Formal Refinement and Model Checking of an Echo Cancellation Unit

Alexander Krupp, Wolfgang Mueller
Paderborn University/C-LAB
Paderborn, Germany

Ian Oliver
Nokia Research Center
Helsinki, Finland

Abstract

This article presents an approach, which combines theorem proving-based refinement with model checking for state based real-time systems. Our verification flow starts from UML state diagrams, which are translated to the formal B language and are model checked for real-time properties. By means of the B language and a B theorem prover, refined state diagrams are verified against their abstract representation. The approach is presented by means of the refinement of a digital echo cancellation unit.

1. Introduction

This paper describes the application of a methodology, which efficiently integrates state diagram based model checking and formal refinement based on the B language[1]. For tools we apply the RAVEN model checker[12] and Clearsy's Atelier-B toolkit. The presented case study is performed with a digital echo cancellation unit for a mobile phone, which filters crosstalk from the phone's speaker to the microphone audio data.

Our verification flow is based on automatic translation of UML state diagrams to the B language for formal refinement automation. A representation similar to finite state machines is used for model checking. When generating specifications from state diagrams, we arrive at synchronously communicating non-deterministic finite state machines, which are executed at cycle-accurate basis and directly correspond to models, which can be verified by a BDD-based model checker. For refinement automation, we focus on the refinement of a non-deterministic cycle-accurate model to a time-annotated deterministic model. We present an approach, which very efficiently applies the Atelier-B theorem prover and the underlying refinement concepts of the B language. In contrast to related approaches, our results demonstrate that the proof of the generated code requires almost no user interaction with the prover as well as low run times of the prover for automatic deduction.

2. Related Work

The Boyer–Moore Theorem Prover (BMTP) and HOL are the two classical approaches to theorem proving in the domain of electronic design automation. BMTP and HOL are both interactive proof assistants for a given set of higher order logic axioms and inference rules[5, 11]. Model checking in the domain of electronic design automation is due to the pioneering work of Clarke et al. in [6] and their BDD-based SMV model checker. SMV verifies a given set of synchronously communicating state machines with respect to properties given by a set of formulae in tree temporal logic, namely CTL (Computation Tree Logic).

There are several works integrating model checkers into theorem provers and vice versa. PVS (Prototype Verification System) is a theorem prover where the PVS *specification language* is based on higher-order predicate logic. Shankar et al. enhance PVS with tools for abstraction, invariant generation, program analysis (such as slicing), theorem proving, and model checking to separate concerns as well as to compute concurrent systems properties[13]. STeP (Stanford Temporal Prover) was implemented in Standard ML and C [9]. STeP integrates a model checker into an automatic deductive theorem prover. The input for model checking is given as a set of temporal formulae and a transition system, which is generated from a description in a reactive system specification language (SPL) or a description of a VHDL subset. Berezin has introduced the SyMP framework, which integrates a model checker into a HOL-based theorem prover for general investigations on effectiveness and efficiency[3]. His work is on the applicability for domain-specific computer-assisted manual proofs where main examples come from hardware design. Mocha [2] is a model checker enhanced by a theorem prover and a simulator to provide an interactive environment for concurrent system specification and verification.

In the context of the B theorem prover, Mikhailov and Butler combine theorem proving and constraint solving[10]. They focus on the B theorem prover and the Alloy Constraint Analyser for general property verification. Fokkink et al. employ the B method and combine it with *μCRL* [4]. They describe the use of B refinement in combination with

1530-1591/04 $20.00 (c) 2004 IEEE

model checking to arrive at a formally verified prototype implementation of a data acquisition system of the Lynx military helicopters. They present the refinement of a system implementation from a first abstract property specification.

All these approaches consider timeless models and do not cover refinement of finite state machines with real-time properties. Only Zandin investigates real-time property specification with B by the example of a cruise controller[15]. However, he reports significant problems with respect to the complexity of the proof during refinement.

In contrast to the HOL based and interactive approaches, we present a model checking based approach with the RAVEN model checker in conjunction with the Atelier-B theorem prover for formal refinement automation with the goal to avoid manual user interaction. We focus on the verification of real-time systems and on the refinement from cycle-accurate to time-accurate models based on an efficient mapping from state diagrams to B. We present our approach by the real world example of an echo cancellation unit of a mobile phone.

3. Real-Time Model Checking with RAVEN

We apply the RAVEN (Real-Time Analyzing and Verification Environment) real-time model checker, which extends basic model checking for real-time systems verification by additional analysis algorithms[12]. In RAVEN, a model is defined by a set of synchronously communicating finite state machines (I/O-Interval Structures) and the specification by Clocked CTL (CCTL) formulae. The latter extends classical CTL by time-bounds.

As an example consider the CCTL property specification, which defines that a consumer input buffer must not be blocked in order to guarantee sufficient continuous workload, i.e., each accepted delivery request must be followed by loading an item at the input within 100 time units after acceptance:

```
AG((consumer.state = consumer.accept)
  -> AF[100]((loader.state = loader.wait)
             & AX(loader.state = loader.load)
            )
  )
```

4. Refinement with B

B stands for a methodology, a language, and a tool-set for the specification, design, and coding of software systems introduced by Abrial[1]. B is based on viewing a program as a mathematical object and the concepts of pre- and postconditions, of non-determinism, and weakest precondition comparable to VDM and Z.

In B, a user writes a first initial system specification B_n and refines that specification by n refinement steps to B_0. For that, n additional B specifications are created: B_{n-1}, ..., B_0. The final B_0 specification has to comply to a well-defined executable B subset always denoted as B_0 in B. A B-toolkit such as Atelier-B can finally generate programming language code such as C++ from B_0. Refinement in B means to replace a B specification M by a B specification M' where M' has to define operations with identical signatures. Nevertheless, M' may work on a different internal state or a different specification of the operation. The important requirement is that specification M' must be a replacement of M: under equal conditions operation $S_{M'}$ does never produce a result or state which is not entailed by the refined (abstract) operation S_M. Refinement typically reduces non-determinism and abstract functions until a deterministic B_0 implementation is reached.

5. Combined Verification Flow

Classical HW/SW Co-design typically starts with a functional model (e.g., C program), which is iteratively transformed into a state-oriented, cycle-accurate model from which HW and SW components can be derived [14]. In a later step, the state-oriented model is typically (back)annotated by timing information for the verification of its real-time behaviour. In this context, the critical steps are the generation and enhancement of the cycle-accurate model and its annotation by timing information. Our approach focuses on both steps and develops an efficient code generation proving that the final model is a correct refinement of the original one by means of Atelier-B, a B-based theorem prover and code generator. In our verification environment, we apply state diagrams as graphical means since they are widely used for documentation, specification, model checking, as well as well investigated for VHDL, Verilog, and C code generation of state-oriented system specifications. Through integration of model checking and formal refinement, we can investigate the cycle-accurate models by simulation and check their properties through model checking by the RAVEN model checker. Thus, after model checking, we generate a corresponding initial B language model. Refinements of state-oriented models cover the refinement of states into hierarchical states, the modification of state transitions like elimination of non-determinism and removal of self-loops. Furthermore, we cover the annotation by timing information and denote the corresponding annotated model as SD^T and B^T. The B prover verifies if B^T is a refinement of the B model, which then also verifies that SD^T is a correct refinement of the previous state diagrams model SD. Through the B environment, further refinement towards B_0 implementation level is possible from which we can automatically generate C code for a proven implemen-

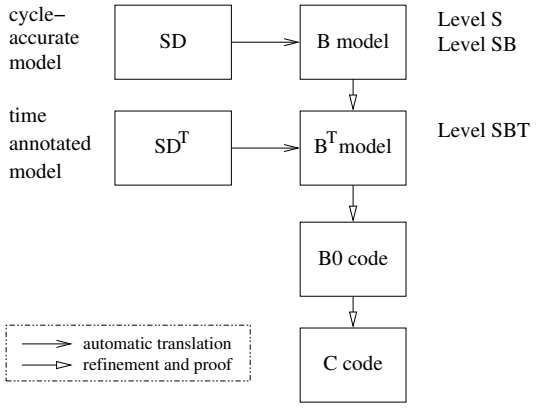

Figure 1. Verification flow

tation. Figure 1 gives an overview of the design flow from SD to SD^T and, correspondingly, from B to B^T. Implementation in B_0 and C code generation may follow. We denote the levels of abstraction used in B as level S (Structure), SB(Structure & Behaviour), and SBT (Structure & Behaviour & Timing). The individual levels are outlined in more details in the remainder of this article. The outlines are based on the example of an echo cancellation unit, which is introduced at the beginning of the next section.

6. Refinement of the Echo Cancellation Unit

The echo cancellation unit of a mobile phone filters the crosstalk from the phone's speaker from the other sound. To prevent a feedback loop, that crosstalk has to be suppressed in the microphone audio packets. This is achieved through application of a digital filter, which correlates received audio data with output audio data. We model the flow of digital audio packets between functional entities of this unit. This basically resembles a producer-consumer synchronisation problem. In the next subsection, we give an overview of the model and present some properties to be proven. Thereafter, we outline the refinement of a B specification from cycle-accurate (S, SB level) to time-accurate level (SBT level). Numbers of generated proof obligations and other experimental results and achievements on an automatic proof are presented in the next section.

6.1. Overview

For echo cancellation of recorded audio, the audio I/O interface decodes data from the network and generates output to a speaker. Simultaneously, data from the microphone are recorded, filtered, and generated for output to the network.

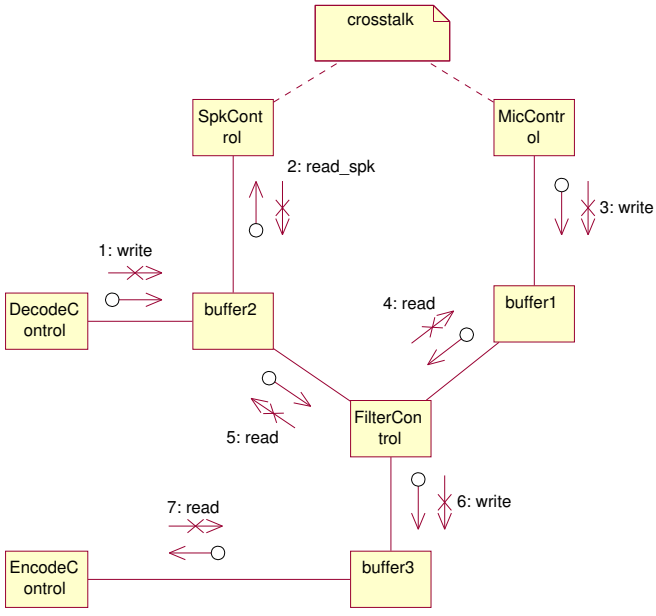

Figure 2. Collaboration diagram for echo cancellation

The producer-consumer synchronisation basically consists of two producers, two consumers, three buffers, and a filter, which can be considered both a producer and a consumer. Data and message flow is shown in the UML collaboration diagram in Figure 2. The model describes an echo cancellation mechanism, which is working on packets of audio data. A decoding process *DecodeControl* decodes packets received from the network and inserts an audio packet into a buffer *buffer2*. From that buffer, audio packets are sent to a speaker through process *SpkControl*. At the same time a process *MicControl* encodes audio from a microphone to packets, which are inserted into another buffer, *buffer1*. The filter process is called *FilterControl* here. It takes a packet from the speaker buffer *buffer2* after it has been sent to the speaker and uses it to filter a packet taken from the microphone buffer *buffer1*. The filtered packet is inserted into *buffer3*. From there it is consumed by the network encoding process *EncodeControl* and sent to the network. In that process, buffer overflows as well as buffer underflows are harmful. The side conditions for buffers are:

1. An item is sent to the buffer only if the buffer is not full, otherwise, the buffer will overflow;
2. the consumer has to react only if an item is available in the buffer, otherwise, the buffer will underflow;
3. an empty buffer will underflow and a full buffer will overflow, if a read and a write are performed on it at the same time.

In the state diagrams of the buffers, only the general states *running*, *overflow*, and *underflow* are introduced. The states *overflow* and *underflow* are failure states and should never be entered. When entering them, the system has to be reset. An integer variable keeps the count of items in the buffer.
Goals for model and specification are:

1. to dimension the buffers to avoid failure states;
2. to match process delays to enable synchronous processing of audio packets;
3. to maintain the structure and semantics through development (refinement);
4. to verify safety and liveness criteria.

To demonstrate the main concepts of our verification flow, the following sections focus on the refinement of the *FilterControl* component.

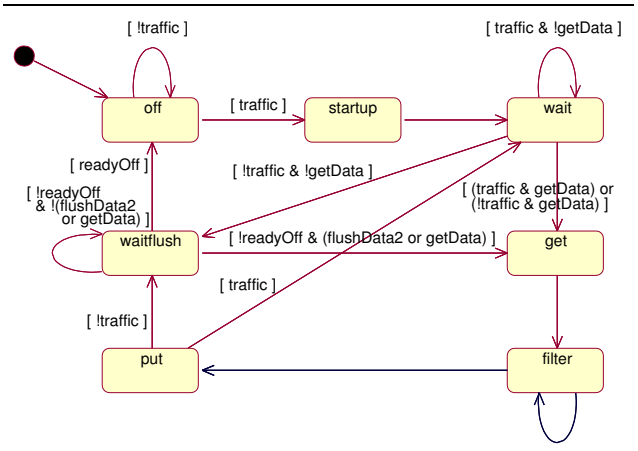

Figure 3. State Diagram *FilterControl_SB*

6.2. S Level - FilterControl

The *FilterControl_SB* state diagram (Figure 3) models a producer and a consumer. Its behaviour is explained in the next subsection, where we are dealing with cycle-accurate state transitions. At structural level (S level), a B specification only captures static properties, i.e., value propagation. Only dependencies of B output values on inputs and states are specified at this level.

The B specification for *FilterControl_S*, for instance, declares 18 variables, an invariant, and two operations. One variable represents the state of *FilterControl*, the others represent inputs and outputs. The state is implemented as a B enumeration type, which ranges over the states given in the state diagram. The invariant defines the input/output relationship between variables. It specifies that if their values are consistent, a selected transition can be executed. B variables are marked inconsistent after initialisation or transition execution. Consistency for those variables is established through the operation *setInputs*, which receives as parameters the current input values. The B specification additionally includes the operation *doTransition* to perform a state transition. It is executed if input and output values are consistent. At S level, *doTransition* specifies a non-deterministic choice of all possible states of the state variable. We denote this a *generic transition*, as the target state is non-deterministically selected from all states.

6.3. SB Level - FilterControl

At SB level, we refine the state transitions as shown in Figure 3. The SB level B specification basically refines the S level operation *doTransition* by introducing behaviour in the form of refined state transitions. Generally, refinement is performed by replacing only some non-deterministic elements by deterministic ones. Consequently, transitions become either restricted to a specific non-determinism or they are deterministic.

The *FilterControl_SB* as given in the diagram performs as follows. A transition from *off* to *startup* is executed when input traffic is announced. Thereafter, state *wait* is entered. A self loop in the diagram indicates, that state *wait* is kept while no data is available in traffic mode. When traffic mode is turned off with no available data, state *waitflush* is entered. In case of data availability, state *get* is always entered. From state *get* a transition leads to state *filter* from which a non-deterministic transition to state *put* is defined, which either stays in *filter* or changes to *put*. It is introduced here because we want to leave the exact timing condition open. In state *put*, if input traffic is enabled, a transition to state *wait* is performed, which basically completes the data processing loop. A final transition to state *waitflush* is required to flush the buffer. In *waitflush* it is determined, whether to change to state *off* or state *get* due to other system states. I.e., as soon as the previous two buffers are empty and microphone and decoding module are turned off, *FilterControl* is turned off as well. Otherwise, if data becomes available again, it will be processed by changing to state *get*.

At a first glance, the separation of the B specification into S and SB level is not very meaningful. In fact, they are only introduced in the context of the Atelier-B toolkit. Experiments have shown that the separation significantly reduces the number of generated proof obligations and thus greatly reduces the number of interactive proofs. The separated specification resembles an interface description at S level and an internal behavioural description at SB level and thus also supports a structured top down methodology for theorem proving.

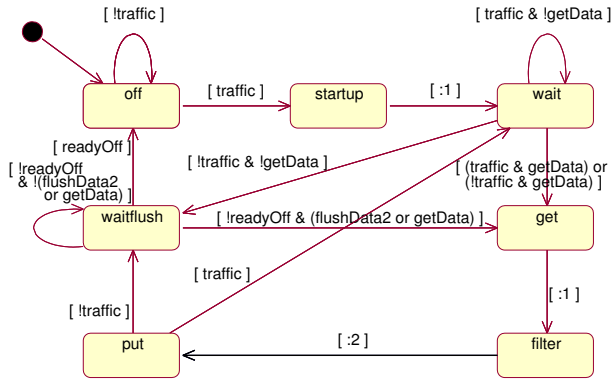

Figure 4. State Diagram *FilterControl_SBT*

6.4. SBT Level - FilterControl

At SBT level, we again refine the state transitions encapsulated in the B operation *doTransition*. Here, all nondeterministic transitions are modified to either to deterministic or timed deterministic transitions. Consider the refined example in Fig. 4. As a simplified refinement, we remove the self-loop of state *filter* and assign a time delay value of 2 to the transition to state *put*.

For B specification, it has to be noted here, that we have defined a timed variant of B, which we denote as BT+.[1] BT+ extends B by the notion of time delays in transition specifications by means of a *DELAY* statement. When selecting a timed transition, an associated timer is initiated. The transition is fired as soon as the timer elapses.

6.5. RAVEN Model Checking

For RAVEN model checking of the echo cancellation unit at SBT level, let us consider the following CCTL specifications.

```
SPEC
  s1    := AG !(buffer1.s=buffer1.overflow )
  l     := AG EF (Controller.s=Controller.traffic)
  v8eq2 := AG( !EG[13](
                 (Controller.s=Controller.traffic)
                   & (buffer1.empty & buffer2.empty)))
ANALYSE
  a1a := MAX VALUE OF buffer1.count
               IN TRUE
  a1b := MIN VALUE OF buffer1.count
               FROM buffer1.count>0 WITHIN [0,INFINITY]
  b1  := MAX TIME
               FROM (Controller.s=Controller.disconnect)
               TO (Controller.s=Controller.idle)
```

The first specification *s1* is a safety condition. It means that on all possible paths of execution, the buffer should never enter state *overflow*. The second specification *l* is a liveness condition. It means that on all possible paths of execution at least one path leads to state *traffic*. The third specification *v8eq2* states that when starting from state *traffic* it is not allowed for *buffer1* and *buffer2* to be empty for 13 time units. Additionally, the example has three definitions for quantitative analysis. The first formula checks for the maximum count of buffer items over all possible execution paths. The second one computes the minimum value of the buffer count after it has increased to a value greater than 0. The third formula checks for the maximum steps in time units from state *disconnect* to state *idle*. Note, that such specifications can also be used to determine minimum or maximum reaction times.

The verification run with RAVEN shows, that a buffer overflow does not occur in our model. Moreover, it computes the exact maximum and minimum numbers of buffer elements. Overall execution time for the verification run was 0.5 seconds under Linux on an Intel P4 with 2.2GHz and 1GB RAM.

7. Experimental Results

The formal refinement for the introduced verification flow shows quite promising results for refinement automation. Table 1 gives an overview of the verification results with an *Atelier B 3.6* pre-release. The table shows numbers of the previously introduced example, where *EXECUTOR*, *SYNCHRONIZER*, and *TickTimer* implement the time-oriented synchronous model of computation for the state diagrams. For model consistency, we have already introduced *TickTimer* at S level. More details of the complete proof with *Atelier B* and RAVEN and the example is given in [7, 8] in details.

The table has four columns: B component identifier, the number of obvious proof obligations, the number of other proof obligations, and the percentage of proven proof obligations. Proof obligations (POs) are theorems, which are automatically generated by Atelier-B. *Obvious proof obligations* are POs, which are found to be *true* at generation time. Almost the entire proof was accomplished by *Atelier B* automatically. A single proof obligation of the *TickTimer* component[2] could not be proven by the automatic prover. The proof of that PO was performed through one simple command of the interactive prover of Atelier-B. Thereafter, the refinement of the echo cancellation unit example was completely proven by *Atelier B*.

As for model checking, the refinement was performed under Linux on an Intel 2.2Ghz P4. The time for B type checking and proof obligation generation was 271 sec. The proof of the 76 proof obligations in *Force 1* mode took 20 sec. The execution time of the interactive prover was less than a second.

1 In practice, BT+ is just needed to specify time annotated state transitions. However, a simple pre-processor would be able to convert BT+ into standard B [7]

2 *TickTimer* is introduced as an extra component to trigger the timeouts

Modules/ Levels	Obvious Proof Obligations	Non Obvious Proof Obligations	Proof Percent
Level S:			
EXECUTOR_S	5	0	100
SYNCHRONIZER_S	3	0	100
TickTimer_S	31	7	100
GLOBAL_CONSTANTS	1	0	100
Controller_S	35	0	100
DecodeControl_S	29	0	100
EncodeControl_S	38	0	100
FilterControl_S	77	0	100
MicControl_S	29	0	100
SpkControl_S	32	0	100
buffer1_S	34	1	100
buffer2_S	40	1	100
buffer3_S	34	1	100
Level SB:			
Controller_SB	156	0	100
DecodeControl_SB	64	0	100
EncodeControl_SB	178	0	100
FilterControl_SB	481	0	100
MicControl_SB	64	0	100
Level SBT:			
EXECUTOR_SBT	220	31	100
SYNCHRONIZER_SBT	7	0	100
TickTimer_SBT	53	13	100
Controller_SBT	223	2	100
DecodeControl_SBT	137	0	100
EncodeControl_SBT	320	1	100
FilterControl_SBT	746	5	100
MicControl_SBT	137	0	100
TOTAL	3669	77	100

Table 1. Proof obligations, Atelier B 3.6b

8. Conclusions

We have investigated formal refinement of finite state machines from a cycle-accurate to a time-accurate model in B with Atelier-B. For complementary verification, we applied the RAVEN model checker. In that scenario, we have proposed a verification flow and performed a B refinement for levels, which we denote as S, SB, and SBT level. The refinement starts at structural level and in succession adds static, behavioural, and timing properties to the B model. The applicability of our approach was demonstrated by the industrial case study of the echo cancellation unit of a mobile phone. All of our current internal investigations have shown, that our translation from RAVEN to the proposed B subset enables efficient proof with Atelier B without any significant manual interference. This is a promising result in the direction of refinement automation and for the wider acceptance of formal refinement with theorem proving. For the case study, the plain proof execution times was 0.5 sec for model checking with RAVEN and 291 sec for theorem proving with Atelier B. The latter number includes the proof of 76 proof obligations in 20 sec.

Though our current results are promising for the refinement of finite state machines, we still need to perform detailed studies for the complementary refinement of other properties. That means, not just to perform refinements on state diagrams but also to consider complementary property specifications like as they can be given by a limited OCL (Object Constraint Language) subset.

Acknowledgement

This work has been supported by the IST Project PUSSEE (IST-2000-30103).

References

[1] J. Abrial. *The B-Book*. Cambridge University Press, 1996.

[2] R. Alur and T. Henzinger. Reactive modules. In *LICS'96*, 1996.

[3] S. Berezin. *Model Checking and Theorem Proving: A Unified Framework*. PhD thesis, Carnegie Mellon University, 2002.

[4] S. Blom, W. Fokkink, J. F. Groote, I. van Langevelde, B. Lisser, and J. van de Pol. *μCRL*: A toolset for analysing algebraic specifications. In *Proceedings of CAV'01*, 2001.

[5] R. Boyer and J. Moore. *A Computational Logic Handbook*. Number 23 in Perspectives in Computing. Academic Press, Inc., 1988.

[6] E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. *Lecture Notes in Computer Science*, 131:244–263, 1981.

[7] A. Krupp and W. Mueller. Deliverable D4.3.1: Specification of real-time properties. Technical report, IST - Project PUSSEE, Dec. 2002.

[8] A. Krupp and W. Mueller. Deliverable D4.3.2: Refinement and verification of real-time properties. Technical report, IST - Project PUSSEE, Apr. 2003.

[9] Manna, Z. et al. STeP: the Stanford Temporal Prover. Technical Report STAN-CS-TR-94-1518, Stanford University, June 1994.

[10] L. Mikhailov and M. Butler. An approach to combining B and Alloy. In D. B. et al., editor, *ZB'2002*, Grenoble, France, Jan. 2002. Springer-Verlag.

[11] M.J.Gordon. *Introduction to HOL*. Cambridge University Press, Cambridge, 1993.

[12] J. Ruf. RAVEN: Real-time analyzing and verification environment. *Journal on Universal Computer Science (J.UCS), Springer, Heidelberg*, Feb. 2001.

[13] N. Shankar. Combining theorem proving and model checking through symbolic analysis. In *CONCUR 2000*, 2000.

[14] N. Wehn and M. Smola. Einsatz von Hardwarebeschreibungssprachen beim Entwurf komplexer Multimedia-Bausteine. In *2. GI Workshop - Hardwarebeschreibungssprachen und Modellierungsparadigmen*, 1996.

[15] J. Zandin. Non-operational, temporal specification using the B method - a cruise controller case study. Master's thesis, Department of Computing Science, Chalmers University of Technology and Gothenburg University, 1999.