T U M

I N S T I T U T F Ü R I N F O R M A T I K

Safety-Critical System Development Methodology

F. Hoelzl, M. Spichkova, D. Trachtenherz

TUM-I1020
November 10

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-11-I1020-0/1.-FI
Alle Rechte vorbehalten
Nachdruck auch auszugsweise verboten

©2010

Druck: Institut für Informatik der
Technischen Universität München

Safety-Critical System Development Methodology

Florian Hölzl, Maria Spichkova, David Trachtenherz

Institut für Informatik, Technische Universität München
Boltzmannstr. 3, 85748 Garching, Germany
{hoelzlf, spichkov, trachten}@in.tum.de

Abstract. The purpose of this work[1] is to integrate verification techniques in real industrial development processes – from informal textual specification and analysis of requirements to a verified implementation. Therefore, we present methods necessary to bridge the gap from informal requirements towards formal specification and from there to executable implementation. We show which development steps are necessary and how the specifications and models are verified using both automatic and interactive techniques.

1 Introduction

Embedded software-based systems development has become a most challenging field of software engineering research and industrial application. These systems underlie real-time requirements, they are safety critical, they must be highly reliable, and they are distributed over multiple processing units. Typical example systems can be found in modern cars and airplanes, but also in power plants, production facilities, and electronic consumer products. Building correct control software becomes more and more complicated in these application domains because the size and the complexity of such systems increases from day to day.

Currently, software correctness is accounted for through extensive testing efforts, but it is well known that testing cannot prove correctness of systems, but it can only demonstrate the presence of errors in exemplary cases. On the contrary formal verification methods can show total correctness of systems w.r.t. the intended, specified behavior. Both automatic and interactive verification techniques have been successfully applied in integrated circuits and processor design. However, software verification turned out to be much more difficult, since software serves a large variety of purposes: there is software working close to the hardware environment like operating systems and driver software, while application software implements the customer functionality.

We need defined methodologies to build and verify these different kinds of software. This is particularly difficult because a complete embedded system product consists of all these parts working tightly together: application software is

[1] This work was fully funded by the German Federal Ministry of Education, Science, Research and Technology (BMBF) in the framework of the Verisoft XT project. The responsibility for this article lies with the authors.

executed in the context of an operating system, which in terms is executed on a hardware execution unit and interacts with peripheral components like sensors or actuators. In order to obtain a fully verified embedded system all the technical issues have to be considered. We have to verify the hardware, the operating system and the application software.

We are well aware that system development in an industrial setting seldom proceeds from requirements to implementation in a single run, but uses review and feedback cycles to address changes found as development progresses. However, for the sake of clarity of presentation we do not provide a process model with change cycles, but stick to the presentation of the development artifacts and the applied verification techniques.

The paper is organized as follows: Section 2 discusses the motivation for the presented approach and related work, this includes the discussion of the differences between the methodology, we propose here, and the mentioned case studies our methodology has evolved from. Section 3 introduces the development methodology as a whole, while Sections 4 – 8 discuss the different artifacts, while Section 9 concludes this paper.

2 Motivation and Related Work

There is a number of works on integration of different system models and verification techniques. For instance, the Ptolemy approach (see [8]) introduces a general way to combine heterogeneous models of embedded systems. A prominent example of integration of verification techniques is a combination of Model Checking and Deduction for I/O-Automata done by O. Müller and T. Nipkow (see [19]). However, to our best knowledge there are no other works on achieving a *pervasive formal development process* for embedded applications starting with informal textual specification and leading to verified machine-code. This direction has been touched for the first time in [3], though only for upper layer of automotive systems and focused on later verification phases – in contrast, the contribution of the work presented here is covering the entire seamless pervasive development process.

The first steps towards a methodology for development of verified embedded system have been done in [4,5].For example, a typical setting found in the automotive domain, a time-triggered operating and communication bus system, has been verified [16,3]. In this paper we deal with the verification of the application software. In comparison to the problem frame approach of M. Jackson [14] as well as the 4-variable model of Parnas and Madey [21], we present a pervasive formal development methodology for embedded systems starting from an informal textual specification of the requirements and going all the way to verified application code. Earlier results of the Verisoft project [3] have shown the methodology for later verification phases, in particular the relation between the application model and its execution environment, e.g. the operating system.

The proposed methodology evolved from experience gained through three case studies on embedded control systems: two industrial case studies from the

automotive area one case study from the business information systems area [6]. In the remainder of the paper we introduce the models and methods applied to achieve a seamless development methodology for embedded applications incorporating suitable verification techniques.

Related work for single phases of the presented methodology is given in each section it pertains to.

3 Development Methodology

Our methodology starts with the first phase of a real development process – the informal specification of the requirements – and concentrates on the verification aspects during the course of system development. Thus, we deal not only with *formal specifications*, but rather with *verification-oriented formal specifications* and *refinement-based verification* of safety-critical distributed systems (see also [26]).

Figure 1 illustrates the structure of the proposed development methodology in a top-down manner: from an informal specification through multiple transformation steps we get a verified formal specification, a verified executable model and also a verified C code implementation. The boxes represent development artifacts, the dark arrows show the dependency relations, i.e., which artifact is used as input for the development of the successor artifact. The light arrows show the proof relations between the artifacts.

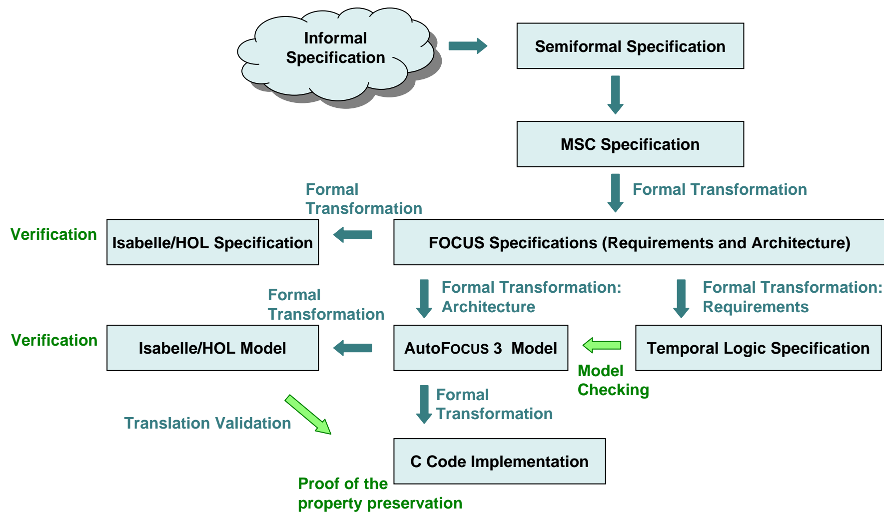

Fig. 1. Development Methodology

The *informal specification* of the requirements captures the relevant aspects of the system to be developed. It is usually given as a requirements document in

natural language. On the one hand the informal specification allows flexibility to capture the relevant aspects, but on the other hand it lacks precision from the formal point of view. While flexibility is desirable in the early stages of system development, the later phases, and verification methods in particular, need artifacts described in a formal language. Therefore, our first step is to derive a *tabular semi-formal specification* of the requirements. We use specific pre-defined syntactic patterns to transform the informal specifications by adding more details and precision.

The tabular semiformal specification can be also rewritten to a *Message Sequence Charts* (MSCs) representation [13,15] according to the approach presented in [24]. The purpose of using MSCs for specification of highly interacting systems is to obtain a better overview in comparison to a textual representation. However, this kind of specification does not give advantages in representation for all kind of systems. We consider this step as an optional one to be included if necessary.

The MSC specification or the semiformal specification, respectively, is translated to a specification in Focus [7], a framework for formal specifications and development of distributed interactive systems. This framework is preferred here over other specification frameworks since it has an integrated notion of time and modeling techniques for unbounded networks (where we have replications of system components of the same kind), provides a number of specification techniques for distributed systems and concepts of refinement. Moreover, Focus specifications are much more readable and manageable than specification done according to approaches like B-method [2] or Z [27] – the advantage of graphical notation is extremely important when we are dealing with systems of industrial size. Focus supports a variety of specification styles, which describe system components by logical formulas or by diagrams and tables representing logical formulas.

In general we represent in Focus two kinds of specifications: a *requirements specification* of the system and its *architecture specification* (corresponding to the black and the glass box view on the system, respectively). Both of them are extracted from the MSC specification and/or the semiformal specification. This representation prepares the ground to verify the system architecture specifications against the system requirements by translating both to the theorem prover Isabelle/HOL [20] via the framework "Focus on Isabelle" [25]. Dealing with the "Focus on Isabelle" approach we can influence the complexity of proofs and their reusability already during the specification phase. This is due to the treatment of specification and verification/validation methodologies as a single joint methodology with the main focus on the specification part. Moreover, using this approach one can perform automatic correctness proofs of syntactic interfaces for specified system components.

In some cases inconsistencies can still remain in the specification, model or code even after verifying certain properties – most often an important property was ignored as nicely stated by Donald E. Knuth's famous saying: "Beware of bugs in the above code – I have only proved it correct, not tried it.". Thus, not only verification techniques, but also testing and simulation must belong

to the development process. Therefore, as the next step of the methodology, we translate the architecture specification to a representation in the related CASE tool AutoFocus 3 [11], a scientific research prototype, which is a tool implementation based on the Focus approach. We can now use the simulation and model-checking facilities of this tool.

The requirements specification will be translated from Focus to temporal logic. This representation gives us a basis to model-check the AutoFocus 3 model against. The transformations from Focus to temporal logic and to the AutoFocus 3 representation are formal and schematic, given some constraints on the Focus specification are obeyed.

The AutoFocus 3 model is also exported to Isabelle/HOL to prove its properties – the AutoFocus 3 model is in general a *refinement* of a Focus specification, thus its properties can be slightly different, i.e., more strict, from the ones specified on the Focus layer. On the other hand, the proof schema, which has been developed for the Focus specifications, can be (partially) reused. Finally, the AutoFocus 3 model is transformed to a corresponding C code by a code generator. We can show that this step preserves properties of the model [12]. Altogether, the methodology guides us from an informal specification via stepwise refinement to a verified formal specification, a corresponding executable verified model, and also a corresponding verified C code implementation.

The proposed development methodology makes it possible to perform formal verification on different levels of system abstraction and development, ranging from formalisation of system requirements to program code. In this section we address verification techniques applicable for different specification techniques used in this project for different system development phases. We opt to use both techniques, model checking and interactive theorem proving, because they provide different features needed for different verification tasks and complementing each other. While model checking provides automatic verification for property notations of limited expressiveness like LTL, interactive theorem proving allows for using powerful notations like HOL at the price of semi-automatic verification, i.e., requiring user interaction to find a proof.

After the requirements and the system architecture have been formalised in Focus, they can be translated to Isabelle/HOL (cf. [25]) and the refinement relation between them can be validated. In most cases not only one refinement step is needed, but a number of them. Thus, we can use the idea of refinement layers and of a refinement-based verification [26]: we see any proof about a system as the proof that a more concrete system specification is a refinement of a more abstract one.

In AutoFocus 3 functional properties can be specified using temporal logic notations, especially LTL (Linear Temporal Logic [17]). Temporal properties can be checked using model checking tools, e.g., SMV [18]. That way, system properties expressible in LTL can be verified by exporting AutoFocus 3 models to SMV and model checking the corresponding temporal formulae [22], as has been performed, for instance, for selected safety-critical properties in [9]. To cover the cases where using model checking have its typical proplems, functional

properties can be formulated and proven in Isabelle/HOL. For this purpose a code generator has been developed for creating Isabelle/HOL representations of AutoFocus 3 models, as described in [28].

The final product of the development process is C code generated from AutoFocus 3 models. In order to obtain the formal guarantee the correctness of the code, the behavioural equivalence of the Isabelle/HOL representation of the model and generated code can be proven in Isabelle/HOL.

Table 1 gives short description how the statistics about the two industrial case studies from the automotive area[2] looks like. The size of the AutoFocus 3 model is approximately equal to that of the corresponding Focus specification. Until now formal verification for the case studies has only been completed within [9] on the AutoFocus layer using model checking, discovering several errors, e.g., modelling errors, inconsistencies between requirements and model, incomplete requirements. The pervasive verification of the second case study starting on the Focus layer and ending on the code layer is work in progress.

Table 1. Case Studies statistics

Methodology Artifact	Case study 1	Case Study 2
Semiformal Specification (atomic requirements)	30	70
MSC Specification (MSCs)	16	20
FOCUS Specifications (components)	10	ca. 70
FOCUS Specifications (hierarchy levels)	2	4
Code (lines of generated code)	ca. 3.000	ca. 17.000

4 Semiformal Specification

Based on an initial set of requirements, the informal specification[3] is structured and subsequently specified with the help of pre-defined text patterns. Furthermore, the logical interface of the system as well as the system states are identified. For this purpose we use a simplified version of an approach presented in [9,10], which we extended according to the needs of the overall development approach.

An informal specification consists of a set of words, which can be distinguished into two categories: content words and keywords (relation words). Content words are system-specific words or phrases, e.g. *"system is initialized"* or

[2] The first case study (referred in [9,24]) was motivated and supported by DENSO CORPORATION. The second case study [29] is ongoing and supported by Robert Bosch GmbH.

[3] The presented methodology focuses only on functional requirements, including timing aspects.

"Off-button is pressed". The set of all content words forms the logical interface of the system, which can be understood as some kind of (domain specific, system-dependent) glossary that must be defined in addition. Keywords are domain-independent and form relationships between the content words (e.g. "if", "then", "else"). A semiformal specification consists of a number of requirements described using the following textual patterns:[4]

WHILE	⟨*Some state*⟩
IF	⟨*Some event occurs or some state changes*⟩
THEN	⟨*Some event occurs or some state changes*⟩

An *event* describes a *point in time*, in which the system observes or does something; the duration of the event is not important, e.g., "driver presses a button". A *state* describes a system or component state *within some time period*, e.g., "a button is pressed". Strictly speaking, all states of a state space are disjunct, but in some cases it is more efficient to use a state hierarchy that must be described separately.

The semiformal specification can be given in a simple tabular form, where the columns are, e.g., unique requirement identifier, semiformal description of the requirement, names of the corresponding MSC, remarks, alternative description, and sentences from the informal specification, which were reformulated to the semiformal requirement. Using such a simple tabular description to structure the information from the informal specification, we can find out missing information quite fast. Furthermore, we identify possible synonyms that must be unified before proceeding to a formal specification. Analysis of the semiformal specification document should also yield sentences, which need to be reformulated or extended.

5 MSC Specification

The Message Sequence Charts (MSCs) representation [13,15] is widely used in telecommunication applications – for specification of systems that have complex interaction patterns and a comparatively small state space. They interact mostly according to their inputs and not so much according to their states and state changes. However, if a system has a large state space and the main (re)action of this system depends on its states and results in state changes, a graphical MSC specification becomes hardly readable. Thus, we gain no benefit w.r.t. the textual, tabular one and the effort to build an MSC specification is practically wasted. Therefore, we consider the MSC-based development step an optional one, which should be included only if necessary and useful in the application domain.

For example, for the system specified in [6], where the interaction between the system components was very intensive, the MSC specification was really wise,

[4] In some cases either the WHILE-part or the IF-part can be omitted.

the case study represented in [9,24] was also appropriate for this step. However, for the case study from the Verisoft XT project [29] this step turned out to be useless, because the system was primarily state-based.

6 FOCUS Specifications: Requirements and Architecture

A system in Focus is represented by its components that are connected by communication lines called *channels*, and are described in terms of its input/output behavior. Any Focus specification characterizes the relation between the *communication histories* for the external *input* and *output channels*. Thus, the formal meaning of a specification is exactly this external *input/output relation*. The components can interact and also work independently of each other.

A specification can be elementary or composite – composite specifications are built hierarchically from the elementary ones. Elementary specifications are divided in Focus into untimed, timed, and time-synchronous according to their level of time abstraction. In the methodology we use the timed kind of specifications only. This simplifies the verification (see also [25]) and allows for the schematic formal translation to the AutoFocus 3 model. Hence, we use a discrete notion of time, which allows us to precisely specify system components, and to compose them without causality anomalies that may occur in the untimed treatment.

7 Executable AutoFocus 3 Model

During the last step, we have obtained a formal model of the system architecture specified as a Focus specification. In order to obtain an executable model next, we transform this specification into an AutoFocus 3 model. AutoFocus 3 [23] is a scientific CASE tool prototype[5] implementing a modeling language based on a graphical notation and a restricted version of the formal Focus semantics, in particular the time-synchronous setting. The system structure specification is similar to the Focus architecture specification. It captures the static aspects of the system description. We specify a network of communicating components working in parallel (assuming a global synchronized time frame).

The AutoFocus 3 model is an execuarble one. Thus, we can validate the model using the AutoFocus 3 simulator to get a first impression of the system under development and possibly find implementation errors that we introduced during the manual transformation of the Focus specification into a AutoFocus 3 model. Automatisation of this transformation is future work.

8 C Code Implementation

In order to obtain executable code from an AutoFocus 3 model, we have implemented a C code generator [12]. As a result, we gain the advantage of being able

[5] `http://af3.in.tum.de/`

to compute memory consumption at compile time. We can also compute worst case execution times, since all operations and function calls are non-recursive, e.g., we can estimate the execution times with the tool *aiT* by AbsInt [1].

The code generation step is the last formal transformation in our methodology. The correctness of this step will be shown by paper and pencil proof of the generation algorithm similar to the proof for the Isabelle/HOL exporter [28]. Here, we show that the C0 program is an admissible simulation of the AutoFocus 3 model.

9 Conclusions

We have presented a methodology for development of safety-critical embedded software systems extended with pervasive verification techniques. We cover the development process from the initial informal requirements over several intermediary formal models to an implementation in a C language subset. The artifacts of the different stages can be created through schematic transformations, which are, depending on the development state of the tools, automatic or at least tool-supported. These artifacts are verified during each development phase: we apply both automatic and semi-automatic interactive verification techniques.

The feasibility of the proposed approach was evaluated on a number of case studies that cover different application areas of embedded control systems: information processing systems and automotive systems.

The ultimate goal of our approach, which is not entirely comparable to any other approach we know, is to obtain a seamless development process with precisely defined phases and methodologies to build fully verified software-based embedded systems: from textual requirements to verified code, from application software to execution environments.

References

1. AbsInt Angewandte Informatik. Worst-Case Execution Time Analyzers.
2. J.-R. Abrial. *The B-book: assigning programs to meanings*. Camb.Univ.Press, 1996.
3. J. Botaschanjan, M. Broy, A. Gruler, A. Harhurin, S. Knapp, L. Kof, W. Paul, and M. Spichkova. On the Correctness of Upper Layers of Automotive Systems. *Formal Aspects of Computing (FACS)*, 20(6):637–662, 2008.
4. J. Botaschanjan, A. Gruler, A. Harhurin, L. Kof, M. Spichkova, and D. Trachtenherz. Towards Modularized Verification of Distributed Time-Triggered Systems. In *FM 2006: Formal Methods*, pages 163–178. Springer, 2006.
5. J. Botaschanjan, L. Kof, C. Kühnel, and M. Spichkova. Towards Verified Automotive Software. In *2nd International ICSE workshop on Software*. ACM, 2005.
6. M. Broy, J. Fox, F. Hölzl, D. Koss, M. Kuhrmann, M. Meisinger, B. Penzenstadler, S. Rittmann, B. Schätz, M. Spichkova, and D. Wild. Service-Oriented Modeling of CoCoME with Focus and AutoFocus. pages 177–206, 2008.
7. M. Broy and K. Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer, 2001.

8. J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.
9. M. Feilkas, A. Fleischmann, F. Hölzl, C. Pfaller, S. Rittmann, K. Scheidemann, M. Spichkova, and D. Trachtenherz. A Top-Down Methodology for the Development of Automotive Software. Technical Report TUM-I0902, TU München, 2009.
10. A. Fleischmann. *Model-based formalization of requirements of embedded automotive systens*. PhD thesis, TU München, 2008.
11. F. Huber, B. Schätz, A. Schmidt, and K. Spies. AutoFocus - A Tool for Distributed Systems Specification. In *Proceedings of FTRTFT'96*, number 1135 in LNCS, pages 467–470. Springer, 1996.
12. F. Hölzl. The AutoFocus 3 C0 Code Generator. Technical Report TUM-I0918, Technische Universität München, 2009.
13. ITU-TS. Recommendation Z.120: Message Sequence Chart (MSC). Geneva, 1996.
14. M. Jackson. *Problem Frames: Analysing and Structuring Software Development Problems*. Addison-Wesley, 2001.
15. I. Krüger. *Distributed System Design with Message Sequence Charts*. PhD thesis, TU München, 2000.
16. C. Kühnel and M. Spichkova. Fault-Tolerant Communication for Distributed Embedded Systems. In *Software Engineering and Fault Tolerance*, Series on Software Engineering and Knowledge Engineering, 2007.
17. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer, 1992.
18. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
19. O. Müller and T. Nipkow. Combining Model Checking and Deduction for I/O-Automata. In E. Brinksma, R. Cleaveland, K. G. Larsen, T. Margaria, and B. Steffen, editors, *TACAS*, volume 1019 of *LNCS*, pages 1–16. Springer, 1995.
20. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
21. D. L. Parnas and J. Madey. Functional documents for computer systems. *Science of Computer Programming*, 25:41–61, 1995.
22. J. Philipps and O. Slotosch. The Quest for Correct Systems: Model Checking of Diagramms and Datatypes. In *APSEC'99*, pages 449 – 458. IEEE CS, 1999.
23. B. Schätz. Mastering the Complexity of Reactive Systems: the AutoFocus Approach. In F. Kordon and M. Lemoine, editors, *Formal Methods for Embedded Distributed Systems: How to Master the Complexity*, pages 215–258. Kluwer Academic Publishers, 2004.
24. M. Spichkova. From Semiformal Requirements To Formal Specifications via MSCs. Technical report.
25. M. Spichkova. *Specification and Seamless Verification of Embedded Real-Time Systems: FOCUS on Isabelle*. PhD thesis, 2007.
26. M. Spichkova. Refinement-based verification of interactive real-time systems. In *REFINE 2008*. ENTCS, 2008.
27. M. Spivey. Understanding Z – A Specification Language and Its Formal Semantics. Cambridge Tracts in Theoretical Computer Science 3. Camb. Univ. Press, 1988.
28. D. Trachtenherz. Ausführungssemantik von AutoFocus-Modellen: Isabelle/HOL-Formalisierung und Äquivalenzbeweis. Tech. Rep. TUM-I0903, TU München, 2009.
29. Verisoft XT Project. `http://www.verisoftxt.de`.