Early Fault Detection in Industry Using Models at Various Abstraction Levels*

Jozef Hooman[1,2], Arjan J. Mooij[2], and Hans van Wezep[3]

[1] Computing Science Department, Radboud University Nijmegen, The Netherlands
[2] Embedded Systems Institute, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{jozef.hooman,arjan.mooij}@esi.nl
[3] Interventional X-Ray Department, Philips Healthcare, Best, The Netherlands
hans.van.wezep@philips.com

Abstract. Most formal models that are used in the industry are close to the level of code, and often ready to be used for code generation. Formal models can also be analysed and verified in order to detect any faults. As the first formal models are often such code-level models, their analysis not only reveals a lot of detailed design faults, but also the more relevant conceptual faults in the design and the requirements. Our observations are based on our experiences in an industrial development project that uses a commercial tool for formal modelling, compositional verification, and code generation. In addition to the provided tool functionality, we have introduced formal techniques to detect conceptual faults during the earlier design and requirements phases. To this end we have made additional formal models, both for the requirements and for the early designs at various abstraction levels. We have analysed these models using simulation and interactive visualization, and we have compared them using refinement checking.

1 Introduction

The formal methods that are currently the most successful in the industry are methods with commercially supported tools that provide code generation. An example is the industrial tool VDMTools [7] for the formal language VDM++ [11]. Similarly, the B-method [1], which has been used to develop a number of safety-critical systems, is supported by the commercial Atelier B tool [6]. The SCADE Suite [10] provides a formal industry-proven method for critical applications with both verification and code generation.

We report about our experiences with the industrial application of formal methods and the introduction of techniques for early fault detection [18,19]. In particular, we report about our experiences in an industrial development project at Philips Healthcare to develop control software for an interventional X-ray system. A brief description of this type of system and the developed control components can be found in Section 2.

* This publication was supported by the Allegio project, as part of the Dutch national program COMMIT, and the ITEA project Care4Me.

J. Derrick et al. (Eds.): IFM 2012, LNCS 7321, pp. 268–282, 2012.

This development project uses a formal approach called Analytical Software Design (ASD) [5,21] that is supported by the commercial tool ASD:Suite [29] of the company Verum. The ASD approach uses two types of formal models: design models and interface models; both types of models are described using state machines in a tabular notation. A prerequisite for the introduction of ASD is a layered architecture with a particular communication pattern between components. Control components can be realized by an ASD design model which implements one interface model; in addition, each component can use any number of interfaces. The use of formal interface models supports concurrent software development and prevents certain integration faults.

ASD:Suite can formally verify whether each design model together with the used interface models refines the implemented interface model. The ASD approach is compositional as each component can be verified in isolation, using only the interface model of any component with which it interacts. In addition, properties like absence of deadlocks and livelocks can be checked. These formal checks can reveal subtle faults in the components, such as race conditions. An analysis of earlier applications of ASD at Philips Healthcare showed that units containing ASD components have less reported defects than other units [16].

A very important feature in industrial contexts is that ASD:Suite can use the design models to generate implementation code in a number of programming languages (C, C++, C#, Java). Code generation adds immediate value [12] to the modelling efforts that are involved in formal approaches. This also prevents the introduction of certain implementation faults related to thread creation and synchronization. Section 6 provides more details about the ASD approach.

Problem statement. In industrial development processes, the first formal models are usually close to the level of code, whereas the earlier design and requirements phases are based on informal documents. As a result, formal modelling and analysis not only reveal a lot of detailed design faults, but also the more relevant conceptual faults in the design and the requirements. These conceptual faults are often costly to repair in a detailed design phase. Moreover, as faults from several development phases are detected, it gives the impression that applying the formal approach itself is very time consuming.

Our approach. In addition to the commercial tool ASD:Suite for formal modelling, compositional verification and code generation, we have introduced formal techniques to detect conceptual faults during earlier design and requirements phases. To this end we have made additional formal models, both for the requirements and for early designs at various abstraction levels. In comparison to code-level models like ASD, such models may ignore any restrictions imposed by specific implementation technologies, they may rely on additional assumptions, and they can use simplified external interfaces.

We have analysed these models using simulation and interactive visualization, and we have compared them using refinement checking; see also the schematic overview in Fig. 8. In the next three paragraphs we describe the kinds of faults that we have thus detected in early development phases. In traditional

development approaches, these faults would probably be detected later on during the test and integration phase, where they are more expensive to repair.

Modelling and analysing requirements. Throughout the development process, we have observed that many faults are due to some unclarities in the requirements, and that such faults often lead to time-consuming redesigns. Our first priority was to increase the confidence in the requirements. In addition to the traditional documentation, we have made formal, executable models of the required system behaviour, as is also advocated by [8].

Making formal requirements models directly triggers all kinds of questions about the interpretation of the requirements. In addition, we have validated the requirements models using simulation. To support discussions with domain experts and non-technical stakeholders, we have connected the simulations to an interactive visualization of a physical view on the system. This is described in Section 3.

Modelling and analysing designs. The ASD approach imposes some restrictions on the design models, e.g., to ensure that they can be verified compositionally. To validate high-level design decisions, we have made several formal, executable models of early designs that do not (yet) adhere to these restrictions. We have validated these models using simulation. In early development phases, we have thus revealed various design faults, often related to feature interactions, that lead to deadlocks, livelocks, and functional errors.

During the development phases, we have iteratively analysed increasingly detailed design models. The use of interactive visualizations (as discussed for the requirements models) also proved useful for the design models. To prepare the final application of ASD, we have defined a pattern for simulating the ASD components in detailed design models. More details are given in Section 4.

Comparing requirements and designs. When making several formal models, it often happens that small discrepancies are introduced. To some extent, these can be discovered using the light-weight simulations discussed before. For a more profound comparison, we have built a compiler that transforms a requirements model and a design model to the input language of a refinement checker.

The use of refinement checkers to compare models has revealed subtle design faults that are difficult to find by simulation. However, exhaustive verification easily hits the state space explosion problem, and scalability to industrial sizes is still challenging. In Section 5 we explain how we have addressed this.

2 Control Components for an Interventional X-Ray System

The experiences described in this paper are based on our work in a development project at Philips Healthcare. This project concentrates on the development of control components for interventional X-ray systems as depicted in Fig. 1. Such

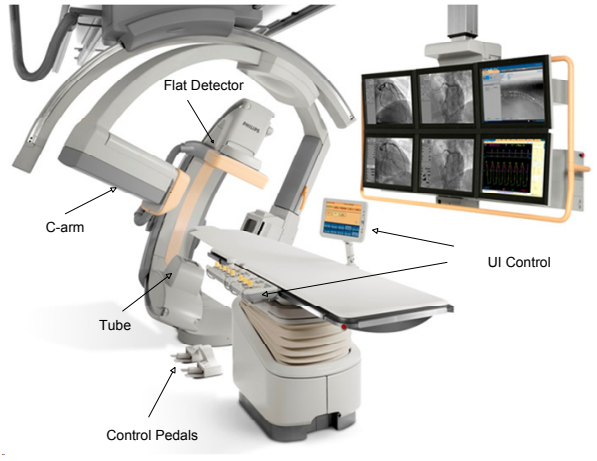


Fig. 1. Interventional X-Ray system

systems are used for minimally-invasive cardiac, vascular and neurological procedures, such as placing a stent via a catheter. During such a medical procedure, the surgeon is guided by real-time images showing, for instance, the position of the catheter inside the patient. These images are constructed from the amount of X-ray that is detected after sending X-ray beams (generated by the tube) through the patient.

The system under development consists of two X-ray planes (called lateral and frontal) that can be used in isolation or together (called biplane). Each plane can apply three types of X-ray that vary in the amount of X-ray that is emitted:

- Fluoroscopy: low dose, for interactive viewing and positioning;
- Exposure SingleShot: high dose, for recording a single image;
- Exposure Series: high dose, for recording a series of images.

The clinical users can control this system using six pedals and one hand-switch. Moreover, each of these inputs can be replicated multiple times. Three of the pedals are used to start Fluoroscopy, corresponding to the planes. For Exposure, there is one pedal to switch between the planes, and there are two pedals to start the two types of Exposure. In addition, Exposure Series can also be started using the hand-switch.

Apart from the user inputs, there can be several reasons for interrupting the X-ray beams once started, or for preventing the X-ray beams to start in the first place; such conditions are called run-conditions and start-conditions respectively. Examples include technical problems with the hardware, but also conditions related to physical safety such as an open door.

Given the earlier experiences [16,20] of Philips Healthcare with ASD, it was decided to develop the main control components using ASD. Moreover, their

external interfaces were already specified using ASD. The use of ASD is motivated by the aim to shorten the test and integration phase, which is usually long to ensure a high level of quality. Starting from the application of the ASD approach, we have experimented with the introduction of formal techniques to describe the requirements and the global design. The aim was to find faults as early as possible to improve the efficiency of the development process.

3 Modelling and Analysing Requirements

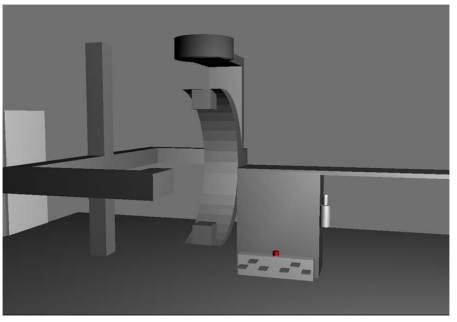
Since it is very costly to correct requirements faults during detailed design, we propose the use of formal techniques to detect such faults as early as possible. To obtain industrial acceptance and fast feedback, we have made executable models that can be simulated. In addition, the models have been simulated in combination with an interactive visualisation of the externally visible behaviour.

Executable models. To model the requirements, we have used the Parallel Object-Oriented Specification Language (POOSL) [9,28]. POOSL is a very expressive formal language with timing, predefined data types, statistical distributions and synchronous communication along channels, similar to CSP [25]. The semantics of POOSL is defined as a timed probabilistic labelled transition system. Models can be simulated by means of the tools supporting POOSL.

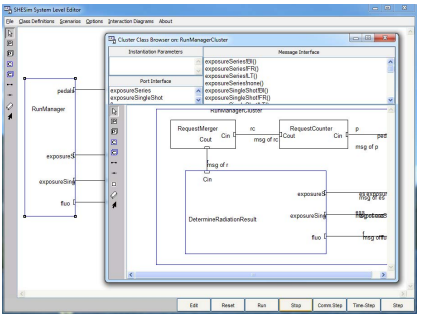
Interactive visualisation. To discuss the requirements with domain experts, we have investigated the use of interactive visualizations. The visualizations provide an attractive and understandable graphical user interface, but the logic follows from a simulation of the formal, executable requirements model.

In earlier work [23] we have used interactive 2D animations based on Flash. More recently we have started to explore the use of interactive 3D animations based on Blender [2]; see Fig. 2(a). Blender is an open source tool that combines a 3D modelling environment with an interactive game engine. Our first impression is that some situations (pedal states, active X-ray beams, etc.) are more easily understood using a realistic graphical view. In comparison to the professional animations that are used in the industry for explaining their products, our visualizations are interactive and the graphical aspects are separated from the internal logic. In this way, our interactive visualizations can also be used later in the development process to evaluate technical models of the architecture and detailed design.

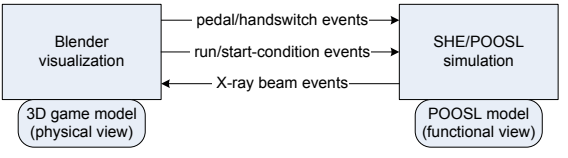
By connecting a Blender model (see Fig. 2(a)) and a POOSL model (see Fig. 2(b)) via sockets, we obtain a simulation that combines two views on the system: the physical hardware and the control logic, respectively. As shown in Fig. 2(c), inputs to the system are forwarded by Blender to POOSL, whereas resulting actions are transferred from POOSL to Blender. This combination can also be used to analyse the effectiveness of the full clinical workflows for executing the use cases. We can even imagine that such techniques can be used for training purposes, e.g., when the real system is not (yet) available.



(a) 3D visualization for simulation



(b) Modelling and simulation



(c) Coupling models

Fig. 2. Modelling and analysing requirements using Blender and POOSL

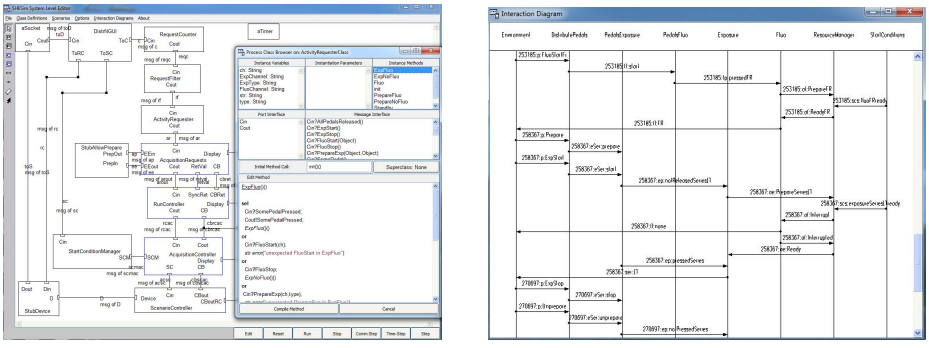
Results. We have started with modelling the types of X-ray and their relations. At any moment in time, only one type of X-ray may be active; in particular, the Exposure types have priority over Fluoroscopy. This basic behaviour was well documented, but the interpretation of the informal text was not always easy. For instance, regarding the intended result of simultaneous X-ray requests through different pedals and hand-switches. In these cases, our light-weight simulations quickly clarified the intended interpretation of the descriptions.

In a later phase, we considered error handling by adding start-conditions and run-conditions to the models. For instance, as Exposure Series can only be started after an additional preparation phase, it needs to be decided when the start-conditions should be checked, and whether any ongoing Fluoroscopy needs to be stopped at the beginning of the preparation phase. The formal, executable models turned out to be useful to make some implicit domain knowledge explicit. In this way time-consuming re-designs can be avoided.

4 Modelling and Analysing Designs

To enable a successful application of ASD, the design should meet a number of constraints. During our project, it turned out to be difficult to devise a design that meets these ASD constraints. The main limiting factors are that ASD:Suite concentrates on single components only and that it does not support simulation.

To get some insight in the essential structure and interaction between the components, we have first made several abstract design models and simulated them using POOSL. The graphical part of a POOSL model shows the structure



(a) Modelling and simulation (b) MSC generated by simulation

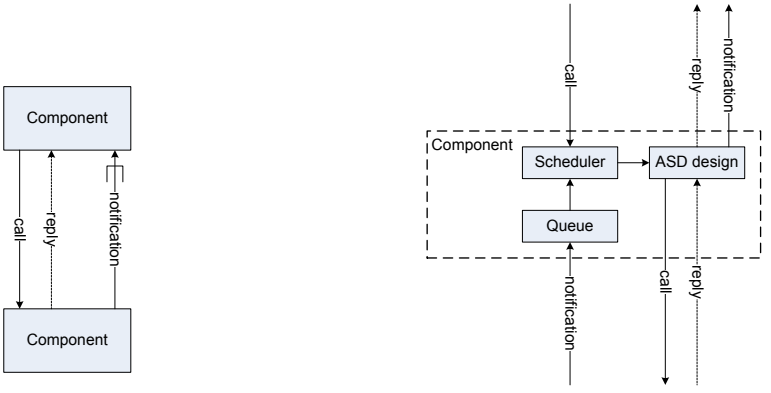
Fig. 3. Modelling and analysing designs using POOSL

of components and their interaction, as depicted in Fig. 3(a). Components can be clustered into hierarchies. At the lowest level, a component is described in a textual CSP-like language. During the simulation, the POOSL tool shows the internal state of each component, and also the interactions between the components in terms of a Message Sequence Chart (MSC); see Fig. 3(b). These features enable a developer to perform a detailed analysis of whether and how the design supports the typical use cases and their interactions.

Abstract designs. The first design model that we made focused on the component structure and the basic component interaction, without considering message parameters and error handling. After a number of iterations, a first satisfactory abstract design was obtained. An important aspect of this design is the decoupling between the fast interactions with the users of the system and the more time-consuming control of the X-ray devices in the system. This design was informally validated using simulation and inspection of the generated MSCs by industrial domain experts.

Afterwards, this design has been refined gradually with message parameters and error handling. Some restructuring was needed to keep the components small and to achieve the required behaviour. For instance, one of the early design models revealed that after a specific scenario the X-ray beam was erroneously not switched off, although all pedals were released. Clearly such important faults would also be detected immediately during the test and integration phase, but the benefit of formally analysing the design models is that the fault is detected earlier, and hence it is faster and cheaper to repair.

ASD-based designs. The application of ASD requires a layered architecture where components can interact as described in Fig. 4(a). The components in higher layers may perform function calls (resulting in a reply) on components at lower layers, but not the other way around. Components in lower layers may



(a) Component interactions (b) Modeling ASD components in POOSL

Fig. 4. ASD interaction patterns

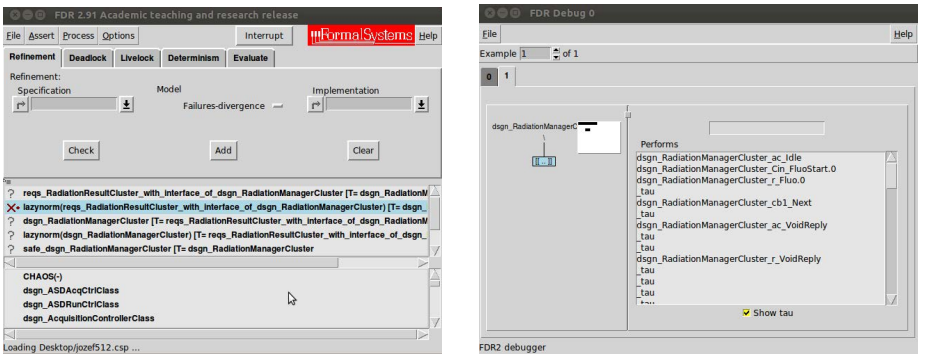
send asynchronous notifications to components in higher layers. There should be no direct interaction between components in the same layer. In this way, ASD can achieve absence of deadlocks by construction.

The POOSL models that contain ASD design models need to take this into account. Notifications have to be buffered in one queue per component. The ASD semantics also prescribes specific scheduling rules for the order in which the calls and queued notifications are processed. This semantics can be captured in POOSL by a cluster consisting of three components, as shown in Fig. 4(b).

5 Comparing Requirements and Designs

To check the consistency between the requirements and design models, we have used the formal refinement checker FDR2 [13,14]; see the screenshot in Fig. 5(a) together with a debug trace for a detected fault in Fig. 5(b). To this end, we have built a compiler that translates two models (using a subset of the POOSL language) to a CSP model, which is the input of FDR2; see also Fig. 8. An alternative refinement checker would be the Process Analysis Toolkit (PAT) [27], which is also based on a CSP dialect. In comparison to simulations, refinement checkers can automatically verify a lot of subtle scenarios.

The FDR2 tool supports, amongst others, trace refinement and failures-divergence refinement [14]. As explained in Section 6, Verum's tool ASD:Suite successfully manages to apply FDR2 to the verification of components. However, directly comparing full design and requirements models using FDR2 easily becomes infeasible. In the following, we explain how we were able to use FDR2 to find subtle discrepancies between the requirements and design models. We conclude with some typical results that we have obtained.



(a) Main window of FDR2 (b) Debug trace generated by FDR2

Fig. 5. Comparing requirements and design models using FDR2

Making the models finite state. Tools like FDR2 apply explicit state space exploration, and hence the state space of the models must be finite. The models that we consider consist of several components. In general, even if the components have an unbounded state space, it is possible that in their composition only a finite part is reachable. However, to analyse any model using FDR2, not only the state space of the entire model must be finite, but also the state space of each of the components must be finite.

The notification queues that are used in ASD designs are unbounded. Verum provides various suggestions for users of ASD:Suite to deal with this for single components, but these are not generally applicable to full design models. We have manually introduced an upperbound on the size of each queue. If the bound is violated, a special event is generated. The upperbound is valid in the design if a trace refinement indicates that this special event cannot occur.

For some external events, such as run-conditions, the bound on the queue size depends on timing aspects (the speed with which events are generated and consumed). To do partial verification in these cases, we have restricted the number of these external events that can be generated.

Making the refinement checks feasible. The requirement that the state space must be finite is just a minimum requirement. As FDR2 applies explicit state space exploration, the size of the state space requires constant attention when applying FDR2 to full models. In our project, generating the state spaces of the models is not the biggest issue. After generating the state spaces, however, FDR2 applies a normalization step before the real refinement checking begins. For complex specifications, it is known [25] that this normalization step can take a long time, and that the normalized version can be much larger than the original. To be able to find at least some traces that distinguish the requirements and design models, we have used FDR2's function "lazynorm" [26,24,25] that does not attempt to normalise the specification completely before carrying out the refinement check.

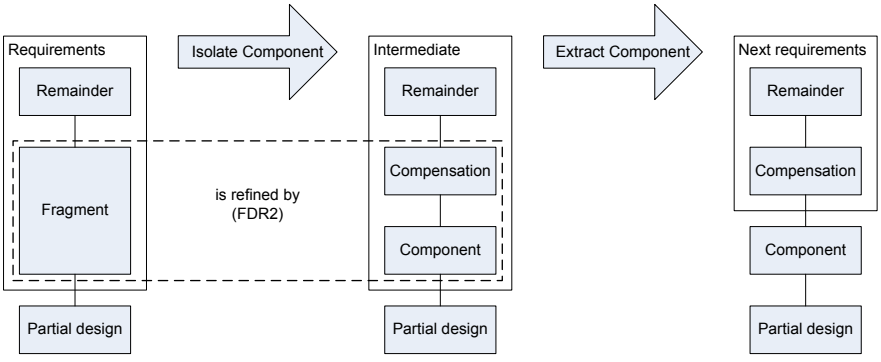


Fig. 6. Decomposition by extracting components

Most of the faults that we have detected using checks for failures-divergence refinement were actually at the level of trace refinement. Failures-divergence refinement implies trace refinement, and the latter seems easier to check. When feasibility is an issue, we have switched to checking for just trace refinement.

Decomposing models by extracting components. Another way to make refinement checking feasible in practice is to decompose a single refinement check on large models into several separate refinement checks on smaller models. In particular, we have considered a decomposition related to a typical design process that iteratively identifies design components based on the requirements. The idea is to gradually transform a requirements model into a design model; the intermediate stages combine some design components with the remaining requirements.

Consider the upper part of Fig. 6, i.e., ignore the blocks labelled "Partial design". The starting point at the left is a requirements model. After having defined a suitable design component, this component is related to a fragment of the requirements model. The aim is to show that this requirements fragment can be replaced by the composition of two blocks, viz., the component and some compensation. That is, the component and the compensation together should be a refinement of the original fragment. We have constructed the compensation manually, but ideally this would be automated using techniques like submodule construction [17,4] or equation solving [22]. If we can construct a compensation, then we have managed to isolate the component.

To finish this step, we extract the component from the requirements. Thus we obtain a model consisting of a part of the requirements (the remainder of the requirements and the compensation) and a model consisting of a part of the design (the design component). Afterwards we apply this approach again on the requirements part, as indicated by the blocks labelled "Partial design" in the full version of Fig. 6. In this way, the design model grows gradually and the formal refinements deal with a part of the original requirements only.

The validity of this decomposition approach depends on the associativity and congruence properties of composition. We have applied this approach mainly to

some conceptually simple components in the design that have a large state space, for example, because of internal counters. By extracting these components, the state space of the remaining models is reduced drastically.

Restricting the interfaces. The design models have to deal with the real technical interfaces, whereas the requirements models may make some simplifying assumptions. To compare requirements and design models, we have added some components that perform the conversions between these interfaces. In particular, we have extended the design model with components that translate the more detailed design interfaces to the simpler requirements interfaces, thus hiding some of the technical details.

In some cases, we have also restricted the possible external events. Although this means that we are not performing a full verification, again it has led to the detection of faults that were not found otherwise (see also the guiding techniques proposed in [15]). This can also be used for comparing a design model with a requirements model that is only correct under some assumptions.

Results. This kind of analysis reveals very subtle faults earlier in the development process than using traditional development approaches. For instance, we have detected some discrepancies between our models with respect to the requirement that in certain situations X-ray requests are cancelled when a pedal is released quickly after it has been pressed. In some cases, the problem turned out to be an inaccuracy in our requirements model.

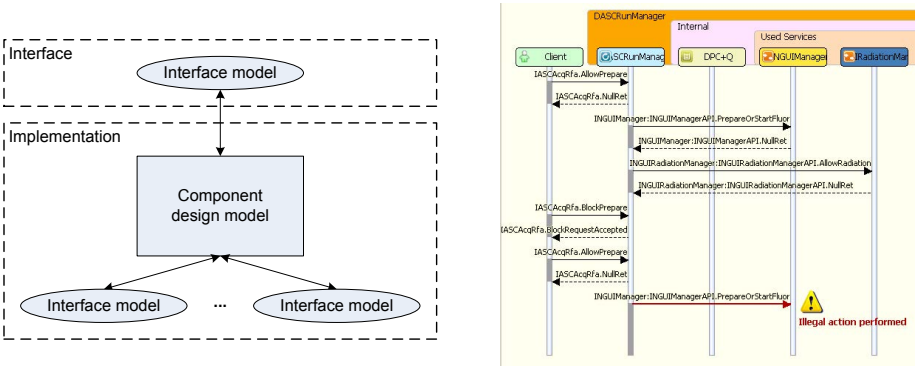
Note that our design model is not failures-divergence equivalent to the requirements model. For instance, when biplane Fluoroscopy cannot be started because of a start-condition, instead only one of the planes without a start-condition (if any) should be started. In the requirements model this is specified as a non-deterministic choice between the lateral or the frontal plane, whereas the design model implements this choice deterministically, which is fine.

6 Detailed Design with ASD

Once we had obtained confidence in the design, the individual components were realized using the ASD approach. We briefly describe the four ASD phases: interface modelling, design modelling, formal verification, and code generation. A small example that illustrates ASD can be found, for example, in [20].

Interface modelling. First, the internal and external interfaces of all components have been modelled using ASD. Such an interface model is a state machine that defines which calls and notifications are allowed and in which order. For instance, it may state that a *StartXRay* call is always followed by a *Started* or *StartFailed* notification; afterwards, a next *StartXRay* call may only occur after a *StopXRay* call. Thus, an interface model can be seen as a contract about the interaction protocol between components.

The tool ASD:Suite includes a number of basic consistency checks on the interface models. Most important is that the interface model must be complete in



(a) Verification condition (b) MSC generated by verification

Fig. 7. Verification of an implementation with respect to an interface

the sense that the response to each call or notification must be defined explicitly in all states. If a call or notification should not occur in a state, then it can be declared to be illegal.

Design modelling. After the definition of the interface models, which was a joint team effort, the components were developed concurrently. Components with data manipulations were implemented manually and tested to check compliance with their interfaces. The control components that do not involve any data manipulations were implemented using ASD design models. An ASD design model is a state machine, but, in contrast to an interface model, it must be deterministic. It defines how the component responds to calls and notifications, e.g., by performing calls and notifications to other components.

Formal verification. ASD:Suite can verify each design model for properties like absence of deadlocks and livelocks. In addition, it can verify whether each component is correct with respect to its interfaces, which means that

- no illegal calls are performed on the used interfaces; and
- the design model composed with all used interface models is a failures-divergence refinement of the implemented interface model (these are called implementation and interface respectively in Fig. 7(a)).

The motivation for these verifications is that the component interfaces are a frequent source of faults during integration. Such a verification can already eliminate such faults during the design. However, it does not guarantee the functional correctness of the components nor the entire design.

These verifications are implemented in ASD:Suite by internally transforming the models into CSP and then invoking the refinement checker FDR2; see also Fig. 8. To make the verification feasible, it is important to keep (the state space of) the components small and to limit the number of notifications.

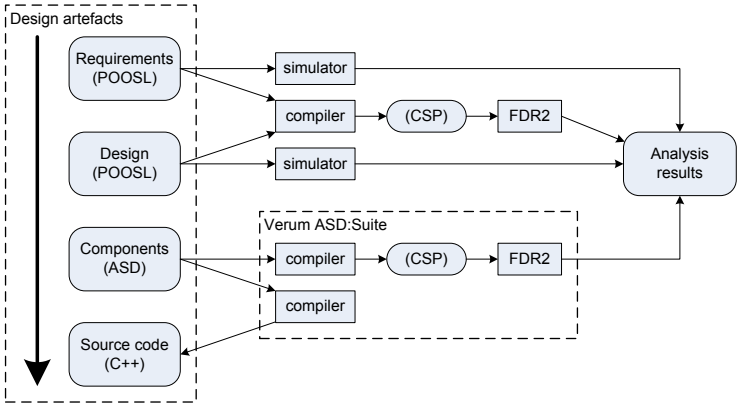


Fig. 8. Overview of models and transformations

This kind of verification typically reveals all kinds of race conditions due to the non-deterministic arrival order of events. For instance, many detected faults were related to the run-conditions, which can interrupt the normal execution flow at almost any point in time. These faults manifest themselves as refinement errors or as calls or notifications that are illegal according to the interface models. Each fault is reported as a Message Sequence Chart (MSC) covering the component and its interfaces; see Fig. 7(b).

Code generation. Finally, when the design models have been verified, ASD:Suite can generate code from the design models. Currently, ASD:Suite offers a choice between a number of programming languages (C, C++, C#, or Java).

7 Conclusions and Further Work

The first formal models that are made during development are usually close to the code level. The danger is that the formal analysis of such models reveals all kinds of faults from all previous development phases. In our industrial project, we have observed that this complicates the application of formal methods and causes delays in the project planning. To detect faults earlier, i.e., in the appropriate development phase, we have introduced additional formal models at various levels of abstraction and earlier in the development process.

In our industrial project, the formal ASD approach has been used, supported by the commercial tool ASD:Suite of the company Verum. We have added a number of techniques on top of this approach, as summarized in Fig. 8:

- modelling: to define the functionality precisely;
- simulation: to explore the functionality described in the models;
- visualization: to provide a user interface based on a physical system view;
- refinement checking: to rigorously compare pairs of models.

The main benefit of applying such techniques is early fault detection. By not only applying them to design models, but also to requirements models, developers can create a better understanding of the requirements in earlier stages of the development process.

It is well-known that unclear requirements lead to costly redesigns. This becomes very visible in the context of formal approaches such as ASD, because redesigns also require that the interface models have to be adapted, and that a large number of formal verification conditions has to be satisfied again. To use ASD more effectively, the conclusion is, also at Philips Healthcare, that more attention should be paid to the definition of the requirements.

Making formal models of the requirements, and analysing these models using simulation and interactive visualization has proved to be very useful to remove a number of unclarities. On the other hand, we have also observed that not all faults can be found practically in this way. Using refinement checking, we have detected additional faults in scenarios where a number of fast user commands are received while the system is still busy processing the first command. It is almost impossible to explore such subtle scenarios using interactive simulation.

Further work. There are various directions for further work. First of all, we are planning to evaluate the use of multiple formal models during the development of another, larger industrial system. In particular, we anticipate the need for additional kinds of analysis, such as a performance analysis for response times. It is also interesting to explore to what extent a good requirements model can formally guide the development of the design model.

On the tooling side, a compiler from (a restricted class of) POOSL design models to ASD components would be a nice extension. An all-in-one tool covering all these techniques might be interesting, but automated model transformations are our first priority. For the applicability of refinement checking, it is very relevant to investigate whether techniques such as submodule construction and equation solving, as mentioned in Section 5, can be applied at an industrial scale. It is also interesting to explore whether model-based testing [3] can be applied to the design models as an alternative for the refinement checking.

References

1. Abrial, J.R.: The B-book: assigning programs to meanings. Cambridge University Press, New York (1996)
2. Blender, `http://www.blender.org/`
3. Boberg, J.: Early fault detection with model-based testing. In: Proceedings of Erlang Workshop 2008, pp. 9–20. ACM (2008)
4. von Bochmann, G.: Using First-Order Logic to Reason about Submodule Construction. In: Lee, D., Lopes, A., Poetzsch-Heffter, A. (eds.) FMOODS/FORTE 2009. LNCS, vol. 5522, pp. 213–218. Springer, Heidelberg (2009)
5. Broadfoot, G.H., Broadfoot, P.J.: Academia and industry meet: Some experiences of formal methods in practice. In: Proceedings of APSEC 2003, pp. 49–58 (2003)
6. ClearSy: Atelier B, `http://www.atelierb.eu/en/`
7. CSK Systems Corporation: VDMTools, `http://www.vdmtools.jp/en/`

8. Easterbrook, S.M., Lutz, R.R., Covington, R., Kelly, J., Ampo, Y., Hamilton, D.: Experiences using lightweight formal methods for requirements modeling. IEEE Transactions on Software Engineering 24(1), 4–14 (1998)
9. Eindhoven University of Technology: Software/Hardware Engineering (SHE) - Parallel Object-Oriented Specification Language (POOSL), http://www.es.ele.tue.nl/poosl/
10. Esterel Technologies: SCADE Suite, http://www.esterel-technologies.com/products/scade-suite/
11. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object-oriented Systems. Springer, New York (2005)
12. Fitzgerald, J.S., Larsen, P.G.: Balancing Insight and Effort: The Industrial Uptake of Formal Methods. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) Formal Methods and Hybrid Real-Time Systems. LNCS, vol. 4700, pp. 237–254. Springer, Heidelberg (2007)
13. Formal Systems (Europe) Ltd: FDR2, http://www.fsel.com/
14. Formal Systems (Europe) Ltd and Oxford University Computing Laboratory: Failures-Divergence Refinement – FDR2 User Manual, 9th edn. (2010)
15. Goga, N., Romijn, J.: Guiding Spin Simulation. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 176–193. Springer, Heidelberg (2004)
16. Groote, J.F., Osaiweran, A., Wesselius, J.H.: Analyzing the effects of formal methods on the development of industrial control software. In: Proceedings of ICSM 2011, pp. 467–472. IEEE (2011)
17. Haghverdi, E., Ural, H.: Submodule construction from concurrent system specifications. Information & Software Technology 41(8), 499–506 (1999)
18. Holzmann, G.J.: Early fault detection tools. Software - Concepts and Tools 17(2), 63–69 (1996)
19. Holzmann, G.J.: Formal Methods for Early Fault Detection. In: Jonsson, B., Parrow, J. (eds.) FTRTFT 1996. LNCS, vol. 1135, pp. 40–54. Springer, Heidelberg (1996)
20. Hooman, J., Huis in 't Veld, R., Schuts, M.: Experiences with a Compositional Model Checker in the Healthcare Domain. In: George, C. (ed.) FHIES 2011. LNCS, vol. 7151, pp. 93–110. Springer, Heidelberg (2012)
21. Hopcroft, P.J., Broadfoot, G.H.: Combining the box structure development method and CSP for software development. ENTCS 128(6), 127–144 (2005)
22. Larsen, K.G., Xinxin, L.: Equation solving using modal transition systems. In: Proceedings of LICS 1990, pp. 108–117. IEEE Computer Society (1990)
23. Li, L., Hooman, J., Voeten, J.: Connecting technical and non-technical views of system architectures. In: Proceedings of CPSCom 2010, pp. 592–599 (December 2010)
24. Roscoe, A.W., Armstrong, P.J., Pragyesh: Local Search in Model Checking. In: Liu, Z., Ravn, A.P. (eds.) ATVA 2009. LNCS, vol. 5799, pp. 22–38. Springer, Heidelberg (2009)
25. Roscoe, B.: Understanding Concurrent Systems. Springer (2010)
26. Ryan, P.Y.A., Schneider, S.A., Goldsmith, M.H., Lowe, G., Roscoe, A.W.: The Modelling and Analysis of Security Protocols: the CSP Approach. Pearson Education (2000)
27. Sun, J., Liu, Y., Dong, J.S.: Model checking CSP revisited: Introducing a process analysis toolkit. In: Proceedings of ISoLA 2008. CCIS, vol. 17, pp. 307–322. Springer (2008)
28. Theelen, B.D., Florescu, O., Geilen, M., Huang, J., van der Putten, P.H.A., Voeten, J.: Software/hardware engineering with the parallel object-oriented specification language. In: Proceedings of MEMOCODE 2007, pp. 139–148. IEEE (2007)
29. Verum Software Technologies: ASD:Suite, http://www.verum.com/