T U M

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

AutoFocus Tool Chain

F. Hoelzl, M. Spichkova, D. Trachtenherz

TUM-I1021
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-I1021-0/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

AutoFocus Tool Chain

Florian Hölzl, Maria Spichkova, David Trachtenherz

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

Abstract. This work presents the tool support for a model-based development methodology for verified software systems. We focus in this discussion on the design, implementation and the verification phase of the overall methodology developed for safety-critical embedded systems.[1] In particular, we show how design models are transformed into C code and Isabelle/HOL theories by code generators. We discuss the applied AutoFocus 3 tool chain and its basic principles emphasizing the verification of the system under development as well as the check mechanisms we applied to raise the level of confidence in the correctness of the implementation of the automatic generators.

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.

To support the contemporary embedded system development CASE tools are used in industry – they allow a simple and (mostly) intuitional design of distributed systems and applications. Executable code is generated directly from the models developed using these CASE 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 of the system. Nevertheless, verification requires significant effort. Strictly speaking, it is impossible today to verify every property of every component in a system, when considering industrial software development. Thus, only the most critical parts of a system can 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.

Nonetheless, in some cases, even after verifying certain properties, inconsistencies can still remain in the specification, model or code – most often an

[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.

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." Thus, not only verification techniques, but also testing and simulation must belong to the development process.

Therefore, a methodology is needed, which not only allows this integration into the process of modeling, but also reduces the verification and integration effort. For this purpose we propose a tool chain emphasizing the verification of the system under development. In the remainder of this article we outline a development methodology for embedded safety-critical systems. We concentrate here on the AutoFocus 3 tool chain, which is employed in the design, implementation and verification phases of the development methodology. In particular we present the different parts of the tool chain and discuss both the motivation and the benefits.

2 Development Methodology

Figure 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.

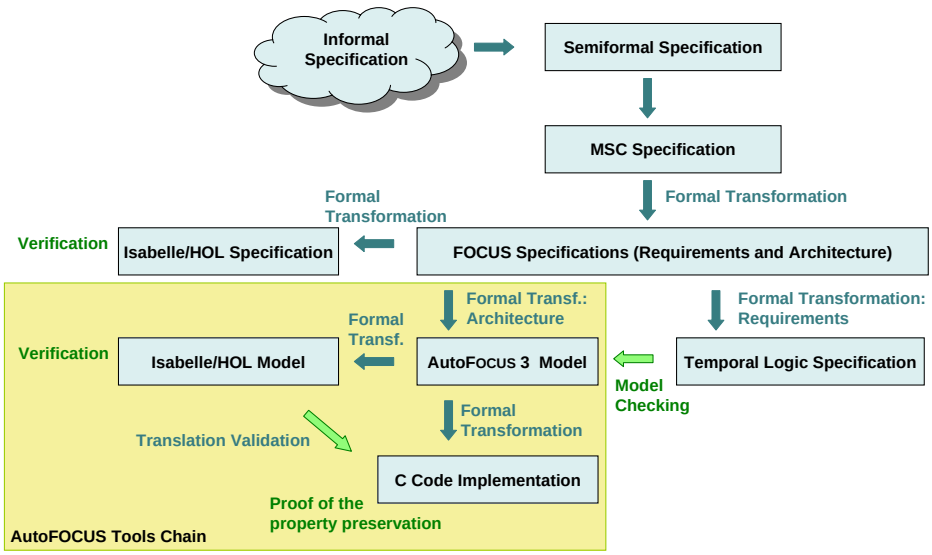

Fig. 1. Development Methodology

Beginning with a *requirements specification*, 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 [5].

The tabular semiformal specification can be also rewritten to a *Message Sequence Charts* (MSCs) representation [6,9,10] according to the approach presented in [17]. The purpose of using MSCs for specification of highly interacting systems is to obtain a better overview in comparison to a textual representation.

The MSC specification or the semiformal specification, respectively, is translated to a specification in FOCUS [3], 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 [19] – 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*. 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 [12] via the framework "FOCUS on Isabelle" [18].

In the remainder of this paper we discuss the highlighted part of the methodology: the AUTOFOCUS 3 tool chain.

As the next step of the methodology, we translate the architecture specification to a representation in the related CASE tool AUTOFOCUS 3 [8], 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 [7].

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 (which can be verified using model checking against).

3 The AutoFOCUS Tools Chain

As depicted in Fig. 1 the methodology is based on the software design tool AUTOFOCUS 3 and the theorem prover environment Isabelle/HOL. We have combined these two tools by implementing a generator for the formal transformation of the design model into the theorem prover model. Furthermore, the tool chain also provides another code generator for producing a C code implementation of the design model. In this section, we discuss the tool chain and show how the different parts are interlinked with each other. 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.

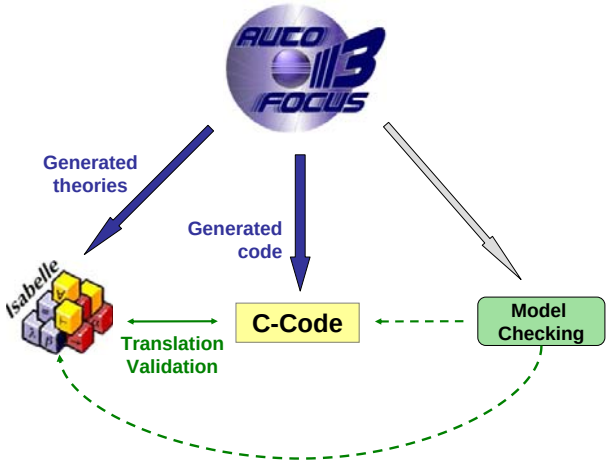

Fig. 2. Generation Tool Chain and Validation Mechanisms

Fig. 2 shows the relevant parts of the current tool chain. 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 central track 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. First, 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 (modulo an abstraction from the latter to the former), 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, e.g., class diagrams without method implementations) 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 provide the basis for the specification of the generators.

Clearly, since our code generators are also pieces of software themselves they are also prone to implementation mistakes as is any other software product. However, there are ways to gain a high confidence in the correctness of the generators.

We have used the following mechanisms in order to validate the code generator implementations using the design language semantics as the starting point:

- A paper and pencil proof has shown the behavioral equivalence between the AutoFocus 3 model and the generated C code [7].
- A second paper and pencil proof has shown the behavioral equivalence between the AutoFocus 3 model and the generated Isabelle/HOL theory [20].
- For each developed system, translation validation between the generated code and the theory shows their behavioral equivalence. This further enhances the confidence in the generator implementations by exploiting possible implementation flaws if the translation validation does not succeed. The translation validation builds on the C code verification in Isabelle/HOL as presented in [16].
- The two generators have been implemented independently by different developer teams based on the respective equivalence proofs.

Note, that the model can be non-deterministic, while the C0 program is deterministic. Still, this is correct according to the argument that the C0 program exposes the non-determinism in an unfair way: it always uses the same resolution. However, in order to do the translation validation between the Isabelle/HOL

model obtained from the AutoFocus 3 model and the C0 program, both the exporter and the generator must resolve the non-determinism in the same way to be behaviorally equivalent. In particular this means that transition segments of an automaton and the mappings defined in a function specification must be totally ordered. We use the unique object identifiers of the AutoFocus 3 model objects for this total order.

4 AutoFOCUS

AutoFocus 3 [8,15,14] is a scientific CASE tool prototype[2] implementing a modeling language based on a graphical notation (see Fig. 3) and a restricted version of the formal Focus semantics, in particular the time-synchronous frame.

We give a brief introduction of the current language.

The system structure specification 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 subcomponents introducing both local communication and communication to the environment through the interface of the parent component.

Atomic components have their behavior specified using one 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

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

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.

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 future work.

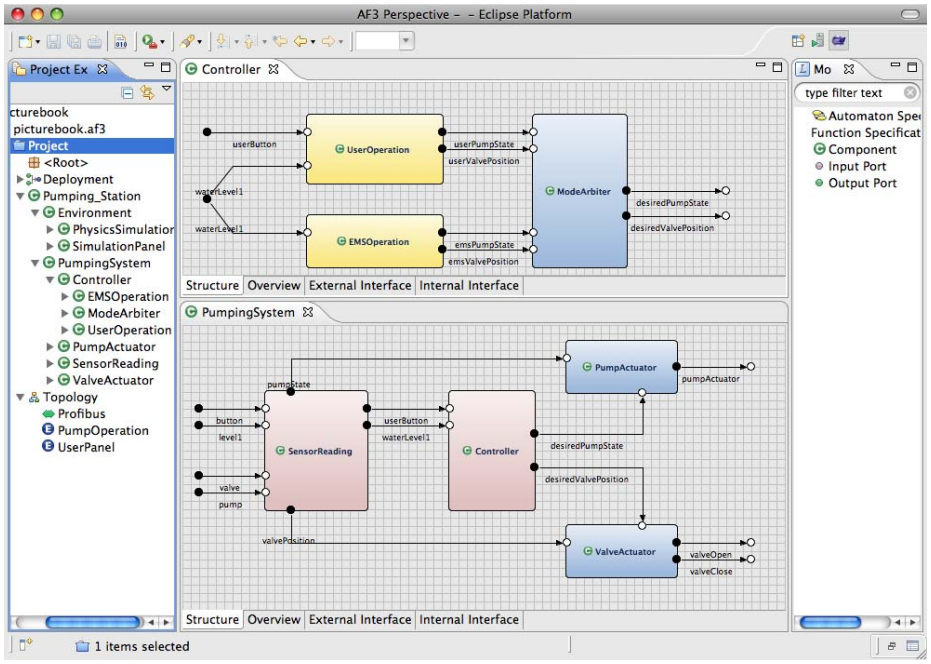

Fig. 3. AutoFocus 3 design tool

5 The Isabelle/HOL Generator

Formal verification is integral part of the presented methodology (cf. Sec. 2). Formally specified requirements can be verified using the Isabelle/HOL theorem prover (Fig. 1). For this purpose we have developed an Isabelle/HOL code generator, which automatically generates Isabelle/HOL theories from AutoFocus models. The code generator supports following AutoFocus language features:

– Data Dictionary: Data type definitions, function definitions, including recursive functions and functions on user-defined types.

– Atomic components: input/output automata with multiple control states and local variables for stateful components , function specifications for stateless components directly mapping inputs to outputs.
– Composite components: components consisting of multiple subcomponents connected by channels, strong and weak causality (delayed and instantaneous output) for subcomponents, including determination of correct execution order for weakly causal processing paths.

The behavioural equivalence between AutoFocus models and their generated representation in Isabelle/HOL has been shown in a paper-and-pencil proof [20].

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. Thus, generating code for the root component of an AutoFocus model yields a set of theory files encoding the AutoFocus model in Isabelle/HOL.

The generated code and proofs on it make use of extensive basic theory libraries [21]. The libraries comprise several aspects:

– Theories for AutoFocus message stream processing, especially generic definitions and proof results for processing finite and infinite data streams by components with arbitrary transition functions.
– Theories for definition and usage of temporal logic specifications, especially generic definitions and proof results for numerous temporal operators on arbitrary time intervals, which make it easy to define syntax and semantics for common linear-time temporal logic notations like LTL and MTL.
– Generic definitions for representation of AutoFocus components, especially transition functions for atomic and composite components, proof results for components' transition functions and additional stream processing results for AutoFocus components for finite and infinite data streams.

Based on these basic theories and semantic analysis of the AutoFocus model the Isabelle/HOL exporter generates code both for representing the model and for definition and proof of theorems that support subsequent verification of model's properties in Isabelle/HOL. This way, we were able to formally verify several requirements to the AutoFocus model from the Verisoft XT case study, which were formalised as LTL properties, by encoding them in Isabelle/HOL and proving there correctness for the Isabelle/HOL model representation automatically generated by the Isabelle/HOL code exporter.

6 The C0-Code Generator

In order to obtain executable code from an AutoFocus 3 model, we have implemented a C code generator [7], more precisely a generator for C0 code, a C language subset constructed for usage with the Isabelle/HOL verification environment as discussed in [16]. [13] and [11] present the verification of a non-optimizing C0 compiler, which was itself written in C0. C0 differs from C by restricting the language. Well-known, hazardous features, like pointer arithmetic, are forbidden in C0, while other restrictions, like the non-nested use of function calls, ease the reasoning and verification with Isabelle/HOL.

The C0 code generated from AutoFocus 3 models does not need language features still available in C0 like dynamic memory allocation, pointers or arrays.

As a result of these further restrictions, we gain the advantage of being able 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 [20]. Here, we show that the C0 program is an admissible simulation of the AutoFocus 3 model.

In summary this concludes our methodology as presented in Figure 1, but that is of course not the end of the complete development process: we need to deploy the code into the execution environment and also verify this deployment. However, this is currently out of our scope and left to future work.

7 Conclusions

We have presented the AutoFocus 3 tool chain for a model-based development methodology for verified software systems. Our focus was on the design, implementation and the verification phase of the overall methodology. We have shown the transformation of design models into C code and Isabelle/HOL theories by code generators and explained how the AutoFocus 3 tool chain and its principles can be applied to verify a system under development.

The applicability of the tool chain has been demonstrated by two case studies on embedded control systems, both being industrial case studies from automotive area. The case study referred to in [4,17] was motivated and supported by DENSO CORPORATION and yields approx. 3 KLOC of generated code. The ongoing case study [22] is supported by Robert Bosch GmbH and yields approx. 17 KLOC of generated code and 38 KLOC of generated Isabelle/HOL theories, respectively.

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. M. Broy and K. Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer, 2001.
4. 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.
5. A. Fleischmann. *Model-based formalization of requirements of embedded automotive systens*. PhD thesis, TU München, 2008.
6. D. Harel and P. S. Thiagarajan. Message Sequence Charts. In L. Lavagno, G. Martin, and B. Selic, editors, *UML for Real: Design of Embedded Real-Time Systems*, pages 77–105. Kluwer Academic Publishers, 2003.
7. F. Hölzl. The AutoFocus 3 C0 Code Generator. Technical Report TUM-I0918, Technische Universität München, 2009.
8. 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.
9. ITU-TS. Recommendation Z.120: Message Sequence Chart (MSC). Geneva, 1996.
10. I. Krüger. *Distributed System Design with Message Sequence Charts*. PhD thesis, TU München, 2000.
11. D. C. Leinenbach. *Compiler Verification in the Context of Pervasive System Verification*. PhD thesis, Saarland University, July 2008.
12. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
13. E. Petrova. *Verification of the C0 Compiler Implementation on the Source Code Level*. PhD thesis, Saarland University, Computer Science Department, May 2007.
14. 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.
15. B. Schätz and F. Huber. Integrating Formal Description Techniques. In J. M. Wing, J. Woodcock, and J. Davies, editors, *FM'99*, volume 1709 of *LNCS*, pages 1206–1225. Springer, 1999.
16. N. Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, TU München, 2006.
17. M. Spichkova. From Semiformal Requirements To Formal Specifications via MSCs. Technical report.
18. M. Spichkova. *Specification and Seamless Verification of Embedded Real-Time Systems: FOCUS on Isabelle*. PhD thesis, 2007.
19. M. Spivey. Understanding Z – A Specification Language and Its Formal Semantics. Cambridge Tracts in Theoretical Computer Science 3. Camb. Univ. Press, 1988.
20. D. Trachtenherz. Ausführungssemantik von AutoFocus-Modellen: Isabelle/HOL-Formalisierung und Äquivalenzbeweis. Tech. Rep. TUM-I0903, TU München, 2009.
21. D. Trachtenherz. *Eigenschaftsorientierte Beschreibung der logischen Architektur eingebetteter Systeme (Property-Oriented Description of Logical Architecture of Embedded Systems)*. PhD thesis, TU München, 2009.
22. Verisoft XT Project. `http://www.verisoftxt.de`.