Formal Specification in VHDL for Hardware Verification

Ralf Reetz

Verysys GmbH
Rudower Chaussee 5
D-12489 Berlin
Germany
ralf@verysys.com

Klaus Schneider, and Thomas Kropf

Institut für Rechnerentwurf und Fehlertoleranz
Universität Karlsruhe
D-76128 Karlsruhe
Germany
Klaus.Schneider@informatik.uni-karlsruhe.de
Thomas.Kropf@informatik.uni-karlsruhe.de

Abstract

In this paper, we enrich VHDL with new specification constructs intended for hardware verification. Using our extensions, total correctness properties may now be stated whereas only partial correctness can be expressed using the standard VHDL assert *statement. All relevant properties can now be specified in such a way that the designer does not need to use formalisms like temporal logics. As the specifications are independent from a certain formalism, there is no restriction to a certain hardware verification approach.*

1 Introduction

As VHDL [1] is an important IEEE standard for describing digital circuits, many commercial design tools are based on this hardware description language. Originally created for simulation, this language has recently been used also for formal verification [2] to ensure the correctness of designs.

However, VHDL itself is only intended for describing the *implementation* of a system for synthesis or simulation. For capturing the system specification, only the **assert** statement is given to simplify the analysis of lengthy simulation results. Due to the original purpose of this construct, only simple safety properties can be stated which are not sufficient for formal hardware verification. For verification, at least additional constructs for specifying liveness and fairness properties are required to state that some event will actually happen once or infinitely often. Moreover, the environment of the current design reflecting all reasonable inputs to the design has to be modeled appropriately.

In this paper, we propose to enrich VHDL by new constructs in such a way that all the necessary specifications can be written directly in a slightly extended VHDL. The extension is based on a verification scenario, called *verification bench*, in strong analogy to the usual simulation scenario, known as test benchs. Furthermore, we significantly extend the existing specification capabilities of VHDL: The existing **assert** statement only allows to capture *partial* correctness whereas our extension allows to specify also *total* correctness of VHDL programs, i.e. now the verification of program termination is possible.

We have incorporated the new constructs as part of the tool FLOWER, which is an experimental environment for the formal semantics and verification of VHDL [3]. As we are aware of the fact that existing design tools do not support our extensions, means for translating the extended VHDL sources into standard VHDL are given.

The paper is structured as follows. First, we give a brief overview about other approaches to VHDL verification. Then, we present the verification scenario, based on the verification bench and the new specification constructs with their syntax and semantics. We then give some examples and conclude the paper with some remarks on further directions of research.

2 State of the Art

The basis of all formal approaches to VHDL is a formal semantics of VHDL. Unfortunately, the IEEE standard for VHDL does not provide this such that various approaches to giving a formal semantics to VHDL have been investigated [4, 5, 3]. For brevity and conciseness of the paper, we assume that a formal semantics for VHDL based on transition systems as presented e.g. in [3] is given and focus only on the semantics of our new constructs.

In general, two approaches to hardware verification can be distinguished: verifying the equivalence of two implementations and property verification. For the former, standard VHDL is sufficient as only two implementation descriptions are necessary. Different tools from companies like AHL, CHRYSALIS, VERYSYS etc. are already avail-

able to check the equivalence of two designs. Usually, only synchronous VHDL is supported [6, 7].

On the other hand, property verification of VHDL designs requires to write down specifications to be checked for an implementation given in VHDL. Since a correctness result is always relative to the specification, it is important to be sure that the specification is the one that is really wanted. Thus being able to easily set up clear, precise, and concise formal specifications is essential. Most verification tools which support VHDL implementations require that the specification is given in the formalism, the verification tool is based on. For example, underlying formalisms are temporal logics as in CV/CVC[1], timing diagrams translated into temporal logics [8], ω-automata for language inclusion [9], and first-order logics [10].

All these approaches bear problems if a designer wants to write a formal specification:

- The designer is forced to learn new formalisms like temporal logics which are different from VHDL and are often hard to learn. Even for experts, it is in some cases difficult to set up a precise specification [11].
- Specifications in other formalisms have to refer to the formal semantics of the VHDL program. Thus the designer has to know e.g. the notion of time and execution states to which the temporal operators of the logic relate to.
- The specification paradigm normally used in formal methods does not have any relation to the well-known test bench concept, used by designers for simulation-based validation.

To eliminate these problems, we suggest to write down specifications in VHDL itself – avoiding the use of any other formalism for specification. Early approaches for describing specifications in VHDL [12, 13] have not been developed further as at that time no formal semantics and no verification tools for VHDL were available. Our new approach consists of the concept of a *verification bench* that is described by new language constructs in VHDL. Doing this, the problems described above may be solved as follows:

- Implementations and specifications intended for formal verification may be both written directly in VHDL. No formal languages have to be learned although the full expressiveness of other formalisms as e.g. ω-automata [14] is reached.
- The verification bench concept is an extension of the usual test bench approach. Thus the designer is accustomed to its intention and basic principles.
- For simulation purposes, the new language constructs used for specification can be removed to obtain standard VHDL. Thus if the verification fails, the generated counterexample can be directly simulated using standard tools.

The addition of new language constructs is necessary if formal verification has to be based on VHDL descriptions of the implementation and the specification without altering the standard semantics of VHDL. The reason for this is that formal verification of a module has to consider the behavior of this module under *all reasonable* inputs of the module, while the simulation semantics of VHDL is based on a *single* input sequence. Consequently, VHDL modules do not have free inputs [1] in the sense that these free inputs can arbitrarily change their values each simulation cycle. Instead, free inputs in VHDL must be initially fixed to some value and are not able to change in the following, hence they behave as constant values.

Moreover, the **assert** statement as the only existing specification construct in VHDL is insufficient as only restricted safety properties can be stated. In order to reason about *total* correctness that also covers the termination of selected statements, we need additional concepts.

The practicability of the approach has been demonstrated by the tool *FLOWER*, which has been implemented to translate verification bench descriptions into standard model checking problems [3]. In *FLOWER*, a full simulation cycle-based semantics is implemented, which especially supports delta-delays and arbitrary non-zero delays at the same time. The implementation has been used to perform actual verification runs, including the generation of counterexamples. However, the methodology proposed in this paper is independent of the chosen verification tool and also of the chosen VHDL semantics and may be used for other verification systems – or even other hardware description languages – as well.

3 Verification bench

3.1 Test bench vs. Verification bench

Today, the usual method to ensure correctness of a design described in VHDL is the simulation of a test bench. A test bench consists of three components: the *stimuli generator*, the *implementation* and an *observer*, as shown in figure 1. The stimuli generator deterministically produces inputs for the implementation, while the observer analyzes both the inputs and outputs, and produces a report if both do not fit together. As the stimuli generator describes the environment of the implementation, there are no inputs to the test bench, hence test benchs are closed systems.

In this paper, we suggest to adapt the well-known concept of a test bench for the needs of formal verification. Thus we construct a 'verification bench' (figure 2).

[1] http://www.cs.cmu.edu/ modelcheck/cv/project.html

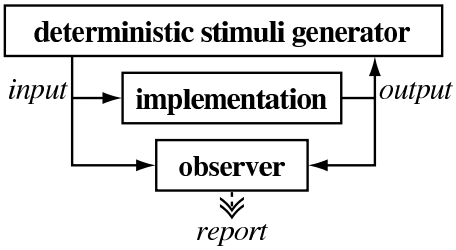

Figure 1: Test bench

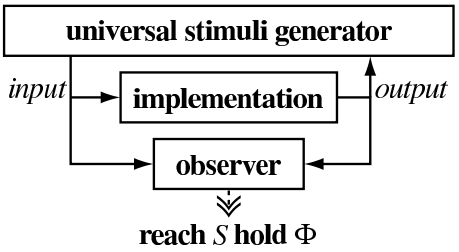

Figure 2: Verification bench

A verification bench consists of the same parts as a test bench, however the deterministic stimuli generator of the test bench is replaced by a 'universal' stimuli generator. This is due to the fact that formal verification must consider *all possible input sequences* of the implementation, not only a concrete sequence, which is sufficient for simulation. A verification bench is still a closed system such that there is no need to have free inputs. The introduction of free inputs would require to change the existing semantics of VHDL[2].

In order to describe the universal stimuli generator for the verification bench, we introduce a new VHDL function T**'arbitrary**. The argument is any type T, and the result is a value of the same type T. Whenever this function is called, the result is some arbitrary, nondeterministically chosen value; and calling this functions never produces an error. In our formal semantics, each call for T**'arbitrary** leads to a universal quantification in the formula associated with the statement, i.e. the semantics reflects that the considered property holds for all values of type T.

During the translation into a finite state transition system, the formal semantics for T**'arbitrary** is simply captured by defining a new input variable with domain T. As input values can not be predicted, these variables may change arbitrarily.

3.2 Formal specifications in VHDL

All specifications of a VHDL program are captured in the observer part of a verification bench. As the observer which is also a VHDL program may also contain some local state variables to store previous events, and runs as a process parallel to the implementation, we can view the observer as an accepting finite-state ω-automaton [14]. Without additional VHDL constructs, the acceptance condition of this ω-automaton can only be given with the **assert** statement.

However, even if we consider both the observer and the implementation under all possible input sequences in a verification bench, **assert** is still not sufficient for writing down powerful specifications in form of observers. To illustrate the shortcomings of **assert**, we have to explain some notations of the VHDL transition system semantics used in the following.

The IEEE standard semantics of VHDL is operational and can thus be understood as a state transition system. Thus, we have defined predicates[3] $\mathsf{enter}(S)$ and $\mathsf{leave}(S)$ that hold for a VHDL statement list S whenever its execution starts or terminates, respectively. $\mathsf{enter}(S)$ and $\mathsf{leave}(S)$ are sets of states in a transition system reflecting the semantics of a given VHDL program.

The statement **assert** Φ only allows to prove that if a certain point of the program is reached, a condition Φ must hold. Using temporal logic[4], we define the semantics of **assert** Φ as $\mathsf{AG}[\mathsf{enter}(S) \rightarrow \Phi]$, or equivalently, $\mathsf{AG}[\mathsf{leave}(S) \rightarrow \Phi]$.

Hence, **assert** Φ can only be used to assure that if a certain point of the program is reached, a property Φ must hold. Nothing can be done to ensure or to detect if the program state is reached or not, hence, **assert** can not be used to reason about the *termination of statements*. As the observer may have in its implementation additional state transitions, the resulting specification language would have the expressiveness of ω-automata with only safety properties as acceptance conditions.

In order to allow more powerful specifications, other constructs have to be added. We propose to introduce the statement '**reach** S **hold** Φ' to VHDL, where S is a VHDL sequential statement list and Φ is a boolean expression in VHDL. The semantics of **reach** S **hold** Φ is defined as $\mathsf{AG}[\mathsf{enter}(S) \rightarrow [\Phi \,\underline{\mathsf{W}}\, \mathsf{leave}(S)]$, where $\underline{\mathsf{W}}$ is the 'strong when' operator[5]. Hence, **reach** S **hold** Φ means

[2] VHDL does not have free inputs. If an input of an entity is left open, the input is set to a predefined value (e.g. see [1, pp. 165, lines 492–493]).

[3] Of course, these predicates must consider the context of S, but we neglect this in the following.

[4] G is the temporal logic *always* operator that states that a property must hold from a certain point of time on and A is an operator that requires that the property must hold for all possible computation sequences [15].

[5] $[a \,\underline{\mathsf{W}}\, b]$ holds at a certain point of time t_0, iff b must hold at least once after t_0 and a holds when b holds for the first time after t_0. There is

that whenever the point of the program is reached where the execution of S is started, it *must terminate and then* Φ *must hold*.

The difference between S; **assert** Φ and **reach** S **hold** Φ is that with the latter we can express that S terminates. In particular, the sequence S; **assert** Φ does not cover this fact, as we can only conclude from the semantics that $\mathsf{AG}[\mathsf{leave}(S) \rightarrow \Phi]$ holds, which is the same as $\mathsf{AG}[\Phi\ \mathsf{W}\ \mathsf{leave}(S)]$. As in the latter formula only the weak W operator occurs, it is not necessarily the case that S will ever terminate. Adding the new specification construct to VHDL hence extends the specification capabilities from *partial correctness* to *total correctness*[6].

Moreover, using **reach** S **hold** Φ we can control the paths on which the program point has been reached as we consider two program states, namely $\mathsf{enter}(S)$ and $\mathsf{leave}(S)$ instead of a single one. For example, we can also observe the relationship between certain signals.

The allowed positions of the key words **reach** and **hold** within a sequence of statements may depend on the actual used formal semantics of VHDL. If a formal semantics on the lowest time abstraction level [17], the simulation cycle, is used, then all positions are allowed. If a formal semantics is used, which executes all sequential statements between two wait statements within one step (as it is usually done for e.g. verifying fully synchronous circuits), **reach** and **hold** are only allowed to appear directly after a (not necessarily the same) wait statement.

A lot of results have been found about the expressiveness of various formalisms used for specifying and verifying properties. For example, it is well-known that nondeterministic Büchi automata are more powerful than deterministic ones [14]. However, it has been shown that if universal quantification is added, then both are equally expressive [18]. We have chosen the new specification constructs for the verification bench in such a way that we can model deterministic Büchi automata with universal quantification with the observers: the state transitions of an finite-state automaton can be directly expressed in an observer written in VHDL and the universal quantification is covered by $\mathsf{T'}$**arbitrary** function calls. The acceptance condition of a Büchi automaton requires that a propositional property Φ must hold infinitely often for each computation sequence. This can be modeled with the following VHDL program[7]:

```
process is begin
  reach
    wait until Φ;
  hold TRUE;
  null;
end process;
```

Hence, our specification language is as expressive as deterministic Büchi automata with universal quantification, and hence as expressive as nondeterministic Büchi automata [18]. As it is well-known that these are at least as expressive as all other known ω-automata [14], and as expressive as some arithmetic approaches [19], and even more powerful than linear temporal logic [20], we have a very powerful specification language.

3.3 Simulating Verification Benchs

For simulation with an existing VHDL simulator, the function calls $\mathsf{T'}$**arbitrary** have to be replaced by (arbitrary) concrete values of type T. If a verification run yields in a countermodel, this countermodel provides concrete values for each call of $\mathsf{T'}$**arbitrary**. Proceeding this way, a countermodel can be visualized by the possibilities of a common VHDL simulator.

reach S **hold** Φ statements can be replaced by an appropriate combination of report and assertion statements. In that way, a formal specification using a verification bench can be checked for some concrete stimuli by simulation. For example, **reach** S **hold** Φ can be replaced for simulation by the following piece of VHDL code:

```
report "entered..." severity NOTE;
S
report "...leaved." severity NOTE;
assert Φ report "Φ is false" severity ERROR;
```

The described **reach** S **hold** Φ property is not fulfilled if for a reported message "entered...", no according message "...leaved." is reported or if a message "Φ is false" is reported.

3.4 Verification Workflow

The design flow of our method requires that the designer writes a verification bench for verification similar to writing a test bench for simulation. We then compute automatically a finite state machine from this verification bench according to a fixed semantics for VHDL [3]. Additionally, all **reach** S_i **hold** Φ_i statements are collected and form now a specification of the form

$$\bigwedge_{i=1}^{n} \mathsf{AG}[\mathsf{enter}(S_i) \rightarrow [\Phi_i\ \underline{\mathsf{W}}\ \mathsf{leave}(S_i)]$$

also a 'weak when' operator $[a\ \mathsf{W}\ b]$ that states that a must hold for the first time when b holds, but if b never holds then $[a\ \mathsf{W}\ b]$ holds also (see also [16] for further discussion).

[6] This distinction of correctness by termination is done in dynamic logics used for program verification: **reach** S **hold** Φ is a VHDL equivalent of the dynamic logic formula $\langle S\rangle\Phi$ and S; **assert** Φ is equivalent to $[S]\Phi$.

[7] To see that the program actually implements $\mathsf{AGF}\Phi$ consider the following proof: abbreviating the statement **wait until** Φ with S, we can first conclude that $\mathsf{AG}[\mathsf{enter}(S) \rightarrow [\mathsf{T}\ \underline{\mathsf{W}}\ \mathsf{leave}(S)]]$ by the definition of reach/hold. This is equivalent to $\mathsf{AG}[\mathsf{enter}(S) \rightarrow \mathsf{F}[\mathsf{leave}(S)]]$ by the semantics of the strong when operator. According to the semantics of the **wait** construct, $\mathsf{leave}(S)$ can only hold when Φ holds. As S is entered infinitely often, we can finally conclude that $\mathsf{AGF}\Phi$ must hold.

The above formula is not a CTL formula such that we can not directly use standard CTL model checkers. However, the translation method presented in [16] can be used to translate the above formula to the following equivalent CTL formula:

$$\bigwedge_{i=1}^{n} \mathsf{AG}\left[\begin{array}{l}\mathsf{enter}(S_i) \rightarrow \\ \qquad \mathsf{A}[(\neg\mathsf{leave}(S_i)) \,\underline{\mathsf{U}}\, (\Phi_i \wedge \mathsf{leave}(S_i))]\end{array}\right]$$

As a back-end tool, we currently use the CTL model checker SMV [21] to check that the above formula holds for the generated finite state structure. Using model checking as verification technique, we currently have to restrict the given VHDL implementation descriptions. For example, we have to assume that only data types over a finite domain occur (otherwise we could not translate to a finite-state machine), and we are not able to handle generate statements.

4 Examples

In the following, some typical properties of an environment or of an implementation will be presented.

4.1 Lift Controller

Consider a lift controller, which has a detector in its environment. The detector sets the input signal of the controller `door_open` to `TRUE` if the door of the lift is open. All events on `door_open` lag between 2 ns and 7 ns behind the rising edge of a clock signal. The stimuli generator representing this environment is given below.

```
process is
  type R is range 2 to 7;
begin
  wait until Clk = '1';
  door_open <=
  BOOLEAN'arbitrary after
  TIME'VAL(R'arbitrary);
end process;
```

The output signal `go_up` of the lift controller makes the lift to go up and its output signal `go_down` makes the lift to down. Below an observer is given, which specifies that the door of the lift is never open when the lift is moving. It works as follows: If at some time, the door is open and the lift is moving, the wait statement terminates and the specification statement will be executed. Then `FALSE` should hold. As `FALSE` never holds, the lift controller would violate the specification.

```
process is begin
  wait until door_open
  and (go_up or go_down);
  reach hold FALSE;
end process;
```

4.2 Loop Termination

A specification statement must not always appear in context with a wait statement. Consider the loop statement shown below. The specification statement specifies that this loop will terminate at some time, regardless of any wait statement.

```
process is begin
  ...
  reach
    while x < y loop
      ...
    end loop;
  hold TRUE;
  ...
end process;
```

4.3 A simple Protocol

Always when there is a rising edge for signal `request`, 3 μs to 7 μs later a falling edge on signal `enable` should follow (for simplicity, assume that as long `enable` as has not been falling, `request` would not fall and rise again).

```
process is begin
  wait until request = '1';
  reach
    wait until enable for 3 us;
  hold not enable'EVENT;
  reach
    wait until enable for 7 us;
  hold enable = '0';
end process;
```

5 Current Work

As most of our efforts are currently invested in building a new verification system C@S[8], we currently adapt the approach presented in this paper to the new context. C@S itself is not based on the use of VHDL, it rather has its own system description language PURR that is more suited for verification than VHDL. PURR is a superset of ESTEREL [23] enhanced by many features useful in the context of hardware verification [22, 24]. Also in PURR, we provide separated means to denote implementations and associated specifications. However, we have extended the approach of this paper in several ways:

- The **arbitrary** construct of VHDL allows only the selection of an arbitrary value of a given type. In PURR we have generalized this notion to $\mathbf{CHOOSE}\ x.\Phi$ such that a value of a given type that satisfies an additional property Φ is chosen[9].

[8] See `http://goethe.ira.uka.de/hvg/cats/` or [22] for more details.

[9] This is nothing else than Hilbert's choice operator [25].

- In the setting described in this paper, it is not possible to directly constrain input sequences. In the C@S system, a **REQUIRES** statement allows to forbid all inputs that do not satisfy the given requirements. In particular, we can express fairness constraints known from model checking.

- As PURR is based on ESTEREL, it is possible that a single thread can fork into two or more other threads[10] while running a program. Hence, the simple **reach** S **hold** Φ can no longer be used to reason about control points since we must be able to control which 'reach' is related to which 'hold'. Thus, we use labeled assertions **ASSERT**(p_1, Φ) where p_1 is a name that labels the current control point and Φ is a temporal logic formula that must hold whenever **ASSERT**(p_1, Φ) is reached. If S is a statement where no fork to new threads occurs, then **reach** S **hold** Φ is equivalent to

ASSERT$(p_1, \mathsf{AG}[p_1 \rightarrow [\Phi \,\underline{\mathsf{W}}\, p_2]])$;
S;
ASSERT(p_2, T);

6 Conclusions

In this paper we have presented a new specification paradigm for VHDL. Properties are specified using deterministic ω-automata with universal inputs and can be written directly in form of an observer module in VHDL. New statements, namely T' **arbitrary** and **reach** S **hold** Φ have been introduced such that a powerful specification language is obtained. For compatibility with existing tools, we have also given means to translate our VHDL descriptions into standard VHDL to carry out e.g. a simulation with a commercial simulator. We claim that the use of a FSM-based specification paradigm is more natural and easier to learn for a designer who is not familiar with formal methods.

We currently have implemented a backend to a CTL model checker for our extended VHDL semantics. As for the correctness proof the product Kripke structure of the design and the accepting automata has to be constructed, the size of the verifiable designs is limited. Our approach is however not restricted to FSM-based techniques and hence, we are also exploring theorem proving based correctness proofs.

Acknowledgements

We are grateful to the anonymous Euro-DAC reviewers for their valuable feedback. Especially the comments on partial vs. total correctness and statement execution allowed us to considerably improve the paper.

[10] The semantics of PURR and ESTEREL assure however, that each program can only generate a finite number of threads.

References

[1] ANSI/IEEE Std 1076–1993. *IEEE Standard VHDL Language Reference Manual*. IEEE, New York, USA, June 1994.

[2] A. Gupta. Formal hardware verification methods: A survey. *Journal of Formal Methods in System Design*, 1:151–238, 1992.

[3] R. Reetz and T. Kropf. Evaluating possibilities for formally sound simulation and verification of VHDL. In S. Singh and M. Sheeran, editors, *Proceedings of the 3rd Workshop on Designing Correct Circuits*, Electronic Workshops in Computing, Baastad, Sweden, September 1996. Springer Verlag.

[4] D. Borrione. Special Issue on VHDL Semantics. *Formal Methods in System Design*, 7(1/2), August 1995.

[5] C. Delgado Kloos and P.T. Breuer, editors. *Formal Semantics for VHDL*, volume 307 of *The Kluwer international series in engineering and computer science*. Kluwer Academic Publishers, Madrid, Spain, March 1995.

[6] Andreas Kuehlmann and Florian Krohm. Equivalence checking using cuts and heaps. In *Proc. 34th Design Automation Conference*, Anaheim, CA, June 1997.

[7] O. Coudert, C. Berthet, and J.C. Madre. Verification of sequential machines using boolean functional vectors. In L. Claesen, editor, *IMEC-IFIP Workshop on Applied Formal Methods for Correct VLSI-Design*, 1990.

[8] W. Damm, B. Josko, and R. Schlör. Specification and verification af VHDL-based system-level hardware-design. In E. Börger, editor, *Specification and Validation Methods for Programming Languages and Systems*. Oxford University Press, 1994.

[9] K. Fisler and P. Kurshan. Verifying VHDL designs with COSPAN. In T. Kropf, editor, *Formal Hardware Verification – Methods and Systems in Comparison*, volume 1287 of *Lecture Notes in Computer Science*, pages 206–247. Springer Verlag, state of the art report edition, August 1997.

[10] M. Bickford and D. Jamseck. Formal specification and verification of VHDL. In M. Srivas and A. Camilleri, editors, *First international conference on formal methods in computer-aided design*, volume

1166 of *Lecture Notes in Computer Science*, pages 310–326, Palo Alto, CA, USA, November 1996. Springer Verlag.

[11] S.D. Johnson, P.S. Miner, and A. Camilleri. Studies of the single pulser in various reasoning systems. In T. Kropf and R. Kumar, editors, *Proc. 2nd International Conference on Theorem Provers in Circuit Design (TPCD94)*, volume 901 of *Lecture Notes in Computer Science*, pages 126–145, Bad Herrenalb, Germany, September 1994. Springer-Verlag. published 1995.

[12] L.M. Augustin, B.A. Gennart, Y.Huh, D.C. Luckham, and A.G. Stanculescu. Verification of VHDL designs using VAL. In *25th Design Automation Conference*, pages 48–53. ACM/IEEE, 1988.

[13] J. van Tassel and D. Hemmendinger. Toward Formal Verification of VHDL Specifications. In *International Workshop on Applied Formal Methods for Correct VLSI Design*, pages 409–418. IFIP WG 10.2/WG 10.5, North-Holland, 1990, 1989.

[14] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 133–191, Amsterdam, 1990. Elsevier Science Publishers.

[15] E.A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 996–1072, Amsterdam, 1990. Elsevier Science Publishers.

[16] K. Schneider. CTL and equivalent sublanguages of CTL*. In C. Delgado Kloos, editor, *Proceedings of International Conference on Computer Hardware Description Languages and their Applications*, pages 40–59, Toledo,Spain, April 1997. IFIP, Chapman and Hall.

[17] R. Reetz and T. Kropf. A Flowgraph Semantics of VHDL: A Basis for Hardware Verification with VHDL. In C.D. Kloos and P.T. Breuer, editors, *Formal Semantics for VHDL*, volume 307 of *The Kluwer international series in engineering and computer science*, chapter 7, pages 205–238. Kluwer Academic Publishers, Madrid, Spain, March 1995.

[18] K. Schneider. *Ein einheitlicher Ansatz zur Unterstützung von Abstraktionsmechanismen der Hardwareverifikation*, volume 116 of *DISKI (Dissertationen zur Künstlichen Intelligenz)*. Infix Verlag, Sankt Augustin, 1996. ISBN 3-89601-116-2.

[19] J.R. Büchi. Weak second order arithmetic and finite automata. *Z. Math. Logik Grundlagen Math.*, 6:66–92, 1960.

[20] P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56:72–99, 1983.

[21] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell Massachusetts, 1993.

[22] K. Schneider and T. Kropf. A unified approach for combining different formalisms for hardware verification. In M. Srivas and A. Camilleri, editors, *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, volume 1166 of *Lecture Notes in Computer Science*, pages 202–217, Palo Alto, USA, November 1996. Springer Verlag.

[23] B. Berry, P.Courronné, and G. Gonthier. *Programming of Future Generation Computers*, chapter Synchronous programming of reactive systems, an introduction to Esterel. Elsevier Science Publisher (North Holland), 1988. INRIA Report 647.

[24] T.Kropf, J. Ruf, M. Wild, and K.Schneider. A synchronous language for modeling and verifying real time and embedded systems. Technical report, Universität Karlsruhe, Institut für Rechnerentwurf und Fehlertoleranz, 1997.

[25] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[26] T. Kropf. Benchmark-Circuits for Hardware-Verification. In T. Kropf and R. Kumar, editors, *Proc. 2nd International Conference on Theorem Provers in Circuit Design (TPCD94)*, volume 901 of *Lecture Notes in Computer Science*, pages 1–12, Bad Herrenalb, Germany, September 1994. Springer-Verlag. published 1995.