Experience Report on Designing and Developing Control Components using Formal Methods

Ammar Osaiweran[1], Tom Fransen[2], Jan Friso Groote[1], and Bart van Rijnsoever[2]

[1] Eindhoven University of Technology, Eindhoven, The Netherlands
[2] Philips Healthcare, BU Interventional X-ray, Best, The Netherlands
{a.a.h.osaiweran,j.f.groote}@tue.nl,{tom.fransen,bart.van.Rijnsoever}@philips.com

Abstract. This paper reports on experiences from an industrial project related to developing control components of an interventional X-ray system, using formal techniques supplied by the Analytical Software Design approach, of the company Verum. We illustrate how these formal techniques were tightly integrated with the standard development processes and the steps accomplished to obtain verifiable components using model checking. Finally, we show that applying these formal techniques could result in quality software and we provide supporting statistical data for this regard.

Key words: Formal methods in industry; Analytical Software Design; component-based software; Software quality.

1 Introduction

This paper demonstrates experiences of developing control components of an interventional X-ray imaging system, using a formal development approach, called the Analytical Software Design (ASD). The work was carried out in one of industrial projects of the business unit Interventional X-Ray (iXR), at Philips Healthcare.

Figure 1 presents an example of such type of systems, depicting a number of movable parts such as a patient table, a stand that holds X-ray collimator and image detector. It also shows graphical user interfaces that facilitate managing details of patients and their clinical examinations and visualizing live images.

The X-Ray equipment is used to support minimally invasive, image-guided surgery to, for instance, improve throughput of patient blood vessels by inserting a stent via a catheter where the physician is guided by X-ray images. This way open heart surgery is avoided resulting in increasing productivity, more effective treatments and reduce healthcare costs by shorter hospital stays and higher throughput.

Since the healthcare domain is quickly evolving, many challenges are imposed to such type of X-Ray systems. This includes, for example, rapidly supporting the increasing amount of medical innovations, new clinical procedures and smooth

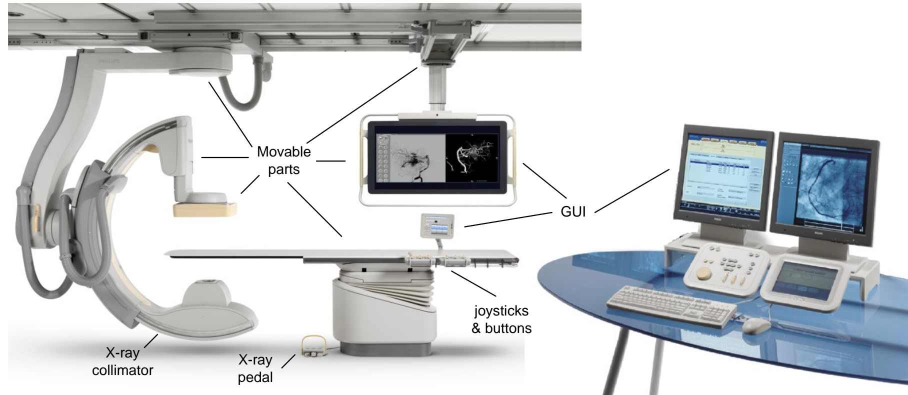

Fig. 1. Interventional X-ray system

integration with products of third part suppliers. Indeed, this requires a flexible software architecture that can be easily extended and maintained without the need of constructing software from scratch.

To achieve this goal, Philips Healthcare is gradually shifting to a component-based architecture with formally specified and verified interfaces. The development of such type of components is supported by a commercial formal verification tool called the ASD:Suite, supplied by the company Verum [18]. The aim is to build high quality components that are mathematically verified at the design phase by eliminating defects as early as possible in the development life cycle, and thus reducing effort and shortening time devoted to testing and integration.

Early reports show that applying formal techniques of ASD resulted in better quality code compared to software developed in more conventional approaches [8]. Therefore, these formal techniques are becoming more and more credible for developing software at Philips Healthcare [7, 6, 2, 13, 9].

The X-ray machines comprise embedded software which includes a number of software modules. One of the key modules is the Backend Orchestration, which is mainly responsible of controlling workflow steps required to achieve clinical examinations using X-ray.

The purpose of this paper is to report on our experience of how we tightly integrated the ASD approach and its formal techniques with the standard development processes for developing control components of the Orchestration module. It also focuses on steps followed to design components of the Orchestration module that preceded the steps of modeling and developing the components using the ASD technology, highlighting limitations encountered during the design process.

The paper demonstrates how these design steps effectively helped us designing verifiable components. We illustrate peculiarities of these components that facilitate verifying them compositionally following the ASD recipe, avoiding the

state space explosion problem of the behavioral verification using model checking supported by the ASD:Suite. Finally, the paper investigates the effectiveness of using ASD to the quality of the module, demonstrating defects reported along the development of the module. We show that errors escaped the ASD formal verification were easy to locate and fix, not deep design or interface errors.

This paper is structured as follows. Section 2 addresses the ASD approach to the limit needed in this paper. In Section 3 the context of the Orchestration module within the X-Ray system is introduced. Section 4 demonstrates steps of incorporating the ASD approach to the standard development processes for developing components of the Orchestration module. Section 5 details steps accomplished for designing components of the Orchestration module, and the peculiarities that facilitate verifying them easily using model checking. Section 6 provides statistical data regarding the end quality results of the module.

2 Principles of Analytical Software Design

ASD is a component-based, model-driven technology that incorporates formal mathematical methods such as Sequence-Based Specification (SBS) [14], Communicating Sequential Processes (CSP) [15] and its model checker Failure Divergence Refinement (FDR2) [3] to software development.

A common design practice in ASD is to identify a software design as interacting components, communicating with one another or their environment via communication channels (interfaces). Using ASD, functionality of a system is distributed among responsible components in levels (e.g., hierarchical structure), to facilitate systematic construction and verification of components in isolation.

At the left of Figure 2 an example structure of components is depicted. It includes a controller (Ctr) that controls a motor and a sensor assumed to be attached to the patient table. Here, we assume that the motor component is responsible of moving the patient table to the left and to the right. The sensor sends signals to the top controller in case there is an object in the course of a movement to prevent collisions with patients or other parts of the system. We use this example system along with the description of the ASD approach in this section.

Any ASD component is developed using two types of models complementing each other: the interface and design models. The interface model of a component does not only include the signatures of methods to be invoked on the component but also the externally visible behavior and the protocol of interaction exposed to client components at an upper level. It excludes any behavior with lower-level components. The interface model can also be used to describe the external behavior of components not developed using ASD. This way the interface model can represent legacy code, hardware and manually coded modules.

The actual detailed behavior of the component is described by a design model, which includes interactions with used components at a lower level. The specification of models is supported by an ASD industrial tool, called the ASD:Suite.

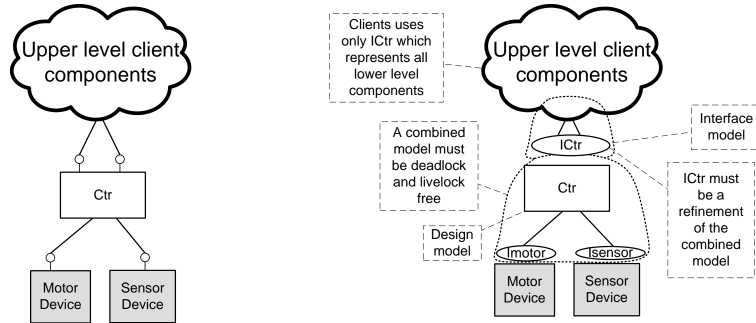

Fig. 2. Example of structured components

The ASD interface and design models are state machines, but described in tables. Each model consists of a number of tables, each of which represents a state in the state machine. An example specification of the interface model of the motor component is presented in Figure 3. It describes the external behavior of the motor towards the top controller providing the behavior of the very basic movements.

The specification depicts two tables that represent two states: *UnInitialized* and *Idle*. Every table comprises a number of rows called rule cases, each of which includes a number of items, such as the interface name (channel), stimulus event, a list of responses and a transition to a new state.

	Channel	Stimulus event	Predicate	Response	State update	Next state	Comment	Tag
1	UnInitialized<>							
2	IMotorCtr	initialize		IMotorCtr.NullRet		Idle		
3	IMotorCtr	uninitialize		Illegal		-		
4	IMotorCtr	moveLeft		Illegal		-		
5	IMotorCtr	moveRight		Illegal		-		
6	IMotorCtr	stopMovement		Illegal		-		
7	Idle<IMotorCtr.initialize>							
8	IMotorCtr	initialize		Illegal		-		
9	IMotorCtr	uninitialize		IMotorCtr.NullRet		UnInitialized		
10	IMotorCtr	moveLeft		IMotorCtr.NullRet		Idle		
11	IMotorCtr	moveRight		IMotorCtr.NullRet		Idle		
12	IMotorCtr	stopMovement		IMotorCtr.NullRet		Idle		

Fig. 3. The tabular specification of the Motor interface in the ASD:Suite

As can be seen from the specification, all possible input stimuli are listed in the tables, so that ASD users are forced to fill-in all corresponding items for the sake of specification completeness. This often results in finding newly unaddressed scenarios and thus initiating discussion with different stakeholders in early phases of the development life cycle.

The ASD:Suite ensures consistency and correctness by automatically generating the tabular specification to corresponding mathematical models such as CSP [10] and source code implementation in different languages such as C++ or C# (following the state machine pattern in [5]). Usually, any changes to the generated CSP models or the source code are not recommended. Details of such systematic translations are irrelevant for this paper.

ASD components are built and formally verified in isolation to allow, for instance, parallel, multi-site development. The isolated, compositional verification is especially essential to circumvent the state space explosion problem when FDR2 is used for formal verification. Below we summarize steps required to develop an ASD component, considering developing the *Ctr* component depicted in Figure 2 at the right as an example.

1. *External behavior specification.* Initially, the interface model of the component is created. Interactions with used components at a lower level are excluded from this specification. For instance *ICtr* is the interface model of the *Ctr* component, where interactions with the sensor and the motor interfaces are not included. *ICtr* specifies how the clients are supposed to use *Ctr*.
2. *External specification of boundary components.* Likewise, interface models of used components at the lower level are specified. These models describe the external behavior exposed to the component being developed. For instance, *Isensor* and the *Imotor* interface models specifies the external behavior visible to the *Ctr* component. Any internal interactions not visible to *Ctr* are not included.
3. *Concrete, functional specification.* Upon the completion of specifying the external behavior, a design model of the component is constructed. It includes detailed behavior and interactions with used components. For instance, design model of *Ctr* comprises method invocations from and to the *Motor* and the *Sensor* components.
4. *Formal behavioral verification.* In this step the ASD:Suite translates all ASD models to corresponding CSP processes for verification using the FDR2 model checker. Verification includes an exhaustive check on the absence of deadlocks (crashes or failure to proceed with any action), livelocks (hanging due to entering an endless loop of internal events and not responding to external commands), and illegal (unexpected) interactions for a combined CSP model that includes the design and the used interface models. When an error is detected by FDR2, ASD:Suite visualizes a sequence diagram and allows users to trace the source of error back in the models. To clarify this step using the *Ctr* component example, the ASD:Suite systematically constructs a combined model that composes *Ctr* and *Imotor* and *Isensor*. Then, the behavioral verification checks whether *Ctr* uses the motor and the sen-

sor interfaces correctly, such that no deadlocks, livelocks, illegal calls, race conditions, etc. are present.

5. *Formal refinement check of external specifications.* After that, ASD:Suite checks whether the design model created in step 3 correctly refines the interface model of step 1 using failures and failures-divergences refinement. Errors are also visualized and traced to the models to allow easy debugging. Once the formal refinement check is succeeded, the interface model represents all lower levels components. Hence, integrating concrete components is often done without errors. For instance, when the Ctr design of step 3 refines of the *ICtr* interface of step 1, *ICtr* formally represents all lower level components including *Ctr*, the Motor and the Sensor.
6. *Code generation.* After all formal verification checks succeeded, source code can be generated and integrated with the rest of the code.
7. *Iterative development of components.* Each interface model can be used as a building block for refinement of new design models. Hence, this allows developing ASD components top-down, middle-out or bottom-up, in parallel with developing the manually coded modules.

3 The context of the Orchestration module

The embedded software of the X-ray equipment is divided into concurrent subsystems; among these are the Backend, the Frontend and the Image Processing (IP), see the deployment in Figure 4. The subsystems communicate with one another via standardized, formally verified ASD interfaces. These interfaces are made formal in order to ensure equal understanding of the intended behavior among separate teams developing the subsystems and to reduce communication overhead.

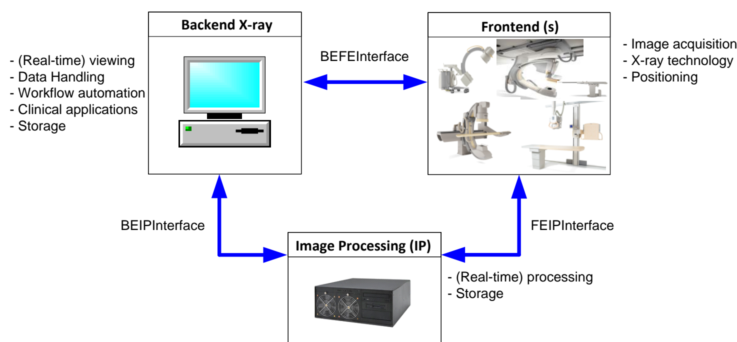

Fig. 4. Subsystems with distinct responsibilities and formal interfaces

Each subsystem comprises a number of software units, each of which includes various modules that encapsulate a number of software components, with well-defined interfaces and responsibilities. Below we briefly address the functionality of the subsystems from a high-level perspective to the extent required for introducing components of the Orchestration module.

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

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

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

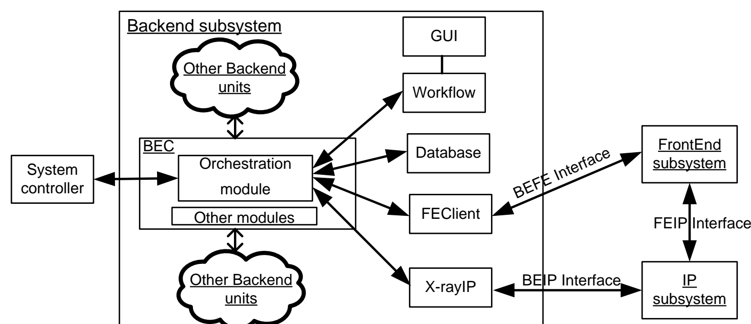

Fig. 5. Relation of Orchestration as a black-box with other units

The Backend includes a total of 12 software units. One of these units is the Backend controller (BEC), which includes the Orchestration module as one of its control modules. Figure 5 depicts the deployment of the Orchestration module in the Backend surrounded by a number of concurrent (i.e., multiple processes include multiple threads) units on the boundary.

The impetus of introducing the Orchestration module was the result of migrating from decentralized architecture, where units were working on their own, observing changes in the system through a shared blackboard and then react

upon them, to a more centralized one. The main challenge imposed on the decentralize architecture was the need to know the overall state of the entire system and whether all units are synchronized with one another in predefined states. Further, extensibility and maintainability were complex to achieve and utterly challenging.

Therefore, the Orchestration module is used as a central module that is responsible of coordinating the activities related to clinical examinations in the system and ensuring that all parties are synchronized in predefined states. The module is mainly responsible of coordinating a number of phases required to achieve the clinical examinations and harmonizing the flow of events between the concurrent interacted subsystems, preventing potential deadlocks, livelocks, race conditions, and illegal interactions. These phases are depicted in Figure 6 and summarized below.

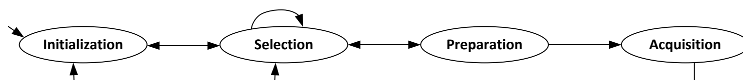

Fig. 6. Global system phases

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

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

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

automatic positioning of the motorized movable parts such as the tables and the stands.

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

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

4 Developing the Orchestration module

We detail the activities performed to develop components of the Orchestration module, through a total of six consecutive increments. The development process involved 2 full-time and 2 part-time team members. Each increment included two members who were involved not only in developing the Orchestration module but also in building other modules of the BEC unit. The team attended ASD training courses, to learn the fundamentals of the ASD approach and its accompanying technologies. Team members had sufficient programming skills, but limited background in formal mathematical methods. During the first three increments one ASD consultant was present who devoted half of his time helping the team to quickly understand the ASD approach and its technology.

Control components that include state machines were implemented using the ASD approach, whereas non-control components such as those used for data computation or manipulation were developed using the conventional development approach, in parallel to developing the ASD components.

Steps of developing components of the Orchestration module. The development process within the context of iXR is an evolutionary iterative process. That is, the entire software is developed through consecutive increments, each of which requires regular review and acceptance meetings by several stakeholders. Figure 7 depicts the flow of events performed in a single increment for developing components of the Orchestration module. It presents how the ASD approach was combined with the standard development approach in industry.

At the start of each increment, lead architects identify a list of features to be implemented together with related requirements. After the features and the requirements are approved by various stakeholders, the development team provides project and team leaders with work breakdown estimations that include,

for instance, required functionalities to be implemented, necessary time, potential risks and effort, and dependencies with other units that may block the development progress of these features.

Based on the work breakdown estimations, team and project leaders prepare an incremental plan, which includes the features to be implemented in a chronological order, scheduled with strict deadlines to achieve each of them. Team leaders use the plan as a reference to monitor the development tasks during regular weekly progress meetings.

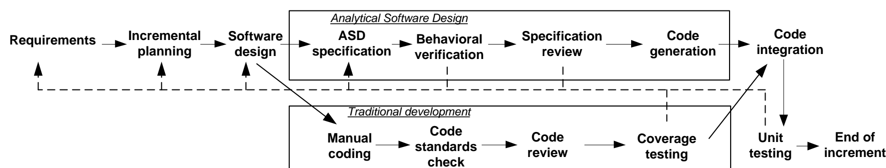

Fig. 7. Integrating ASD processes in a development increment

The actual building of software components begins with an approved design that includes components with well-defined interfaces and responsibilities. Such a design often results from iterative design sessions and a number of drafts.

When the intention is to use ASD, the design differentiates between control (state machines) and non-control components (e.g., data manipulation, computation and algorithms, (de-)serializing xml strings ..etc). Non-control components are developed using conventional development methods, while control components are usually constructed using ASD.

Non-control components are coded manually, so that checking coding standards is mandatory. Such a check is performed automatically using the TIOBE technology [17, 1]. After that, the code is thoroughly reviewed by team members before it becomes a target of coverage testing.

For coverage testing, development teams are required to provide at least 80% statement coverage and 100% function coverage for the manually written code, using the NCover technology [16]. Upon the completion of coverage testing, the code is integrated with the rest of product code, including the automatically generated code from ASD models. Formal verification in ASD takes the place of coverage testing, which is typically not necessary for the ASD generated code.

Then, the entire unit becomes a target of unit test, usually accomplished as a black-box. The entire code is then delivered to the main code archive, managed by the IBM clearcase technology [11], where the code is integrated with the code delivered by other team members responsible of developing other units. At the end of each increment developers solve problems and fix defects reported during the construction of the components.

Below we concentrate more on the design phase detailing steps of designing and constructing components of the Orchestration module using ASD.

5 Design of the Orchestration module

The Orchestration module was one of the first modules which were built using ASD. At that point in time the ASD tooling used to construct the models was still very immature and difficult to use. Apart from that the team members were new to ASD and were confronted with the steep learning curve although the ASD approach hides all formal details from end users.

There was a lack of design cookbooks, guidelines, design patterns or steps that help designers to not only design quality components but also more importantly to construct formally verifiable components using model checking. As a result the first version of the Orchestration module suffered from some problems. For example, some models were over-specified, too complex to understand and model checking took a substantial amount of time for verification. During a subsequent development increment we decided to refactor the module based on the knowledge gained.

The next section discusses the steps we took to get to a better (ASD) design. After that, we demonstrate peculiarities of design components that facilitate verifying them compositionally following the previously addressed ASD recipe (see Section 2).

5.1 Design Steps

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

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

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

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

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

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

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

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

During the creation of the ASD models, requirements were referenced in ASD tables using tags. Since ASD forces specification completeness, a number of new (missing) requirements were found. This revealed omissions and gaps in requirements early in the development process, before verification or even implementation.

5.2 The resulting ASD components

The final structure of the Orchestration components is depicted in Figure 8. Below we detail their peculiarities that effectively had helped verifying them compositionally in a reasonable time using the ASD:Suite.

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

The *Orchestration* controller state machine includes states that capture the overall modes of the system. That is, whether the system is busy activating, performing selection procedures, or performing image acquisition. The Orchestration controller, for instance, does not know which particular type of selection is performed but it knows that the selection procedure is active or has finally succeeded or failed. Detailed procedures of these phases are the responsibility of lower-level components. The same concept applies to all other modes, e.g.,

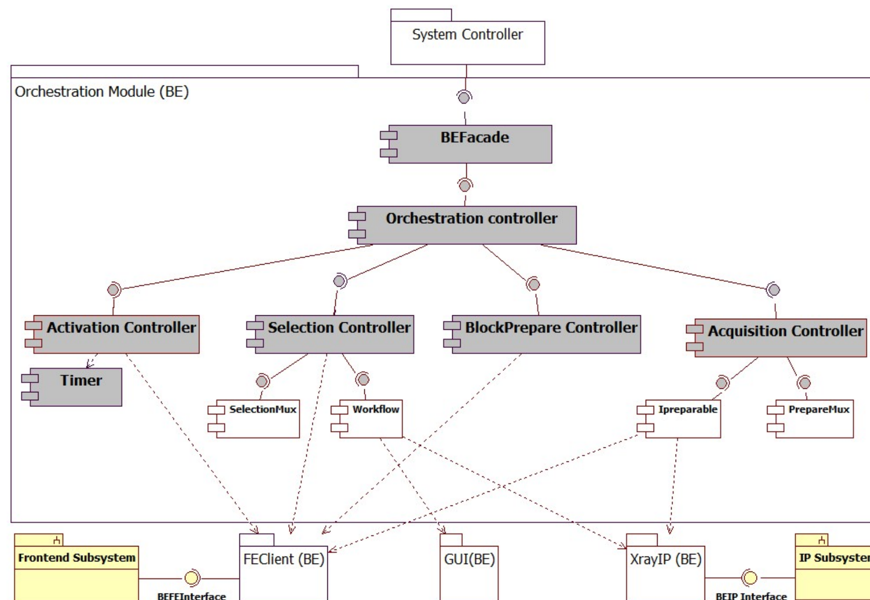

Fig. 8. Decomposition of Orchestration components

activation and acquisition. The Orchestration controller component mainly coordinates the behavior of the used components positioned at the lower-level, give permissions to start certain phase and ensures that certain procedures are mutually exclusive and run to completion. It also ensures that units are synchronized back to a predefined state when a connection with other subsystems is re-established (e.g., reselecting previously selected patient).

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

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

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

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

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

5.3 Constructing the ASD components following the ASD recipe

Components of the Orchestration module were realized in a mixture of top-down and bottom-up fashions. Each ASD design model is verified in isolation with the direct interface models of lower-level components, providing that these interface models are refined by corresponding design and other interface models. The compositional construction and verification is visualized in Figure 9 and is self-explainable. Both Orchestration and FEClient units were constructed concurrently. The FEClient team provided the IFEClient ASD interface model to the Orchestration team as a formal external specification describing the protocol of interaction between the two units, and the allowable and forbidden sequence of events crossing the boundary.

Table 1 depicts the final statistical data of components of the Orchestration module. It mainly shows that components were verified in a reasonable time using model checking. The first column lists the names of the components. The second column represents the total number of ASD models of each component, which presents the sum of one design model plus the interface models of the boundary ASD and non ASD components.

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

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

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

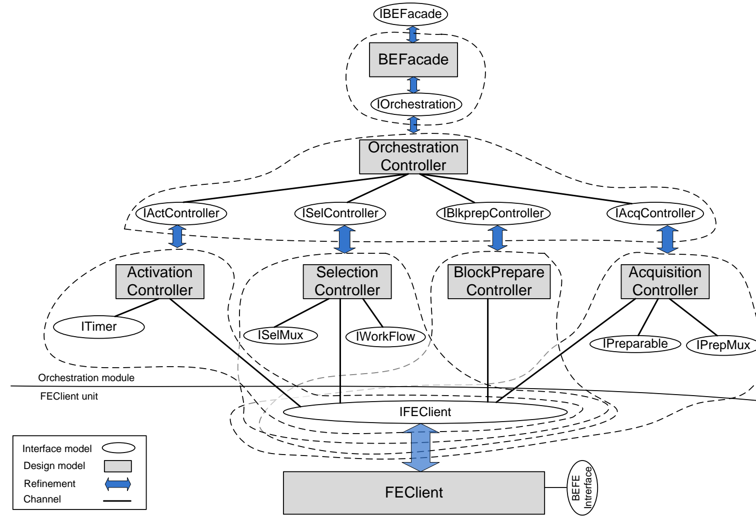

Fig. 9. Compositional construction and verification of components. Handwritten components are hidden.

Table 1. The ASD models of the Orchestration

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

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

Using this technology, the entire unit was systematically tested under a statistical quality control, based on usage models that specify the usage scenarios as state machines. The description of usage models is similar to the ASD tabular specification but extended with probabilities of usage. This revealed a few errors but most were not directly related to ASD code. For example, incorrect han-

dling of data in the manually written code and cases were usage models specify additional behavior not implemented yet by the Orchestration.

6 Quality results of the Orchestration module

The development activities of first three increments resulted in a release of the product to the market. The other three increments were devoted to extending the module with additional functionalities and new features. It is notable that components of the Orchestration were easy to maintain and to extend due to the high-level description of ASD specification, and the high abstract behavior of the components. In general, it was easy to adapt the models and generate new verified code.

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

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

The development team submitted detailed reports related to errors encountered along the construction of the Orchestration module. This includes errors found not only during testing but also during implementation and integration in case such errors hinder other teams developing other units. These defects were submitted to a defect tracking system, which is part of a sophisticated code management system.

We carefully investigated these defects trying to determine their impact on the end quality of the module and to recognize typical type of errors left behind by the ASD technology. Our analysis resulted in the followings.

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

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

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

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

After carefully analyzing the detailed reports of these defects we found that the errors were easy to find and to fix, not profound design or interface errors. Table 2 depicts a summary of these errors. The error severity codes are as follows.

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

We use extra codes to specify whether the error was caused/found during design ("D"), implementation ("I"), integration ("G") or system testing ("T").

Table 2. Summary of errors found during the construction of the Orchestration module

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

The development activities of the Orchestration module yield a total of 19,601 LOC, with an average rate of 0.4 defect per KLOC. This favorably compares to the standard of 1-25 defects per KLOC for software developed in industrial settings [12].

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

The quality of the manually coded components depends on many other factors such as code reviews, automatic code standard checks and coverage testing. Unit testing had provided key benefits of preparing coverage reports, detecting potential memory leaks and optimizing memory usage. The total number of test code written for the Orchestration module is 3966 lines of code.

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

Although there were some delays on the deliverable of the Orchestration module due to spending more time in learning ASD and obtaining verifiable design, there was less time spent in testing compared to the other manually coded modules of the same and other units. This at the end led to less time spent to resolve problems found in testing at later stages compared to manually coded modules [4].

Finally, feedbacks and comments from team and project leaders were very positive, and the module appeared to be stable and reliable. The module was robust against the increasing evolution and the frequent changes of requirements. Team members appreciated the end quality of the software, relating that to the firm specification and formal verification technologies provided by the ASD approach.

7 Conclusions

In this paper we reported about experiences with integrating formal commercial approach called ASD with traditional industrial practices. The approach was used for developing control components of a module called the Orchestration in a real industrial project at Philips Healthcare. We demonstrated how formal techniques of ASD were combined with the standard development processes in industry, highlighting some issues encountered during the design phase and the formal verification using model checking.

Subsequently, we provided steps followed for designing verifiable components and we showed that the resulted components were easy to verify using model checking. The peculiarities of such verifiable components were explained in more depth. Moreover, due to evolution of requirements components were robust against frequent changes and easy to maintain and modify because of the high-degree of abstraction of their behavior.

Finally, we demonstrated that the ASD technology eliminated errors earlier in the design phase and that the resulted quality of ASD components was remarkable. However, there were a few errors left behind by the technology but they were rather easy to find and fix. Feedback from project leaders was very positive and the module appeared to be stable and reliable. Although there were some delays in the deliverables of ASD components, there was less time spent for fixing errors at later stages of the project.

References

1. *Philips Healthcare - C# Coding Standard, Version 2.0.* `http://www.tiobe.com/content/paperinfo/gemrcsharpcs.pdf`, 2012.
2. G. H. Broadfoot. ASD case notes: Costs and benefits of applying formal methods to industrial control software. In *FM 2005: Formal Methods*, volume 3582 of LNCS, pages 548–551. Springer (2005), 2005.
3. FDR homepage. *http://www.fsel.com, 2012.*
4. B. Folmer. Personal communication (backend project leader). 2010.

5. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.
6. J. F. Groote, A. Osaiweran, and J. H. Wesselius. Analyzing a controller of a power distribution unit using formal methods. In *Proceedings of the Fifth International Conference on Software Testing, Verification and Validation (ICST 2012, Montreal, Canada, April 18-20, 2012)*, page (in press). IEEE.
7. J. F. Groote, A. Osaiweran, and J. H. Wesselius. Experience report on developing the front-end client unit under the control of formal methods. In *Proceedings of the 27th ACM Symposium on Applied Computing, The Software Engineering Track (ACM SAC-SE 2012, Riva del Garda, Italy, March 25-29, 2012)*, page (in press). ACM.
8. 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 International Conference on Software Maintenance (ICSM 2011)*, pages 467–472, 2011.
9. 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), Presymposium Proceedings*, pages 92–109. UNU-IIST Report 454, McSCert Report 5. http://www.iist.unu.edu/ICTAC/FHIES2011/Files/fhies2011_8_17.pdf.
10. P. J. Hopcroft and G. H. Broadfoot. Combining the box structure development method and CSP for software development. *Electr. Notes Theor. Comput. Sci.*, 128(6):127–144, 2005.
11. IBM ClearCase. *http://www-01.ibm.com/software/awdtools/clearcase/, 2012.*
12. S. McConnell. *Code Complete, Second Edition.* Microsoft Press, Redmond, WA, USA, 2004.
13. A. Osaiweran, M. Schuts, J. Hooman, and J. 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 31,2012)*, page (in press). Electronic Proceedings in Theoretical Computer Science.
14. S. J. Prowell and J. H. Poore. Foundations of sequence-based software specification. *IEEE Transactions on Software Engineering*, 29(5):417–429, 2003.
15. A. W. Roscoe. *The theory and practice of concurrency.* Prentice Hall, 1998.
16. The NCover home page. *http://www.ncover.com/, 2012.*
17. TIOBE homepage. *http://www.tiobe.com, 2012.*
18. Verum homepage. *http://www.verum.com, 2012.*