

Verified System Development with the AutoFocus Tool Chain

Maria Spichkova

Institut für Informatik, TU München,
Boltzmannstr. 3, 85748 Garching, Germany
spichkov@in.tum.de

Florian Hölzl

fortiss GmbH,
Guerickestr. 25, 80805 München, Germany
{hoelzl,f, trachten}@fortiss.org

David Trachtenherz

This work presents a model-based development methodology¹ for verified software systems as well as a tool support for it: an applied AUTOFOCUS 3 tool chain and its basic principles emphasizing the verification of the system under development as well as the check mechanisms we used to raise the level of confidence in the correctness of the implementation of the automatic generators.

1 Introduction

Software-based system development has become one of the most challenging fields of software engineering research and industrial application. To support the contemporary system development in industry CASE tools are used – they allow a simple and (mostly) intuitional design of distributed systems and applications, and executable code is generated directly from the models developed using these tools. In state-of-the-art industrial development, quality assurance is performed by extensive testing of the generated code. However, testing can only demonstrate the absence of errors for exemplary test cases, but not the correctness of the system. In opposite to testing, formal verification delivers a correctness proof for safety critical properties but requires significant effort. Thus, only the most critical parts of a system must be verified, and the whole process of specification and verification must be set up in a way that minimizes the overall effort – the verification process must be integrated into model-based development of safety-critical systems. On the other hand, testing and simulation must belong to the development process, because in some cases, even after verifying certain properties, inconsistencies can still remain in the specification, model or code – most often an important property is overlooked 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.”

There is a number of works on integration of different system models and verification techniques. For instance, the Ptolemy approach (see [7]) 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 [16]). 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

¹This 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.

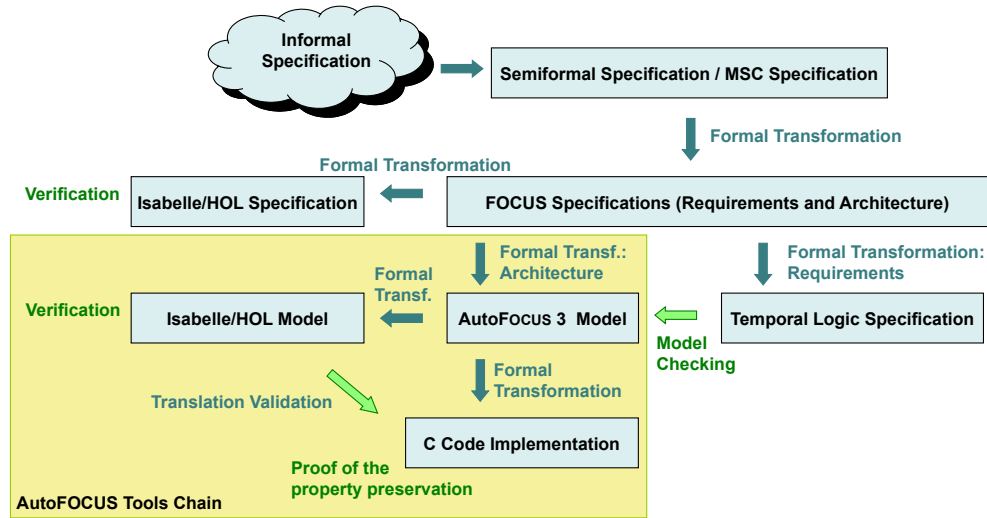


Figure 1: Development Methodology

communication bus system, has been verified [14, 3]. In this paper we deal with the verification of the application software. In comparison to the problem frame approach of M. Jackson [13] as well as the 4-variable model of Parnas and Madey [18], 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.

We outline here a development methodology for safety-critical systems with focus on the tool chain, which is employed in the design, implementation and verification phases of the methodology. The tool chain emphasizes verification of the system under development – it not only allows integration into the process of modeling, but also reduces the verification and integration effort.

2 Development Methodology

Fig. 1 illustrates the structure of the 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.

Beginning with a *requirements specification* (we focus here only on functional requirements, including timing aspects), which captures the relevant aspects of the system to be developed in flexible yet informal way, we derive a *tabular semi-formal specification*. This first step raises the level of precision by transforming the free text requirements into a structured form using specific pre-defined syntactic patterns as presented in [9]. 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 “*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 pattern, which is easily understood also by engineers unfamiliar with the formal methods:

WHILE *Some state*
IF *Some event occurs or some state changes*
THEN *Some event occurs or some state changes*
ELSE *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”. Using such a simple 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. This specification can be also rewritten to a *Message Sequence Charts* (MSCs, cf. [10]) representation 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.

Subsequently, the specifications are translated to FOCUS [6], a framework for formal specifications and development of distributed interactive systems. FOCUS 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, as well as graphical notation, which is extremely important when we are dealing with systems of industrial size. In general we represent in FOCUS two kinds of specifications: a formal specification of *system requirements* and the corresponding *architecture specification*. 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 [17] via the framework “FOCUS on Isabelle” [23].

As the next step, we translate the architecture specification to a representation in the related CASE tool AUTOFOCUS 3 [21, 2] to simulate the system. The requirements specification can be translated from FOCUS to temporal logic, which gives basis to model-check the model. 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. This model can also be exported to Isabelle/HOL to prove its properties – it 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, but the proof schema, which has been developed for the FOCUS specifications, can be (partially) reused.

The last step in our methodology is transformation of the model to a corresponding C code by a code generator: we show that the C program is an admissible simulation of the AUTOFOCUS 3 model. This concludes our methodology, but that is of course not the end of the complete development process: we need to deploy the code into the execution environment. 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 applicability of the methodology has been demonstrated by two case studies from automotive area. The first case study [8, 24], developing an Adaptive Cruise Control (ACC) system with Pre-Crash Safety functionality, was motivated and supported by DENSO Corporation, the second case study [25],

developing a Cruise Control System with focus on system architecture and system verification, was supported by Robert Bosch GmbH, Table 1 gives short description how the statistics about these two case studies, the size of the AUTOFOCUS 3 model is approximately equal to that of the corresponding FOCUS specification.

In each of the case studies we had a final number of system states was final according to the AUTOFOCUS 3 modeling paradigm, the most complex component was the component specified the main logic of the system. In the case of the Cruise Control System we split the specification of the main system logic into four components according to the formal decomposition approach [25] introduced within the system development methodology. There is a large number of approaches in area of decomposition (see, e.g., [11, 20, 27]). The main difference and the main contribution of our decomposition methodology is that it was developed for such a system architecture, where we already know (or, more precisely, have already specified them in a formal way) systems or components properties and need to decompose this whole properties collection to a number of subcomponents to get readable and manageable specifications.

An sample-property proven for the Cruise Control System looks like follows: *If the driver pushes the Accelerate-button while the system is on and none of the switch-off constraints occurs, the system must accelerate the vehicle during the next time unit respectively to the current speed and the predefined acceleration schema.* This means also that the system must analyze the information from sensors to check whether any switch-off constraints occurs, i.e. if the battery voltage is too low or if the gas pedal sensor fails.

Methodology Artifact	ACC / Pre-Crash Safety	Cruise Control: Architecture
Semiformal Specification (requirements)	30	70
MSC Specification (MSCs)	16	20
FOCUS Specification (components)	10	ca. 80
FOCUS Specification (hierarchy levels)	2	4
Isabelle Specification (lines of code)	-	ca. 12,500
Isabelle Model (lines of code)	-	ca. 38,000 (generated)
Verification of Isabelle Specification	-	ca. 8,500 lines of proof
Partial Verification of Isabelle Model	-	ca. 5,000 lines of proof
C-Code (lines of generated code)	ca. 3,000	ca. 17,000

Table 1: Case Studies Statistics

3 AutoFOCUS

AUTOFOCUS 3 [2, 21] is a scientific prototype implementing a modeling language based on a graphical notation designed to support the modeling of distributed, timed, reactive systems. The tool a restricted version of the formal FOCUS semantics, in particular the time-synchronous frame and supports different layers of models, their simulation and analysis. Main aspects of AUTOFOCUS 3 are hierarchical component decomposition of the system, modeling behavior of application parts of the system as well as modeling of technical architectures including control units and communication busses.

We give a brief introduction of the current language. In particular, we introduce three modeling views: *data type*, *system structure*, and *component behavior* specifications.

Every computer-based system can be seen as a data processing system. Thus programming and

modeling languages are based on some mechanism for specifying data items or more general data types. AUTOFOCUS 3 uses a functional language to specify data types, this provides a set of predefined types, such as Boolean and Integer. Based on these, we can define more complex types using variant or enumeration types and record-like types. User-defined functions allow us to implement additional operations for our data types.

Although in general data types and user functions may be recursive, we restrict them to be non-recursive for the domain of embedded systems, because this eases several tasks. Firstly, we obtain statically computable bounds for memory consumption and execution times. For embedded software this is often highly desirable, because of restricted hardware resources and the need to respect given timing bounds and constraints. Secondly, current model-checkers rely on bounded types.

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). Each component has a syntactic interface described by a set of ports. Each port is either an input or an output port, has a data type and an initial value. Furthermore, each component is declared to be weakly causal or strongly causal. Weak causality models instantaneous reaction, while strong causality models a delayed reaction. The network of components is formed by connecting ports with channels. From the semantics point of view, we need to avoid weakly causal feedback loops (Brock-Ackermann anomaly). Fortunately, this can be checked easily by the tool's model constraint checker.

System structure specifications may be separated into hierarchic views in order to deal with larger models. Components can be refined into a set of sub-components introducing both local communication and communication to the environment, e.g., through the interface of the parent component.

Atomic components have their behavior specified using one of the following variants: a stateful *automaton specification* or a stateless *function specification*. An automaton specification describes an input/output automaton. It consists of a set of control states, a set of typed data state variables, and a set of transitions. One of the control states is marked as the initial state. Every data state variable is also initialized to a given value.

Each transition has a source and a target control state. Furthermore, each transition specifies patterns of messages received via the input ports of the respective component and preconditions over the inputs and the data state variables. A transition can fire, if its source state is the current control state, the current input values match its input patterns, and the precondition is fulfilled. If a transition fires, the current state of the component changes to its target state, the output values of the component are updated according to the output pattern specification of the transition, and the data state variables are also updated as specified by the postcondition part of the transition specification. Note that more than one transition might be enabled for a given component state and set of input port values. Thus, component behavior can be non-deterministic.

Sometimes it is cumbersome to give an automaton specification for simple components. In particular, if the component does not need an internal state, a simple mapping from input values to output values is convenient. In AUTOFOCUS 3 this is modeled by a function specification. This is a simple table where each line defines such a mapping from inputs to outputs. Again, the table can define non-deterministic choices. From the semantic point of view function specifications can be seen as restricted automaton specifications.

Using the three views introduced above, we obtain an *executable model*. We can now 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 an ongoing work.

4 Generation Tool Chain

Our goal is to establish a sound verification environment with a high confidence in the correctness of the environment and thus of correctness the designed system, for this purpose we have combined AUTOFOCUS 3 and the theorem prover environment Isabelle/HOL by implementing a generator for the formal transformation of the design model into the theorem prover model, as well as a code generator for producing a C code implementation of the design model.

The central part is the transformation of the design model into the C code, i.e., the implementation path of the product under development. This is supported by two verification mechanisms: semi-automatic verification with a theorem prover and fully automatic verification by means of a model-checker. These flanking measurements fulfill two purposes. On the one hand they verify actual properties of the design model and the implementation code, On the other hand differing results of both techniques indicate an implementation fault in one of the generators.

When building a tool chain based on automatic generators it is vital to take great care about what one is doing – one must understand the semantics of the generation input, e.g., the AUTOFOCUS 3 model's semantics, and the target language, e.g., the C language. Since automatic code generation only makes sense, if the behavior of the generated program is equivalent to the behavior of the input model, we must show that the transformation is preserving the semantics. We strongly emphasize that the semantics of the design language has been defined before implementing the generators. Unfortunately, many code generation environments do not follow this up-front approach. They either do not care about behavior at all (i.e., generate structural outputs only) or consider the semantics of the generation input to be defined by the semantics of the generation outputs (i.e., the semantics of the design model is defined by the semantics of the generated code). The first case is of no further interest to us, since the generator only produces hull code that needs to be extended manually. The second case forces the user of the input language to work around flaws of the generator as we have seen with bugs in compilers. Usually, such flaws are exploited by flaws in the developed system, while in our approach the semantics of the design language provides the basis for the specification of the generators.

4.1 The Isabelle/HOL Translator

Formally specified requirements can be verified using a theorem prover: for this purpose we have developed a translator, which generates Isabelle/HOL theories from AUTOFOCUS models. The behavioural equivalence between AUTOFOCUS models and their generated representation in Isabelle/HOL has been shown in a paper-and-pencil proof [26].

The Isabelle/HOL code for an AUTOFOCUS model is created as follows. Firstly, the user initiates code export for the data dictionary, which generates a theory containing data type declarations and function definitions used in the model. Then the user may generate code for any of the components in the model: when selecting an atomic component, a theory will be generated containing the input/output interface definition for the component and the transition function originating from the automaton or function specification used to define the component's behaviour; when selecting a composite component, recursively the theories for all subcomponents will be generated and, ultimately, the theory for the considered component, whose transition function performs data transmission between the subcomponents and invokes all subcomponents' transition functions. Generating code for the root component of an AUTOFOCUS model yields a set of theory files encoding this model in Isabelle/HOL, as well as proof of theorems that support subsequent verification of model's properties in Isabelle/HOL.

4.2 The C0-Code Generator

In order to obtain executable code from an AUTOFOCUS 3 model, we have implemented a C0 code generator. A paper and pencil proof (cf. [12]) has shown the behavioral equivalence between the model and the generated code.

C0 is a C language subset constructed for usage with the Isabelle/HOL verification environment as discussed in [22]. It differs from C by restricting the language – a number of hazardous features, like pointer arithmetic, are forbidden in C0, which eases the reasoning and verification with Isabelle/HOL. As a result of these restrictions, we gain the advantage of being able to compute memory consumption at compile time as well as worst case execution times automatically, since all operations and function calls are non-recursive. [19] and [15] present the verification of a non-optimizing C0 compiler, which was itself written in C0 with the tool *aiT* by AbsInt [1].

5 Conclusions and Future Work

We have presented the model-based development methodology for verified software systems as well as the corresponding AUTOFOCUS 3 tool chain to explain how this tool chain and its principles can be applied for verification. Our focus was on the design, implementation and the verification phase of the overall methodology. The feasibility of the proposed approach has been demonstrated by two industrial case studies from automotive area: one case study was motivated and supported by DENSO Corporation, the second case study was supported by Robert Bosch GmbH.

The formal transformations from FOCUS to Isabelle/HOL as well as to AUTOFOCUS 3 are specified in the presented methodology as schematic, but not as automatic ones. Automatization of these steps is an ongoing work. We are currently also complementing the AUTOFOCUS 3 language with a description language for distributed execution environments based on embedded control units and bus systems. Furthermore, a deployment model combines the logical architecture with the technical architecture, thus resulting in a complete description of the executable system. We have to extend our methodology into this area by giving suitable verification methods on these new models, as done for a time-triggered operating and bus system in [14].

References

- [1] *AbsInt Angewandte Informatik. Worst-Case Execution Time Analyzers.*
- [2] *The AUTOFOCUS³ homepage.* <http://af3.fortiss.org>.
- [3] J. Botaschanjan, M. Broy, A. Gruler, A. Harhurin, S. Knapp, L. Kof, W. Paul & M. Spichkova (2008): *On the Correctness of Upper Layers of Automotive Systems.* *Formal Aspects of Computing (FACS)* 20(6), pp. 637–662, doi:10.1007/s00165-008-0097-0.
- [4] J. Botaschanjan, A. Gruler, A. Harhurin, L. Kof, M. Spichkova & D. Trachtenherz (2006): *Towards Modularized Verification of Distributed Time-Triggered Systems.* In: *FM 2006: Formal Methods*, pp. 163–178, doi:10.1007/11813040_12.
- [5] J. Botaschanjan, L. Kof, C. Kühnel & M. Spichkova (2005): *Towards Verified Automotive Software.* In: *2nd International ICSE workshop on Software*, ACM, doi:10.1145/1082983.1083199.
- [6] M. Broy & K. Stølen (2001): *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement.* Springer.

- [7] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs & Y. Xiong (2003): *Taming heterogeneity - the Ptolemy approach*. *Proc. of the IEEE* 91(1), pp. 127–144, doi:10.1109/JPROC.2002.805829.
- [8] M. Feilkas, F. Hölzl, C. Pfaller, S. Rittmann, K. Scheidemann, M. Spichkova & D. Trachtenherz (2009): *A Top-Down Methodology for the Development of Automotive Software*. Tech. Rep. TUM-I0902, TU München.
- [9] A. Fleischmann (2008): *Model-based formalization of requirements of embedded automotive systems*. Ph.D. thesis, TU München.
- [10] D. Harel & P. S. Thiagarajan (2003): *Message Sequence Charts*. In L. Lavagno, G. Martin & B. Selic, editors: *UML for Real: Design of Embedded Real-Time Systems*, Kluwer Academic Publishers, pp. 77–105.
- [11] C. Hofmann, E. Horn, W. Keller, K. Renzel, M. Schmidt, W. Horn & B. Anger: *Approaches to Software Architecture*.
- [12] F. Hölzl (2009): *The AutoFocus 3 C0 Code Generator*. Tech. Rep. TUM-I0918, TU München.
- [13] M. Jackson (2001): *Problem Frames: Analysing and Structuring Software Development Problems*. Addison-Wesley.
- [14] C. Kühnel & M. Spichkova (2007): *Fault-Tolerant Communication for Distributed Embedded Systems*. In: *Software Engineering and Fault Tolerance*, Series on Software Engineering and Knowledge Engineering.
- [15] D. Leinenbach (2008): *Compiler Verification in the Context of Pervasive System Verification*. Ph.D. thesis, Saarland University.
- [16] O. Müller & T. Nipkow (1995): *Combining Model Checking and Deduction for I/O-Automata*. In Ed Brinksma, R. Cleaveland, K. G. Larsen, T. Margaria & B. Steffen, editors: *TACAS, LNCS 1019*, Springer, pp. 1–16, doi:10.1.1.11.1585.
- [17] T. Nipkow, L. C. Paulson & M. Wenzel (2002): *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*. LNCS 2283, Springer, doi:10.1007/3-540-45949-9.
- [18] D. L. Parnas & J. Madey (1995): *Functional Documents for Computer Systems*. *Science of Computer Programming* 25, pp. 41–61, doi:10.1016/0167-6423(95)96871-J.
- [19] E. Petrova (2007): *Verification of the C0 Compiler Implementation on the Source Code Level*. Ph.D. thesis.
- [20] J. Philipps & B. Rumpe (1999): *Refinement of Pipe-and-Filter Architectures*. In J. M. Wing, J. Woodcock & J. Davies, editors: *World Congress on Formal Methods (FM'99)*, LNCS 1708, Springer, pp. 96 – 115, doi:10.1007/3-540-48119-2_8.
- [21] B. Schätz (2004): *Mastering the Complexity of Reactive Systems: the AUTOFOCUS Approach*. In: *Formal Methods for Embedded Distributed Systems: How to Master the Complexity*, Kluwer Academic Publishers, pp. 215–258.
- [22] N. Schirmer (2006): *Verification of Sequential Imperative Programs in Isabelle/HOL*. Ph.D. thesis, TU München.
- [23] M. Spichkova (2007): *Specification and Seamless Verification of Embedded Real-Time Systems: FOCUS on Isabelle*. Ph.D. thesis, TU München.
- [24] M. Spichkova (2010): *From Semiformal Requirements To Formal Specifications via MSCs*. Technical Report TUM-I1019, TU München.
- [25] M. Spichkova (2011): *Architecture: Requirements + Decomposition + Refinement*. *Softwaretechnik-Trends* 31:4.
- [26] D. Trachtenherz (2009): *Ausführungsemantik von AutoFocus-Modellen: Isabelle/HOL-Formalisierung und Äquivalenzbeweis*. Tech. Rep. TUM-I0903, TU München.
- [27] D. Wild, A. Fleischmann, J. Hartmann, C. Pfaller, M. Rappl & S. Rittmann (2006): *An Architecture-Centric Approach towards the Construction of Dependable Automotive Software*. In: *Proc. of the SAE 2006 World Congress*, doi:10.4271/2006-01-1222.