View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Radboud Repository

Radboud Repository

Radboud University Nijmegen

PDF hosted at the Radboud Repository of the Radboud University Nijmegen

The following full text is a publisher's version.

For additional information about this publication click this link.
http://hdl.handle.net/2066/151782

Please be advised that this information was generated on 2017-12-05 and may be subject to change.

Int J Softw Tools Technol Transfer (2016) 18:93–108
DOI 10.1007/s10009-015-0374-1

REGULAR PAPER

Evaluating the effect of a lightweight formal technique in industry

Ammar Osaiweran[1] · Mathijs Schuts[2] · Jozef Hooman[3,4] · Jan Friso Groote[1] · Bart van Rijnsoever[2]

Published online: 2 April 2015
© The Author(s) 2015. This article is published with open access at Springerlink.com

Abstract We evaluate the effect of applying the commercial formal technique Analytical Software Design (ASD) to an industrial project. In ASD, interfaces and software designs are modelled using a formal tabular notation. The ASD tool set supports formal checks of these models, such as deadlock freedom and interface compliance. In addition, full code can be generated from design models. ASD has been applied at Philips Healthcare to develop parts of the software of interventional X-ray systems. We report about the experiences with the embedding of ASD into the development processes. The quality of the resulting code and the productivity has been analysed and compared to code developed with other techniques. We observe that the use of ASD leads to a strong reduction of the number of defects and an increase in productivity. The results are also compared to the literature about standards and related projects at other companies.

✉ Ammar Osaiweran
osaiweran@gmail.com

Mathijs Schuts
mathijs.schuts@philips.com

Jozef Hooman
jozef.hooman@tno.nl

Jan Friso Groote
j.f.groote@tue.nl

Bart van Rijnsoever
bart.van.rijnsoever@philips.com

1 Eindhoven University of Technology, Eindhoven, The Netherlands

2 Philips Healthcare, Best, The Netherlands

3 Embedded Systems Innovation by TNO, Eindhoven, The Netherlands

4 Radboud University, Nijmegen, The Netherlands

Keywords Formal methods · Industrial application · Code generation · Component-based design · Compositional verification

1 Introduction

The use of formal techniques in software engineering has been advocated for many years, with many arguments, see, e.g. [25]. The main aim is to detect design flaws early using techniques with a solid mathematical basis. The industrial application of such techniques, however, is not trivial. It might easily lead to large efforts on modelling and analysis, requiring experts in logic. Moreover, scalability is a well-known problem of many techniques. Hence, the use of lightweight formal methods has been promoted [24,31]. These methods hide a lot of the mathematical details from the user and do not aim at generic modelling and analysis techniques. By specializing on a particular type of design and a particular set of properties, more efficient and effective approaches can be developed. Additionally, the possibility to generate code from formal models is important for industrial acceptance. This avoids the error prone manual translation from models to code and is expected to improve productivity.

In this article, we evaluate the Analytical Software Design (ASD) [8,28] method which can be classified as a lightweight formal technique, since it is based on a set of well-chosen restrictions which make it possible to combine model checking and code generation. We list the main characteristics (more explanation can be found in Sect. 3):

– The approach is restricted to control components where decisions depend on incoming events and not on the data parameters of these events. Interface specifications and models of the implementation of components

(called design models) are expressed in a limited form of state-transition tables. The underlying formal models that define the semantics of the models are invisible to the user.
- Only a pre-defined set of properties can be checked. The underlying formal model checker is hidden for the user; counterexamples are shown using sequence diagrams. The restrictions of ASD enable fully automated verification.
- Scalability is obtained by verifying a design model using only the interfaces of the components it uses, without knowing their implementation.
- From design models, full code can be generated, including logging and tracing facilities.

The method is supported by the commercial tool ASD:Suite from the company Verum [43]. Typically, a 2-day course of Verum is sufficient to start applying the method.

The aim of our research is to investigate whether the use of ASD results in tangible improvements. Our goal is to answer the following questions:

- Can ASD deliver product code of good quality?
- Does ASD always produce near zero-defect software? If not, which types of defects can be expected?
- Does ASD require more development time compared to traditional methods? What about the productivity using ASD?

To answer these questions, we discuss the experiences in an industrial project at Philips Healthcare in which the ASD method has been used. We compare the collected quantitative facts to other development projects at Philips Healthcare and related projects from the literature.

This paper is structured as follows. Section 2 contains a discussion of related work. In Sect. 3, the ASD approach is introduced as far as needed to understand our study. In Sect. 4, we briefly sketch the industrial context. Section 5 provides details about the industrial development process using ASD, including the main issues and limitations encountered, and data about the developed ASD models. Facts about the generated code and defects found during testing are presented in Sect. 6. In Sect. 7, we analyse the cause and the type of defects that could escape the formal techniques. Section 8 compares the results of the ASD code with other code developed at Philips and the industry standards worldwide. An overview of related projects using formal techniques can be found in Sect. 9. Finally, Sect. 10 contains our conclusions.

2 Related work

There are a number of formal techniques that can be applied to source code. Relevant for the application domain of this paper is the bounded model checker CBMC for C and C++ programs [10], which supports both fixed checks (e.g., array bounds, safety) and user-specified assertions. A related checker for C is CPAchecker [11]. Since Philips aims at a more model-based development with well-defined interfaces between independently developed components, this type of formal techniques has not been selected.

An interesting model checker in the domain of embedded systems is Uppaal [42]. It uses an attractive visual representation of timed automata which can be simulated and checked using user-defined properties. In many respects, Uppaal is rather complementary to ASD; a comparison between the two methods can be found in [16]. The main drawback of Uppaal for Philips is that it does not support code generation and for large systems it easily suffers from the well-known state explosion problem.

Clearly, there are many tools that support code generation from models. For instance, the Rational Rhapsody Developer [29] supports code generation for a number of programming languages based on UML diagrams. Matlab [36], which is often used for model-based development in the embedded domain, also supports code generation for several targets based on graphical models. These type of tools, however, have no or only limited support for formal verification.

Related to ASD is the commercial tool VDMTools [14] which supports code generation from models specified using the VDM++ language [19]. VDM (The Vienna Development Method) is based on a formal specification language which became an ISO standard in 1996. VDM++ is an extension which supports object orientation and concurrency. Likewise, the tool Atelier B [12] has been used to develop a number of safety-critical systems using the B method [1]. The SCADE Suite [17] provides formal techniques for specification, verification and code generation. These techniques are quite generic and the correctness proofs for VDM and B models may require interactive theorem proving. ASD is much more restricted than the approaches mentioned above to achieve a high level of automation and to support compositional verification.

Note that the authors of this paper were not involved in the decision of Philips to use ASD. Besides the characteristics of ASD in comparison to other tools, this decision was also based on non-scientific aspects such as costs, the local availability of training and support, and the possibility to influence the future direction of the tool development. The authors were also not involved in the development of ASD and are not associated to the company Verum. They have only used ASD or observed its use at Philips.

3 Analytical software design

In this section, we provide a short exposition of the ASD approach. ASD [8,28] is a component-based technology that

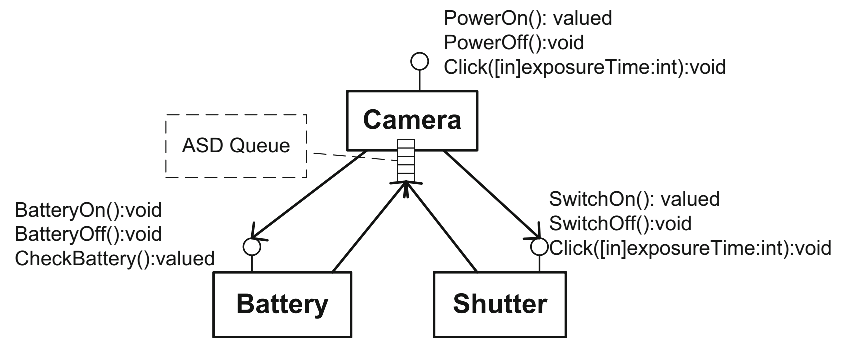

Fig. 1 Overview of the Camera example

aims at enabling the application of formal methods into industrial practice by a combination of the Box Structure Development Method [40] and the Communicating Sequential Processes (CSP) formalism [23]. The ASD:Suite is a development tool that embeds the ASD technology into a software design environment.

The ASD approach distinguishes two types of models which are both described by a similar tabular notation: *ASD interface models* and *ASD Design Models*. These models are state machines described in a tabular format, following the Sequence-Based Specification technique, to force consistent and complete specifications [9].

The interface model specifies the external behaviour of a component without referring to any internal behaviour. This forms the formal contract of interaction between the component and its clients. A design model implements a certain interface model, and typically uses services of other components by referring to their interface models.

The model checker of ASD:Suite verifies that calls to these so-called *server* components are correct with respect to their interface models. Moreover, it checks that the design model conforms to the implemented interface model. The tool ASD:Suite allows code generation from design models to a number of programming languages (C, C++, C#, Java).

It is not needed that server components are realized using ASD. Since the approach is intended for control components, server components that involve data manipulations will be implemented by other techniques; such components are called *foreign components*. The ASD approach is compositional [26], because verification of a design model uses only external interfaces of server components, without knowing their internal implementation.

In ASD, communication between client and server components is asymmetric, using synchronous calls and asynchronous callbacks. On the one hand, clients issue synchronous calls to server components, where the client is blocked until the server accepts the call and eventually returns it to the client. The return of a call may either be void or valued. On the other hand, a server can communicate with its clients by asynchronous callbacks. Callbacks are stored in a so-called ASD callback queue (FIFO).

We explain the ASD technology in more detail using a small Camera example. It consists of a Camera component that controls a Shutter and a Battery component, see Fig. 1. The Camera component may receive three requests from external users, namely *PowerOn* to start-up the device, *PowerOff* to shutdown the device and *Click* to take a picture.

We use this example to explain the ASD interface models in Sect. 3.1 and design models in Sect. 3.2. Formal verification is described briefly in Sect. 3.3.

3.1 ASD interface models

To develop an ASD component, first an interface model has to be specified. The interface model is used as a formal means to describe the allowed sequences of method calls and replies. The interface model specifies the interaction protocol between a component and its clients. All internal behaviour which is not visible to client components is excluded from the interface specification.

An example of the tabular specification of an ASD interface model is depicted in Fig. 2, which shows the interface model of the Camera component. This interface model contains four states: *Off*, *SwitchingOn*, *On*, and *TakingPicture*. An interface model must be complete in the sense that in each state the response to all input stimuli must be defined. Each row, also called a *rule case*, specifies the response to a certain stimulus and the next state. Input stimuli that are not allowed in a state are declared *Illegal* in the *Actions* field.

A stimulus event may have an "+" postfix to indicate that the call has a valued reply. The values of the reply are specified in the Actions field. To model non-deterministic cases, a rule case can be duplicated; see for instance rule cases 3 and 4 of Fig. 2. In rule case 3, the Camera component receives a *PowerOn* request which returns with value *OnOK* and then transits to the *SwitchingOn* state. In rule case 4 the component fails on the request and remains in the *Off* state.

To model internal events, such as internal callbacks, rules can be triggered by so-called modelling events (prefixed by '*INT*' in the *Interface* columns in Fig. 2). When a modelling event cannot happen, a *Disabled* action is assigned. For instance, in the *SwitchingOn* state three of the four modelling

Fig. 2 The ASD interface model of the Camera component (ICamera)

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

Fig. 3 The ASD interface model of the Battery component (IBattery)

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

events may happen. They abstract from internal behaviour related to initializing internal components.

Similarly, Figs. 3 and 4 depict the ASD interface models of the Battery (IBattery) and the Shutter (IShutter) components. They both describe the external behaviour available for client components (in this case the Camera component). In these figures, the *Illegal* and the *Disabled* rule cases are hidden.

3.2 ASD design models

Once the interface model of a component is completed, the design model can be created. The ASD design model extends the interface model with more detailed internal behaviour. It is deterministic and may include method invocations to its server components.

Fig. 4 The ASD interface model of the Shutter component (IShutter)

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

Fig. 5 DCamera: ASD design model

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

Figure 5 depicts the design model that uses the interfaces of the Battery and the Shutter components. The specification is straightforward and refines the interface model of Fig. 2 with all required internal details. For instance, rule case 3 specifies that when the Camera component receives a *PowerOn* request, it checks the internal status of the Battery by sending the *CheckBattery* call and then transits to the *CheckingBattery* state where two return values are expected. Based on the return value from the Battery, the Camera component sends either the *OnOK* or *OnFailed* value to its client and then transits to a next state.

The semantics of a design model is such that callbacks from server components are non-blocking and can always be received. For this, they are put into an ASD callback queue (one callback queue per component). Incoming calls from clients are serialized, that is, at any point in time at most one call is executed until completion. Callbacks from server components have priority over client calls. After the completion of a rule case, the callback queue is inspected. If the queue is not empty, the rule case corresponding to the first callback at the head of the queue is executed. When the callback queue is empty and no client call is being processed, a new client call is accepted. Invoking a call or callback which is illegal will halt the component.

3.3 Formal verification using model checking

Formal verification can be applied to a set of interface and design models, such as the models for the Camera component in Fig. 6. The ASD:Suite automatically translates the ASD tabular specifications to corresponding CSP models and verifies them using the formal refinement checker FDR2 [18] (FDR is an abbreviation of Failures-Divergence Refinement). All CSP and FDR2 details are hidden from end-users.

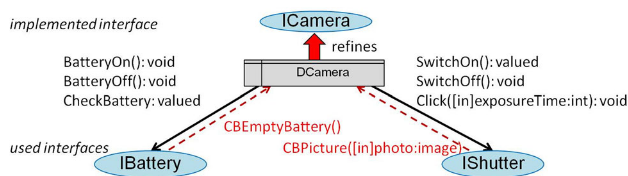

Fig. 6 Structure of ASD models related to the Camera component

The ASD:Suite provides a fixed set of properties that can be be verified. The main checks performed by the ASD:Suite are:

- Checking behavioural correctness: this checks whether the design model correctly uses its server interfaces. This is accomplished by detecting and correcting deadlocks, livelocks, illegal calls, and checking for determinism of the design model. For our Camera example this means that the above checks will be applied to the *DCamera* design model in combination with *IShutter* and *IBattery* interface models.
- Checking refinement of the implemented interface: this checks whether the interface model of a component is correctly refined by the design model in combination with the server interfaces. For our Camera example, this checks whether the *DCamera* design model, in combination with the *IShutter* and *IBattery*, refines the top-level interface model *ICamera*. Hence, *ICamera* can be used by another top design model without considering the details of the lower level components. This compositional way of verification avoids the well-known state space explosion problem and enables industrial scalability, because components can be checked in isolation. It requires, however, a careful design of the system such that the components themselves are kept small.

4 Description of the project

We describe a development project of Philips Healthcare in which ASD has been used. Philips Healthcare develops a number of highly sophisticated medical systems, used for various clinical applications. Our study concerns the development of an interventional X-ray (iXR) system, which is depicted in Fig. 7. These systems are used for minimally invasive surgery. For instance, guided by X-ray images a surgeon can place a stent via a catheter, thus avoiding open heart surgery.

The software architecture of the iXR system is divided into a number of subsystems, including the so-called 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 patient table, and the stand that holds the generator and the detector, see Fig. 7.

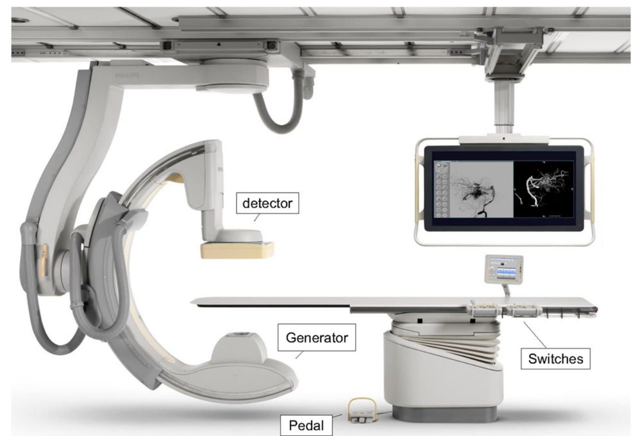

Fig. 7 An interventional X-ray system

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

To clarify the terms used along the paper, a software unit indicates a package that contains code related to certain functionality. A unit is a collection of smaller modules each of which is responsible of more specialized and detailed functionality. A module comprises a number of software components, where each component is implemented in one or more C# files.

The FE subsystem includes 22 units, two of which are constructed from scratch and are the target of this study: the Application State Controller (ASC) and the Frontend Adapter (FEA). Both units comprise a number of modules that include 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 hand-switches. The unit counts, filters, and ensures priorities of such requests before instructing other units controlling the lower level drivers to start X-ray image acquisition. 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 interfacing with another external subsystem through a network. It exchanges

information related to patients and their examination 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 may press a pedal or a hand-switch at any time, even if the internal components or the hardware is not prepared or configured yet for image acquisition.

5 The process of developing ASD components in iXR

The software of iXR systems is developed following 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 on the work during three increments for developing ASD components for both the ASC and the FEA units, from January 2011 until August 2011. The work was done by a team of five 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 foreign components. The ASD technology was tightly integrated with the traditional development processes. Below we describe the main phases concerning the incorporation of ASD into the development processes.

Pre-study phase The three increments were preceded by a pre-study period, during which the team attended a 1 week ASD course to get familiar with the approach and its formal technologies. The course was limited to learning how to use the ASD:Suite, for example, how to fill-in the tables, how to verify them using model checking, and how to integrate the generated code with other handwritten code.

Furthermore, during the pre-study period the reference architecture of the FE subsystem was discussed, to get consensus about the functionality of the units among all team members. Next the ASD team explored various design alternatives and approaches to define the ASD components. This phase took a considerable amount of time, because it turned out that the team still had to learn a number of aspects of ASD. 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.

There was a lack of ASD design guides, cookbooks and patterns that could aid the team to incorporate the new technology into its 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. Because object-oriented design and programming was the dominating approach at iXR, 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 [20], trying to model them 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 successful application of the technology requires changing the development culture and the mind-set.

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 design clearly distinguishes control components from other data and computational components. The designs were iteratively reviewed and re-factored until they were approved by all team members. After that, design of components and their responsibilities were documented in informal documents.

Modelling the ASD components After the design phase, 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 gradually refining them by detailed design models and server 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 early questions of what actions must be taken to input events not addressed by the incomplete informal documents. Indeed, the technology helped the team to find omissions and gaps in the initial set of requirements and designs. It initiated early discussions with various stakeholders. Subsequently, this increased the quality of requirements and designs during early phases of development.

However, due to the required completeness of the tables, some ASD interface models were too large, hard to review, and hard to maintain. This resulted in further decompositions, leading to smaller components, thus increasing readability and maintainability. The required completeness was also challenging during early stages of development. For a complex system like the FE, lead architects tend to concentrate on important, high abstract aspects of the system and leave the details to later stages of the project. Although one could choose to work on a subset of the interface, the corresponding specification must be complete.

Table 1 lists the developed ASD components for the ASC and the FEA units. Each component includes one imple-

Table 1 Modelling and verification statistics of the ASD components

Component	Models	Rule cases	States	Transitions	Time(s)	ELOC
ASC unit						
AcquisitionController	6	432	16,912	79,840	2	1685
AcquisitionRequests	5	837	192,320	1,027,032	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	45,560	76,233	3	1133
RequestFilter	3	381	51,790	226,910	1	994
RunController	4	842	8,058,224	41,150,288	444	5126
FEA unit						
AcqCtrlAcqRequests	3	353	2802	7805	<1	465
BETStateless	3	404	8640	38,704	<1	518
CmdStateless	3	62	12	24	<1	704
CxaAdpMain	7	2794	139,824	461,292	50	5944
DAcqCtrl	4	1300	117,412	364,066	13	3206
DActivation	3	103	4480	14,688	<1	585
Decoupler	2	11	3	3	<1	239
DWrapper	4	408	8928	46,736	<1	674
FEProxyVE	6	5270	597,096	1,764,366	289	9645
FEProxyVEStateless	6	202	4976	19,320	<1	1753
FSStateless	3	31	220	636	<1	554
UGStateless	3	88	44,388,432	76,939,995	6853	951

mented interface model that captures the external behaviour, one design model that includes the external and internal behaviour of the component, and a number of interface models of server components. The number of models for each component can be found in column 2 and the sum of all rule cases in column 3. The other columns are explained below.

Formal verification Each ASD model is verified using model checking. With a single button click, the model checker detected various deadlocks, livelocks, illegal scenarios, and race conditions, which required changes to the models. In some cases, solving these defects caused a redesign of the ASD components, especially when the fix made model checking not feasible due to the size of the induced state space.

Another important reason for redesign was the lack of abstraction in the behaviour of some components. For example, the use of callback events which can enter the ASD queue in any order might imply that FDR2 needs hours or even days to calculate the state space that captures all possible execution scenarios. Adding a new event, e.g., due to changed requirements, may make verification virtually impossible. Hence, components were re-factored to make model checking feasible again.

In general, the design style influences the time needed for model checking. Note that, within our industrial context, waiting for a long time is usually not acceptable due to the tight deadlines of the incremental planning. Worst, before a defect 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 defects. A number of design guidelines to improve the model checking speed were proposed [21]. Hence, the team avoided a number of design styles that may needlessly increase the state space and the time required for verification [27].

Based on the knowledge gained, the team eventually obtained a set of formally verified components. Columns 4–6 of Table 1 contain the output data from FDR2 for the most time-consuming check, i.e., the refinement check for the design models. For each component, the table shows the state space generated by FDR2: the number of states and transitions, and the time in seconds. These data indicate that most of the components were verified within an acceptable range of states and transitions, calculated in a reasonable time by FDR2.

Code generation and integration As soon as ASD components had been formally verified and further reviewed to ensure that fixing the model checking defects did not break the intended behaviour, 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), generated in the C++ programming language. The effective lines of

code denote all source code excluding blanks and comments.

Experience shows that, in a more conventional development method, integrating components is a nightmare, due to the substantial effort required to ensure that all components work together correctly. 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 handwritten or legacy code that did not undergo formal verification caused some errors and delays. These errors were not counted because they emerged during the implementation phase and are part of the normal coding and debugging processes. Furthermore, they are not considered as behavioural defects because these issues appeared in the wrappers connecting ASD generated code, which is implemented in one technology (Microsoft C++ .Net), and the legacy code, which is implemented using a different technology (Microsoft Component Object Model).

Testing An apparent advantage of ASD is that testing time is shortened since formal verification replaces white-box testing of the internal structure of the generated code. But, an apparent limitation of ASD is that only a fixed set of properties are verified. It is not possible to express domain specific properties or to relate events between implemented and server interfaces. Hence, testing the required behaviour is still needed.

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

6 Data collection and analysis

In this section, we collect and analyse the project data to investigate the impact of the use of ASD on the quality of the developed software. We further compare the end quality of the units which incorporate ASD with the other units of the FE. To accomplish this goal, we started with collecting, for every unit, the total number of effective lines of code that had been newly developed plus the changed legacy code. We restricted ourselves to the period bounded by two baselines, representing the start and the end of the three increments.

There were a total of 202 submitted defects. These defects were uncovered during the in-house subsystem tests and are not post-release defects or found after delivery to the field. We analysed all defects separately and partitioned them into coding defects and others (e.g., due to documents, requirements or design issues). This led to a list of 104 defect reports related to coding. For each coding defect, the corresponding unit and the internal component were identified.

Table 2 Statistical data of representative units of the FE

Unit	Effective lines of code		Defects		Defects/KELOC	
	ASD	HW	ASD	HW	ASD	HW
ASC	14,006	5784	13	10	0.92817	1.72891
FEA	25,238	9489	1	18	0.03962	1.89693
IGC	0	6326	0	35	N/A	5.53272
SC	0	3340	0	0	N/A	0
SIM	0	6202	0	0	N/A	0
IDS	0	2650	0	7	N/A	2.64151
NGUI	0	2848	0	0	N/A	0
PandB	0	3161	0	1	N/A	0.31636

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 those which 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.

To study the impact of ASD, we separated the ASD code and the handwritten (HW) code and analysed them in isolation. Both are quantified in the second and third columns of Table 2. The handwritten code of the first two units represents glue code that connects ASD generated code with other non-ASD external components. It also represents non-control code that cannot be modelled in ASD:Suite such as data manipulation or computations [example (de-)serializing xml strings].

We distinguished ASD defects from those related to the handwritten code, as listed in the fourth and fifth columns. Next the defect rate of ASD and non-ASD code was computed, as shown in columns 6 and 7.

Note that some of the manually coded units exhibit zero defects, because most of the changes were on the level of interfaces and not on the core internal behaviour of the units. Furthermore, the ASC and the FEA were constructed from scratch while other units were reused from previous released products.

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 contained one defect. After studying the corresponding defect report we found that the defect 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.

For the FEA unit, most defects were related to the handwritten code. This is different for the ASC unit, although the defect rate for the ASD part is slightly better than the manually written code. The amount of defects appeared in ASD components of the ASC unit motivated us to investigate the behaviour of the components in depth and to study the nature and the type of the detected defects. We discuss this in detail in the subsequent section.

In general, the defects of the ASC appeared due to unintended, unexpected behaviours (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 stopped). None of these defects 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). This is because deadlock and illegal scenarios were formally checked by the fixed set of verification properties of ASD.

To explain why the FEA unit included fewer defects, we observed that its required functional behaviour is far less complex than the behaviour of the ASC unit. Moreover, the FEA unit implements an ASD interface model that specifies an important interaction protocol between the FE and other subsystems. This interface and its intended behaviour were thoroughly reviewed—not only by the FE team but also by other teams of the other subsystems.

7 Analysing the complexity of ASD components in relation to defects

The purpose of this section is to study the complexity of each ASD component and determine whether there is a correlation between complex models and the error density. We also study the root causes of the defects appeared in ASD components that could escape formal verification, without providing technical details about the functionality of the ASD components. To do so, we began by analysing the ASD components individually, especially those related to the ASC unit, trying to identify the responsible component that contributed most to the defects.

Initially, this appeared to be challenging because we did not possess any systematic means to measure the complexity of components at the model level. Therefore, we assessed the complexity of components using two other means. The first way is subjective in the sense that we evaluated the models concerning understandability and reviewability of the models, based on our "common sense". The second way is more objective because we chose to systematically analyse the generated code, using available code analysis tools and techniques. The two steps are detailed below.

For the first way, we evaluated the readability of only 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. Column two of Table 3 includes the results of the assessment. For example, the *RunController* component is considered to be complex because 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. For instance, in a certain state there are 31 rule cases for the *FailedSC* stimulus event to deal with all possible combinations of variable values. A fragment with three of these rule case is shown in Fig. 8.

As a contrary case, the user-guidance component *UGStateless* of the FEA unit is considered to be very easy because it contains only two states without any predicates. The component is enabled or disabled to allow or block the flow of information traffic to other components. Although the component is easy 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. Because these events are stored in the queue of the component in any arbitrary order, FDR2 took substantial time to calculate the state space of the component.

Next, we distributed the defects to the respective components, as depicted in the last column of Table 3. As can be seen, most of the ASD defects reside in the *RunController* component, unveiling an apparent correlation between the complexity of the component and the density of defects.

In the second way, we performed a static analysis of the generated code, seeking similar correlations between code complexity and defect density. The motivation was that complexity of the models may also be reflected in the corresponding generated code. We used the *SourceMonitor* tool Version 3.2 [41] to analyse the generated code.

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). In general, generated code tends to take up more space than manually written code. In our application domain there are hardly any memory limitations for code (the main concern is the storage of X-ray images). It would lead to a larger maintenance effort if one would maintain the generated code manually, but clearly this is not advisable.

As can be seen in the table, the *RunController* component also appears to be very complex compared to other components. Notable is that the 157 Max CC of the *RunController* component resides in the code corresponding to the rule cases of the *FailedSC* stimulus event presented earlier in Fig. 8. In

Table 3 Statistical data of ASD components

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

	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. 8 An example of complex rule cases

the code, the rule cases are represented by a single method containing 30 related if-else statements.

The number of defects found in the *RunController* component motivated us to study the type of these defects and their evolution. In fact, four of the seven defects had a similar cause, namely missing updates of state variables before a state transition. The team solved these defects by adding more rule cases with different predicates and also additional state variables, which increased the complexity even more. Another defect was caused by forgetting to store a value. Two defects were due to missing requirements.

A potential solution to avoid such defects is to reduce the complexity by decomposing the *RunController* component into a number of smaller components instead of increasing the complexity by adding more details and state variables to the component when fixing the defects. The *RunController* clearly shows that bug fixing should be done with care and may lead to redesign activities.

8 Quality and performance results

In this section, we evaluate the end quality and productivity of the developed ASD units, by comparing them against the worldwide industry standards reported in the literature. The best sources we could find are [30,37–39], where 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 (FE) subsystem. Code quality is discussed in Sect. 8.1, whereas productivity is addressed in Sect. 8.2.

8.1 Code quality

In [35], 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 [30] presents an average of 1.7 coding defects 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 [30]).

Furthermore, McConnell presents in [38] (p. 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 [37] 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 defect density based on the project size, where our system is expected to include 4–100 defects per KLOC (p. 652, Table 27-1).

At Philips Healthcare, for each increment the internally delivered code should exhibit at most six defects per KLOC with an average productivity of 2 LOC per staff hour. Any code that includes more defects, 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 the ASD technology delivered high quality code, averaging the ASD code of the ASC and FEA units to only 0.36 defects per KLOC. Hence, if we consider the ranges of software quality mentioned above, the ASD code appears to be of good quality.

8.2 Productivity

The productivity of ASD is compared to the literature in terms of the number of lines of code per staff hour (using 132 h per month, based on 22 days, 6 h a day). In [30], Jones presents a productivity figure of 435 ELOC for C++ per staff month (p. 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).

In [15], Cusumano et al. studied the data of a number of projects worldwide, 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 % of generated code.

A study of McConnell [37] (p. 522) showed that a formal Cleanroom project could deliver nearly 5.61 LOC per staff hour [34]. 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, [37] (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 (FE) subsystem ranges between 700 and 10,000 LOC per staff year with a nominal value of 2000 LOC per staff year (i.e., 0.4–6.3 with a nominal value of 1.3 LOC per staff hour).

Consequently, we can use the above measures to compare the productivity of the ASD developed units. The total time spent developing the ASD components was 2378 h, affording an average of 16.5 ELOC per staff hour. The total time spent developing the two units, including the time spent for non-coding overhead, was 5701 h, which favourably yields 9.6 ELOC per staff hour. Therefore, if we consider the above-mentioned range of 0.4–6.3 LOC per staff hour of [37], the productivity of ASD appears to be much better. This high productivity is due to a number of factors:

- The ASD:Suite is a model-based tool that provides an easy to use graphical user interface. Specification of models is done by filling the ASD tables with actions selected from an ordered list using clicks. Compared to traditional text editors, the graphical interface of ASD is more productive.
- The code is obtained automatically by a click of a button.
- Integrating ASD components is automatic and requires no extra work.
- The internal structure of ASD generated code need not be tested since it is formally verified (but black-box testing is required).
- ASD tracing and logging facilities finding the source of bugs and quickly repairing them.

9 Related projects using formal techniques

In this section, we report about a number of industrial projects that incorporated formal methods into software development, highlighting their achieved quality and productivity. Our starting point was a survey and a comprehensive review of applications developed using formal methods, including 70 references to the literature, in [44]. Furthermore, we searched other projects using web search engines and visited a number of searched home pages hosting the formal techniques.

Initially, the idea was to restrict ourselves to the last 10 years, but we found very few publications reporting quantitative evidences that demonstrate the impact of formal

Table 4 List of projects incorporated formal techniques in software development

Year	Project	Technology	Size (KLOC)	Prog. language	D/KLOC	LOC/man-hour	Phase counting defects
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	≤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 after delivery
2007	Shuttle, Paris airport	B Method	158	Ada	–	–	–
2008	Mobile FliCa	VDM	264	C++	Zero	–	After system release
2012	Philips, Frontend	ASD	39.2	C++	0.36	16.5 (ELOC)	Subsystem test

techniques in industry. Most publications contain detailed case studies of applying formal methods at different stages of software development and facts about the performance of the formal method tools and did not describe the performance of industrial projects. Hence we searched further back, until the late 1980s.

Table 4 summarizes the results by listing 13 projects that are similar to ours and which provide sufficient data about code quality and productivity. The last line describes our own FE project. The projects are listed in chronological order, highlighting the formal technique used, 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 defects were counted. We give a brief explanation of the first 13 projects.

The first three projects were selected from [34], where Linger lists 15 projects where the Cleanroom formal engineering method was used, summarizing the results achieved for each project. All systems exhibit quality figures that range between 0 and 5 defects per KLOC with an average of 3.3 defects per KLOC. Compared to the mentioned range of 30–50 defects/KLOC in traditional development, Linger concluded that systems developed with formal methods achieve remarkable quality. Since productivity figures were not available for 12 of these projects, Table 4 contains the three projects with quality and productivity data:

1. The IBM COBOL Structuring Facility, with a team of six developers (it was their first development project). The product exhibits 3.4 defects per KLOC and several major components were certified without experiencing any defect. The average productivity was 5.6 LOC per man-hour.
2. The development of a Satellite controller carried out by the Software Engineering Laboratory at NASA. The system included 40 KLOC of FORTRAN and was certified with 4.5 defects/KLOC. The productivity was 5.9 LOC/person-hour, resulting in an 80 % improvement over previous averages known in the laboratory.
3. Complex system software which was developed by 50 people at IBM using various programming languages. The system exhibited 2.6 defects/KLOC, where five of its eight components experienced no defects during testing. The team used the Cleanroom method for the first time [22].

The MaFMeth project [6] incorporated VDM and the B method in software development. The project included 8000 lines of generated code. However, developers estimated the actual code size to be 3500 LOC in case the same functionality is implemented without any code reuse. The reported defects were found during unit testing. Defects found during validation testing or defects found after releasing the system were not available. Productivity was 13.6 LOC per hour with a defect density of 0.9 defect per KLOC.

The B Method was used to develop safety-critical components of the automatic train operating system, the metro line 14 in Paris [2,5]. Members of the development and validation teams were newcomers to 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 defect 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 on schedule [5].

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++ [13]. The system exhibited less than one defect per KLOC. The defects were found during coverage testing and after product release. Productivity was above industry norms but there were no figures provided in the paper. Productivity using Prolog was less than C++ because of the high abstract level and the rigorous way the core Prolog was generated. Developers involved were skilled in formal methods.

Abstract State Machines (ASM) were used in the development of a software package developed at Siemens called FALKO [7]. The package was redesigned from scratch due to its complexity. The newly developed package included roughly 11,900 LOC of C++, generated and manually written, developed in nearly 66 man-weeks effort. Two defects were found after product release and were fixed directly in the generated code. The end quality was 0.17 defect per KLOC and the productivity was 2.2 LOC per hour.

In [33], 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 defects found were reported.

The VDM toolset was used for developing some components of a business application, called TradeOne [19]. 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 exhibits 6.1 lines of C++ per staff hour. The defect rates of both subsystems are less than one defect per KLOC. The defects were reported during integration testing and there were no defects 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 1 year using SPARK [4]. The overall productivity of the TIS core system was 6.3 LOC of Ada per man-hour. The system did not exhibit any defect 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 [2,3]. 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 were no data available about the total time spent in development or the number or type of defects encountered along the construction of the software.

Sony Corporation applied VDM to the development of firmware for a smart card IC chip in an industrial project called the Mobile FeliCa [32]. The developers produced 140,241 formal lines of VDM specification including test cases of 66,412 lines and 34,460 lines of comments written in a natural language. The formal specification guided the implementation of 264,000 lines of C++ code, produced by a team of 50–60 members. The project was accomplished with an average productivity of 11.9 lines of VDM per hour. Before the integration tests, 440 errors were found and fixed. No defects were reported by customers since the first system release. In general, developers observed an increase in quality and they could deliver the system in time.

As a general conclusion we can say that the formal techniques used in these projects had favourably increased the productivity and the quality of the developed systems. 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. The literature about the projects mentioned above does not contain discussions regarding the weaknesses and the main difficulties encountered when applying the techniques.

10 Conclusions

Based on our analysis of the use of the ASD approach in an industrial development project, related to industrial standards

and similar projects, we answer the questions raised in the Sect. 1.

Can ASD deliver product code of good quality? The ASD technology allows generating formally verified code from formal models. Compared to the industry standards of Philips and those reported worldwide, the ASD technology could clearly deliver product code that exhibits good quality figures. The developed units appeared to be stable and reliable against the frequent changes of requirements. But, obtaining this level of quality depended on many factors like the experience of developers and the level of abstractions in the designs, for instance.

Does ASD always produce near zero-defect software? If not, which types of defects can be expected? As we saw before, although ASD components were formally specified and verified, some defects were found during testing. Since ASD only checks a limited set of properties, it may not always lead to defect-free software. However, our study shows that the formally developed software contains very few defects compared to the industry standards. In general, most defects were easy to find and to fix. Furthermore, we collected statistical figures related to worldwide projects that incorporated other formal methods in software development. Although the number of these projects was very limited, we found that most projects exhibited good quality and productivity figures.

Does ASD require more development time compared to traditional methods? What about the productivity using ASD? The presented data indicate an 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. Another important fact is that less or even no time was spent integrating and manually testing the generated code; activities which are usually time-consuming and uncertain. The productivity was higher because ASD prevented problems earlier rather than detecting and fixing problems at later stages, which is time-consuming and costly.

Finally we discuss some of the limitations of the ASD method that might prevent large-scale introduction into the industrial workflow:

- The approach is limited to event-based control components. It is not suitable for low-level real-time controllers and data-intensive components. Designers might find it difficult to decide what to do in ASD and what not.
- ASD assumes a hierarchical control architecture with synchronous method calls from top to bottom and asynchronous callbacks in the other direction. Although this gives a clear structure, it is not always easy to construct such a hierarchy, especially because components should be small and the number of callbacks limited to allow model checking. Moreover, when software engineers are used to object-oriented designs this might require a paradigm shift.
- Large state-transition tables become difficult to review and to maintain; there are hardly any structuring mechanisms, e.g., to indicate that a certain transition is common to a set of states.
- When preparing work breakdown estimations, it is difficult to estimate the time needed for verification.
- There is no systematic means to evaluate and analyse the complexity of ASD models, e.g., to detect early that model checking might take too long or to decide that refactoring is needed. For our analysis, we determined the complexity based on the corresponding generated code. But models that generate complex code do not necessarily produce huge state spaces when checked formally.

Summarizing, our investigation shows that the ASD technology could deliver good quality software with high productivity. But we also indicated that large-scale introduction is not trivial and requires good insight in the possibilities and impossibilities of the method.

Acknowledgments We would like to thank the anonymous reviewers for their useful remarks and suggestions for improvement.

Open Access This article is distributed under the terms of the Creative Commons Attribution License which permits any use, distribution, and reproduction in any medium, provided the original author(s) and the source are credited.

References

1. Abrial, J.-R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, New York (1996)
2. Abrial, J.-R.: Formal methods: theory becoming practice. J. Univ. Comput. Sci. **13**(5), 619–628 (2007)
3. Badeau, F., Amelot, A.: Using B as a high level programming language in an industrial project: Roissy val. In: ZB 2005: Formal Specification and Development in Z and B. Lecture Notes in Computer Science, vol. 3455, pp. 334–354. Springer, Berlin (2005)
4. Barnes, J., Chapman, R., Johnson, R., Widmaier, J., Cooper, D., Everett, B.: Engineering the Tokeneer enclave protection system. In: Proceedings of the 1st International Symposium on Secure Software Engineering (2006)
5. Behm, P., Benoit, P., Faivre, A., Meynadier, J.-M.: Meteor: A successful application of B in a large project. In: FM'99: Formal Methods. Lecture Notes in Computer Science, vol. 1708, pp. 369–387. Springer, Berlin (1999)
6. Bicarregui, J., Dick, J., Woods, E.: Quantitative analysis of an application of formal methods. In: FME'96: Industrial Benefit and Advances in Formal Methods. Lecture Notes in Computer Science, vol. 1051, pp. 60–73. Springer, Berlin (1996)
7. Borger, E., Pappinghaus, P., Schmid, J.: Report on a practical application of ASMs in software design. In: Abstract State Machines—Theory and Applications. Lecture Notes in Computer Science, vol. 1912, pp. 361–366. Springer, Berlin (2000)

8. Broadfoot, G.H., Broadfoot, P.J.: Academia and industry meet: some experiences of formal methods in practice. In: APSEC '03: Proceedings of the Tenth Asia–Pacific Software Engineering Conference Software Engineering Conference, pp. 49–58. IEEE Computer Society, London (2003)
9. Carter, J.M., Poore, J.H.: Sequence-based specification of feedback control systems in Simulink. In: Proceedings of the 2007 Conference of the Center for Advanced Studies on Collaborative Research, CASCON '07, pp. 332–345. IBM Corporation, New York (2007)
10. CBMC: Bounded Model Checking for Software. http://www.cprover.org/cbmc (2015)
11. CPAchecker: The Configurable Software-Verification Platform. (2015)
12. ClearSy, Atelier B: Industrial Tool Supporting the B Method. http://www.atelierb.eu/en (2012)
13. Clement, T., Cottam, I., Froome, P., Jones, C.: The development of a commercial "shrink-wrapped application" to safety integrity level 2: the dust-expert™ story. In: Computer Safety, Reliability and Security. Lecture Notes in Computer Science, vol. 1698, pp. 216–225. Springer, Berlin (1999)
14. CSK Systems Corporation: VDMTools. Industrial Tool Supporting VDM++. http://www.vdmtools.jp/en (2014)
15. Cusumano, M., MacCormack, A., Kemerer, C.F., Crandall, B.: Software development worldwide: the state of the practice. IEEE Softw. **20**(6), 28–34 (2003)
16. Doornbos, R., Hooman, J., van Vlimmeren, B.: Complementary verification of embedded software using ASD and Uppaal. In: Innovations in Information Technology (IIT'12), pp. 60–65 (2012)
17. Esterel Technologies: SCADE Suite. Model Based Development Environment Dedicated to Critical Embedded Software. http://www.esterel-technologies.com/products/scade-suite (2014)
18. FDR homepage: http://www.fsel.com (2014)
19. Fitzgerald, J., Larsen, P., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object-Oriented Systems. Springer TELOS, Santa Clara (2005)
20. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley, New York (1995)
21. Groote, J., Kouters, T., Osaiweran, A.: Specification guidelines to avoid the state space explosion problem. In: Fundamentals of Software Engineering. Lecture Notes in Computer Science, vol. 7141, pp. 112–127. Springer, Berlin (2012)
22. Hausler, P.A.: A recent cleanroom success story: the Redwing project. In: 17th Annual Software Engineering Workshop. NASA Goddard Space Flight Center, Greenbelt (1992)
23. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, New York (1985)
24. Heitmeyer, C.: On the need for practical formal methods. In: Formal Techniques in Real-Time and Fault-Tolerant Systems. Lecture Notes in Computer Science, vol. 1486, pp. 18–26. Springer, Berlin (1998)
25. Holloway, M.C.: Why engineers should consider formal methods. In: 16th Digital Avionics Systems Conference, pp. 16–22. IEEE Press, New York (1997)
26. Hooman, J.: Lecture Notes in Computer Science. Specification and compositional verification of real-time systems, vol. 558. Springer, Berlin (1991)
27. Hooman, J., Huis, R., Schuts, M.: Experiences with a compositional model checker in the healthcare domain. In: FHIES 2011, LNCS, vol. 7151, pp. 93–110. Springer, Berlin (2012)
28. Hopcroft, P.J., Broadfoot, G.H.: Combining the box structure development method and CSP for software development. Electron. Notes Theor. Comput. Sci. **128**(6), 127–144 (2005)
29. IBM Rational Rhapsody family: http://www.ibm.com/software/products/en/ratirhapfami (2015)
30. Jones, C.: Software Assessments, Benchmarks, and Best Practices. Addison-Wesley Longman, Boston (2000)
31. Jones, C.B., Jackson, D., Wing, J.: Formal methods light. Computer **29**(4), 20–22 (1996)
32. Kurita, T., Nakatsugawa, Y.: The application of VDM to the industrial development of firmware for a smart card IC chip. Int. J. Softw. Inform. **3**(2), 343–355 (2009)
33. Larsen, P.: Ten years of historical development "bootstrapping" VDMTools. J. Univ. Comput. Sci. **7**(8), 692–709 (2001)
34. Linger, R.C.: Cleanroom software engineering for zero-defect software. In: Proceedings of the 15th International Conference on Software Engineering, pp. 2–13. IEEE Computer Society Press, Los Alamitos (1993)
35. Linger, R., Spangler, R.: The IBM cleanroom software engineering technology transfer program. In: Software Engineering Education. Lecture Notes in Computer Science, vol. 640, pp. 380–394. Springer, Berlin (1992)
36. MATLAB: MathWorks. http://www.mathworks.com (2015)
37. McConnell, S.: Code Complete, 2nd edn. Microsoft Press, Redmond (2004)
38. McConnell, S.: Software Estimation: Demystifying the Black Art. Microsoft Press, Redmond (2006)
39. Mills, H.: Certifying the correctness of software. In: 25th HICSS, Hawai, pp. 373–381 (1992)
40. Mills, H.: Stepwise refinement and verification in box-structured systems. Computer **21**, 23–36 (1988)
41. SourceMonitor homepage: http://www.campwoodsw.com/sourcemonitor.html (2014)
42. Uppaal, UP4ALL, Uppsala: http://www.uppaal.com (2015)
43. Verum homepage: http://www.verum.com (2014)
44. Woodcock, J., Larsen, P., Bicarregui, J., Fitzgerald, J.: Formal methods: practice and experience. ACM Comput. Surv. **41**(4), 1–36 (2009)