Symbolic Verification of Sequential Circuits Synthesized with CALLAS[1, 2]
Extended Summary

Thomas Filkorn Michael Payer[3] Peter Warkentin
Siemens Corporate Research and Development
Otto-Hahn-Ring 6
D-8000 Munich 83, Germany

Abstract

We present a solution to the verification problem of high-level synthesis. The high-level synthesis system CALLAS takes as input an algorithmic specification, in VHDL, and produces as output an EDIF netlist. Both, the specification and the generated netlist can be interpreted as finite state machine descriptions. Then, in this context, the verification problem is reduced to proving the behavioral equivalence of both machines. For this equivalence proof we use the symbolic verifier of the CVE System (CVE = Circuit Verification Environment). Recent improvements of the verifier allowed equivalence proofs of machines with up to 260 binary state variables.

1 Introduction

In this paper, we address the problem of verifying high-level synthesis [17]. By verification we mean to establish the behavioral equivalence of the input design specification and the low level hardware description which is synthesized.

High-level synthesis systems are on their way out of the research labs into industrial use. These systems synthesize complex sequential circuits which cannot be validated with simulation. On the other hand, a typical high-level synthesis systems contains several 10K lines of code and such its correctness cannot be proven.

Our approach exploits advances in symbolic state machine verification [6, 8]. With recent improvements of the system described by Filkorn in [8] we could verify sequential circuits with up to 260 binary state variables. The basic idea of this paper is to interpret the original high-level specification, usually in behavioral VHDL [14], as a finite state machine and to verify with algorithmic methods that the original finite state machine and the synthesized finite state machine are behaviorally equivalent. With this technique we obtain a rigorous proof for the equivalence of specification and implementation; moreover, the equivalence proof is completely automatic and does not require any user interaction.

Corella et al. [5] use a theorem prover for the verification problem in high-level synthe-

[1] This research was partially supported by JESSI AC-8
[2] 6th International Workshop on High-Level Synthesis, Laguna Nigel, CA, U.S.A., 1992
[3] Email: payer@zfe.siemens.de

sis. The authors propose to exploit information generated by the high-level synthesis system HIS [3] to guide the proof process. In particular, the VHDL specification is back annotated with the generated schedule. The circuit, including both the controller and the data path, is then verified against the specification annotated with the schedule. This approach has one drawback: It relies upon the correctness of the generated schedule.

McFarland [16] and Camposano [2] prove the correctness of certain behavior preserving transformations used in high-level synthesis. This idea helps to achieve "correctness by construction" but can neither prove the correctness of the synthesis system nor the correctness of the synthesized circuit.

Claesen et al. [4, 10] present the SFG-Tracing methodology for verification of MOS layouts generated by CATHEDRAL-II. SFG-Tracing relates the algorithmic signals in the specification to their occurrence in space and time in the implementation and can deal with circuits of up to 32000 transistors. The semantics of the implementation is extracted by a switch-level symbolic simulation.

We use the symbolic verification system CVE [9, 19] to verify the SIEMENS synthesis system CALLAS [7]. CALLAS accepts a VHDL subset with a well defined state machine semantics. This state machine must show the same behavior as the synthesized sequential circuit and, hence, can be checked with a state machine verifier.

Fig. 1 gives an overview of the interplay of synthesis, simulation, and verification. The synthesis system CALLAS processes a VHDL specification and generates an EDIF netlist and a VHDL structural description of the synthesized sequential circuit. This way, both the behavioral specification and the synthesized structure can be compared vith a conventional simulator. The same VHDL specification is also transformed into a finite state machine description in Prolog. This transformation preserves the semantics of the original description. The symbolic verification system CVE is then used to verify the behavioral equivalence of both generated descriptions.

Section 2 gives an overview about the Siemens synthesis system CALLAS. In Section 3 we briefly describe the VHDL subset used and explain how a VHDL process is interpreted as a finite state machine. In Section 4 we present some first encouraging results. In Section 5, finally, we discuss further work and directions for future research.

2 CALLAS Overview

CALLAS supports the synthesis of combinational and sequential circuits. For a detailed description of the synthesis system the reader is referred to [7, 15, 18].

CALLAS maps the algorithmic description onto a synchronous, digital circuit which is separated into a data part and a control part. The synthesized circuit behaves as a Moore-type finite state machine; any path from an input port to an output port of the circuit is cut by at least one register.

In a first step global data-flow analysis techniques [1] are used to compile the behavioral description into an initial data-flow graph (DFG) and an initial control graph(CG). The DFG represents a register-transfer level netlist of the data part; the CG represents a state transition graph storing the behavior of the control part of the synthesized circuit.

Designers like to see a relation between the algorithmic description and the synthesized circuit. Thus, CALLAS maps the algorithmic description onto an initial structure without resource constraints. This structure is iteratively improved with various optimizations that may be applied in an arbitrary sequence. The designer can rely on an automated optimization or can guide the synthesis process interactively.

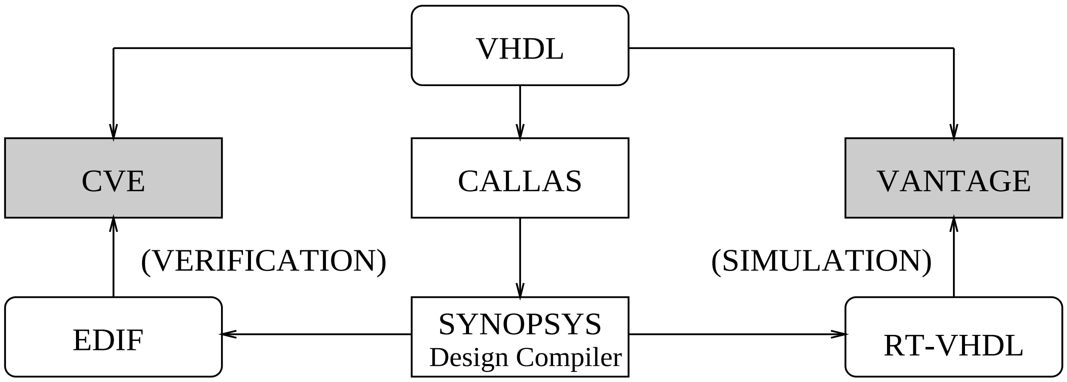

Figure 1: Overview of the Verification System

An initial allocation is done by a simple one-to-one mapping of operations onto functional units. These units are connected in a maximally parallel and maximally chained way. Conflicting assignments to variables of the behavioral description caused by *if, case* or *loop* constructs are resolved by multiplexors. Registers are inserted to cut feedback loops and to control the reading and writing of interface signals.

CALLAS uses enable registers. Reading of an input signal is implemented as loading of an input register. Writing of an output signal enables an output latch. This implementation is consistent with the semantics of the VHDL subset supported by CALLAS [11]. The CALLAS VHDL subset supports the *wait until* construct, which is related to a unique clock signal, to express timing specifications. The *wait until* construct may be used at any location in the program and arbitrarily often. With an algorithmic specification as above the interface timing is completely determined (in terms of clock cycles). This concept allows to check the correctness of the algorithmic specification by simulation or a formal verification. The algorithmic architecture of a design entity to be synthesized may be replaced by the register-transfer architecture after synthesis.

Our subset disables the specification of asynchronous circuits or of Mealy behavior [18]. CALLAS includes various transformations on the DFG and CG to achieve the specified interface timing and to optimize the area and delay of the final circuit. Testability aspects are also considered. The fault coverage of the synthesized circuits is very high (>98 % for most examples).

The essential transformations on the internal control and data-flow graph include timing relevant optimizations such as removal of superfluous edges in the CG (false path analysis), removal of unnecessary data transfers between registers, and control graph reduction [18]. Area and delay optimizations comprise register minimization, operator sharing, multiplexor optimization [20], arithmetic and logic transformations [7], optimization of the data-part/controller interface, flattening of complex functional units, and partitioning and logic minimization.

3 State Machine Semantics of the CALLAS VHDL Subset

CALLAS uses a subset of VHDL [14] with a well defined semantics as specification language. For the purpose of this paper we describe a more restrictive subset.

A CALLAS-specification is written in a subset of behavioral VHDL where the clock and reset signals are considered as special input signals. A specification must consist of a single process whose statements are the usual control constructs such as if-then-else, while, etc., and constructs for expressions and assignment. We distinguish between *variables* and *signals*. A signal can only appear in the process body if it has been defined in the entity description. The data types are bit fields of arbitrary but finite length with the natural semantics of operations on these data types.

Explicit `wait` statements are not allowed; instead, they have to be hidden by the use of a special predefined `cycl` procedure (see Fig. 2) which deals with the input signals *reset* and *clk*. This `cycl` procedure is used to define the timing behavior of the specification and makes the process sensitive to the *clk* signal. The `cycl` procedure guarantees that a signal of the specification can only change at rising clock edges.

This "cycl" procedure is used to define the timing behavior of the specification in terms of clock cycles. The process is sensitive only to the "clk" signal. The "cycl" procedure guarantees that a signal of the specification can only change at rising clock edges. (In fact, "cycl" is a VHDL procedure which deals with the clock and the synchronous reset signals of the circuit and hides the "wait until (clk = '1')" statement —which is sensitive to the clock— from the user.)

The process body has to be encapsulated in the structure shown in Fig. 3. This construction is mandatory and is used for the correct handling of the synchronous reset.

Fig. 4 shows a specification of the counter benchmark [12]. Observe, that the sequence of operations in a VHDL specification resumes in the beginning after having reached the end.

When simulated, such a specification behaves as a Moore-type finite state machine; Figure 5 symbolizes the input/output behavior of such an FSM.

```
cycl(reset, clk, 1);
if (reset = '1') then
else
  outest_loop: loop

  -- <process body>

    exit outest_loop when (reset = '1');

  end loop outest_loop;
end if ;
```

Figure 3: Gross Structure of a CALLAS Process Body

The inputs of the finite state machine are given by the input signals[1] of the entity and the outputs by the output signals, respectively.

The state of the finite state machine is determined by the position of the statement that the simulator will process next and by the values of all variables and output signals[2] at that position. The next state is determined by the position of the next statement to be processed and by the new values of the affected variables and output signals. The value of a variable or output signal (of the entity) is determined by the statement to be executed in the current position. The state set contains one additional component which "stores" the previous value of the input signal "clk". This mechanism is needed to react on the rising edge of the clock. Finally, the output (of the finite state machine) is given by those state components that represent the output signals. In the sequel we describe how this finite state machine is actually generated.

To make the transformation process more explicit, we first transform the control constructs and the implicit loop of the process into an equivalent program in some algorithmic

[1] To be more precise, when we speak of "the input signals" then we mean all possible combinations of values of the input signals.

[2] Here we mean the output signals listed in the entity description.

```
procedure cycl (signal reset, clk : in bit; n : in natural) is
begin
  for i in 1 to n loop
          wait until (clk = '1'); exit when (reset = '1'); end loop;
end;
```

Figure 2: The Predefined `cycl` Procedure

language with *goto*, *unconditional goto* and *labels* for the targets of the goto statements. Secondly, we label each unlabeled statement (see Fig. 4).

Finally, we have to take care of the `WAIT UNTIL (clk = '1');` and `EXIT WHEN (reset = '1')` constructs of the `cycl` procedure. The VHDL process is sensitive to the rising edge of the clock signal, i.e., signals may change at these events; moreover, processing is halted until this event occurs. We simply introduce one additional internal variable *last_clk* and model a call to the `cycl` procedure as shown in Fig. 6.

The signal assignments, denoted by "<=", of the transformed program do not yet have the semantics of the original specification: The output signals change their value with every state transition. We solve this problem as follows:

We interpret the signal assignments just like variable assignments and construct the corresponding Moore-type machine in the obvious way.

Now observe that input signals can only change at a clock transition from '0' to '1' and then remain stable until the next "rising edge" of the clock signal; i.e., at some point of a symbolic simulation of this finite state machine the state does not change anymore. These states become the states of the final finite state machine. The transition function of the final finite state machine is then determined by the composition of the transition functions of the original machine. The final machine can be computed by a fixpoint iteration.

We have described the construction of the finite state machine rather informally; a detailed formulation is left to a subsequent paper. A similar approach has been described by Hou et al. in [13]. The "explicit synchronous model" is very close to our micro machine; the "implicit synchronous model" corresponds to our macro machine.

Observe, that the output signals always change at the right time and at most once per cycle, which conforms to the VHDL semantics as defined in the VHDL language reference manual [14].

4 Results

To show the feasibility of our approach, we present some real-time measures for the verification process in Table 2. The experiments were made on a Siemens workstation WS 30-450, which has approximately twice the speed of a SUN 3/60 and on a SPARC2, respectively. The verifier and the transformation process make extensive use of BDDs. A memory limit of 12 MB was sufficient to store the BDDs.

We have formulated several behavioral (algorithmic-level) benchmarks, namely PREFETCH, Traffic Light Controller (TLC), Greatest Common Divisor (GCD), and COUNT in the CALLAS VHDL-subset. We simulated the designs with the VANTAGE simulator and synthesized with CALLAS. For logic optimization and mapping onto the Siemens library Advancell_D we used the SYNOPSYS Design Compiler.

Table 1 summarizes the complexity of the behavioral specifications in terms of lines of code (loc), word length of the data part (wl),

```
library callas;
use callas.callas.all ;

entity count is
 port (
   reset     : in bit;                        -- declare(input, [reset], bit),
   clk       : in bit;                        -- declare(input, [clk], bit),
   countin   : in bit_vector (3 downto 0);    -- declare(input, [countin], bit_vector(3 downto 0)),
   up        : in bit;                        -- declare(input, [up], bit),
   count     : in bit;                        -- declare(input, [count], bit),
   countout  : out bit_vector (3 downto 0) );-- declare(output, [countout], bit_vector(3 downto 0)),
end count;

architecture be of count is
begin
p: process
    variable   i      : bit_vector (3 downto 0); -- declare(variable, [i], bit_vector(3 downto 0)),
    constant   cnull4 : bit_vector := "0000";    -- declare(constant, [cnull4], bit_vector, 0000),
    constant   cone4  : bit_vector := "0001";    -- declare(constant, [cone4], bit_vector, 0001),
 begin
  i := cnull4;                             -- label(21), i := cnull4,
  countout <= cnull4;                      -- label(22), countout <= cnull4,
  cycl(reset, clk, 1);                     -- label(23), iteGoto((clk = 1) and (last_clk \= 1), 24, 23),
  if (reset = '1') then                    -- label(24), iteGoto(reset = 1, 39, 25),
  else                                     -- label(25), nop,
    outest_loop: loop
      if (count = '0') then                -- label(27), iteGoto(count = 0, 28, 29),
        i := countin;                      -- label(28), i := countin,
      elsif (up = '1') then                -- label(29), iteGoto(up = 1, 30, 32),
        i := i + cone4;                    -- label(30), i := i + cone4,
      else                                 -- label(31), goto(33),
        i := i - cone4;                    -- label(32), i := i - cone4,
      end if;                              -- label(33), nop,
      countout <= i;                       -- label(34), countout <= i,
      cycl(reset, clk, 1);                 -- label(35), iteGoto((clk = 1) and (last_clk \= 1), 36, 35),
      exit outest_loop when(reset = '1'); -- label(36), iteGoto(reset = 1, 38, 37),
    end loop outest_loop;                  -- label(37), goto(27),
  end if;                                  -- label(38), nop,
end process;                               -- label(39), goto(21),
end be;                                    -- label(99), end
```

Figure 4: Correspondance of Specification and Intermediate Form

cycle			1			2			3			...
clock			↑			↑			↑			
	-		x			x			x			
in	-		x			x			x			
	-		x			x			x			
	-	x			x			x			x	
out	-	x			x			x			x	
	-	x			x			x			x	
	-	x	x	x	x	x	x	x	x	x	x	
others	-	x	x	x	x	x	x	x	x	x	x	
	-	x	x	x	x	x	x	x	x	x	x	

Figure 5: I/O-Behavior of a VHDL specification.

```
label(x), iteGoto((clk = 1) and (last_clk \= 1), y, x),
label(y), iteGoto(reset = 1, z, u),
label(z), goto <restart>
label(u), goto <continue>
```

Figure 6: Modelling of the `cycl` Procedure

number of variables and signals (var), number of loops (loop), number of if/case constructs (if/case), and the maximum nesting. The three columns labeled "FG Format" show the number of nodes, variables, and paths in the CALLAS internal representation, the so-called flow graph, are listed in Table 1. The CALLAS VHDL frontend inserts several additional variables in order to resolve expressions and to meet the semantics of reading and writing VHDL interface signals [11]. The two rightmost columns "FSM1" and "FSM2" show the (uncritical) time needed to construct the CVE-internal BDD representation of the VHDL specification; here "FSM1" lists the time for the construction of the first finite state machine and "FSM2" lists the time for the fixpoint iteration, respectively.

Table 2 lists the size of the synthesized designs (Siemens library Advancell_D) and the verification runtime for different structural optimizations.

In our VHDL formulation of the PREFETCH benchmark the variables and signals have a total of 260 bits. To generate the first symbolic finite state machine which does not yet respect the VHDL semantics takes 9s. The fixpoint iteration which results in the final finite state machines needs 53s (Table 1). The synthesized circuit contains 163 D-FlipFlops and between 216 and 226 combinational gates (2-input multiplexors, inverters, 2- and 3-input nands, etc., not shown in Table 2.) The EDIF netlist is transformed into the internal CVE representation in approx. 55s. Finally, the (positive) behavioral equivalence check takes between 467s and 552s (Table 2). Thus the whole proof runs in about 9.5 minutes.

In particular, from the TLC entries in Table 2 it can be seen that the verification process is slower for more optimized designs; of major influence here is the number of registers.

The verification of the 4-bit GCD-design takes about 2200s; this example exhibits the "information content" of the output signals as another important factor for the runtime; the

	Alg. VHDL specification						FG format			cpu (s)	
Design	loc	wl	var	loop	if/case	nest	node	var	path	FSM1	FSM2
COUNT	38	4	7	1	3	3	25	12	6	0.7	1.5
PREFETCH	52	32	12	2	2	2	44	20	5	9.0	53.0
TLC	93	2	7	7	5	2	93	28	52	2.2	14.0
GCD	49	4	8	3	2	3	35	14	5	1.3	5.8

Table 1: Complexity of Benchmark Designs. (Lines of VHDL code, word length, number of variables, loops, if/case nodes; maximal nesting depth, number of FG nodes, FG variables, paths through the FG; time for construction of the first finite state machine, time for fixpoint iteration (SPARC2).)

Design	opt	Final Design			Verification		
					cpu (s)		Mem.
		reg	cells	grids	E2BDD	Verif.	MB
COUNT[1]	min	11	38	246	6	9	0.4
	noopt	12	41	266	6	9	0.4
	optm	12	41	266	6	9	0.4
	optom	12	42	272	6	9	0.4
	optrom	12	42	272	6	9	0.4
	optromF	12	41	267	6	9	0.4
PREFETCH[2]	min	163	379	2904	55	552	12
	noopt	163	379	2903	55	552	12
	optm	163	474	3429	56	462	12
	optom	163	473	3426	56	481	12
	optrom	163	476	3430	56	481	12
	optromF	163	389	2913	55	467	12
TLC[1]	min	33	191	1012	16	454	8
	noopt	33	191	1012	16	451	8
	optm	33	198	1021	16	451	8
	optom	33	190	993	16	554	8
	optrom	16	139	666	15	686	8
	optromF	16	131	615	15	796	8
GCD[1]	optrom	15	110	549	13	2184	12

[1]SYNOPSYS Version 2.0a [2]SYNOPSYS Version 2.2a

Table 2: Size of Synthesized Designs and Verification Runtime. (Number of register bits, number of cells, and cell area of the final design after logic optimization and technology mapping (Siemens library Advancell_D); cpu time (Siemens WS30-540/SPARC2) for EDIF to BDD translation and verification (SPARC2) in seconds; memory usage for translation and verification in mbytes.)

more an output signal depends on the internal states of the machine the faster is the verification. In our GCD-design the output does not show any changes until the final result is computed.

5 Conclusions

In this paper we have described a new solution for the verification problem of high-level synthesis. We model the high-level specification and the CALLAS synthesis result as finite state machines and then use the algorithmic verifier CVE to prove the behavioral equivalence of both machines. Our solution is conceptually very simple; it has become computationally feasible with recent advantages in finite state machine verification. Currently we are able to verify sequential circuits with up to 260 binary state variables. We expect that this number will grow considerably over the next years.

Among other examples, we have presented the verification of the PREFETCH benchmark as an example. Here, the complete equivalence check took about 9.5 minutes of cpu time.

In the near future we plan to extend the verifiable CALLAS VHDL subset. Currently we do not exploit that CALLAS synthesizes a separate control finite state machine and data path. Additional enhancements to our solution can be achieved if we use such information as the mapping of variables onto data path registers. This information is provided by CALLAS but currently not used.

To call for the state machine equivalence of both descriptions is too restrictive: A high-level specification should be interpreted as a generic description of a whole set of finite state machines. Then the verification problem of high-level synthesis can be relaxed to an equivalence proof of the synthesis result and *one* member of the generic set.

Acknowledgments

Many colleagues at Siemens Corporate Research were involved in the CALLAS and CVE systems. We thank Wolgang Ecker and Steffen Rumler for the helpful discussions of the VHDL semantics.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[2] R. Camposano. Behavior-preserving transformations in high-level synthesis. In M. Leeser and G. Browne, editors, *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, pages 106–128, New York, 1989. Springer-Verlag. LNCS, Vol. 408.

[3] R. Camposano, R.A. Bergamaschi, C. Haynes, M. Payer, and S. Wu. *High-Level VLSI Synthesis*, chapter The IBM High-Level Synthesis System, pages 79–104. Kluwer Academic Publishers, Norwell, MA, 1991. R. Camposano and W. Wolf, editors.

[4] L. Claesen, F. Proesmans, E. Verlind, and H. De Man. SFG-tracing, a methodology for the automatic verification of MOS transistor level implementations from high level behavioral specifications. In P. A. Subrahmanyam, editor, *Proc. ACM-SIGDA International Workshop on Formal Methods in VLSI Design*, 1991.

[5] F. Corella, R. Camposano, R. Bergamaschi, and M. Payer. Verification of synchronous circuits obtained from algorithmic specifications. In D. Borrione and R. Waxman, editors, *CHDL 91 - Computer Hardware Description Languages and their Applications*, pages 209–227, Marseille, France, April 1991.

[6] O. Coudert, C. Berthet, and J.C. Madre. Verification of synchronous sequential machines based on symbolic execution. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, Grenoble, June 1989. LNCS Vol. 407.

[7] P. Duzy, H. Krämer, M. Neher, M. Pilsl, W. Rosenstiel, and T. Wecker. CALLAS - conversion of algorithms to library adaptable structures. In *VLSI'89*, pages 197–208, Munich, Germany, August 1989. Elsevier.

[8] Th. Filkorn. A method for symbolic verification of synchronous circuits. In D. Borrione and R. Waxman, editors, *CHDL 91 - Computer Hardware Description Languages and their Application*, pages 229–239, Marseille, France, April 1991.

[9] Th. Filkorn, R. Schmid, E. Tidén, and P. Warkentin. Experiences from a large industrial circuit design application. In *Proc. of the 1991 International Logic Programming Symposium*, San Diego, October 1991.

[10] M. Genoe, L. Claesen, E. Verlind, F. Proesmans, and H. De Man. Illustration of the SFG-tracing multi-level behavioral verification methodology, by the correctness proof of a high to low level synthesis application in CATHEDRAL-II. In *Proc. IEEE International Conference on Computer Design: VLSI in Computers & Processors, ICCD-91, Cambridge MA*, pages 338–341, October 1991.

[11] W. Glunz and G. Umbreit. VHDL for high-level synthesis of digital systems. In *Proc. of 1st European Conference on VHDL Methods*, 1990.

[12] Benchmarks for the 5th International Workshop on High-Level Synthesis. Available through electronic mail at HLSW@decwrl.dec.com, 1991.

[13] P.-P. Hou, R.M. Owens, and M.J. Irwin. High-level specification and synthesis of sequential logic modules. In D. Borrione and R. Waxman, editors, *CHDL 91 - Computer Hardware Description Languages and their Application*, pages 131–142, Marseille, France, April 1991.

[14] The Institute of Electrical and Electronical Engineers, Inc., New York. *Standard VHDL Language Reference Manual*, 1988.

[15] M. Koster, M. Geiger, and P. Duzy. ASIC design using the high-level synthesis system CALLAS: A case study. In *Proc. ICCD'90*, pages 141–146, Cambridge, Ma., September 1990.

[16] M.C. McFarland. A practical application of verification to high level synthesis. In *Proc. Workshop on Formal Methods in VLSI Design*, 1991.

[17] M.C. McFarland, A.C. Parker, and R. Camposano. The high-level synthesis of digital systems. *Proceedings of the IEEE*, 78(2):301–318, February 1990.

[18] A. Stoll and P. Duzy. High-level synthesis from VHDL with exact timing constraints. In *Proc. 29th DAC*, 1992.

[19] Erik Tidén and Richard Schmid. Verifying ASICs by symbolic simulation. In *Proc. EUROASIC 90*, pages 461–473, 1990.

[20] N. Wehn, J. Biesenack, and M. Pilsl. A new approach to multiplexor minimization in the CALLAS synthesis environment. In A. Halaas and P.B. Denyer, editors, *Proc. VLSI91*, pages 203–213, 1991.