Model Checking in Industrial Hardware Design *

Jörg Bormann, Jörg Lohse, Michael Payer and Gerd Venzl

Siemens Corporate R&D
D-81730 Munich
Germany
Joerg.Bormann@zfe.siemens.de

Abstract—This paper describes how model checking has been integrated into an industrial hardware design process. We present an application oriented specification language for assumption/commitment style properties and an abstraction algorithm that generates an intuitive and efficient representation of synchronous circuits. These approaches are embedded in our Circuit Verification Environment CVE. They are demonstrated on two industrial applications.

I. Introduction

In todays' hardware design processes the validation phase requires such a large effort that fast alternatives to simulation have a great potential to shorten the overall time for the development of a circuit. One such promising alternative is the use of formal methods [15]. Among these we favour BDD [4] based symbolic model checking, i.e., tools that automatically prove properties about finite state systems such as absence of deadlocks, lifeness properties ("something good will eventually happen"), or safety properties ("something bad will never happen"), and produce input sequences that contradict a property if a proof fails.

The main advantage of these algorithms is that they work automatically and are often capable to examine circuits of industrially relevant sizes within minutes. The user only needs to specify the properties to be checked whereas other formal methods such as theorem proving [3, 5] require a deep knowledge of the circuit internals.

The ease of use allows the integration of a model checker into the traditional hardware design process as a supplementary tool for the various validation phases. It can be used, e.g., to check with some suitable properties whether a circuit indeed meets a designer's intentions, to verify the synthesis step, or to validate post synthesis modifications such as retiming and engineering changes.

*This work has been supported by JESSI AC-8 and ESPRIT project FORMAT

To use a supplementary model checker the designers must only spend a reasonable specific effort. It pays off substantially if the model checker terminates successfully because it can replace many simulation runs. On the other hand the designer knows that almost no effort is wasted if the model checker encounters prohibitive complexity problems. He then proceeds with simulation as usual.

Besides a powerful model checker, this requires appropriate interfaces from and to the traditional hardware design process and a user interface that can quickly be mastered by the designers. To our knowledge the Circuit Verification Environment CVE [2] is the first tool that satisfies these requirements. CVE supports EDIF [16] and VHDL [11, 12] and generates VHDL test benches for counter examples if it detects a design error. CVE is operated from a graphical menu driven user interface.

The designer has to specify the properties to be model checked which forces him to think about whole sets of behaviors. This is uncommon to the current validation practice where the designer concentrates on one simulation run after the other. The necessary familiarization can be considerably facilitated with an application oriented language for the specification of properties. For this purpose we developed CVE's Interval Language (CIL). Another important help is provided by an algorithm that automatically generates a special finite state machine (FSM) representation for synchronous circuits (the *synchronous machine*) which captures restrictions of the input behavior common in a synchronous setting. Consequently these restrictions need not be explicitly stated in a model checking property.

This paper presents CIL (Section II) and the automatic extraction mechanism for synchronous machines (Section III). The relation to other research is discussed in Section IV. Applications of CVE are presented in Section V. Section VI concludes this paper.

II. CVE's Interval Language CIL

Model checking requires the specification of a property in a formal language. To be used in industrial hardware design, the language must be application oriented, comprehensible for designers and efficiently treated by the model checker. This goal is met by CVE's Interval Language CIL which is optimized for the use with synchronous designs as this is the implementation method for most hardware functionality. Synchronous circuits have a special internal representation that performs one state transition per clock cycle. It is generated with the algorithm of Section III. CIL introduces an explicit notion of time which counts the state transitions of the underlying FSM. This establishes a close relation between the real timing of a synchronous circuit and the state transitions of its special internal representation.

CIL is an extension of Boolean VHDL expressions by some constructs for temporal specifications. Since VHDL is the base of CIL, definitions from the circuit design phase (such as type or function declarations) can be used to specify the properties. The language is employed to specify commitments containing the proof goal as well as assumptions which describe the behavior of the environment.

The development of CIL was motivated by expressions such as

$$request|_t \Rightarrow acknowledge|_{x>t}$$

It specifies that every request will be acknowledged. Expressions like this are informally used in data sheets or in discussions among designers. Like these expressions, CIL formulae are built up from *timed predicates* which consist of a state predicate and a temporal specification. The temporal specification describes when the machine should be in a state that satisfies the state predicate.

The state predicate is given in the subset of Boolean expressions in VHDL. The temporal specification refers either to a particular point of time, or to a whole period. A point of time is specified behind the keyword `at`. A period is specified by an interval, which is a uniform representation of three different types (`T`, `T1`, and `T2` are points of time):

`[T1, T2]`, refers to the time between `T1` and `T2` inclusively.

`[T, infinite]`, refers to T and every point after T.

`[T, p]`, refers to the time between `T` and the last point of time before the state predicate `p` is satisfied for the next time.

An interval is preceded by `during` or `within` to specify whether the state predicate holds during the whole period or at least once in the interval.

Times are either integer constants or defined relative to a variable `t` which is universally or existentially quantified by `always` or `finally`.

In commitments, timed predicates with temporal specifications relative to `t` may be preceded by the keyword `possibly` to denote that for every state of the FSM at an arbitrary point of time `t` there must be at least one execution path that satisfies the state predicate at the given time.

Some examples for CIL-expressions are:

`acknowledge = true at 5` specifies that the signal acknowledge is true after the 5th state transition following the initial state of the underlying FSM.

`reset = '0' during [0, infinite]` is true, iff the reset signal is always `'0'`.

```
always((request = '1' at t) implies
    (acknowledge = '1' within [t+1, infinite]))
```
specifies, that every request must be answered by an acknowledge that occurs at least one state transition after the request.

`finally(initialized = '1' during [t, infinite])` requires the signal `initialized` to remain `'1'` after some point of time.

```
always(write = '1' within [t, infinite]) and
    always((write = '1' at t) implies (write =
    '0' during [t+1, t+3]))
```
requires that the `write` signal is infinitely often `'1'` but with at least three state transitions with `write = '0'` in between.

```
always((resetCntr = '1' at t) implies
    (resetCntr = '0' during [t+1, counter=5]))
```
specifies that after the signal `resetCntr` was active, it must remain deactivated until the counter reaches the value 5.

A property to be model checked is specified in a *theorem* which consists of a CIL formula for the assumption (preceded by `assume:`) and a CIL formula for the commitment (preceded by `prove:`).

CIL is supported by a parser and a translator to the temporal logics CTL and LTL for commitments and assumptions, respectively, which are the input language of SVE, the model checker that underlies CVE. CIL expressions are separated into the VHDL expressions and the dynamic structure given by the temporal specifications. The latter is compiled into temporal logic while the VHDL expressions are translated by the VHDL frontend of CVE [11]. This ensures, that user defined VHDL packages can be consistently used in CIL.

III. Synchronous Machines

CIL is generally applicable and provides a comprehensible language for the specification of the properties of any circuit. However, CIL is most intuitive when used in conjunction with a synchronous machine, i.e., an FSM for which the ith entry in the input sequence represents the input of the corresponding synchronous circuit read at the ith clock edge and the ith entry in the output sequence represents the output that is generated after the ith clock edge.

The algorithm presented in this section extracts a synchronous machine from the *macro machine* that is generated by the CVE frontends [11] from almost arbitrary asynchronous designs. Macro machines are FSMs that perform one state transition at every input change. To build a macro machine, the CVE frontends assemble all computations that are invoked in the zero delay model by an input change into one state transition. Consequently a macro machine transits several times within a clock cycle. Our algorithm combines all transitions in a clock cycle and then removes the clock input. See Fig. 2 for a comparison of the macro machine representation of a flip flop and the corresponding synchronous machine.

The advantage of synchronous machines is that the relation between the temporal specifications in CIL and the timing of the circuit is obvious. Moreover, a synchronous machine captures restrictions of the input behavior

program synchronize
Inputs: (λ, δ, (I_s, I_p)) (* macro machine *), Clock

identify old clock bit p (* transition function equals clock *)
if p cannot be identified then report error; stop
$F_1(s, i_0) \quad := \quad (\lambda(1, i_0, 0) \circ \delta(0, i_0, 1) \circ \delta(1, i_0, 1))(s)$
$F_2(s, i_0, i_1) \quad := \quad (\lambda(1, i_1, 1) \circ \delta(1, i_0, 0) \circ \delta(0, i_0, 1) \circ \delta(1, i_0, 1))(s)$
$F_3(s, i_0, i_1) \quad := \quad (\lambda(0, i_1, 1) \circ \delta(1, i_1, 1) \circ \delta(1, i_0, 0) \circ \delta(0, i_0, 1) \circ \delta(1, i_0, 1))(s)$
if not $F_1 = F_2 = F_3$ (* synchronization criterion *) then report error; stop
$\Lambda(s, i) \quad := \quad F_1(s, i)$
$\Delta(s, i) \quad := \quad (\delta(1, i, 0) \circ \delta(0, i, 1) \circ \delta(1, i, 1))(s)$ (and eliminate old clock bit)
if not $I_p = 1$ then $(I_p, I_s) := \delta(1, i', I_p, I_s)$ with an appropriate input i'.
$I \quad := \quad I_s$
return (Λ, Δ, I) (* synchronous machine *)

Figure 1: Abstraction Algorithm for Synchronous Machines

common in a synchronous setting as e.g., the toggling of the clock and the temporal relation between changes of clock and data inputs. Consequently, these restrictions need not be formalized for model checking of synchronous machines whereas they lead to bulky assumptions if the corresponding macro machine is examined.

Fig. 1 presents the abstraction algorithm for the case of circuits that are sensitive to the *rising* clock edge. A similar algorithm applies, if a circuit is sensitive to the falling clock edge. The algorithm assumes that the state of the macro machine contains one particular bit, the *old clock bit*, which stores the last value of the clock input. This assumption is reasonable because the old clock value is necessary to detect clock edges. The old clock bit can be automatically identified since its state transition function always returns the value of the clock input.

To make the old clock bit and the clock input explicit the state transition function and the output function of the macro machine are denoted by $\delta(c, i, p, s)$ and $\lambda(c, i, p, s)$, respectively, where c is the clock, p the old clock bit, i all other inputs, and s all other state variables. (I_s, I_p) denotes the initial state of s and p in the macro machine. For simplicity we write $(F(a, b, c) \circ G(u, v, w))(z)$ instead of $F(a, b, c, G(u, v, w, z))$.

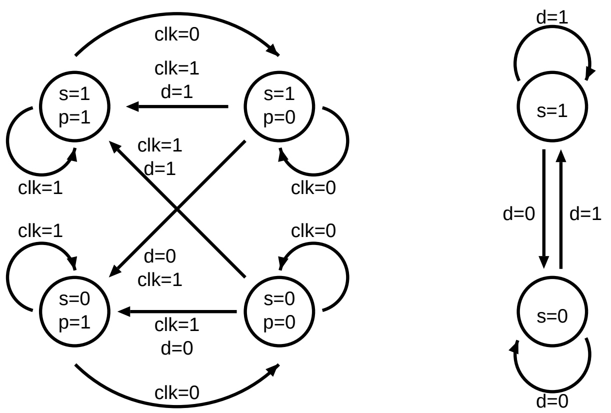

Figure 2: State Graph of the Macro Machine (left) and Synchronous Machine (right) of a Flipflop. (s, p and d denote state, old clock bit and data input, respectively)

The algorithm eliminates the clock input and the old clock bit. It assumes that the circuit is part of a larger synchronous system and receives the input timing given in Fig. 3 where the data inputs change simultaneously when the clock is high. These changes divide a clock cycle into the three intervals I, II, and III. The macro machine computes state and output for each interval by a separate state transition. The main idea of the algorithm is to determine the state transition function Δ of the synchronous machine, such that it computes the state of the macro machine in interval II of every clock cycle. To obtain Δ the state transitions of the macro machine between successive intervals II must be combined. Due to the given clock behavior the clock input and the old clock bit are known in advance. Moreover, the data inputs must be kept stable during these state transitions. This gives the expression that is assigned to Δ in the algorithm.

The algorithm checks whether the corresponding circuit is synchronous and sensitive to the rising clock edge with a *synchronization criterion*. It is satisfied iff under the timing of Fig. 3 the outputs of the zero delay model of the circuit are stable between two rising clock edges. The output of the macro machine in every interval of the clock cycle is determined by the functions F_1, F_2, and F_3 in the same way as described for Δ. The functions must be equal to satisfy the synchronization criterion.

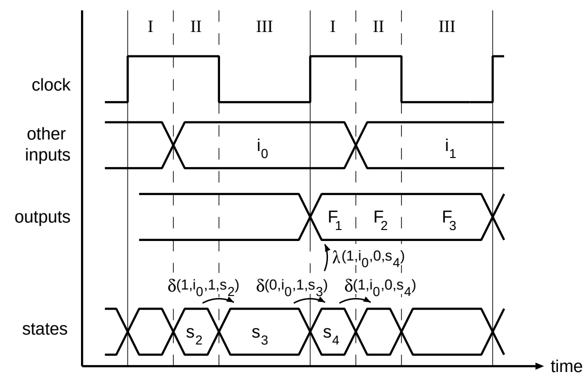

Figure 3: Relation between Input and Output Timing

The algorithm is well adopted to the internal representation of FSMs in CVE, where the state transition and output functions are represented by BDDs [4]. First the concrete values for the clock input and the old clock bit are applied which reduces most of the BDDs considerably. Then the substitutions are performed on these BDDs. The execution time of this algorithm was negligible for all examples we tried so far.

Examples show that synchronous machines have considerably smaller BDD representations than the corresponding macro machines and that often several state bits of the synchronous machine are redundant such that they are eliminated in a post processing step. Additionally, the number of iterations of the model checker is cut in half. As a consequence the execution time of the model checker is reduced to at least one half and for some examples even one thirtieth of the time spent for a verification of the corresponding macro machine. Results are given in Section V.

IV. Comparison to Related Work

How model checking can be applied in hardware design has been studied in academia using tools like HSIS [1] and SMV [14]. HSIS supports extensions of BLIF and Verilog, whereas SMV has an own textual representation of the models under examination. Both tools check properties by model checking of CTL formulae. HSIS additionally provides language containment algorithms for L-Automata. The circuit environment is described by automata and fairness constraints that exclude certain infinite behaviors. Research with these tools examined at which stages of the hardware design process model checking can be used most beneficially [13, 15] and how the complexity of model checking problems can be reduced [7]. Consequently, the user interfaces are made for expert users and are flexible rather than comprehensible.

CTL, the specification language for properties in both tools, is far more expressive than CIL. However, to our experience the expressiveness of CIL is sufficient for typical verification tasks in hardware design. On the other hand, CIL formulae are more comprehensible than the corresponding mixture of automata, fairness constraints, and CTL commitments which are needed to specify properties for HSIS or SMV.

It is an advantage of CIL that it introduces an explicit quantitative notion of time. The Symbolic Timing Diagrams (STD) [8] introduce time in a qualitative sense. The clock signal must be explicitly included. STDs can express properties that cannot be specified in CIL, e.g., a sequence of events leading to some other event. However, such properties can be expressed in CVE if the model is complemented by a test bench that detects these sequences of events. While it is an open issue whether STDs are more useful for designs than a language like CIL, we expect the model checking of a property specified in STDs to be considerably more complex than model checking of the corresponding CIL formulae.

In [9] a VHDL subset is described from which an FSM representation similar to our synchronous machines is directly extracted. This subset is more restrictive than the subset supported by CVE [11]. Our two step approach allows to examine synchronous VHDL descriptions with small asynchronous portions that do not influence the genuine functionality (e.g., asynchronous resets). Most of our industrial examples are of that form. These descriptions are translated with the CVE frontend into macro machines and, after deactivating the asynchronous portions, the synchronous machine is extracted.

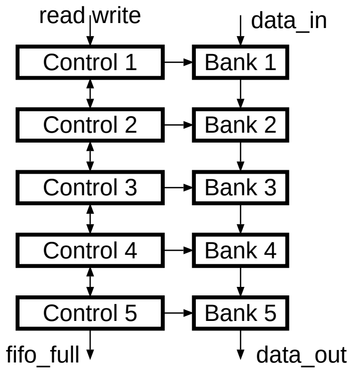

Figure 4: Implementation of 4x5 FIFO

V. Two Industrial Applications

This section presents applications of CVE to two selected industrial hardware designs. One is a generic first–in–first–out memory (FIFO) for which model checking was applied during the design validation phase. CVE quickly spotted a design error. We examine different instances of the FIFO and present the execution times of the model checker and the debugging facility of CVE both for macro machines and the corresponding synchronous machines.

The second example describes the verification of a token ring controller, a real world example of significant complexity. This example illustrates how properties which are easy to specify can check designs to an extent far beyond the capabilities of simulation.

A. *FIFO memory*

The FIFO is a generic VHDL description that shall be used in different designs. It is scalable in size and optimized to reduce the area consumption of the synthesized circuits. It is implemented by banks of latches (cf. Fig. 4). Data move from the first bank to the last. A control mechanism is associated with each bank and determines whether the bank contains valid data. If the bank is empty and the previous bank contains valid data, the data is moved down one bank.

Instances of the generic description were compiled into macro machines. These did not satisfy the synchronization criterion (cf. Section III) because under some operation conditions the latch banks can be transparent simultaneously, so that changes at the data input are immediately observable at the outputs. However, it was easy to obtain a synchronous machine with a test bench where the inputs of the FIFO are buffered by registers and, in addition, the asynchronous reset is deactivated.

The synchronous machine representations of, e.g., the 4x5 FIFO was checked by the CIL property

		depth					
		5		10		15	
width		mc	debug	mc	debug	mc	debug
4	macro	69.3	17.6	323.9	44.4	602.6	69.0
	synch	3.8	3.6	12.3	9.4	55.5	23.1
8	macro	37.4	17.0	852.5	104.3	1336.7	142.1
	synch	4.2	5.5	25.3	20.2	133.9	49.9
12	macro	373.3	66.3	711.0	98.0	6908.5	618.3
	synch	27.7	14.6	49.7	34.0	226.1	82.3
16	macro	208.0	49.8	2953.0	315.0	11436.6	986.3
	synch	15.9	14.9	83.3	49.8	379.8	130.4

Figure 5: Execution times for FIFO memory (in seconds on a SPARC 10 with 128 MB, mc: Time to detect the error, debug: Time to construct the counter example)

```
theorem noLostData;

assume:
  always((write = '1' at t) implies
         (write = '0' during [t+1, t+2])) and
  always((fifo_full = '1' at t) implies
         (write = '0' at t + 1)) and
  always((read  = '1' at t) implies
         (read  = '0' during [t+1, t+2])) and
  always (read = '1' within [t, infinite]);

prove:
  always(
    (write = '1' and data_in = "1010" and
       fifo_full = '0' at t) implies
    (data_out = "1010" within [t, infinite])
  );

end theorem;
```

This property specifies that if the data "1010" was read in at some point of time t, it will eventually show up at the output port. The assumption describes the environment of the FIFO which assures, that

- consecutive write pulses are separated by 2 clock cycles,
- no writes will occur, if the FIFO is full,
- consecutive read pulses are separated by 2 clock cycles, and
- infinitely many reads will happen.

This property is natural for a FIFO. It is a specialized form of the general property that no data is lost and it tests the control of the FIFO which is its critical part. The verification of this property, indeed, exhibited a subtle design error that occurs, if a full FIFO receives a read pulse that is immediately followed by a write pulse. The input sequence that exhibits the faulty behavior consists of 26 steps, where the last 10 steps have to be repeated infinitely often.

Fig. 5 presents the execution times for the verification and the debugging of the above property based on synchronous machines and compares them with the verification of an equivalent property about the macro machine representation. A comparison of the execution times shows that model checking of synchronous machines can be 30 times more efficient than model checking of equivalent problems of macro machines.

This example shows how CVE can be used successfully. Since it requires almost no effort to start the model checker, parts of a design can be checked at very early stages where a bug is quickly located and fixed. Note that usually small scale instances of designs are sufficient to detect the most subtle errors in the control.

B. *Token Ring Controller TC*

In one of the most complex verification tasks performed with CVE a token ring controller (TC) was examined which is part of the clients in a particular industrial bus system. The TCs at the bus organize the bus access by a token ring protocol. This protocol is automatically configured in the startup phase and reconfigured in inconsistent situations. The timing for the TC is provided by one timing generator (BT) for each TC. All BTs in the bus system are synchronized.

Both blocks were described in synthesizable VHDL. The designers of the TC had to cope with the problems that arise from the complexity of distributed control. Moreover, the TC is connected to the bus via an interface unit that delays data sent and received by some clock cycles. This additionally increases the logical complexity. Therefore, the validation by simulation took three times as much as the development of the blocks.

Since we used the existing VHDL descriptions the circuits were described in great detail so that we had to examine a minimal configuration (Fig. 6) where the bus has only one client. The corresponding model consists of one TC, one BT, an abstraction of the bus interface unit in the form of a delay block, and a multiplexor to model bus errors. This configuration was considered to be sufficient by the designers because important functionality can be verified with it. It was also used during extensive simulation. For the treatment with CVE the model was transformed into a synchronous machine.

The verification of the TC requires the computation of the set of reachable states to optimize the model checking. For this reachability analysis a hierarchical approach [6] and redundancy elimination [10] were employed. Even then, the reachability analysis took 4 days on a SPARC 10 with 128 MByte memory. After this effort important properties could quickly be verified. Among them were lifeness properties about correct execution of the configuration procedure and token passing. These properties could be specified in single lines of CIL code and verified

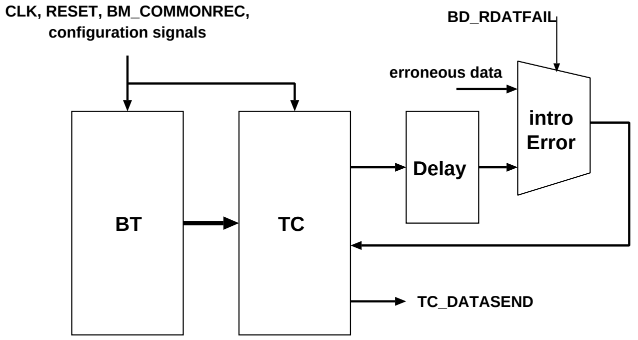

Figure 6: Minimal Configuration of a Bus System with TC

within 1 to 2 hours. The most impressive property described the bus access: If data should be sent to the bus (i.e., signal BM_COMMONREC active) the TC will allow the bus access (i.e., signal TC_DATASEND active), provided that there is a point of time after which no more bus errors occur. The CIL notation of this commitment is

```
prove: always((BM_COMMONREC = '1' at t) implies
(TC_DATASEND = '1' within [t, infinite]));
```

CVE verified the property within 4.5 hours. This short property examines the following operations:

- Detection that no token is present,
- Configuration procedure,
- Detection, that no partner TC is in the bus system,
- Creation of the token,
- Passing of the token,

with the guarantee that bus errors occuring at arbitrary points of time cannot drive the TC into a deadlock situation. This single verification therefore greatly increased the confidence in the design. This example shows that CVE can handle complex industrial designs that are described in the level of detail that is common in the validation phase.

VI. Results and Future Work

The applications presented show that CVE is ready for the use in industrial hardware design. It reduces the interaction with the designer to a minimum since all algorithms execute automatically thus minimizing the learning effort. CVE offers interfaces which allow a seamless integration with the established design flow as a powerful supplement to simulation. Properties to be model checked by CVE are described using VHDL test benches and the application oriented, comprehensible language CIL. CIL has less expressive power than other property languages but our examples show this to be sufficient.

CIL can be used for the specification of properties of any circuit but it is most useful for the application to synchronous designs. They are represented in CVE by synchronous machines which provide an appropriate level of detail. The abstraction mechanism that generates this representation was presented. Experimental results showed that synchronous machines are up to 30 times more efficiently model checked than the more general macro machines generated by the CVE frontends.

Future work will extend CIL to reduce the need for test benches. These test benches often introduce shift registers to store aspects about the past of the circuit. CIL can naturally be extended to specify these additional shift registers without additional VHDL code.

Acknowledgments

The authors would like to thank D. Werth, T. St. Pierre, A. Rademacher, D. Emmer, V. Weyl, and in particular T. Filkorn and P. Warkentin for their support of this work.

References

[1] A. Aziz, F. Balarin, S.-T. Cheng, R. Hoyati, T. Kam, S.C. Krishnan, R.K. Ranjan, T.R. Shiple, V. Singhal, S. Tasiran, H.-Y. Wang, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. HSIS: A BDD-based environment for formal verification. In *31st ACM/IEEE Design Automation Conference*, pages 454–459, June 1994.

[2] J. Bormann, T. Filkorn, J. Lohse, M. Payer, G. Venzl, and P. Warkentin. CVE: An industrial formal verification environment. Internal report, 1994.

[3] J. Bormann, H. Nusser-Wehlan, and G. Venzl. Formal design in an industrial research laboratory: lessons and perspectives. In *Designing Correct Circuits. 2nd IFIP WG10.2/WG10.5 Workshop*, pages 193 – 213, 1992.

[4] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[5] H. Busch and G. Venzl. Proof–aided design of verified hardware. In *28th ACM/IEEE Design Automation Conference*, pages 391 – 196, 1991.

[6] H. Cho, G. D. Hachtel, E. Macii, B. Plessier, and F. Somenzi. Algorithms for approximate FSM traversal. In *30th ACM/IEEE Design Automation Conference*, pages 25 – 30, 1993.

[7] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *Conference Record of the 19th Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 343 – 354, 1992.

[8] W. Damm, B. Josko, and R. Schlör. Specification and verification of VHDL-based hardware designs. In E. Börger, editor, *Specification and validation methods for programming languages and systems.* Oxford University Press, 1994. To appear.

[9] A. Debreil and P. Oddo. Synchronous designs in VHDL. In *Proceedings Euro-DAC'93 with Euro-VHDL'93*, pages 486–491. IEEE Computer Society Press, September 1993.

[10] A. J. Hu and D. L. Dill. Reducing BDD size by exploiting functional dependencies. In *30th ACM/IEEE Design Automation Conference*, pages 266 – 271, 1993.

[11] J. Lohse, J. Bormann, M. Payer, and G. Venzl. VHDL-translation for BDD-based formal verification. Internal report, 1994.

[12] *IEEE Standard VHDL Language Reference Manual.* The Institute of Electrical and Electronical Engineers, Inc., New York, IEEE Std 1076-1987 edition, 1988.

[13] K. L. McMillan and J. Schwalbe. Formal verification of the encore gigamax cache consistency protocol. In *International Symposium on Shared Memory Mulitprocessors*, April 1991.

[14] K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[15] K.L. McMillan. Fitting formal methods into the design cycle. In *31st ACM/IEEE Design Automation Conference*, pages 314–319, June 1994.

[16] P. Stanford and P. Mancuso. *EDIF Electronic Design Interchange Format, Reference Manual for Version 2 0 0.* Electronic Industries Association, Washington D.C., 1989.