Benefits of Applying Formal Methods to Industrial Control Software

J.F. Groote[1], A.A.H. Osaiweran[1], and J.H. Wesselius[2]
[1] Eindhoven University of Technology, Eindhoven, The Netherlands
[2] Philips Healthcare, CardioVascular X-ray, Best, The Netherlands
{j.f.groote, a.a.h.osaiweran}@tue.nl, jacco.wesselius@philips.com

Abstract

Formal methods are being applied to the development of software of various applications at Philips Healthcare. In particular, the Analytical Software Design (ASD) method is being used as a formal technology for developing defect-free control software of highly sophisticated X-ray machines. In this paper we analyze the effects of applying ASD in the development of various control software units. We compare the quality of these units with other units developed in traditional development methods. The results indicate that applying ASD as a formal technology for developing control software results in better quality code.
keywords: Formal Methods; Analytical Software Design; Model Checking; Software Quality

1 Introduction

In industrial systems control software is becoming increasingly complex with more concurrency playing a crucial role. In conventional software development of such type of systems, errors are considered as inevitable. Techniques for early defect prevention are widely encouraged as software practitioners are pushed to get software into execution quickly on tight schedules.

Establishing the correctness of these systems is widely known to pose serious challenges for traditional testing techniques, used by conventional design development methods. Selective test cases are invented with prior awareness of code internals, often done by the code developers themselves or specialized test personnel, mainly to cover key functions, error cases, etc. On completion of testing, software is known to pass certain tests, but can still fail for cases not tested.

It is claimed that formal methods allow the development of complex software under a firm mathematical foundation resulting in high quality, more correct software compared to conventional design methods. For example, model checking techniques have been widely applied to the

verification of discrete behavior of various industrial critical systems [11, 13]. Virulent concurrency errors have been discovered that would not have been unveiled through traditional testing. In some circumstances these uncovered errors caused serious damage or loss of property [8].

For the purpose of obtaining high quality software, Philips Healthcare is extensively investigating and applying formal methods in the development of its safety critical software components. More precisely, Philips Healthcare incorporates the Analytical Software Design[1] (ASD) method [1] to the development of various software components of X-ray machines.

The ASD method employs state machine models to formally specify and verify behavior of systems. From these models, source code can be generated automatically. When ASD models have been formally verified, the code generated from such models is considered to be correct meaning a.o. that sets of components match their prescribed interfaces. ASD employs a design method that mitigates the state space explosion problem by compositionally designing and verifying components in isolation.

Analyzing the quality effects of applying formal technologies to large-scale systems is a barely addressed issue. The best we could find is [2, 16], where it is claimed that near-zero defects can be obtained compared to traditionally developed software.

The purpose of our study is to carefully analyze the effects of formal methods on the quality of developed software providing third-party evaluation. The target of this study is the software of a complex X-ray machine. To accomplish this aim we compare the defect rates of a number of software units that incorporate formal methods with others developed using conventional methods. For each unit we carefully analyze every defect submitted along the development of the unit.

As we will see the results may appear incredible since the widespread view in industry is that applying formal mathematical methods on sizable software products is impractical. The results indicate that better quality software can be obtained from formal technologies compared to software developed by traditional development methods. This paper is arranged as follows. Section 2 sketches the basic concepts of ASD. In Section 3 we show how ASD is being applied in the development of various software units. We compare the effectiveness of applying ASD in Section 4.

2 Principles of Analytical Software Design

ASD is a component-based, model-driven technology that combines the application of formal mathematical methods such as Sequence-Based Specification (SBS) [10], Communicating Sequential Processes (CSP) [14] and the model checker Failure Divergence Refinement (FDR) [4] with software development methods such as Stepwise Refinement, and Component-Based Software Development [3].

A fundamental principle of ASD is to consider a software design as interacting components, communicating with one another or their environment via channels. As a common practice in ASD, system functionality is decomposed into components in levels (e.g., hierarchical structure)

[1] Supplied by Verum Software Technologies B.V., the Netherlands, www.verum.com.

to systematically develop and verify these components in isolation. For example, Figure 1 at the left depicts a hierarchal distribution of system components that include a controller (*Ctr*), a sensor and a lock.

Developing any ASD software component typically requires two models: an interface model and a design model. The interface model specifies the external behavior of the component, whereas the design model describes the concrete behavior. Both interface and design models are state machines described in a tabular format, see Figure 1.b, which depicts the specification of the Sensor interface model, described using the ASD industrial tool, called the ASD ModelBuilder.

To ensure correctness and consistency, formal mathematical models such as CSP [9] and source code implementation such as C++ or C# (following the state machine pattern in [6]) can be generated automatically from ASD models. The details of such translations are omitted here as they are not relevant for this article.

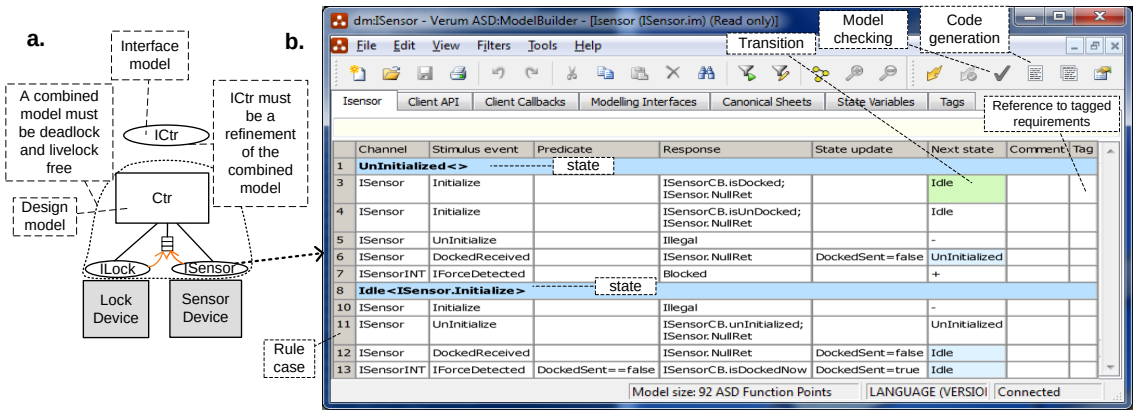

Figure 1: a. Design and interface models. b. The ASD ModelBuilder

The objective of incorporating model checking in ASD is that, unlike testing, model checking is comprehensive, and can cover all possible execution scenarios. Unlike conventional verification, it is automatic, as the model checking tool requires no human intervention. Such verifications can be completed in a day's effort. Verification and code generation of these models are done automatically with the click of a button.

Testing is not carried out for code generated from ASD models. Traditional testing such as function and statement coverage is performed for the handwritten part of the unit. The complete unit is further tested as a black box before the code is delivered to the system.

Below we summarize the steps required for developing an ASD component, given a structure of components. We consider the *Ctr* component from Figure 1 as an example.

1. *External behavior specification.* First, the interface model of a component under development is specified, such that it describes the external behavior exposed to its clients. All interactions with used components located at a lower level are not included in the specification. For example *ICtr* is the interface model of the forthcoming *Ctr* component, where interactions with the lock and the sensor components are not present.

2. *External specification of boundary components.* Similarly, the interface models of components located at the lower level are created. They describe also the external behavior exposed to the component being developed. For instance, the *ILock* and *ISensor* interface models describe the external behavior exposed to the *Ctr* component. All other internal interactions at lower levels not visible to *Ctr* are ignored.

3. *Concrete, functional specification.* After that, a design model of the component is created. The concrete behavior of the component is described including the interaction with used components. For example the *Ctr* design model includes method invocations from and to the lower level *Lock* and *Sensor* components. Invoked methods might supply data in their parameters. This data is not checked in the behavioral verification.

4. *Formal behavioral verification using model checking.* In this step CSP processes can be generated from the interface and design models constructed previously. A combined model that includes the parallel composition of the design model plus the interface models of the used components is generated automatically. The model is checked for deadlock, livelock, and illegal invocations using FDR; these checks are generated automatically using the ModelBuilder. Additional properties can be specified in CSP and verified against the combined model if required.

5. *Formal refinement of external and internal specifications.* The combined model must be a correct refinement of the interface model of the component being developed because the interface model is used by the client components. The formal refinement check is established using the failure or failure-divergence refinement supported by FDR, where the interface process is the specification and the combined model is the implementation. When the formal refinement check is accomplished, the interface model represents all lower level components.

6. *Code generation.* In this step source code is generated and integrated with the rest of the system in the target programming language.

7. *Recursive development of components.* For each component at a higher or lower level the steps 1 to 7 can be repeated until the system is completed. This provides the possibility to develop components in a top-down, middle-out, or bottom-up fashion, in parallel with developing some manually coded modules.

3 The application of ASD in software development

Philips Healthcare incorporated the ASD technology in the development of control software at the end of 2006. Initially, the technology was used to formally specify and verify protocols of interactions among internal interfaces of subsystems of an X-ray machine. One of the primary subsystems incorporating ASD is Back-end Xray (BeX) [18, 15, 17].

Below we report about two consecutive projects of BeX starting from January 2008 till the end of 2010. The projects include a total of 36 software designers, architects, and engineers,

of which 9 attended ASD training courses. ASD imposes a learning curve, and therefore extra efforts and investments are required before reaping its benefits. At the earlier stages of applying ASD, four part-time ASD consultants were present, devoted approximately half of their time helping developers to quickly learn the technology.

In this section we sketch how ASD has been incorporated in the development process of several software units of BeX, highlighting the flow of events followed during the project.

3.1 Incorporating ASD to the development of BeX

Software units were developed in a series of consecutive increments, each of which included the implementation of a subset of user functions. Since ASD comprises formal technologies, incorporating the method requires certain adaptations to the traditional development process. Figure 2 depicts the flow of ASD events in a development increment. Note that these steps are preceded by brainstorming sessions where team members explore several design alternatives without being precise.

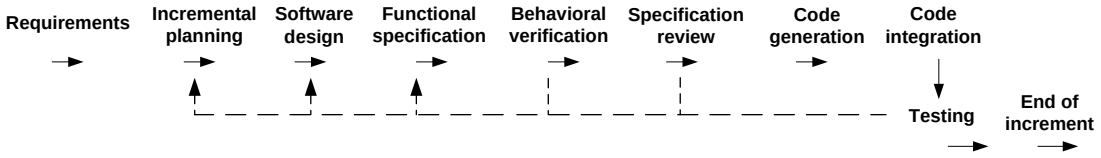

Figure 2: The ASD processes in a development increment

Requirements. This step included the definition of the requirements for function, reliability, performance, characterization of usage conditions, target programming language for code generation, and the operating system.

Incremental planning. In this step functions to be implemented through each increment were selected, with established work breakdown estimations and a tight schedule. For each function to be implemented the time, efforts, deadlines, risks, etc., were clearly identified.

Software design. In this step the distribution of components was accomplished, with well-defined responsibilities and interfaces. Designs of software components commenced as working drafts until team reviews had been accomplished, and design improvements resulting from each team review session were incorporated.

The effort of obtaining a suitable ASD architectural design for some units was higher than normal since ASD does not support all design or architectural patterns, with which the developers were acquainted. For example, the technology is hardly suitable for modeling the object-oriented design patterns [6], and ASD components must strictly be distributed in levels, where a component is only allowed to communicate with others located at the direct subsequent level.

Functional specification. In this step, each ASD component under development was specified in isolation following the ASD recipe. The external and concrete behavior of each component was described using the ASD ModelBuilder. Whenever a design did not suit the ASD specification or verification, the structure of the software was adapted.

Behavioral verification. For each unit, the behavioral verification using model checking was done in a component-wise manner. Race conditions, deadlocks, livelocks, and illegal interactions violating the interaction protocols were discovered, causing adapting the behavioral model or redesigning the software design.

It is notable that the state space explosion kicked in during verification of various components. We learned that alternative designs can help to avoid this problem and make verification doable [7]. In some cases, the explosion of states of a complex component was circumvented by decomposing the component further into a number of smaller components.

Specification review, code generation, and code integration. The specification of all ASD models had to be reviewed by team members, row-by-row, for traceability and correctness against the requirements. Once verification was completed, the design models were automatically translated into the target language, in this case, C#. Changes to generated code were not permitted. The generated code was integrated to the rest of the product code by implementing glue code of proper adapters and wrappers. Integration of the code of ASD components was always smooth with no error ever reported. Integration errors occurred when integrating ASD code with the manually developed code. Other errors were due to the data part of the generated code which was not formally verified.

Testing. Since code generated from ASD models was already verified using model checking, the code was not a target of function coverage or statement coverage tests, which applies to all manually written code of each software unit. Unit testing was started after the generated code was integrated with the manually written code. The units were further examined using statistical testing, supplied by the ASD method, for certifying compliance of software components.

End of increment. This step was mainly devoted to solving problems and fixing defects raised during the development of the units. Few defects related to the ASD code were committed. After a careful analysis of the cause of these defects we found that the main source was the data part of the code. Correctness verification of data is not supported by ASD at the moment of writing this article. Defects related to the control part of the generated code were barely found. After all defects had been fixed, the subsequent increment was started, implementing new user functions.

Three units of BeX used ASD for the development of their control parts. The following table depicts the statistical data related to the units. For each unit the total number of specified design and interface models is depicted. The total number of rule cases specified for each unit is also shown. A rule case is a row in a table of an interface or design model, specified and reviewed by team members. The table also depicts the total number of states generated by the model checker to check potential deadlocks. In case a unit comprises more than one design model, we sum up

Unit	Design models	Interface models	Rule cases	States	Transitions	Verification time
Orchestration	8	26	2,857	15,954,291	68,895,475	1,847 sec
FEClient	1	15	5779	1,996,830	5,249,538	230 sec
XrayIp	1	6	1,051	2,874	6321	0 sec

Table 1: ASD data in BeX units

all generated states of each individual design model. This applies also to the generated transitions and the verification time.

The table gives an insight into the effort spent for specifying and reviewing the ASD models. In fact filling in the tables can be a straightforward activity, but special attention was given to prevent human errors easily caused by cloning rule cases.

Notable is the Orchestration unit, which was initially designed in a way causing a state explosion in many of its components. Since developers could not proceed to code generation without formal correctness using model checking, components were redesigned such that model checking was a straightforward activity. As can be seen from the table the sum of the generated states of all Orchestration components is only 15 million, which can be calculated in half an hour. Generally, when the verification time of a single component exceeds one hour, further decomposition or redesign activities were immediately considered to reduce the complexity.

4 Quality results

We analyzed every defect submitted along the development process of the units. All defects are stored in a bug tracking database, which is part of a code management system. Defects related to each unit were carefully revised, one by one, by analyzing the type and cause of each defect, and how it particularly affected the quality of the code. Defects related to documentation (e.g., specification or requirement documents) are excluded from the calculations.

Table 1 summarizes the accomplished work and reports about the quality results of BeX software units. For each unit the number of effective (logical) lines of code (LOC) written manually, and those generated automatically from ASD models are reported. The total number of submitted defects of each unit is depicted in the table. These numbers represent the errors captured during in-house design, implementation, integration, and testing phases. The last column contains defect rates, e.g., the rate for the Orchestration unit is 0.5 errors per KLOC, and for the FEClient unit is 0.4 errors per KLOC.

As can be seen from the table, the units that include ASD components reveal minor reported defects, averaging to 0.86 defects per KLOC. This level of quality compares favorably to the standard of 1-25 defects per KLOC for conventionally developed software in industrial settings [12]. Defects left behind by ASD correctness verification tend to be straightforward faults easily found and fixed, not deep interface or design errors.

Typical errors found in the units developed with ASD were misspellings of variables in the

ASD used	Unit	Lines of code				Defects			
		Manual LOC	ASD LOC	Total LOC	ASD%	Manual defects	ASD defects	Total defects	Defects /KLOC
No	Acquisition	6,140	0	6,140	00.00%	33	0	33	5.375
No	BEC	7,007	0	7,007	00.00%	44	0	44	6.279
No	EPX	7138	0	7138	00.00%	7	0	7	0.981
No	FEAdapter	13,190	0	13,190	00.00%	18	0	18	1.365
Yes	FEClient	15,462	12,153	27,615	44.01%	9	2	11	0.398
Yes	Orchestration	3,970	8,892	12,862	69.13%	3	4	7	0.544
No	QA	23,303	0	23,303	00.00%	90	0	90	3.862
No	Status Area	8,969	0	8,969	00.00%	52	0	52	5.798
No	TSM	6,681	0	6,681	00.00%	7	0	7	1.048
No	UIGuidance	20,458	0	20,458	00.00%	23	0	23	1.124
No	Viewing	19,684	0	19,684	00.00%	294	0	294	14.936
Yes	XRayIP	14,270	2,188	16,458	13.29%	27	0	27	1.641

Table 2: Statistical data during in-house construction of BeX units

parameters of methods, e.g., having a parameter named 'SelectionType' instead of 'selectionType' caused the generation of two independent variables. There were also some sequencing errors. For instance, there was a case in a unit where external components were activated before the internal components. Due to the high level description of ASD these errors were easily found and fixed, compared to some hardly reproducible errors found in the manually coded modules.

The conventionally developed units did not undergo formal correctness verification. However, the units were strictly examined at different levels of code and design reviews, unit test, integration test, and system test. Traditionally developed units of BeX are already of good quality.

Other factors besides software errors can play a key role for defects to emerge. For example, some defects of the Viewing unit appeared due to migrating to new services supplied by external suppliers. Over 40% of the depicted defects of this unit are cosmetic errors (e.g., "Annotation text: font size not changed"), which don't cause potential failures during the execution of the system.

The members of teams attribute the ultimate quality of the developed units to the rigor and disciplines enforced by the ASD technology. Although the ASD developed code comprises fewer defects, the required development time was higher compared to developing the same code in the conventional way. But this was more than made up for because less time was required to resolve problems found in testing [5].

On completion of the in-house development of the units, the software is sent to the test teams. The teams require unit owners to supply complete test and verification documents, that provide evidences of 100% requirement and function coverage, and at least 80% statement coverage for their code, before any subsystem test activity is started. In general, test teams understand that any code exhibiting over 20 "allowable errors" for the entire subsystem in early testing will be

rejected and go back into design and review. But, this did rarely occur. To insure the quality of delivered code, the code was thoroughly examined by test teams using various test techniques, of which details are outside the scope of this paper.

5 Conclusion

We have demonstrated that formal methods supplied by the ASD technology can substantially influence the quality of industrial control software. We explained how the ASD method was adapted to the development process of various units. We analyzed the effectiveness of the method on sizable industrial software, by comparing a number of units developed using conventional methods with units incorporating formal technologies. The target of this study was the software of a subsystem of a complex X-ray machine, developed at Philips Healthcare.

The rigor of ASD processes eliminates design errors earlier and results in substantially reduced development time. The extra time needed to develop the software in a formal way is more than paid back by the time gained as there are less problems to be resolved in a late stage of the project.

Acknowledgements. We thank Paul Alexander, Tom Fransen, Amit Ray, Ron Swinkels and Marco van der Wijst for their useful comments on the text.

References

[1] 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.

[2] R. H. Cobb and H. D. Mills. Engineering software under statistical quality control. *IEEE Software.*, 7:44–54, November 1990.

[3] I. Crnkovic. *Building Reliable Component-Based Software Systems*. Artech House, Inc., Norwood, MA, USA, 2002.

[4] FDR homepage. *http://www.fsel.com, 2011.*

[5] B. Folmer. Personal communication. 2010.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[7] J. F. Groote, T. W. D. M. Kouters, and A. A. H. Osaiweran. Specifcation guidelines to avoid the state space explosion problem. *Technical Report 10-14, Computer Science Reports*, 2010.

[8] K. Havelund, M. Lowry, S. Park, C. Pecheur, J. Penix, W. Visser, and J. L. White. Formal analysis of the remote agent before and after flight. *Proceedings of 5th NASA Langley Formal Methods Workshop*, 13–15 June 2000.

[9] 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.

[10] J.M.Carter and J.H.Poore. Sequence-based specification of feedback control systems in Simulink®. In *CASCON '07: Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*, pages 332–345, ACM, New York, NY, USA, 2007.

[11] A. Mathijssen and A. J. Pretorius. Verified design of an automated parking garage. In *Proceedings of the 11th international workshop, FMICS 2006 and 5th international workshop, PDMC conference on Formal methods: Applications and technology*, FMICS'06/PDMC'06, pages 165–180, Springer–Verlag, Berlin, Heidelberg, 2007.

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

[13] B. Ploeger and L. Somers. Analysis and verification of an automatic document feeder. In *Proceedings of the 2007 ACM Symposium on Applied Computing (ACMSAC'07)*, pages 1499–1505. ACM, Mar. 2007.

[14] A. W. Roscoe. *The theory and practice of concurrency*. Prentice Hall, 1998.

[15] S. Smits. Automatische testomgeving voor rontgen back-ends. *Afstudeerverslag*, Fontys Hogeschool Technische Informatica, The Netherlands, 2009.

[16] C. J. Trammell, L. H. Binder, and C. E. Snyder. The automated production control documentation system: a case study in cleanroom software engineering. *ACM Trans. Softw. Eng. Methodol.*, 1:81–94, January 1992.

[17] R. van Velzen. Subsystem design specification, BeX platform, internal Philips document. 2011.

[18] M. Wessels. High level performance analysis and dependency management. *Masters thesis*, Eindhoven university, The Netherlands, 2010.