Experiences with a Compositional Model Checker in the Healthcare Domain

Jozef Hooman[1,2], Robert Huis in 't Veld[3], and Mathijs Schuts[3]

[1] Embedded Systems Institute, Eindhoven, The Netherlands
[2] Radboud University, Nijmegen, The Netherlands
[3] Philips Healthcare, Best, The Netherlands

Abstract. This paper describes the use of a formal method to support component-based development in the healthcare domain. The method is based on a commercial tool suite which combines formal modeling, compositional model checking, and code generation. The main approach of the tool suite will be explained and demonstrated from a user point of view. We report about experiences with this approach at the company Philips Healthcare for the design of control software for advanced interventional X-ray systems. This concerns formal interface definitions between the main system components and detailed design of control components.

Keywords: formal modeling, component-based development, model checking, code generation, industrial experience.

1 Introduction

We report about experiences at Philips Healthcare with a commercial tool based on formal methods. The tool is used for the development of control software of interventional X-ray systems. These systems are used for minimally invasive surgery, e.g., improving the throughput of a blood vessel by placing a stent via a catheter where the surgeon is guided by X-ray images. These techniques avoid open heart surgery. This has many benefits in the healthcare domain such as improved productivity, more effective treatments, better success rate, and increased quality of the life of patients. Moreover, this type of image-guide non-invasive surgery leads to lower healthcare costs by shorter hospital stays and higher throughput.

An example of such a system is depicted in Fig. 1, showing a patient table, a C-arm with X-ray tube and detector, and a large screen for images and user guidance. This equipment is placed in the so-called intervention room, where physicians press pedals and operate switches to control the acquisition of X-ray images. In addition, there is an additional control room where medical personnel can view and store images, write reports, and prepare the intervention. Preparation includes, for instance, entering patient data and details about the treatment that determine the amount of X-ray and the movements of the C-arm.

Z. Liu and A. Wassyng (Eds.): FHIES 2012, LNCS 7151, pp. 93–110, 2012.

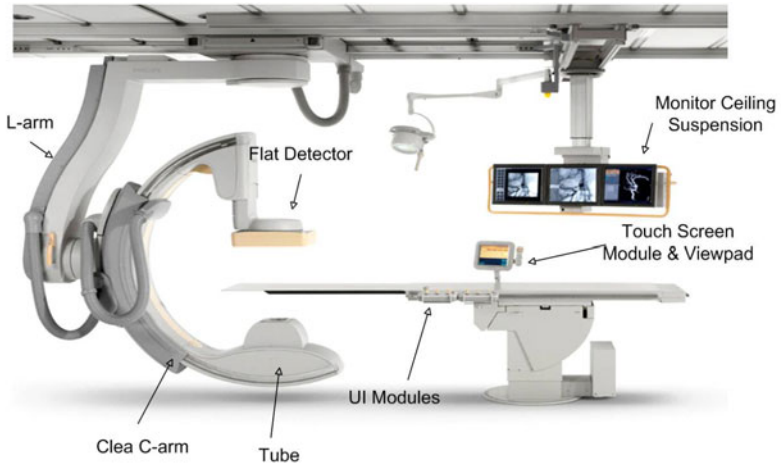

Fig. 1. Interventional X-ray system

To support new medical procedures, it is important to be able to incorporate medical innovations fast in such systems. This is challenging, because high quality standards have to be met. This requires a software architecture that can be adapted and extended easily without long test and integration times, while still maintaining the current high-level of quality.

To meet these goals, Philips Healthcare is migrating towards a component-based software architecture with clearly defined interfaces. Interfaces are defined formally using a formal tool called ASD:Suite of the company Verum [18]. This tool is also used to implement control components of the new architecture based on formally verified design models. The use of this formal technique at Philips Healthcare is motivated by the aim to remove faults as early as possible during the software development process, thus reducing the test and integration time, which is often long and unpredictable. We describe our experiences when applying this approach to a part of an interventional X-ray system.

ASD:Suite of Verum is a development tool that embeds the Analytical Software Design (ASD) technology into a software design environment. ASD [3,13] enables the application of formal methods into industrial practice by a combination of the Box Structure Development Method [15] and CSP [11]. To obtain complete and consistent specifications, a Sequence-Based Specification Method [16] is used where the response to all possible sequences of input stimuli has to be defined. Sequences that cannot happen must be declared illegal explicitly. The sequence-based specifications have been translated into CSP and the FDR model checker [8] is used to verify refinement steps. ASD:Suite hides the CSP details and provides the user with a tabular notation to describe state machines, a standard set of correctness checks, and a nice visualization of error traces.

The ASD approach distinguishes two types of models which are both described by a similar tabular notation: *interface models* and *design models*. The tool ASD:Suite allows code generation from design models for a number of

programming languages (C, C++, C#, Java). A design model implements a certain interface model, typically using services of other components by referring to their interface models. The model checker of ASD:Suite verifies that calls to these used components are correct with respect to their interface models. Moreover, it is checked that the design model conforms to the implemented interface model. The approach is compositional [12], since the verification uses only the interfaces of the used components, without knowing their implementation. It is also not needed that these used components are realized using ASD. Since the ASD approach is intended for control components, used components that involve data manipulations or algorithms will be implemented by other techniques; such components are called *foreign components*.

Related to ASD:Suite are formal methods with commercial tool support and code generation from formal models. For instance, the industrial tool VDM-Tools [5] contains a code generator for the formal language VDM++ [7]. The B-method [1], which has been used to develop a number of safety-critical systems, is supported by the commercial Atelier B tool [4]. The SCADE Suite [6] 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.

This paper is structured as follows. In Sect. 2, the interface models of ASD are explained using a small camera example which resembles a medical imaging device. The use of interface models at Philips Healthcare is described in Sect. 3. Sect. 4 introduces the design models of ASD and model checking, again using the camera example. The application of these models at Philips is discussed in Sect. 5. Concluding remarks can be found in Sect. 6.

2 ASD Interface Models

To illustrate the ASD approach we use a small camera example. We start with an ASD interface model which represents the traces a server offers to its clients. There are two ways of communication between client and server:

- *Procedure calls* from client to server, which are synchronous in the sense that the client has to wait until the server is ready to accept the call. Next the client is blocked until the server returns the call. There are two types of calls:
  - Void calls, which return a void reply to signal the completion of the call
  - Valued calls, which return a value upon completion
- *Callbacks* from server to client, which are asynchronous events that can be sent by the server immediately.

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

- *Internal modeling events*, which can be optional (meaning that they may happen) or inevitable (meaning that they will happen eventually).

For the camera example we have the following sets of calls, callbacks, and internal events:

- APICamera contains three calls:
  - PowerOn(): valued, with two possible return values: OnOK, OnFailed
  - PowerOff(): void
  - Click([in]exposureTime:int): void
- CBCamera contains four callbacks:
  - CBPicture([in]photo:image)
  - CBOn()
  - CBEmptyBattery()
  - CBOnFailed()
- INTCamera contains four internal modeling events to trigger the four callbacks above:
  - PicutureMade, which is inevitable
  - SwitchedOn, which is inevitable
  - SwitchedOnFailed, which is inevitable
  - BatteryEmpty, which is optional

An interface model is represented as a state machine which defines the set of possible traces representing the interface between client and server. Such an ASD interface model plays a similar role as a protocol state machine of UML [2]. An ASD interface not only describes the services offered by the server; it also specifies the calls allowed by the client. So it can be seen as a contract between client and server, similar to the Design by Contract approach [14].

In the Camera example, the ASD interface is shown in Fig. 2. There are four states: Off, SwitchingOn, On, and TakingPicture. In each state the response to all possible stimuli is defined. The "+" behind the name of an event indicates that it is a valued call. The tool ASD:Suite generates for each reachable state a so-called canonical sequence which is a minimal sequence of input stimuli leading to the state from the initial state (which is always the first state mentioned). This canonical sequence is written behind the name of each state. Observe that the state machine is non-deterministic; in state Off there are two possible responses to the valued call PowerOn.

In state Off the calls PowerOff and Click are declared to be Illegal which means that the client should not call these functions in this state. The modeling events are Disabled in state Off. The use of these internal modeling events is illustrated by state SwitchingOn; when modeling event SwitchedOn or SwitchedOnFailed occurs, the camera sends the corresponding callback to the client.

Note that the rules 2, 12, 21 and 30 are hidden; they can be used for invariants which are not discussed in this paper. Fig. 3 shows a visual representation of the state machine of this interface, which is generated automatically by ASD:Suite from the table of Fig. 2.

To design the camera component, two other components will be used: a battery component and a shutter component. The interfaces of these components are shown in Fig. 4 and Fig. 5, respectively, where illegal and disabled events are hidden. Also these two interfaces are non-deterministic. Observe that the

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

Fig. 2. ICamera: Interface Model of the Camera Component

battery interface uses a state variable EmptyDetected to describe the set of allowed traces. Rule 16 of Fig. 4 shows that a Charge call in state BatteryOn does not lead to an externally visible action, but it updates variable EmptyDetected if it is true. Hidden is the rule which expresses that the Charge call is illegal if EmptyDetected equals false.

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

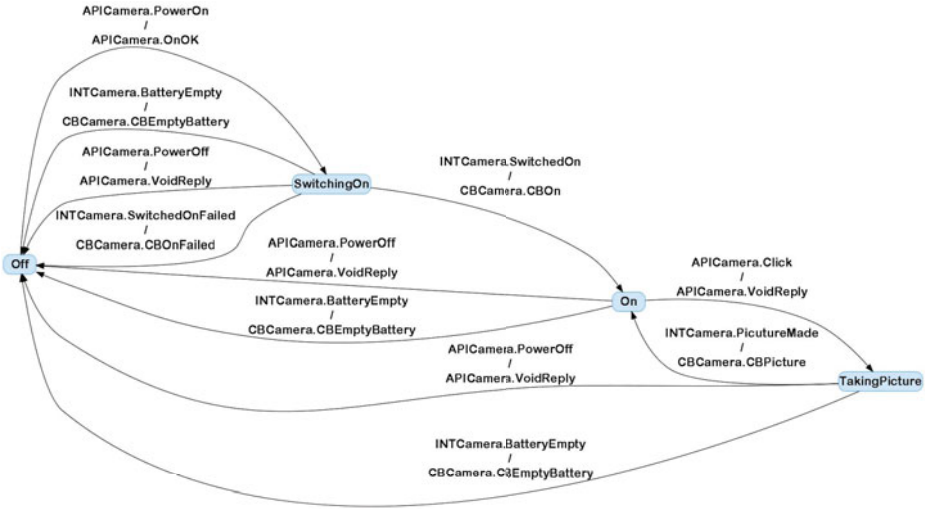

Fig. 3. Graphical Representation of the Interface Model of the Camera

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

Fig. 4. IBattery: Interface Model of the Battery Component

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

Fig. 5. IShutter: Interface Model of the Shutter Component

3 The Usage of Interface Models at Philips Healthcare

The software of the interventional X-ray systems of Philips Healthcare has grown enormously over the last ten years. There is a need to redesign the system architecture to be prepared for a number of challenges:

- Fast support for medical innovations in various clinical segments, such as cardiovascular, neurology, electrophysiology, and surgery.
- Deal with over 1 million product variations, e.g., due to many possible configurations for patient tables, many types of display screens and user input devices, a large number of dedicated devices and image enhancement algorithms supporting specific medical procedures, and combinations of these features.
- Incorporate parts of 3rd party suppliers in a fast and reliable way.
- Allow product service for the next ten years.

To this end, the current system is migrated to a new component-based architecture with clearly defined component interfaces. As a starting point for a new architecture, the system has been split into three main components:

- A *Data Handling* component which deals with patient data, treatment selection, user interaction, interaction with the hospital information system, and the coordination of the other two components.
- An *Image Acquisition* component which deals with the control of the image acquisition chain, including the processing of user input (e.g., via pedals and switches) and the coordination between the X-ray generation (with the right dose for the selected procedure) and the detection of X-ray images.
- An *Image Processing* component which deals with processing and enhancing images, possibly combining it with images from other sources.

The interface between the Image Acquisition and Image Processing components has not been specified in ASD, because this concerns image formats and associated data. The interfaces between Data Handling and the other two components have been defined formally by means of ASD interfaces. These state machines specify an interaction protocol between the components, with several phases such as version exchange, activation & deactivation, data handling, and image acquisition. This leads to large ASD tables. For instance, the interface between Data Handling and Image Acquisition, called *IDHIA*, contains 41 calls, 32 callbacks, and 32 modeling events. The ASD state machine contains 2 state variables and 25 states. In each state the response to the 41 calls and 32 modeling events has to be defined, leading to a table with more than 1800 rule cases.

The three main components of the architecture have been refined and at appropriate places additional ASD interfaces have been defined or are being defined. The general experience is that the formally defined ASD interfaces are very useful. Besides the usual static description, listing all function calls, now also the dynamic behaviour is specified, such as the allowed order of calls and callbacks. Since the interface specifications must be complete, all unclarities have

to be resolved and implicit domain knowledge must be made explicit. An important advantage is that the formally defined state machines allow independent development of the components. Moreover, at Philips the ASD interfaces are used to generate conformance tests. Altogether, these formal interfaces reduce the amount of integration problems.

We observed that the definition and interpretation of ASD interface models is sometimes difficult for software developers. It requires the ability to reason about traces and to abstract from design decisions which is not always easy. The difficulties encountered are partly due to the current status of the tool which does not support a very concise notation. An interface model is a flat state machine and structuring mechanisms such as the hierarchy and concurrency constructs of Harel's Statecharts notation [10] are not allowed. Hence a transition which is common to a number of states (e.g., a reset or error transition) has to be entered manually many times. This makes it difficult to read, to understand, and to maintain large interfaces. Current solution is to draw separate Visio diagrams of more structured charts, but it is difficult to keep model and diagram consistent. In any case, it is good practice to keep interfaces small and split them whenever possible.

4 Design Models and Model Checking in ASD:Suite

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

The model checker of ASD:Suite can be used to check conformance with the implemented interface and consistency with the used interfaces. Complete executable code can be generated from the design model, where a choice can be made between a number of programming languages (currently C, C++, C#, and Java). The camera example is used to explain the ASD design models in Sect. 4.1 and the model checker in Sect. 4.2.

4.1 ASD Design Models

The design of the camera component is depicted in Fig. 6, where ovals represent ASD interface models and the rectangle denotes an ASD design model. The up arrow denotes that the design model refines the interface model of Fig. 2, as will be explained in Sect. 4.2. The design uses the interfaces IBattery of the battery (Fig. 4) and IShutter of the shutter (Fig. 5).

Figure 7 shows the ASD design model for the camera example, hiding illegal and disabled events. Similar to the interface model, the table must be complete in the sense that for all events an action must be defined (which might be Illegal or Disabled). The events now include callbacks of the used interfaces, such as the callbacks CBBatteryEmpty from the battery and CBPicture from the shutter. Similarly, the actions may contain calls to used interfaces. For instance, in state

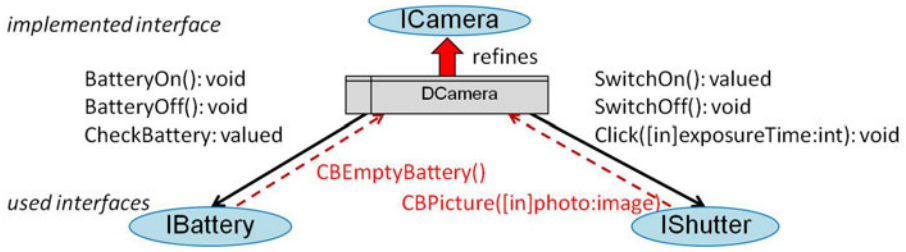

Fig. 6. Design of the Camera Component

Off rule 3 expresses that - as a response to the PowerOn call - the function CheckBattery of the battery is called. As indicated by the "+" behind the name, this is a valued call and in state CheckingBattery the component is waiting for a response. In such a so-called blocking state, all other events are blocked.

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

Fig. 7. DCamera: Design Model of the Camera Component

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

In an ASD design model it is not possible to make control decisions based on parameters of function calls. For instance, in the camera example it is not

possible to make a case distinction on the exposureTime parameter of the Click call. If control would depend on the value of such a parameter, it has to be sent to a foreign component, which is not implemented using ASD. Such a foreign component can analyze the value and return different values or callbacks to indicate the required control.

The semantics of a design model is such that callbacks from used components can always be received and they are put into a so-called *callback queue* (FIFO). Client calls are serialized, that is, at any point in time at most one client call is executed. Callbacks have priority over client calls. Initially, and after the completion of a rule case, first the callback queue is inspected. If this queue is not empty, the rule case corresponding to the first callback is executed. When the callback queue is empty and no call is being processed, a new call may be accepted. An illegal call or callback leads to a halt of the component.

4.2 Compositional Model Checking

The tool ASD:Suite contains a fixed number of checks on design models. Figure 8 shows a screenshot of part of the tool with a Verify window. Verification includes the previously discussed checks on all interfaces, i.e., implemented and used interfaces. In addition there are specific checks for design models, such as a check to ensure that the design model is deterministic. Most important is a check on the consistency of all interfaces. The design should adhere to all interface models of used components and it should conform to the implemented interface. Conformance has been defined formally in the failures-divergence model of CSP [17] and is checked with the underlying FDR2 model checker [8]. Note that FDR2, which is an abbreviation of Failures/Divergence Refinement 2, is in fact a refinement checker.

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

- In used interface IBattery it is possible to get a BatteryOff call in state BatteryOff; this is a race condition between a PowerOff call and a callback CBBatteryEmpty send by the battery components. Note that the callback is put into the callback queue of the camera component while the BatteryOff call is processed. This problem is corrected by improving the battery interface as shown in Fig. 9 where rule 4 now allows a BatteryOff call.
- The model checker complained about an attempt to switch the shutter on when it was already on, which is not allowed by the interface of the shutter as specified in Fig 5. Analyzing this situation, it turned out that in the design model it was forgotten to switch the shutter off when a CBBatteryEmpty has been received (rules 41 and 52 in Fig. 7).
- Similarly, it was forgotten to switch the battery off when an attempt to switch the shutter on fails (rule 32 in Fig. 7).
- As another race condition, the model checker shows that a callback CBPicture might be received in state Off, namely after a CBBatteryEmpty in state TakingPicture. This has been repaired by adding a rule case in the design to receive the callback, but not forwarding it to the client.

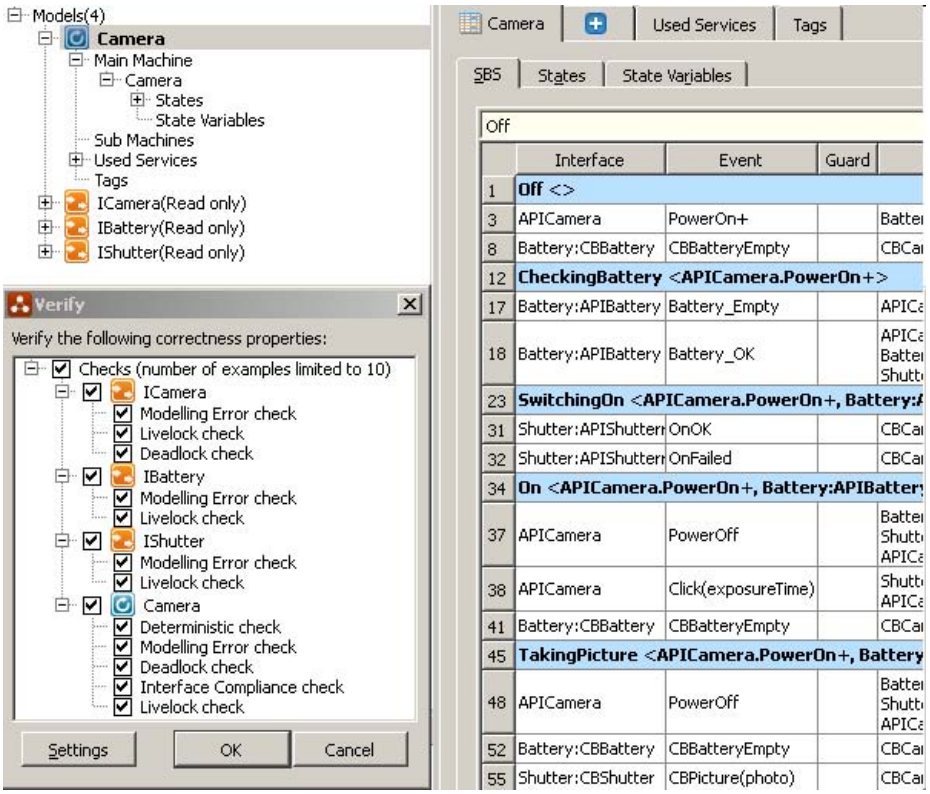

Fig. 8. Screenshot ASD:Suite with Verify Window

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

Fig. 9. Improved Battery Interface IBattery

ASD:Suite has a nice visualization of error traces, which makes it easy to find the cause of an error. Figure 10 shows the visualization of the last problem mentioned above. It shows the lifeline of the Camera component, with a client, the callback queue (called DPC+Q), its used components Battery and Shutter, and an environment which triggers modeling events in the used interfaces.

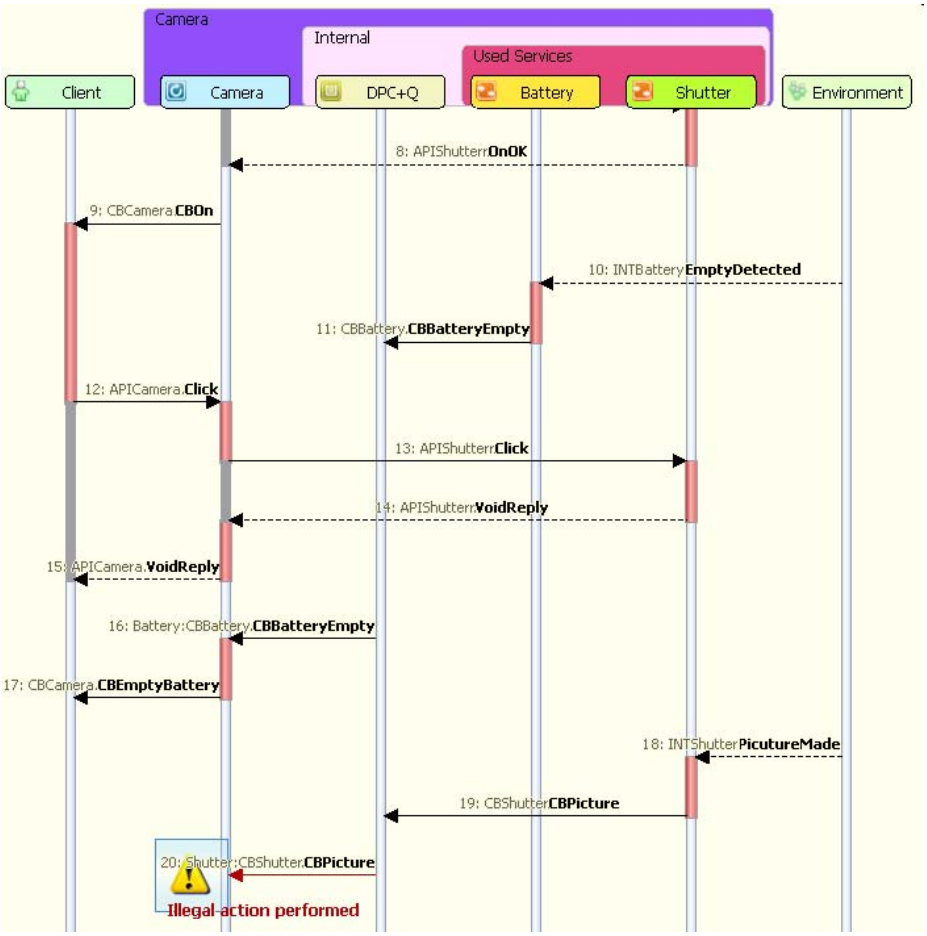

Fig. 10. Visualization of an Error Trace

The corrected design model for the camera is shown in Fig. 11, with changes in rules 11, 32, 41, and 52.

Observe that the verification is compositional since it uses only the interfaces of the used components. Hence, the used components can be developed independently according to their interface. Also note that there is no obligation to develop these components with ASD. Typically, components that involve data manipulations will be implemented differently and conformance to their ASD interface is checked by means of testing.

Finally, observe that the components have a strict communication pattern; a client of a component can only perform synchronous calls and might receive asynchronous callbacks from the component. Similarly, the component itself will only perform synchronous calls on its used components and receive callbacks from them. In this way absence of deadlock is achieved by construction.

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

Fig. 11. Correct Design Model for the Camera

5 The Use of Design Models at Philips Healthcare

In this section we describe experiences with the use of ASD design models at Philips Healthcare. In Sect. 5.1 we report about the attempts to design a component that has to satisfy a large and complex ASD interface. Next Sect. 5.2 summarizes the observations made when applying the ASD approach to the detailed design of components.

5.1 Decomposition of a Complex Interface

At Philips Healthcare, ASD-based design has been applied to components which are responsible for high-level supervisory control. Starting point are the ASD interface models between the main system components as mentioned in Sect. 3. In this section we describe the experiences with ASD design in the Image Acquisition component. Hence, we start with the complex interface model IDHIA, where Data Handling is the client and Image Acquisition the server. First challenge was to design a component which conforms to this complex interface. We describe a few design possibilities considered and list the resulting observations.

The main idea of the design was to start with a component which decomposes the interface into a number of smaller interfaces, based on the phases defined in

the interface protocol: version exchange, activation/deactivation, data handling, and image acquisition. We investigated two possibilities:

– A sequence of components, where the first one (called DVersion) manages the version part of the protocol, using an interface IVersion for detailed control, and forwarding the remainder of the interface to a reduced interface. The second one deals with activation, and the third one with the data handling and image acquisition part. The main idea of the design is depicted in Fig. 12, using the same notation as Fig. 6.

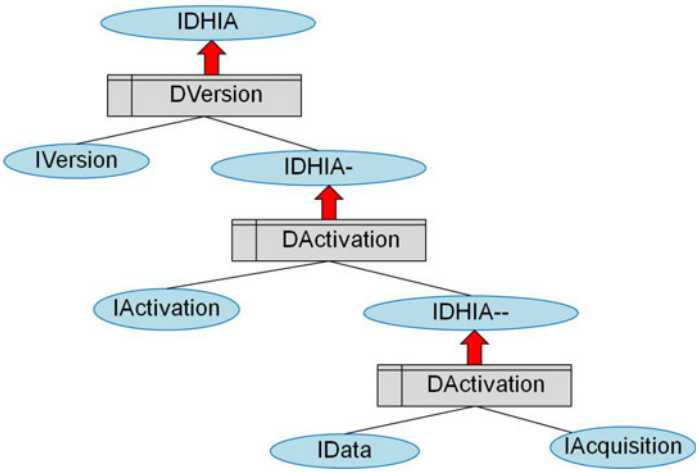

Fig. 12. Design with Sequence of Components

– A single component which has as states the main phases of the protocol (Version, Activation, Data and Acquisition). In each state it forwards relevant calls to the corresponding interface. This design is depicted in Fig. 13.

When trying to implement these design possibilities in ASD, we encountered a number of problems. Main problem was that calls that are illegal in the interface cannot be accepted and forwarded. To avoid these problems, all calls can be made valued (instead of void), returning Accepted or Rejected. But this was not acceptable for the designers of the Data Handling component, because valued calls require an additional state to wait for the result and too many of them lead to a complex design.

The problems with illegal calls also triggered a discussion about robustness. As mentioned in Sect. 4.1, the semantics is such that any illegal client call leads to a halt of the component. To be more robust against illegal calls of non-ASD clients (e.g., 3rd party clients) Verum was requested to allow the possibility to specify some kind of graceful degradation for illegal calls of non-ASD clients. To deal with illegal callbacks of non-ASD used components, a pattern has been

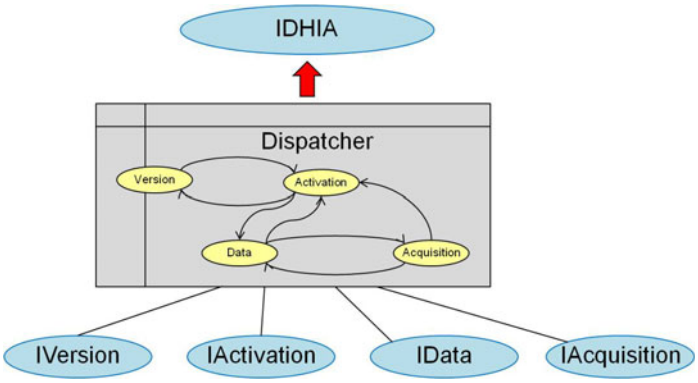

Fig. 13. Design with Dispatcher Component

defined by Verum to deal with such callbacks and to log erroneous callbacks (e.g., to discuss interface errors with suppliers of 3rd party used components).

Finally, we observed that forwarding calls leads to additional race conditions. A race condition shows a design problem to deal with a simultaneous client call and a callback from a used interface where the order of acceptance leads to different states. A pattern has been defined to solve race conditions in a design systematically, but altogether both design options discussed above did not lead to satisfactory implementations in ASD. Verum is working on alternative verification modes where illegal calls can be delegated.

Meanwhile, the choice was made to design a component which contains all (25) states of the interface to be able to decide which calls are illegal. All non-illegal calls are forwarded to appropriate interfaces, similar to the designs pro-posed above. Compared to the solution where all calls are made valued, this leads to easier interfaces. It also avoids additional race conditions. The design, however, is rather complex and not easy to read and to maintain. A small improvement could be achieved by introducing sub-state machines which can be used one-level deep in design models (but not in interface models). This last step, however, required an improvement of interface model IDHIA where the phases of the communication protocol had to be separated more clearly.

5.2 Detailed Design of Control Components

Next we describe experiences with the detailed design of the high-level control part of the Image Acquisition component. To be able to implement the main control components in ASD, a number of constraints have to be taken into account:

- The communication pattern with synchronous function calls from top to bottom and callbacks from bottom to top has to be respected
- Data manipulation and control has to be separated to allow the implementation of the data part in non-ASD components

- To allow model checking and to avoid hitting the state explosion problem:
  - Reduce the number of callbacks that may occur simultaneously
  - Reduce the number of unsolicited callbacks
  - Reduce the number of state variables
  - Keep components small, split components when adding functionality

Although the compositional approach of ASD is very important for scalability and allows the application to large industrial systems, typically the size of some components increased too much during development, especially when exceptions and failure modes were added. Then easily the boundaries of the model checker were met and a number of redesigns were needed to make model checking possible again.

In general, it turned out to be very important to have an extensive preparation phase before applying ASD. The requirements must be clarified with more precision than during conventional software development and extensive discussions on both requirements and design are needed. Since ASD:Suite considers each component and its interface in isolation, additional tooling (including a UML tool, PowerPoint and Visio) has been used to create design overviews.

Once the requirements were clarified and a proper design with small components was established, we usually started with a precise definition of component interfaces in ASD. Next the completion of design models was rather straightforward. Of course, the model checker still finds quite a number of errors, but they are often rather easy to correct. The camera example is a good illustration of the type of errors found and their correction.

6 Concluding Remarks

Summarizing the experiences with the formal development tool ASD:Suite at Philips Healthcare, it is clear that the formally defined interfaces are very beneficial. They not only describe the syntax of calls and callbacks, but also define the expected behaviour in terms of the allowed sequences of events. This reduces the amount of faults detected during integration and improves the possibility to develop components independently. At Philips Healthcare the aim is to certify the main components using conformance tests which are derived from the ASD interface models. Main challenge is to keep the interfaces manageable and maintainable.

The most important advantage of ASD-based design is the strong combination of compositional model checking and complete code generation from the same models, where code and model have the same semantics. By means of the model checker many faults are found early and the nice visualization of error traces makes it easy to correct them. Especially when dealing with complex combinations of device failures, activation/deactivation, and connect/disconnect behaviour it would be almost impossible to obtain a correct implementation without the support of a tool like ASD:Suite.

An analysis of earlier applications of ASD in the Data Handling component at Philips Healthcare [9] showed that units containing ASD components have less

reported defects than other units. Although the work on the Image Acquisition part was not yet completed at the time of writing, the impression is that the problems reported after the integration and test phase of the first increments mainly concerned differences in the interpretation of requirements.

It is important to realize that a number of design constraints have to be fulfilled in order to apply ASD successfully, such as the required communication pattern. At Philips Healthcare, the hierarchical control structure required by ASD supports the desired change from an object-oriented blackboard architecture towards a hierarchical component-based control architecture. On the other hand, during detailed design it is sometimes difficult to make a balanced division into ASD components for the control part and non-ASD components to deal with data manipulations. Moreover, components have to be made smaller than usual to allow model checking.

Another important observation is that the engineers have to realize that testing is still needed and a good set of unit tests is still very valuable. The ASD model checker verifies only the interaction protocol between components; it does not check the relation between calls to the implemented interface and calls to the used interfaces. It would be an interesting challenge to investigate whether the specification and verification of such relations could be added to the tool suite without compromising the ease of use for the average engineer.

Acknowledgments. This work has been supported by the Dutch knowledge workers project Partitura, ITEA project Care4Me, and COMMIT project Allegio. We would like to thank Leon Bouwmeester of Verum for his valuable comments on a draft of this paper.

References

1. Abrial, J.R.: The B-book: assigning programs to meanings. Cambridge University Press, New York (1996)
2. Booch, G., Rumbaugh, J.E., Jacobson, I.: The unified modeling language user guide - the ultimate tutorial to the UML from the original designers. Addison-Wesley object technology series. Addison-Wesley-Longman (1999)
3. Broadfoot, G.H., Broadfoot, P.J.: Academia and industry meet: Some experiences of formal methods in practice. In: 10th Asia-Pacific Software Engineering Conferenc (APSEC 2003), pp. 49–58 (2003)
4. ClearSy: Atelier B, industrial tool supporting the B method (2011), `http://www.atelierb.eu/en/`
5. CSK Systems Corporation: VDMTools, industrial tool supporting VDM++ (2011), `http://www.vdmtools.jp/en/`
6. Esterel Technologies: SCADE Suite, model based development environment dedicated to critical embedded software (2011), `http://www.esterel-technologies.com/products/scade-suite/`
7. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object-oriented Systems. Springer, New York (2005), `http://www.vdmbook.com`
8. Formal Systems (Europe) Ltd: FDR2 model checker (2011), `http://www.fsel.com/`

9. Groote, J.F., Osaiweran, A., Wesselius, J.H.: Analyzing the effects of formal methods on the development of industrial control software. In: Proceedings of the IEEE ICSM 2011, pp. 467–472 (2011)
10. Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming 8(3), 231–274 (1987)
11. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
12. Hooman, J.: Specification and Compositional Verification of Real-Time Systems. LNCS, vol. 558. Springer, Heidelberg (1991)
13. Hopcroft, P.J., Broadfoot, G.H.: Combining the box structure development method and CSP for software development. Electronic Notes in Theoretical Computer Science 128(6), 127–144 (2005)
14. Meyer, B.: Applying "design by contract". IEEE Computer 25(10), 40–51 (1992)
15. Mills, H.D.: Stepwise refinement and verification in box-structured systems. Computer 21, 23–36 (1988)
16. Prowell, S.J., Poore, J.H.: Foundations of sequence-based software specification. IEEE Transactions on Software Engineering 29, 417–429 (2003)
17. Roscoe, A.W.: The theory and practice of concurrency. Prentice Hall (1998)
18. Verum: ASD:Suite (2011), `http://www.verum.com/`