifies the top controller which, in turn, notifies its clients. As a result, these clients might stop the generation of X-ray and display user messages.

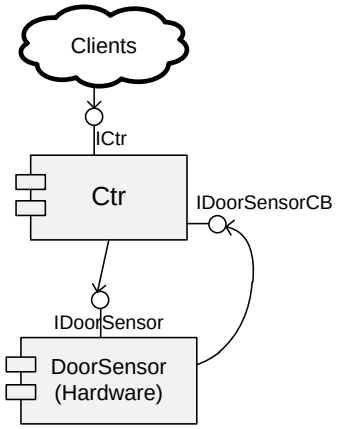

Fig. 2. An example of a controller and a sensor device

3.1 Specification of ASD models

The structure of the ASD models related to the example of Figure 2 is depicted in Figure 3.

In general, each component has an ASD interface model which captures the external behavior related to clients. In our example, a small ASD interface model related to the *Ctr* component is shown in Figure 4, using a screenshot of ASD:Suite version 6.2.0.

The specification is straightforward and consists of four sub-tables, representing states *Created*, *Initializing*, *Initialized* and *initializeFailed*, each having four rule cases (rows in the table). A rule case includes a number of items such as a communication channel (interface), a stimulus event (a method) supplied by optional data parameters, predicates (conditions on the stimulus), response events, transition to a next state, comments and tags to informal requirements.

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* while events that may not happen at a state are declared *Blocked*. The *Null* response is assigned for ignoring stimulus events.

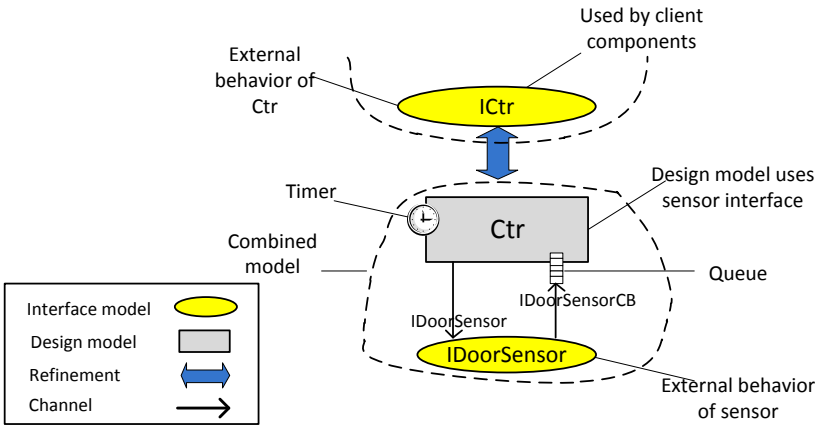

Fig. 3. Structure of ASD models

The specification describes the behavior with respect to clients. In the *Created* state client components can initialize *Ctr* by invoking the *initialize* stimulus through the *ICtr* channel. The *ICtr.NullRet* response indicates the completion of the request after which the *Ctr* transits to the *Initializing* state. In the *Created* state invoking the *unInitialized* stimulus is not allowed.

Internal interactions not visible to clients are specified by means of modeling events. Moreover, ASD components can notify clients using the callback mechanism. For specification readability we usually add the postfix 'INT' to the channel name of internal modeling events and 'CB' to channels of client callback events. For example, rule case 15 specifies that when the sensor internally becomes active, the *Ctr* sends the *stopXray* callback event via the *ICtrCB* callback channel to clients.

The corresponding design model of the *Ctr* component is depicted in Figure 5. It extends the interface model and includes further interactions with the used sensor component. In general, each component has a queue where callback events from its used components are stored. Rule cases dealing with callback events have priority over client calls. In the example, the queue of the *Ctr* component will contain callback events of the used sensor component.

Similar to the interface specification, in rule case 2, a client can initialize the *Ctr*; but in the design model this leads to an initialization of the sensor via the *IDoorSensor* interface and the start of an ASD timer for 3 seconds. Next, the controller transits to the *Initializing* state where it expects a callback event from the sensor or a timeout event from the timer. In both cases, the *Ctr* component informs its clients by means of corresponding callback events. It deactivates the timer if a sensor event is received before the timer expires.

	Channel	Stimulus event	Predicate	Response	State update	Next state	Comment	Tag
1	Created<>							
2	ICtr	initialize		ICtr.NullRet		Initializing		
3	ICtr	unInitialize		Illegal		-		
4	ICtrINT	sensorActive		Blocked		+		
5	ICtrINT	initResutl		Blocked		+		
6	Initializing<ICtr.initialize>							
7	ICtr	initialize		Illegal		-		
8	ICtr	unInitialize		Illegal		-		
9	ICtrINT	sensorActive		Blocked		+		
10	ICtrINT	initResutl		ICtrCB.initOK		Initialized		
11	ICtrINT	initResutl		ICtrCB.initFail		InitializeFailed		
12	Initialized<ICtr.initialize,ICtrINT.initResutl>							
13	ICtr	initialize		Illegal		-		
14	ICtr	unInitialize		ICtr.NullRet		Created		
15	ICtrINT	sensorActive		ICtrCB.stopXray		Initialized		
16	ICtrINT	initResutl		Blocked		+		
17	InitializeFailed<ICtr.initialize,ICtrINT.initResutl>							
18	ICtr	initialize		Illegal		-		
19	ICtr	unInitialize		ICtr.NullRet		Created		
20	ICtrINT	sensorActive		Blocked		+		
21	ICtrINT	initResutl		Blocked		+		

Fig. 4. Interface model of the Ctr component

3.2 Verification of ASD models

ASD components are verified in isolation, using the interfaces of boundary components. There is a fixed set of properties that can be verified. Figure 6 depicts a screenshot of the standard formal checks performed for verifying the models of the *Ctr* component:

- The first two properties verify whether the *ICtr* interface model is livelock and deadlock free.
- The third and the fourth property specify whether the sensor and the timer interface models are livelock free.
- The fifth property expresses that the design model of the *Ctr* component should be deterministic.
- The sixth property specifies the absence of illegal scenarios and queue overflow cases in the design of the component. This check is based on the parallel composition of the design model and the interface models of the used

	Channel	Stimulus event	Predicate	Response	State update	Next state	Comment	Tag
1	Created<>							
2	ICtr	initialize		ICtr.NullRet; DoorSensor:IDoorSensor.initialize; Timer:ITimer.CreateTimer(3)		Initializing		
3	ICtr	unInitialize		Illegal		-		
4	DoorSensor:IDoorSensorCB	initialized		Null		Created		
5	DoorSensor:IDoorSensorCB	initializeFail		Null		Created		
6	DoorSensor:IDoorSensorCB	doorOpen		Null		Created		
7	Timer:ITimerCB	Timeout		Illegal		-		
8	Initializing<ICtr.initialize>							
9	ICtr	initialize		Illegal		-		
10	ICtr	unInitialize		Illegal		-		
11	DoorSensor:IDoorSensorCB	initialized		Timer:ITimer.CancelTimer; ICtrCB.initOK		Initialized		
12	DoorSensor:IDoorSensorCB	initializeFail		Timer:ITimer.CancelTimer; ICtrCB.initFail		InitializeFailed		
13	DoorSensor:IDoorSensorCB	doorOpen		Illegal		-		
14	Timer:ITimerCB	Timeout		ICtrCB.initFail		InitializeFailed		
15	Initialized<ICtr.initialize, DoorSensor:IDoorSensorCB.initialized>							
16	ICtr	initialize		Illegal		-		
17	ICtr	unInitialize		ICtr.NullRet; DoorSensor:IDoorSensor.unInitialize		Created		
18	DoorSensor:IDoorSensorCB	initialized		Illegal		-		
19	DoorSensor:IDoorSensorCB	initializeFail		Illegal		-		
20	DoorSensor:IDoorSensorCB	doorOpen		ICtrCB.stopXray		Initialized		
21	Timer:ITimerCB	Timeout		Illegal		-		
22	InitializeFailed<ICtr.initialize, DoorSensor:IDoorSensorCB.initializeFail>							
23	ICtr	initialize		Illegal		-		
24	ICtr	unInitialize		DoorSensor:IDoorSensor.unInitialize; ICtr.NullRet		Created		
25	DoorSensor:IDoorSensorCB	initialized		Null		InitializeFailed		
26	DoorSensor:IDoorSensorCB	initializeFail		Null		InitializeFailed		
27	DoorSensor:IDoorSensorCB	doorOpen		Null		InitializeFailed		
28	Timer:ITimerCB	Timeout		Illegal		-		

Fig. 5. Design model of the Ctr component

components (this combined model is constructed in the background by the ASD:Suite).

- The seventh property expresses that the combined model mentioned in the previous point is deadlock free. Together with the previous properties it guarantees that the design model of the component uses the interfaces of the used components in a correct way.
- The last two properties are used to check conformance of the combined model (asd_Implementation) with respect to the corresponding *ICtr* interface model (asd_Specification), under both the Failure and Failure-Divergence refinement models as supported by the FDR2 model checker [13].

Observe that when the last two checks succeed, the interface model *ICtr* is a correct representation the design of the *Ctr* component combined with the used components timer and sensor. Hence, clients of the *Ctr* component can be

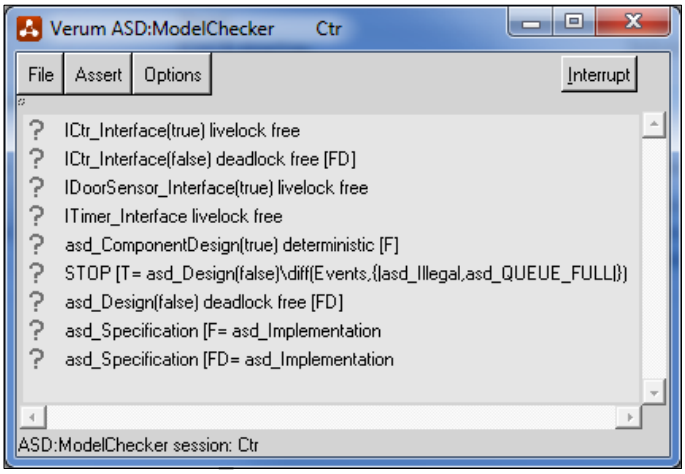

Fig. 6. Formal ASD checks

verified using only the interface model *ICtr*. In general, this mechanism often mitigates the state space explosion problem, since the interface model is usually much simpler than the combination of design model and used interfaces.

4 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 7 outlines the flow of activities in a development increment, highlighting the steps to incorporate both the ASD and the test-driven development (TDD) [4] approaches.

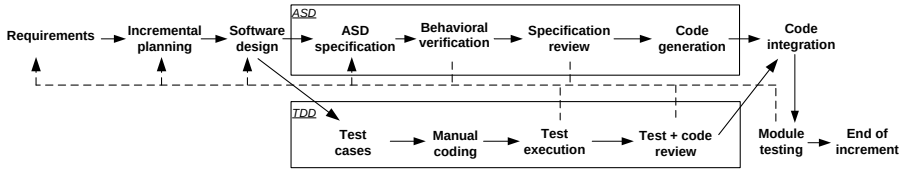

Fig. 7. Steps performed in a development increment

Each increment starts with identifying a list of requirements to be implemented by team members. As soon as requirements are approved by lead archi-

tects, 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 [27]; this means that the effort needed for every task is estimated by two or more experienced software designers in the first phase. In the second phase, software designers need to get consensus on the estimate. The outcome of the estimate is then used in the planning. Not all tasks of the WBS are estimated; some are derived from historical data. Examples are overhead and average time needed to solve a defect.

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

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

4.1 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.

4.2 The Analytical Software Design approach

An overview of the activities in the ASD approach is depicted in Figure 8. Starting point is a structure of the components as described above.

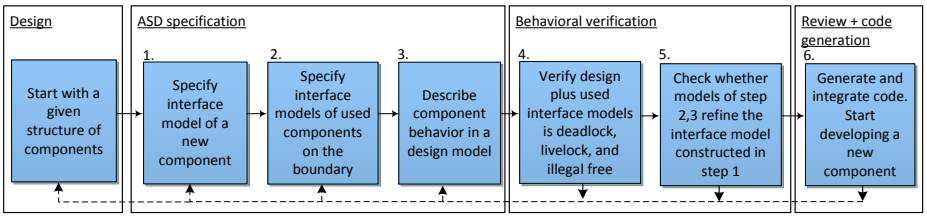

Fig. 8. The ASD approach to develop components

ASD components can be developed in a top-down, bottom-up or middle-out fashion. Each component is developed using ASD according to the steps 1 through 6 of Figure 8:

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

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

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

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

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

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

5 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. 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 9. The

PDU also controls communication signals related to the startup and shutdown of the PCs.

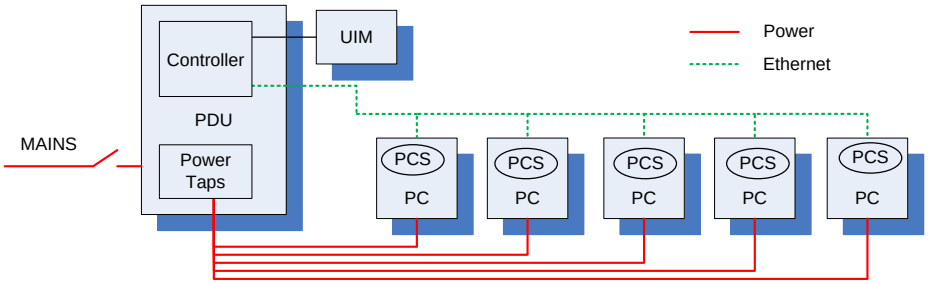

Fig. 9. The PCS in the context of power distribution

As can be seen in Figure 9, 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 10 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.

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 10 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.

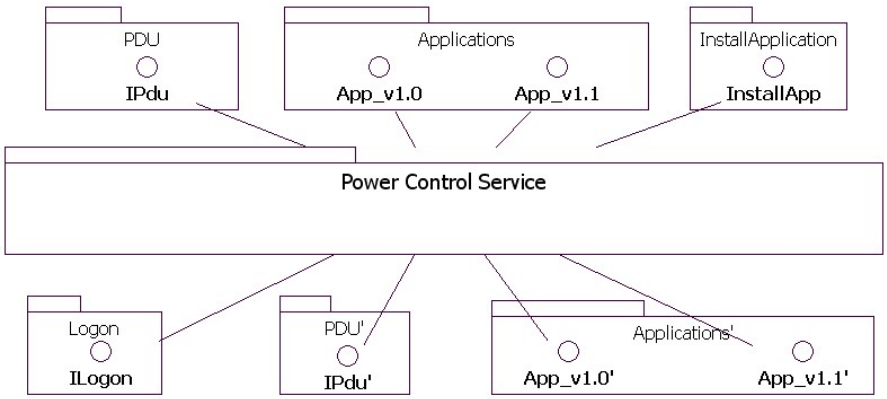

Fig. 10. The PCS as a black-box surrounded by concurrent components

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.

6 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 7. 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 11. 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).

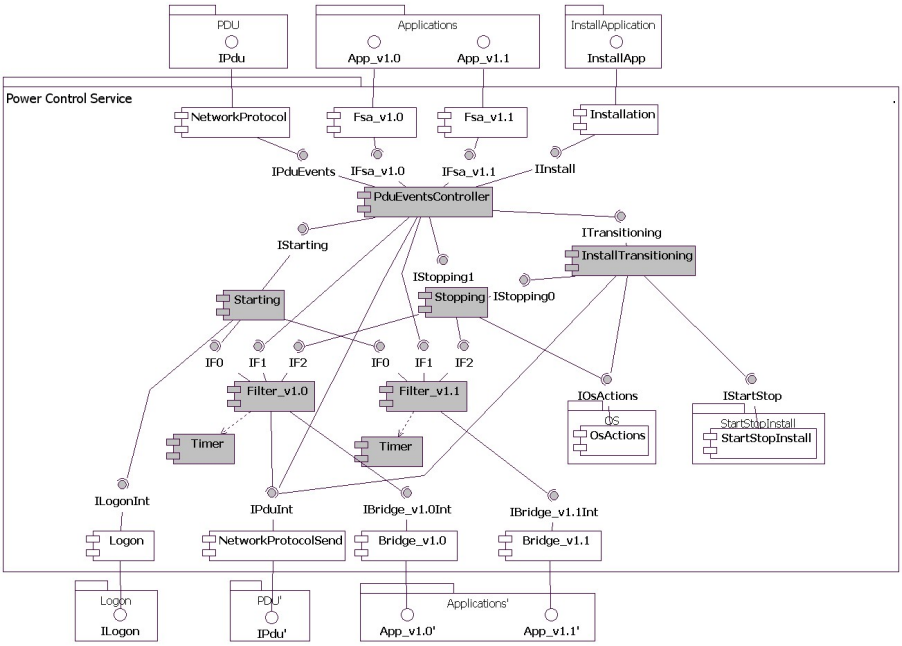

Fig. 11. Components of the PCS

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

- The *PduEventController* component mainly serves commands issued by the external components: the PDU and the InstallApplication, for instance. It contains a top-level state machine that captures overall global states (or modes) of a PC: normal mode, installing, starting/stopping applications, operational, ..etc.
- The *InstallTransitioning* component implements the detailed behavior of the installation mode of the top-level state machine. The component is responsible of safeguarding the detailed transitions from normal mode to installation mode, and vice versa.
- The *Starting* component launches the clinical applications of a PC and logs-on/off the default clinical user. It ensures that clinical applications are successfully started.
- The *Stopping* component is responsible for ensuring that closing the running applications and then shutting down or rebooting the OS is done sequentially.
- The *Filter* components are responsible for starting, restarting, and stopping the applications within a predefined fixed time. They are the facade to the components of located outside the boundary of the PCS.

Experience shows that most novice ASD users tend to design rather large components leading to large ASD models [26, 15]. 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 some visible deviations between hours estimated in the WBS's and actual hours spent for development.

Therefore, the design of the PCS has been decomposed into rather small components, described using small models. Although the ASD approach shown in Figure 8 does not prescribe an order in which the components are realized, we used a top-down, step-wise refinement approach. This effectively helped us distributing responsibilities and maintaining a proper degree of abstraction among all components. In this way we obtained a set of formally verifiable components.

Specification and formal verification of ASD models. The ASD models were specified using the ASD:Suite version 6.2.0, following the ASD recipe. Each component was modeled in isolation with interfaces of boundary components. An example structure of ASD models related to the *Stopping* component is depicted in Figure 12.

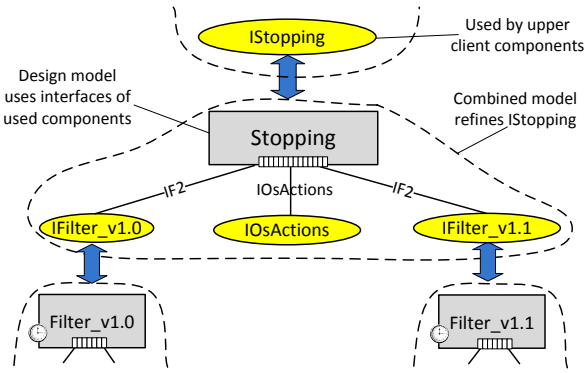

Fig. 12. Structure of ASD models of the *Stopping* component

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

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

The ASD formal properties introduced in Section 3.2 were performed step by step for the models of each component. We first started checking correctness of interface models. When this check succeeded, we searched for illegal scenarios and then for deadlocks in the design model. After that we checked determinism and finally refinement of designs against the interfaces.

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

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

Specification review, code generation and integration. Although the formal verification is very useful to detect errors, it does not guarantee that the design model realizes the intended behaviour. For instance, the correct relation

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

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

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

7 Errors which were not detected by the ASD verification

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

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

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

The ordering error. This error was caused by the impossibility to specify and verify properties about the order of messages of two components in ASD. In our case study, this concerns the *Stopping* and the *Filter* components. Considering Figure 11, the *Stopping* component can receive a request to shutdown the PC from the *PduEventsController* component. The *Stopping* component first instructs the *Filter* component to stop the running applications and then waits for the result before it instructs the *OsActions* component to shutdown the OS.

As specified in rule case 19 in Figure 13 of the *Filter* design model, the *Filter* component starts its timer, instructs the clinical applications to stop, and transits to the *Stopping* state waiting some seconds for a notification from the applications indicating the completion of the stop request. Meanwhile, if the timer expires while waiting for the notification, the *Filter* notifies the *Stopping*

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

Fig. 13. Design model of the *Filter* component

component using the *Stopped* callback and then logs a "FinishedStoppingAfterTimeOut" message; see rule case 28 in Figure 13.

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

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

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

Filter component logged the "FinishedStoppingAfterTimeOut" message; see the sequence diagram of Figure 14.

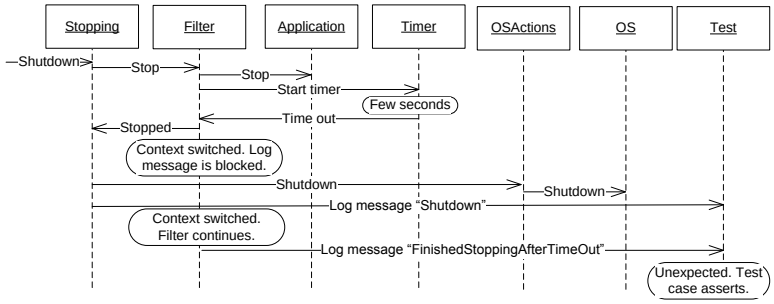

Fig. 14. Error caused by concurrent execution of events due to wrong ordering

This error was easy to find by testing, but it was hard to reproduce due to its concurrent nature. The scenario was not detected by the model checker due to the compositional verification. That is, verification of the *Filter* design model did not include the design of the *Stopping* design model.

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

The multi-client error. Although the model checker of ASD:Suite verified the absence of illegal events, testing showed an illegal event during the execution of the PCS. Figure 15 depicts the structure of the three components involved in the error: the *PduEventController*, the *InstallTransitioning* and the *Stopping* components. The *Stopping* component was initially in the *Created* state, waiting to be initialized by its client components. Upon receiving the *initialize* call, it initializes other lower-level components and then transits to the *Initialized* state, where any other *initialize* call is illegal. However, the *Stopping* component received the first initialize call from the *PduEventController* component, and then the second call from the *InstallTransitioning* component, causing the illegal error in the *Initialized* state.

The reason of not detecting this error using model checking when verifying the *PduEventController* component is that the interface model of the *Install-Transitioning* component exposes only the interaction with the client *PduEvent-Controller* component, excluding any interaction with the *Stopping* component; see Figure 15. More precisely, the *initialize* call from *InstallTransitioning* to the *Stopping* component is excluded from the specification and formal verification, causing a hidden dependency between the *InstallTransitioning* and *Stopping* components not visible to the *PduEventController*.

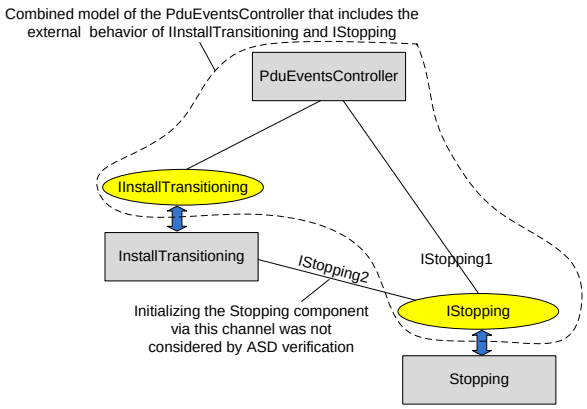

Fig. 15. Model checker could not detect the error due to a hidden dependency

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

8 Results of developing the PCS

Figure 16 depicts code evolution of the manually coded components, after mining the code repository using TIOBE software [28]. The figure shows only the effective lines of code (ELOC), i.e., all blank and comment lines are excluded from calculations. The code was officially placed in the repository at the start of May 2011, with approximately 1,600 ELOC of previously coded components. As can be seen from the figure, the construction of the manually coded components was smooth and gradually evolved throughout all increments. The figure also indicates that there were no major redesign activities caused any removal of the implemented code in any increment.

Similarly, Figure 17 depicts the evolution of test code. The reason of having more testing code than product code at the early stages is that the manually coded components were developed under the control of the TDD technique. As mentioned earlier, the TDD approach implies that test cases have to be written first, before the product code.

Figure 18 sketches the evolution of the ASD code, highlighting 5 versions from 5 stable baselines at the end of each increment, taken from a code management system, called IBM ClearCase [20]. We extracted such figures manually since the ASD code did not comply to the coding standard enforced by the TIOBE technology and hence was excluded from calculation by the technology since the early phase of the development process. As can be seen from the figure, the PCS appeared to already be stable since the start of increment 3. In previous

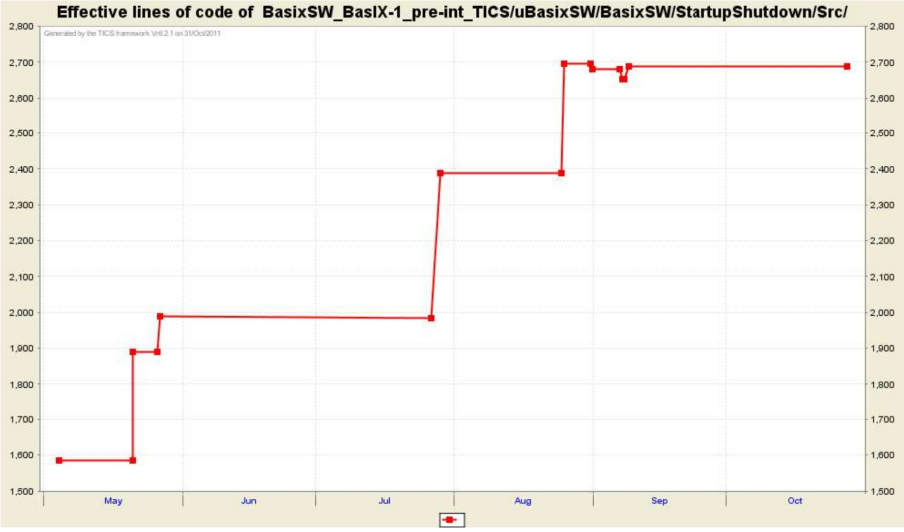

Fig. 16. Evolution of the manually coded components

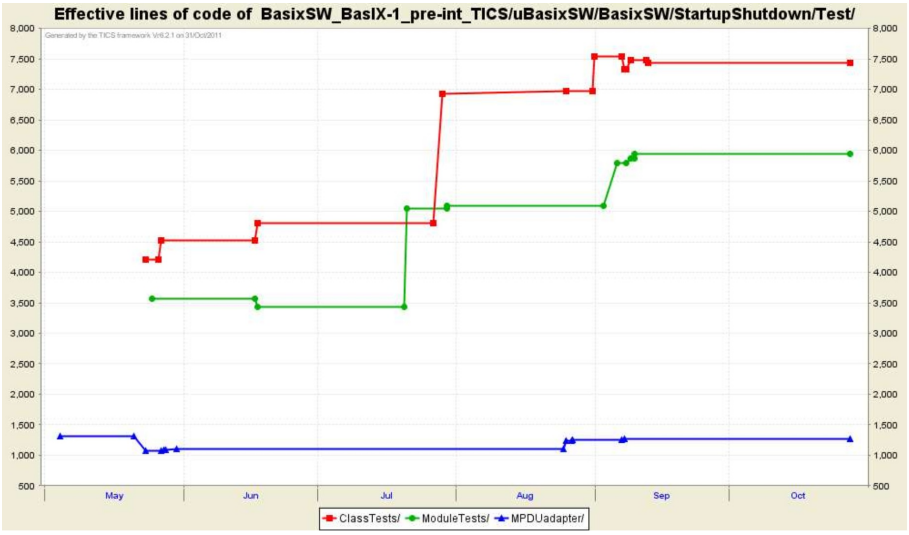

Fig. 17. Evolution of test code

projects where ASD was used [15, 14], major redesigns were needed due to the state space explosion problem. This did not happen in the PCS project since all ASD components are kept small and fit within the limits of the model checker.

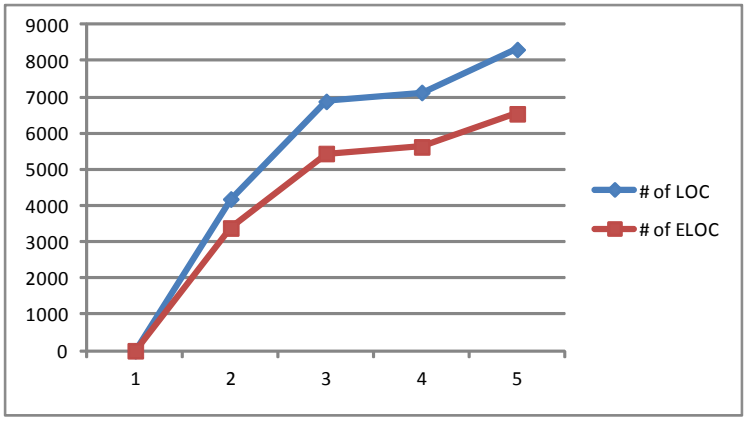

Fig. 18. Evolution of ASD code

In Table 1 we provide statistical data of the final developed ASD components after increment 5, listing all corresponding interface and design models. The first and second column include all ASD interface and design models (IM and DM respectively). The third column shows the number of rule cases of each model. These rule case have been reviewed thoroughly by team members. The fourth and fifth column reveal the states and transitions reported from the model checker FDR to check deadlock freedom (which holds for all models). For the other checks we obtained similar numbers.

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

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

Table 2 depicts metrics related to all developed code. It includes the sum of all total and executable lines of code written for the product and test code.

The entire service includes 17,226 ASD generated and manually written code. It includes a total of 30,264 LOC of test code. The end quality result of the PCS service is remarkable, and the entire service exhibited only 0.17 defect per KLOC. This level of quality is much better than the industry standard defect rate of 1-25 defects per KLOC [22].

Table 3 depicts the hours spent during each increment. The total hours spent for developing the entire service is 1789, with average productivity of 1 effective line of code per hour.

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

Table 1. The ASD models of the power control service

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

Table 2. Statistical data of the power control service

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

Table 3. Hours spent on the power control service

The PCS service was deployed on all PCs, and further tested by independent teams who are responsible of developing the clinical applications on each PC. The result of testing was that no errors were found and the service appeared to function correctly on every PC, from the first run.

Feedbacks received from the independent test teams were very positive, and the service seems to be stable and reliable. Team members of the PCS appreciated the quality of the service, and decided to further incorporate the ASD technology to the development of other parts of the system. The 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.

9 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 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 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.

References

1. A.A.H. Osaiweran and M.T.W. Schuts and J.J.M. 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 (FESCA'12, Tallinn, Estonia, March 31)*, pages in press, 2012.
2. Philips Healthcare - C# Coding Standard, Version 2.0. `http://www.tiobe.com/content/paperinfo/gemrcsharpcs.pdf`, 2011.
3. J.-R. Abrial. *The B-book: assigning programs to meanings.* Cambridge University Press, New York, NY, USA, 1996.
4. K. Beck. *Test Driven Development: By Example.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
5. 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.
6. G. Broadfoot. Introducing formal methods into industry using Cleanroom and CSP. *Dedicated Systems Magazine*, 2005.
7. 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.
8. ClearSy. *Atelier B*, 2011. Industrial tool supporting the B method, `http://www.atelierb.eu/en/`.
9. CSK Systems Corporation. *VDMTools*, 2011. Industrial tool supporting VDM++, `http://www.vdmtools.jp/en/`.
10. Esterel Technologies. *SCADE Suite*, 2011. Model based development environment dedicated to critical embedded software, `http://www.esterel-technologies.com/products/scade-suite/`.
11. 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`.
12. Formal Systems (Europe) Ltd. *FDR2 model checker*, 2011. `http://www.fsel.com/`.
13. Formal Systems (Europe) Ltd and Oxford University Computing Laboratory: Failures-Divergence Refinement – FDR2 User Manual, 9th edn. (2010)
14. 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.
15. 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.
16. C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.
17. 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`.
18. J. Hooman. *Specification and Compositional Verification of Real-Time Systems*, volume 558 of *Lecture Notes in Computer Science.* Springer, 1991.

19. 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.
20. IBM ClearCase. `http://www-01.ibm.com/software/awdtools/clearcase/`, 2011.
21. R.C. Linger. Cleanroom process model. *IEEE Software*, 11(2):50–58, 1994.
22. S. McConnell. *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA, 2004.
23. H. D. Mills. Stepwise refinement and verification in box-structured systems. *Computer*, 21:23–36, 1988.
24. S. Prowell, C. Trammell, R. Linger, and J. Poore. *Cleanroom Software Engineering: Technology and Process*. Addison-Wesley, 1999.
25. S.J. Prowell and J.H. Poore. Foundations of sequence-based software specification. *IEEE Transactions on Software Engineering*, 29:417–429, 2003.
26. M.T.W. Schuts. Improving software development. *Masters thesis*, Radboud University Nijmegen, The Netherlands, 2010.
27. A. Stellman and J. Greene. *Applied Software Project Management*. O'Reilly Media, 2005.
28. TIOBE homepage. `http://www.tiobe.com`, 2011.
29. Verum homepage. `http://www.verum.com`, 2011.
30. J. Woodcock, P.G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys*, 41(4):1–36, 2009.

Science Reports **Department of Mathematics and Computer Science**
Technische Universiteit Eindhoven

If you want to receive reports, send an email to: wsinsan@tue.nl (we cannot guarantee the availability of the requested reports).

In this series appeared (from 2009):

09/01	Wil M.P. van der Aalst, Kees M. van Hee, Peter Massuthe, Natalia Sidorova and Jan Martijn van der Werf	Compositional Service Trees
09/02	P.J.l. Cuijpers, F.A.J. Koenders, M.G.P. Pustjens, B.A.G. Senders, P.J.A. van Tilburg, P. Verduin	Queue merge: a Binary Operator for Modeling Queueing Behavior
09/03	Maarten G. Meulen, Frank P.M. Stappers and Tim A.C. Willemse	Breadth-Bounded Model Checking
09/04	Muhammad Atif and MohammadReza Mousavi	Formal Specification and Analysis of Accelerated Heartbeat Protocols
09/05	Michael Franssen	Placeholder Calculus for First-Order logic
09/06	Daniel Trivellato, Fred Spiessens, Nicola Zannone and Sandro Etalle	POLIPO: Policies & OntoLogies for the Interoperability, Portability, and autOnomy
09/07	Marco Zapletal, Wil M.P. van der Aalst, Nick Russell, Philipp Liegl and Hannes Werthner	Pattern-based Analysis of Windows Workflow
09/08	Mike Holenderski, Reinder J. Bril and Johan J. Lukkien	Swift mode changes in memory constrained real-time systems
09/09	Dragan Bošnački, Aad Mathijssen and Yaroslav S. Usenko	Behavioural analysis of an I²C Linux Driver
09/10	Ugur Keskin	In-Vehicle Communication Networks: A Literature Survey
09/11	Bas Ploeger	Analysis of ACS using mCRL2
09/12	Wolfgang Boehmer, Christoph Brandt and Jan Friso Groote	Evaluation of a Business Continuity Plan using Process Algebra and Modal Logic
09/13	Luca Aceto, Anna Ingolfsdottir, MohammadReza Mousavi and Michel A. Reniers	A Rule Format for Unit Elements
09/14	Maja Pešić, Dragan Bošnački and Wil M.P. van der Aalst	Enacting Declarative Languages using LTL: Avoiding Errors and Improving Performance
09/15	MohammadReza Mousavi and Emil Sekerinski, Editors	Proceedings of Formal Methods 2009 Doctoral Symposium
09/16	Muhammad Atif	Formal Analysis of Consensus Protocols in Asynchronous Distributed Systems
09/17	Jeroen Keiren and Tim A.C. Willemse	Bisimulation Minimisations for Boolean Equation Systems
09/18	Kees van Hee, Jan Hidders, Geert-Jan Houben, Jan Paredaens, Philippe Thiran	On-the-fly Auditing of Business Processes
10/01	Ammar Osaiweran, Marcel Boosten, MohammadReza Mousavi	Analytical Software Design: Introduction and Industrial Experience Report
10/02	F.E.J. Kruseman Aretz	Design and correctness proof of an emulation of the floating-point operations of the Electrologica X8. A case study

10/03	Luca Aceto, Matteo Cimini, Anna Ingolfsdottir, MohammadReza Mousavi and Michel A. Reniers	On Rule Formats for Zero and Unit Elements
10/04	Hamid Reza Asaadi, Ramtin Khosravi, MohammadReza Mousavi, Neda Noroozi	Towards Model-Based Testing of Electronic Funds Transfer Systems
10/05	Reinder J. Bril, Uğur Keskin, Moris Behnam, Thomas Nolte	Schedulability analysis of synchronization protocols based on overrun without payback for hierarchical scheduling frameworks revisited
10/06	Zvezdan Protić	Locally unique labeling of model elements for state-based model differences
10/07	C.G.U. Okwudire and R.J. Bril	Converting existing analysis to the EDP resource model
10/08	Muhammed Atif, Sjoerd Cranen, MohammadReza Mousavi	Reconstruction and verification of group membership protocols
10/09	Sjoerd Cranen, Jan Friso Groote, Michel Reniers	A linear translation from LTL to the first-order modal μ-calculus
10/10	Mike Holenderski, Wim Cools Reinder J. Bril, Johan J. Lukkien	Extending an Open-source Real-time Operating System with Hierarchical Scheduling
10/11	Eric van Wyk and Steffen Zschaler	1st Doctoral Symposium of the International Conference on Software Language Engineering (SLE)
10/12	Pre-Proceedings	3rd International Software Language Engineering Conference
10/13	Faisal Kamiran, Toon Calders and Mykola Pechenizkiy	Discrimination Aware Decision Tree Learning
10/14	J.F. Groote, T.W.D.M. Kouters and A.A.H. Osaiweran	Specification Guidelines to avoid the State Space Explosion Problem
10/15	Daniel Trivellato, Nicola Zannone and Sandro Etalle	GEM: a Distributed Goal Evaluation Algorithm for Trust Management
10/16	L. Aceto, M. Cimini, A.Ingolfsdottir, M.R. Mousavi and M. A. Reniers	Rule Formats for Distributivity
10/17	L. Aceto, A. Birgisson, A. Ingolfsdottir, and M.R. Mousavi	Decompositional Reasoning about the History of Parallel Processes
10/18	P.D. Mosses, M.R. Mousavi and M.A. Reniers	Robustness os Behavioral Equivalence on Open Terms
10/19	Harsh Beohar and Pieter Cuijpers	Desynchronisability of (partial) closed loop systems
11/01	Kees M. van Hee, Natalia Sidorova and Jan Martijn van der Werf	Refinement of Synchronizable Places with Multi-workflow Nets - Weak termination preserved!
11/02	M.F. van Amstel, M.G.J. van den Brand and L.J.P. Engelen	Using a DSL and Fine-grained Model Transformations to Explore the boundaries of Model Verification
11/03	H.R. Mahrooghi and M.R. Mousavi	Reconciling Operational and Epistemic Approaches to the Formal Analysis of Crypto-Based Security Protocols
11/04	J.F. Groote, A.A.H. Osaiweran and J.H. Wesselius	Benefits of Applying Formal Methods to Industrial Control Software
11/05	Jan Friso Groote and Jan Lanik	Semantics, bisimulation and congruence results for a general stochastic process operator
11/06	P.J.L. Cuijpers	Moore-Smith theory for Uniform Spaces through Asymptotic Equivalence
11/07	F.P.M. Stappers, M.A. Reniers and S. Weber	Transforming SOS Specifications to Linear Processes
11/08	Debjyoti Bera, Kees M. van Hee, Michiel van Osch and Jan Martijn van der Werf	A Component Framework where Port Compatibility Implies Weak Termination
11/09	Tseesuren Batsuuri, Reinder J. Bril and Johan Lukkien	Model, analysis, and improvements for inter-vehicle communication using one-hop periodic broadcasting based on the 802.11p protocol

11/10	Neda Noroozi, Ramtin Khosravi, MohammadReza Mousavi and Tim A.C. Willemse	Synchronizing Asynchronous Conformance Testing
11/11	Jeroen J.A. Keiren and Michel A. Reniers	Type checking mCRL2
11/12	Muhammad Atif, MohammadReza Mousavi and Ammar Osaiweran	Formal Verification of Unreliable Failure Detectors in Partially Synchronous Systems
11/13	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
11/14	J.F. Groote, A.A.H. Osaiweran and J.H. Wesselius	Ananlyzing a Controller of a Power Distribution Unit Using Formal Methods
11/15	John Businge, Alexander Serebrenik and Mark van den Brand	Eclipse API Usage: The Good and The Bad
11/16	J.F. Groote, A.A.H. Osaiweran, M.T.W. Schuts and J.H. Wesselius	Investigating the Effects of Designing Control Software using Push and Poll Strategies
11/17	M.F. van Amstel, A. Serebrenik And M.G.J. van den Brand	Visualizing Traceability in Model Transformation Compositions
11/18	F.P.M. Stappers, M.A. Reniers, J.F. Groote and S. Weber	Dogfooding the Structural Operational Semantics of mCRL2
12/01	S. Cranen	Model checking the FlexRay startup phase
12/02	U. Khadim and P.J.L. Cuijpers	Appendix C / G of the paper: Repairing Time-Determinism in the Process Algebra for Hybrid Systems ACP
12/03	M.M.H.P. van den Heuvel, P.J.L. Cuijpers, J.J. Lukkien and N.W. Fisher	Revised budget allocations for fixed-priority-scheduled periodic resources
12/04	Ammar Osaiweran, Tom Fransen, Jan Friso Groote and Bart van Rijnsoever	Experience Report on Designing and Developing Control Components using Formal Methods
12/05	Sjoerd Cranen, Jeroen J.A. Keiren and Tim A.C. Willemse	A cure for stuttering parity games
12/06	A.P. van der Meer	CIF MSOS type system
12/07	Dirk Fahland and Robert Prüfer	Data and Abstraction for Scenario-Based Modeling with Petri Nets
12/08	Luc Engelen and Anton Wijs	Checking Property Preservation of Refining Transformations for Model-Driven Development
12/09	M.M.H.P. van den Heuvel, M. Behnam, R.J. Bril, J.J. Lukkien and T. Nolte	Opaque analysis for resource-sharing components in hierarchical real-time systems - extended version –
12/10	Milosh Stolikj, Pieter J. L. Cuijpers and Johan J. Lukkien	Efficient reprogramming of sensor networks using incremental updates and data compression
12/11	John Businge, Alexander Serebrenik and Mark van den Brand	Survival of Eclipse Third-party Plug-ins
12/12	Jeroen J.A. Keiren and Martijn D. Klabbers	Modelling and verifying IEEE Std 11073-20601 session setup using mCRL2
12/13	Ammar Osaiweran, Jan Friso Groote, Mathijs Schuts, Jozef Hooman and Bart van Rijnsoever	Evaluating the Effect of Formal Techniques in Industry
12/14	Ammar Osaiweran, Mathijs Schuts, and Jozef Hooman	Incorporating Formal Techniques into Industrial Practice