Evaluating the Effect of Formal Techniques in Industry

Ammar Osaiweran[1], Jan Friso Groote[1], Mathijs Schuts[2], Jozef Hooman[3], and Bart van Rijnsoever[2]

[1] Eindhoven University of Technology, Eindhoven, The Netherlands
[2] Philips Healthcare, BU Interventional X-ray, Best, The Netherlands
[3] Radboud University Nijmegen, Nijmegen, and Embedded Systems Institute, Eindhoven, The Netherlands
{a.a.h.osaiweran,j.f.groote}@tue.nl,{mathijs.schuts,bart.van.Rijnsoever}@philips.com,jozef.hooman@esi.nl

Abstract. In this paper we evaluate the effectiveness of applying a formal component-based approach called Analytical Software Design (ASD) to the development of control software of an industrial project at Philips Healthcare. We analyze the performance of the ASD related tasks carried out during the development processes and report about the main issues encountered. Furthermore, we investigate whether introducing these formal techniques to industry could actually improve the quality and the productivity of the developed code compared to software developed by more traditional development methods.

1 Introduction

Philips Healthcare develops a number of highly sophisticated medical systems, used for various clinical applications. One of these systems is the interventional X-ray (iXR) system, which depicted in Figure 1.

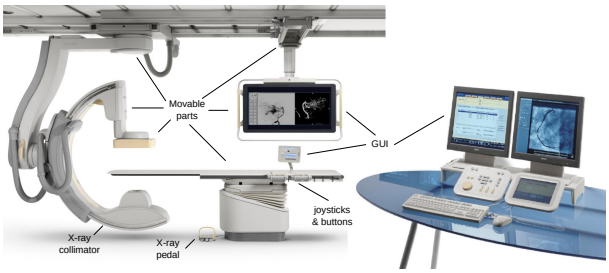

Fig. 1. An interventional X-ray system

The software practitioners at Philips Healthcare developing this type of systems are constantly seeking approaches, tools and techniques to advance current

software development processes. The purpose is to improve the quality of developed code, enhance productivity, lower development costs, shorten the time to market, and increase end-user satisfaction.

As one of these new technologies, Philips Healthcare introduced to its development context a model-driven, formal component-based development approach called the Analytical Software Design (ASD). The approach was exploited for defining formal interfaces and building mathematically verified software components. The ASD approach supports formal techniques by a commercial tool called the ASD:Suite, developed by the company Verum [29].

The focus of this paper is to investigate whether the use of these formal techniques actually resulted in visible improvements to the developed software, demonstrating key issues encountered when incorporating the techniques with the industrial development processes. The target of our investigation are the software components of a subsystem of the X-ray machine, called the *Frontend* subsystem.

The ASD technology was successful in other projects at Philips [23, 21, 24]. In this paper we investigate whether the approach is also successful in other projects with other environments, circumstances and backgrounds. Moreover, we provide a more detailed quantitative analysis regarding the effectiveness of these techniques in industrial practices.

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

- Can these formal techniques deliver product code? And if so, is the code of high or low quality?
- Do these formal techniques require more time in development compared to traditional development?
- What about the productivity using these techniques?
- Do the techniques require specialized mathematicians for a successful application?
- Do these techniques always produce zero-defect software? If not, which types of errors are expected and how many compared to industrial standards?
- Which artifacts should we consider when evaluating these techniques empirically? Should we include the formal models or the related code?

The paper is structured as follows. Section 2 briefly sketches the industrial context. In Section 3 the ASD approach is introduced to the limit needed in this paper. Section 4 details the phases of developing the ASD components and the main issues and limitations encountered. We analyze the data related to the developed code to evaluate the ASD approach in Section 5. In Section 6 we extend our analysis to study the cause and the type of errors that could escape the formal techniques. Section 7 details the end results comparing the ASD code with other code developed at Philips and also with the industry standards reported worldwide. Section 8 details a number of industrial projects that we found incorporated formal techniques in their software development.

Finally, Section 9 contains our conclusions by answering the research questions mentioned earlier.

2 Description of the project

The software that controls the X-ray machine is divided into a number of subsystems, including the Frontend (FE) subsystem. The FE is responsible for creating X-ray images by controlling and managing physical hardware, such as the X-ray generator, the X-ray detector, the table where patients can lay and the stand that holds the generator and the detector, as shown in Figure 1.

Previously, the FE subsystem was developed as a decentralized architecture in the sense that all units work on their own, observing changes of other units via a shared blackboard and react accordingly. The main shortcoming of this type of architecture was the difficulty of knowing the overall system state. More importantly, incorporating innovations or new products of third-party suppliers was very challenging.

Therefore, some of the units of the FE subsystem were redesigned in order to migrate to a new centralized, hierarchical component-based architecture, while others were kept intact and were reused in the new architecture (e.g., the units that control the hardware devices).

The FE subsystem includes 22 units, two of which are the target of this study: the Application State Controller (ASC) and the Frontend Adapter (FEA). Both units comprise a number of modules that includes concurrent components with well-defined interfaces and responsibilities.

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

The FEA unit is mainly responsible for the interfaces with other external subsystems, through a network. It exchanges information related to patients and their exam details with other external parties. The unit is also responsible for monitoring the presence of other remote subsystems and converting incoming information to readable xml and string formats.

The interaction of these units with other external parties is rather complex and error prone. Any party can issue requests or can enter a faulty state. For example, clinical users can press the pedals or the switches at any time, even if the internal components or the hardware are not prepared or configured yet for image acquisition.

Moreover, the FE may lose the connection with the external subsystems due to network outage or errors in the subsystems at any time. Therefore, the ASC and FEA units must always be robust against such situations and should provide safe and convenient behavior to the end-users. In order to guarantee correctness of the developed components, the software practitioners decided to employ the ASD technology aiming at detecting and preventing any potential

errors at early stages of the project and obtaining high quality software at the end of the project.

3 Use of formal methods

The ASD approach supports the development of formally verified software components by exploiting two types of models, both described as state machines and specified using a similar tabular notation: the interface model and the design model. We briefly describe these two types of models:

- The *interface* model is used as a formal means to describe not only the methods to be invoked on a component but also the external behavior. The interface model specifies the interaction protocol and the allowed or inhibited sequences of invoking these methods by clients. Any interactions with used components are not included in the model.
- The *design* model of a component refines and extends the interface model by more internal details. Usually, the design model uses the interface models of other ASD and non ASD components. These interfaces, in turn, are independently refined by other design models (perhaps by other teams), facilitating multi-site, parallel development of components.

In order to allow the ASD approach to scale to industrial applications, the approach is compositional [14] in the sense that components are verified in isolation. This means that the formal verification of a design model uses only the interfaces of the surrounding used components, without considering their internal implementation.

To ensure complete and consistent specifications, the models are described using the sequence-based specification technique [22]. Any ASD interface or design model includes a complete specification in the sense that responses to every possible input stimulus in every state must be described.

By translating the ASD tabular specifications to corresponding CSP models [28], which are checked by FDR2 [28, 9], the tool ASD:Suite can be used to formally verify that the external behavior captured by the interface model is formally refined by the corresponding design model. Moreover, the tool is used to verify a predefined set of properties such as absence of deadlock and livelock under all circumstances of use. ASD:Suite veils the CSP and the FDR2 details from end-users in order to enable practical industrial usage.

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

The ASD component may include a queue to store incoming callback events sent by used components, supporting an asynchronous communication mechanism. The queue runs in parallel (i.e., in a separate thread) with the design model and may cause interleaving with other components.

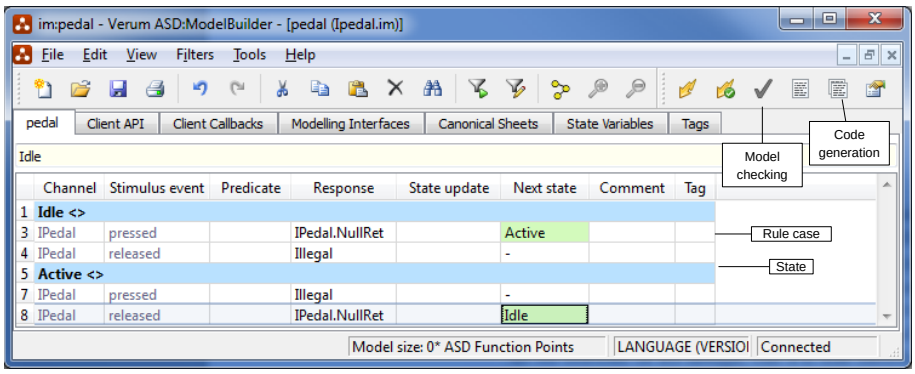

Fig. 2. An example of ASD specification

A vital and very attractive feature in the ASD:Suite is the support of a comprehensive code generation from formally verified design models to a number of programming languages (C, C++, C#, Java).

Figure 2 presents an example of a very small ASD interface model in the ASD:Suite application, describing the behavior of an X-ray pedal. The model includes two states, namely *Idle* and *Active*, each containing all input stimuli events listed in rows called rule cases. ASD practitioners are forced to fill in all rows of all rule cases for the sake of completeness. As can be seen, automatic verification and code generation can be established with the click of a button. The specification is straightforward and self explainable. Another more detailed example illustrating ASD can be found in [15].

4 The developed ASD components

The process of developing software used in the iXR projects is an evolutionary iterative process, i.e., the software is developed through successive increments, each of which requires regular acceptance and review meetings by several parties.

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

Along the three increments, the ASD formal technology was tightly integrated with the traditional development processes. Below we describe the main phases accomplished by the team.

Pre-study phase. The three increments were preceded by a pre-study period, during which the team attended a one week ASD course to get familiar with the approach and its related technologies. The course was limited to learn-

ing how to use the ASD:Suite (e.g., how to fill-in the tables and verify them compositionally using model checking and how to generate and integrate the code).

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

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

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

Since the object-oriented method was the dominating approach of design and implementation, team members started investigating the suitability of ASD for developing Object-Oriented designs. For this purpose, the team reviewed and thoroughly studied the well-known object-oriented design patterns [11], trying to model them alternatively using ASD.

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

Design phase. Based on the knowledge gained from the pre-study period, team members prepared initial design drafts containing hierarchical components with well-defined interfaces and responsibilities, without using object-oriented patterns. The designs were iteratively reviewed and re-factored until they were approved by team members. After that, design of components and their responsibilities were documented in informal documents.

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

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

Indeed, the technology consequently helped the team finding omissions and gaps in the initial set of requirements and the designs and hence initiated early

discussions with various stakeholders. Subsequently, this increased the quality of requirements and designs at early phases of development, and saved development time, comparing to addressing these issues at later stages of the project, using a more traditional development method.

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

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

Table 1 lists the developed ASD components for the ASC and the FEA units, demonstrating the number of ASD design and interface models for each component (column 3), and the sum of specified rule-cases (column 4). Each component includes one implemented interface model, one design model, and a number of used and implemented interface models.

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

Table 1. Modeling and verification statistics of the ASD components

Formal verification. Formal verification started with reviewing the ASD specifications, which were checked row-by-row for correctness and traceability to

the informal requirements. After specification reviews, the models were verified using model checking.

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

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

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

During formal verification, the team realized that different design styles could substantially influence the verifiability of components using formal techniques [25]. Hence, they avoided a number of design styles that may needlessly increase the state space and the time required for verification [15].

Based on the knowledge gained, the team could eventually obtain a set of formally verified components. Each ASD component was verified using the predefined set of ASD properties. Table 1, columns 5-7, includes the output data from FDR2 related to the refinement check (since it is most time consuming) for the design model of each ASD component, showing the states, transitions, and time in seconds. The data indicates that most of the components were verified within acceptable range of states and transitions, calculated in a reasonable time by FDR2. Note that, some components took less than a second for verification, covering all possible execution circumstances of the component.

Code generation and integration. As soon as ASD components had been formally verified and further reviewed to ensure that fixing the model checking errors did not break the intended behavior, the code was generated automatically and integrated with the code of other components via glue code. The last column of Table 1 quantifies the number of the effective lines of code (ELOC), automatically generated in the C++ programming language, excluding blank and comment lines.

Experience shows that, in a more conventional development method, integrating components is a nightmare, due to the substantial effort required to bring all components to correctly work together. Therefore, it was surprising that integrating ASD components with one another was always smooth, did not require any glue code, and often was accomplished without any errors. However,

integrating ASD code with the surrounding components that did not undergo formal verification (e.g., manually developed or legacy code) caused some errors and delays.

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

- The generated code is readable since it is constructed using well-known patterns, such as the object-oriented *State* and *Proxy* patterns;
- The generated code of all components has the same coding shape and structure, following similar coding standards;
- The generated code does not contain ad-hoc solutions, workarounds or tricks;
- The structure of the code allows systematic translation to other languages or other type of models, if needed;
- Changes are done at the model level, not at the low-level code;
- The code is thoroughly verified using model checking; previous and current experiences [23, 24, 21] indicate that errors left behind are simple to find and to fix;
- The support for different programming languages makes models more platform-independent than hand-written code.

Testing. An apparent advantage of ASD is that development time is shortened since white-box testing of the generated code is excluded due to the formal verification using model checking.

But, an apparent limitation of the ASD compositional verification is that it is impossible to formally establish whether the combination of ASD components yields the required behavior. It is not possible to express domain specific properties or to relate events in implemented and used interfaces.

Therefore, the ASC and FEA units were tested as a black-box. Furthermore, at the end of each increment, the FE units including the ASD components were thoroughly and extensively tested by a specialized test team, using various types of testing such as model-based statistical test, smoke test, regression test, performance test, etc, of which details are outside the scope of this paper. As a result of testing, a few errors were detected; details will be given in subsequent sections.

5 Data analysis

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

To accomplish this goal, we took several steps. We started with collecting the total number of effective lines of code that had been newly introduced plus

the changed legacy code, for every unit separately. We restricted ourselves to the period bounded by two baselines representing the start and the end of the three increments.

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

Table 2 depicts the results of our data collection, showing only a representative subset of the FE units. The units that exhibit similar results or were not changed during the increments have been excluded for readability purposes. Hidden from the table is also the amount of reused or legacy code, developed and verified during previous projects.

Given the obtained data, we could estimate the overall defect density of each unit separately, as depicted in the last column of Table 2. Although the ASD units appeared to be slightly better than some other units, at that stage of the analysis process, the effectiveness of the ASD formal techniques were not very conclusive and we felt that with a more refined analysis more insight can be obtained. Note that some of the manually coded units exhibit zero defects, but the reason was that most of the changes were on the level of interfaces and not on the core internal behavior of the units.

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

Table 2. Statistical data of representative units of the FE

Therefore, we decided to separate the ASD code and the handwritten (HW) code and analyze them in isolation. The ASD code and the manually written code are quantified in the second and third column of Table 2. Then, we studied the defects of ASD units once more, distinguishing ASD defects from those related to the handwritten code, as listed in the fourth and fifth columns. Consequently, we could estimate the defect rate of ASD and non ASD code, as depicted in columns six and seven.

As can be inferred from the table, for the two units that incorporate ASD, the quality of ASD code seems to be better than the corresponding manually written

code, especially for the FEA unit, which received only one defect. After studying the corresponding defect report we found that the error was not only related to ASD components but rather to a chain of ASD and non ASD components, due to a missing parameter in a method. Nevertheless, developers of the FEA unit, clearly, could deliver close to zero defects per thousand ASD lines of code.

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

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

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

Most of the detected errors were due to experiencing unintended, unexpected behaviors (e.g., after a user presses and releases a number of pedals and switches it was expected that a particular type of X-ray resumes but it did stop). But, as a result of the fixed set of properties the ASD technology currently supports, none of these errors was due to deadlocks or illegal interactions (e.g., there were no crashes due to null reference exceptions or illegal invocation of methods at some states).

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

6 Analyzing the cause of ASD errors

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

Initially, this appeared to be challenging since we did not possess any systematic means to measure the complexity of components at the model level.

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

Table 3. Statistical data of ASD components

Therefore, we alternatively assessed the components using two other means. First, we evaluated the models concerning the understandability and the review easiness of the models, based on our "common sense". Second, we chose to systematically analyze the generated code, using available code analysis tools and techniques. The two steps are detailed below.

In the first step, we evaluated the readability of the design model of each component, and assigned review codes based on the degree of complexity in reviewing and comprehending the models: VE= Very easy, E=Easy, M= Moderate, C= Complex and VC= Very Complex. Table 3 column 2 includes the result of the assessment.

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

A sample of a complex specification of rule-cases in the *RunController* design model is depicted in Figure 3. Visible in the figure are only 5 rule-cases related to the *FailedSC* stimulus event, which had been duplicated 31 times with different combinations of predicate values in the original model.

On the other hand, the user-guidance *UGStateless* component of the FEA unit is considered to be very easy since it contains only two states without the use of any predicates. The component is enabled or disabled to allow the flow of information traffic to other components. Although the component is easy

	Channel	Stimulus event	Predicate	Response
189	AcquisitionController:AcquisitionControllerCB	FailedSC	eExpReq == none and bDeactivating == false and bRunCondition == false and bAwaitFluoReleased == true	Null
190	AcquisitionController:AcquisitionControllerCB	FailedSC	eExpReq == none and eFluoReq == start and bDeactivating == false and bRunCondition == false and bAwaitFluoReleased == false and bAllExpReleased == true	AcquisitionController:AcquisitionCo...
191	AcquisitionController:AcquisitionControllerCB	FailedSC	eExpReq == none and eFluoReq == start and bDeactivating == false and bRunCondition == false and bAwaitFluoReleased == false and bAllExpReleased == false	Null

Fig. 3. An example of complex rule-cases

to read and to understand, it was the most time-consuming component when verified using model checking, as can be seen in Table 1. The reason is that the component receives a large number of callback events. Since they are stored in a queue, FDR2 took substantial time to calculate all possible orders these callbacks may take.

As a next step, we distributed the errors to the respective components, as depicted in Table 3 column 8. As can be seen, most of the ASD errors reside in the *RunController* component, unveiling an apparent correlation between the complexity of the component and the errors found.

In the second step, we performed a static analysis of the generated code, seeking similar correlations between complex code and the error density. The motivation was that the complexity of the models can also be reflected in the corresponding generated code. We used the *SourceMonitor* tool Version 3.2 [13] to analyze the generated code since the features catered by the tool seemed to be a good fit to our aim.

Table 3 includes some selected code metrics produced by the tool: the average number of methods per class (Avg M/C), the average statements per method (Avg S/M), the maximum cyclomatic complexity (Max CC), the average block depth, and the average cyclomatic complexity (Avg CC).

As can be seen in the table, the *RunController* component also appears to be very complex compared to other generated code of other components. Notable is that the 157 max complexity of the *RunController* component resides in the corresponding code of the rule-cases of the *FailedSC* stimulus event presented earlier in Figure 3. In the code, the rule-cases are represented by a single method (called *FailedSC*) containing 30 related if-else statements.

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

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

7 Quality and performance results

In this section, we evaluate the end quality and productivity of the developed ASD units, by comparing them against the industry standards reported worldwide in the literature. The best sources we could find are [16, 18, 20, 19], where interesting statistics related to a number of projects of different types and sizes are thoroughly described. We concentrate more on those statistics revealed for software systems analogous to the Frontend.

In [26], Linger and Spangler compared the quality of code developed under the Cleanroom software engineering formal method to the industry standard of 30-50 defects per KLOC. Jones in [16] presents an average of 1.7 coding errors per function point (p. 102, Table 3.11), which roughly corresponds to a range of 14-58 defects per C++ KLOC (after consulting Table 3.5 on p. 78 of [16]).

Furthermore, McConnell presents in [19] (page 242, Table 21-11) a breakdown of industry average defect rate based on software size, where our type of software is estimated to include 4-100 defects per KLOC. In [18] McConnell explicitly states an industry average of 1-20 defects per KLOC during the construction of software (p. 521), and also mentioned a range of 10-20 defects per KLOC, in the Microsoft Applications Division, during in-house testing. McConnell classifies the expected error density based on the project size, where our system is expected to include 4-100 errors per KLOC (p. 652, Table 27-1).

At Philips Healthcare, project and team leaders are often concerned with delivering features and function planned and estimated at the start of the incremental development, and not the size of the delivered code. However, the corresponding delivered code should exhibit 6 allowable defects per KLOC with an average productivity of 2 LOC per staff-hour, at the end of each increment. Any code that includes more errors, during the in-house construction, can be rejected and sent back to the developers, but this rarely happened.

From the data presented earlier in Table 2, we conclude that compared to the industry standard introduced earlier the ASD technology could deliver quality code, averaging the ASD code of the ASC and FEA units to only 0.36 defects per KLOC. The entire code of the two units reveals an average of 0.86 defects per KLOC.

Similar to comparing the quality of the units, we compare the productivity in terms of the number of lines of code per staff-hour. McConnell in [18] (p. 522) confirms that it is cheaper and better to develop high-quality software than it is to develop and mend low-quality software, so that it was of no surprise that a formal Cleanroom project could deliver nearly 5.61 LOC per staff-hour [27]. He also mentioned an industry average of 250-300 LOC per work-month (1.9-2.3 LOC per staff-hour), including all non-coding overhead.

Furthermore, McConnell in [18] (p. 653 Table 27-2) lists the expected productivity based on the size of the software product. Given these statistics, the productivity of software similar to the Frontend subsystem ranges between 700 to 10,000 LOC per staff-year with a nominal value of 2,000 LOC per staff-year (i.e., 0.4 to 6.3 with a nominal value of 1.3 LOC per staff-hour).

In [16], Jones presents a productivity figure of 435 C++ ELOC per staff-month (page 73, Table 3.4), which is equal to 3.3 ELOC per staff-hour. Furthermore, he provides figures for the average and best practices for systems software (p. 339, Table 9.7). There, Jones presents a 4.13 and 8.76 as an average and best-in-class function points per staff-month (which is equal to 1.7 and 3.5 as an average and best-in-class LOC per staff-hour, after consulting Table 3.5 on p. 78).

Cusumano et al., in [8] studied the data of a number of worldwide projects, and found a median of 450 LOC per staff-month (3.41 LOC per staff-hour) for the data sample related to the Japanese and European projects. The projects include roughly 48 percent generated code.

Consequently, we can use the above measures to compare the productivity of ASD developed units. The total time spent for developing the ASD components is 2378 hours, affording an average of 16 ELOC per staff-hour. The total time spent for developing the two units, including the time spent for non-coding overhead, is 5701 hours, which favorably yields 9.6 ELOC per staff-hour.

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

8 Other formal techniques used in other projects

In this section we present a number of worldwide industrial projects that incorporated formal techniques in software development and report about their achieved quality and productivity. We considered the work accomplished in [30] and its references (over 70 publications) as a starting point to seek these projects (the work includes a survey and a comprehensive review of formal methods application in industry). Furthermore, we searched other projects using web search engines and by visiting a number of home pages hosting the formal techniques.

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

Table 4 summarizes the results by listing 14 projects that fit our goal. The projects are listed in a chronological order, highlighting the used formal technique, the size of the developed software, the programming language used for

implementation, the defect density, the productivity in terms of the lines of code produced per staff-hour, and the phase where the errors were counted.

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

Table 4. List of projects incorporated formal techniques in software development

Linger in [27] listed 15 projects where the Cleanroom formal engineering method was used, summarizing the results achieved for each project. All developed systems exhibit quality figures that range between 0 to 5 errors per KLOC with an average of 3.3 errors per KLOC. Compared to the mentioned range of 30 to 50 errors/KLOC in traditional development, Linger concluded that the developed systems present remarkable quality.

From the 15 projects, three projects that reveal quality and productivity figures are depicted in Table 4. The other remaining projects do not include productivity figures so they were excluded from our consideration.

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

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

The third Cleanroom project included 50 people, developed complex system software at IBM using various programming languages. The system exhibited 2.6 errors/KLOC, where five of its eight components experienced no errors during testing. The team used the Cleanroom method for the first time [12].

The MaFMeth project [5] incorporated VDM and the B method in software development. The project included 3500 lines of C code, estimated by the developers from 8000 lines of generated code. The reported errors were found during unit testing. Errors found during validation testing or errors found after releasing the system were not available. Productivity was 13.6 LOC per hour with an error density of 0.9 error per KLOC.

The B Method was used to develop safety critical components of the automatic train operating system, the metro line 14 in Paris [1, 4]. Members of the development and validation teams were newcomers in formal methods, but were supported by B experts, when needed. The developed components included 86,000 of mathematically verified Ada code and the system did not experience any error during independent testing or after release. However, the numbers regarding the effort spent for the entire development were missing except for the correctness proofs. Nevertheless, the project was completed successfully and went off according to the schedule [4].

The DUST-EXPERT project incorporated VDM to software development and was successfully released with 15.8 KLOC of prolog and 17.5 KLOC of C++ [7]. The system exhibited less than one error per KLOC. The errors were found during coverage testing and after product release. Productivity was above industry norms but there were no figures provided in the paper. Productivity of Prolog was less than C++ due to the high-abstract level and the rigorous way the core Prolog was generated. Developers involved were skilled in formal methods.

The Abstract State Machines (ASM) were used in the development of a software package, developed at Siemens, called FALKO [6]. The package was redesigned from scratch due its complexity. The newly developed package included roughly 11900 generated and manually written C++ LOC, developed in nearly 66 man-weeks effort. Two errors were found after product release and were fixed directly in the generated code. The end quality was 0.17 and the productivity was 2.2 LOC per hour.

In [17], VDM was used to formally develop some components of the VDM toolset itself. Table 4 includes some metrics related to the VDM-C++ code

generator component, which was formally specified using VDM but manually implemented using C++. The productivity was 12.4 LOC per hour but compared with other components the productivity was less due to its complexity and the involvement of new employees. No figures related to the errors found were reported.

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

The Tokeneer ID Station (TIS) project was carried out by Praxis High Integrity Systems and accomplished by three part-time members over one year using SPARK [3]. The overall productivity of the TIS core system was 6.3 LOC of Ada per man-hour. The system did not exhibit any error whatsoever during reliability testing and also since delivery.

The B Method was successful in developing software components of a driverless shuttle at Paris Roissy Airport [1, 2]. The developed software included 158 KLOC of generated code. The generated code includes lots of duplications due to the lack of sharing in the code and the intermediate steps performed by the code generator. The code is estimated to be 60 KLOC in size after tuning. However, there was no data available about the total time spent in development or the number or type of errors encountered along the construction of the software.

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

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

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

9 Conclusions

In this paper we demonstrated a formal component-based approach called ASD and how its formal techniques were exploited for developing control components of two software units of an X-ray machine, developed at Philips Healthcare. We elaborated more on the issues encountered during its application. The result of our investigation shows that the ASD technology could effectively deliver quality

software with high productivity. Below, we answer the questions raised in the introduction.

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

We highlighted the benefits of the ASD generated code. Compared to the manually written code, the generated code is easy to read and understand. It follows the same coding standard and is implemented using well-known, highly recommended object-oriented patterns. Any highly skilled programmers may use the same patterns if the state machines were manually implemented. However, a corresponding manually written code may contain less lines of code compared to the generated code but this depends on the quality of the programmer. The main concern here is that the generated code exhibit fewer defects since it is formally verified. Furthermore, the productivity is high due to the absence of integration and testing efforts.

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

What about the productivity using these techniques? As can be inferred from the presented data, there is some indication of improved productivity compared to industrial standards. This resulted from the fact that developers were only concerned with models, from which verified code is generated automatically with the click of a button, and hence reducing implementation overhead. Another important fact is that less or even no time was spent for integration and manual testing, which are usually time consuming and uncertain. The time devoted to bug fixing at the end of development is also reduced.

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

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

saw before, although the components were formally specified and verified, still errors were found during testing. Hence, the used techniques do not always lead to defect-free software. However, our study shows that the formally developed software contains very few defects.

Which artifacts should we consider when evaluating these techniques empirically? Should we include the formal models or the related code? Evaluation requires a product in hand to be analyzed so that a challenging task was analyzing the product and the complexity at the models level. As there was no systematic means to analyze the models, we analyzed the corresponding code. We found that complex models do not necessarily produce huge state space but they may be error prone. An interesting future direction is to develop tools and techniques for establishing static analysis of models.

References

1. J.-R. Abrial. Formal methods: Theory becoming practice. 13(5):619–628, may 2007. `http://www.jucs.org/jucs_13_5/formal_methods_theory_becoming`.
2. F. Badeau and A. Amelot. Using b as a high level programming language in an industrial project: roissy val. In *Proceedings of the 4th international conference on Formal Specification and Development in Z and B*, ZB'05, pages 334–354, Berlin, Heidelberg, 2005. Springer-Verlag.
3. J. BARNES, R. CHAPMAN, R. JOHNSON, J. WIDMAIER, D. COOPER, and B. EVERETT. Engineering the tokeneer enclave protection system. *In Proceedings of the 1st International Symposium on Secure Software Engineering*, 2006.
4. P. Behm, P. Benoit, A. Faivre, and J.-M. Meynadier. Meteor: A successful application of b in a large project. In *Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems-Volume I - Volume I*, FM '99, pages 369–387, London, UK, 1999. Springer-Verlag.
5. J. Bicarregui, J. Dick, and E. Woods. Quantitative analysis of an application of formal methods. In *Proceedings of the Third International Symposium of Formal Methods Europe on Industrial Benefit and Advances in Formal Methods*, FME '96, pages 60–73, London, UK, 1996. Springer-Verlag.
6. E. Börger, P. Päppinghaus, and J. Schmid. Report on a practical application of asms in software design. In *Proceedings of the International Workshop on Abstract State Machines, Theory and Applications*, ASM '00, pages 361–366, London, UK, 2000. Springer-Verlag.
7. T. Clement, I. Cottam, P. Froome, and C. Jones. The development of a commercial "shrink-wrapped application" to safety integrity level 2: The dust-experttm story. In *Proceedings of the 18th International Conference on Computer Computer Safety, Reliability and Security*, SAFECOMP '99, pages 216–225, London, UK, 1999. Springer-Verlag.
8. M. Cusumano, A. MacCormack, C.F. Kemerer, and B. Crandall. Software development worldwide: The state of the practice. *IEEE Softw.*, 20(6):28–34, Nov. 2003.
9. FDR homepage. *http://www.fsel.com, 2011.*
10. J. Fitzgerald, P. Larsen, P. Mukherjee, N. Plat, and M. Verhoef. *Validated Designs For Object-oriented Systems.* Springer-Verlag TELOS, Santa Clara, CA, USA, 2005.

11. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.
12. P. A. Hausler. A recent cleanroom success story: The redwing project. In *Seventeenth Annual Software Engineering Workshop*, NASA Goddard Space Flight Center, Greenbelt, MD, December 1992.
13. S. homepage. *http://www.campwoodsw.com/sourcemonitor.html*.
14. J. Hooman. *Specification and Compositional Verification of Real-Time Systems*, volume 558 of *Lecture Notes in Computer Science*. Springer, 1991.
15. J. Hooman, R. Huis in 't Veld, and M. Schuts. Experiences with a compositional model checker in the healthcare domain. In *FHIES 2011*, pages 93–110. LNCS 7151, Springer-Verlag, 2012.
16. C. Jones. *Software assessments, benchmarks, and best practices*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.
17. P. Larsen. Ten years of historical development "bootstrapping" VDMTools. *J. UCS*, pages 692–709, 2001.
18. S. McConnell. *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA, 2004.
19. S. McConnell. *Software Estimation: Demystifying the Black Art*. Microsoft Press, Redmond, WA, USA, 2006.
20. H. Mills. Certifying the correctness of software. In *Proceedings of the 25th HICSS*, pages 373 – 381, Hawai, HI, 1992.
21. A. Osaiweran, M. Schuts, J. Hooman, and J.H. Wesselius. Incorporating formal techniques into industrial practice: an experience report. In *Proceedings of FEASCA 2011 Workshop*, page (In press), Tallinn, Estonia, March 31, 2012. EPTCS.
22. S. J. Prowell and J. H. Poore. Foundations of sequence-based software specification. *IEEE Trans. on Soft. Eng.*, 29(5):417–429, 2003.
23. J.F. Groote, A. Osaiweran, and J.H. Wesselius. Analyzing the effects of formal methods on the development of industrial control software. In *ICSM 2011*, pages 467–472, 2011.
24. J.F. Groote, A. Osaiweran, and J.H. Wesselius. Experience report on developing the front-end client unit under the control of formal methods. In *Proceedings of the 27th ACM SAC-SE*, page (In press), Riva del Garda, Italy, March 25-29, 2012. ACM.
25. J.F. Groote, T.W.D.M. Kouters, and A. Osaiweran. Specification guidelines to avoid the state space explosion problem. In *FSEN*, pages 112–127, 2011.
26. R.A. Sprangler and R.C. Linger. The ibm cleanroom software engineering technology transfer program. In *Proceedings of the SEI Conf. on Soft. Eng. Edu.*, pages 380–394, London, UK, UK, 1992. Springer-Verlag.
27. R.C. Linger. Cleanroom software engineering for zero-defect software. In *Proceedings of ICSE 1993*, pages 2–13, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.
28. A. Roscoe. *Understanding Concurrent Systems*. Springer, 2010.
29. Verum homepage. *http://www.verum.com, 2011*.
30. J. Woodcock, P. Larsen, J. Bicarregui, and J. Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys*, 41(4):1–36, 2009.