VHDL-Translation for BDD-Based Formal Verification*

Jörg Lohse, Jörg Bormann, Michael Payer, and Gerd Venzl
Siemens Corporate R&D
D-81730 Munich
Germany
e-mail: `Joerg.Lohse@zfe.siemens.de`

Abstract

We describe a novel approach to translate a reasonably large subset of VHDL into BDD's. The VDHL subset was chosen to include the commonly used synthesis subsets but is strictly based on the simulation semantics required by [2]. Our results significantly improve on the previously reported ones [5]. We also investigate the inherent complexity of dealing with the VHDL semantics as opposed to translating netlists into BDD-based FSM's.

1 Introduction

Formal verification techniques have proven to be powerful in detecting errors in early design phases when fixes are still inexpensive. Equivalence checking of sequential designs as well as symbolic model checking are techniques used in formal verification. For these techniques, the designs are represented as finite state machines using binary decision diagrams [1, 3] for the state transition and output functions.

An increasing number of digital hardware designs are currently written in VHDL and compiled into netlists using synthesis tools. Designers in industry usually work in tight schedules and thus do not have the time to translate their circuits from VHDL into another language (e. g. SMV [6]) needed for formal verification. Moreover, such a translation task would be error-prone if performed manually. We solve this problem by providing a translator from VHDL into BDD-based finite state machines (FSM's). This way, the designer can feed his/her designs right as they are into formal verification tools. The verification tool we translate the VHDL for also gives feedback in terms of VHDL by providing counter examples as VHDL test-benches which can be exercised right with any VHDL-simulator [10].

Formal verification gives results that are equivalent to exhaustive simulation. Therefore, our translator strictly relies on the VHDL semantics as specified in [2] in terms of simulation. In other words, it does *not* — in contrast to synthesis tools — treat language items like `wait`-statements or sensitivity lists differently from the VHDL semantics. Also, unlike [7] there is no requirement to have a synchronous design and to mark clock pins with attributes.

The finite state machine extraction of VHDL processes has been shown for synthesis [4]. An early translator of a VHDL subset into BDD's was presented by Filkorn et. al. [5]. This translator represen-

*Partially funded by the European Community, grant JESSI AC-8.

```
ENTITY delay IS
  PORT(input: bit; output: OUT bit);
END;

ARCHITECTURE simple OF delay IS
BEGIN
  output <= TRANSPORT input AFTER 1 sec;
END simple;
```

Figure 1: Delay-Element in VHDL.

ted the control flow within processes as state transitions in BDD's and computed their fixed points with BDD's. This fixed point computation of processes only worked for processes with a small number of statements. In this paper, we describe a different approach that overcomes this limitation by treating the control flow separately from the data flow: The control flow is represented in graphs with branching conditions and state transitions represented with BDD's. It also does not require state encoding of sequential statements and fixed point computation for processes.

2 VHDL Subset Representable in FSM's

In this section the subset of VHDL used for formal verification is described. The main restriction is introduced by the finite (i. e. fixed at translation time) number of states. In VHDL, data types like access types (pointers) and files allow to model data of virtually unlimited size. VHDL designs using these data types principally cannot be translated into FSM's.

Delay is another language item that cannot be fully represented in a finite state machine: As an example of this, consider the following VHDL model of a delay element (Figure 1).

At any given time, the value at the output of this delay element is its input's value of 1 second ago. Represented as an FSM, this delay element must store all input values that can occur in one second. According to [2], VHDL implementations must provide for the worst case in which the input can toggle every femtosecond, 10^{15} times a second. Since it is unreasonable to represent 10^{15} possible projected values for the output, we conclude that VHDL's timing cannot be reasonably represented as finite state machines. Any other decision, e. g. to restrict the time of input changes as done in [9], would mean formal verification not to be equivalent to exhaustive simulation anymore.

2.1 Processes and Signals

VHDL models behavior in terms of processes that communicate via signals. At any given time, each process is in one of two possible states: either running or suspended. When all processes are suspended, the signal's values are updated and the simulation time advances. A process is suspended until some signal changes value to which the process is sensitive.

Within processes behavior is described by sequential statements. These sequential statements include function and procedure calls, assignments to variables and assignments to signals. Assignments to signals do not cause the signal's value to be immediately modified. They just modify a signal driver which is locally maintained for each process that writes to a signal. The values of the drivers are assigned to the signals when all processes are suspended. If there is more than one process driving a signal, the values of all drivers will be passed to a resolution function and the function's result will be assigned to the signal.

```
ENTITY counter IS
 GENERIC ( capacity : integer := 256 );
 PORT    ( reset    : IN bit;
           clk      : IN bit;
           count    : BUFFER integer RANGE 0 TO capacity-1 );
END counter;

ARCHITECTURE behavior OF counter IS
BEGIN
  p: PROCESS
     BEGIN
       IF reset = '1'
       THEN
         count <= 0;
       ELSE
         count <= (count + 1) MOD capacity;
       END IF;
       WAIT UNTIL clk'event AND clk = '1';
     END PROCESS;
END behavior;
```

Figure 2: 8-Bit Counter in VHDL.

3 VHDL Translation

The translation of VHDL into a BDD-based FSM is performed in two phases. We assume that VHDL has been parsed, any overloading of subprograms, operators and enumeration literals resolved as well as all VHDL-expressions attributed with types.

In Phase 1 declarations are annotated with BDD's and processes are compiled into control graphs. The control graphs include state transitions represented as BDD's. In Phase 2 the control graphs are compiled into an FSM and fixed-point computations and optimizations are performed on this FSM.

3.1 Translation into BDD's and Control Graphs

The declarations are processed by annotating them with BDD's: For signals (including the ones introduced by attributes such as `stable` and `delayed`) and variables, BDD variables (index in [3]) representing their current values are introduced. Constants are annotated by their value represented as BDD's.

Processes are compiled into control graphs. A node in the control graph describes control flow as a Boolean condition (represented as a BDD) and references to two possible successor nodes. The node also includes a list of BDD-based state transitions with one entry for each state transition to variables and signal drivers. These state transitions are performed at the given point of control. Signal drivers and variables that are not changed at that point are not included in the list of transitions.

In the control graph, all nodes that represent `wait`-statements are marked. To determine the initial state of the variables and signal drivers as well as the wait statement to start with, the control graph is simulated. The signal drivers, signals and variables are initialized with their initial values and updated during simulation of the control graph until the first `wait`-statement is reached. This `wait`-statement is marked as the `wait`-statement at the initial state and the simulated values of the variables and signal drivers are entered as part of the FSM's initial state. This implements the initialization phase in LRM Section

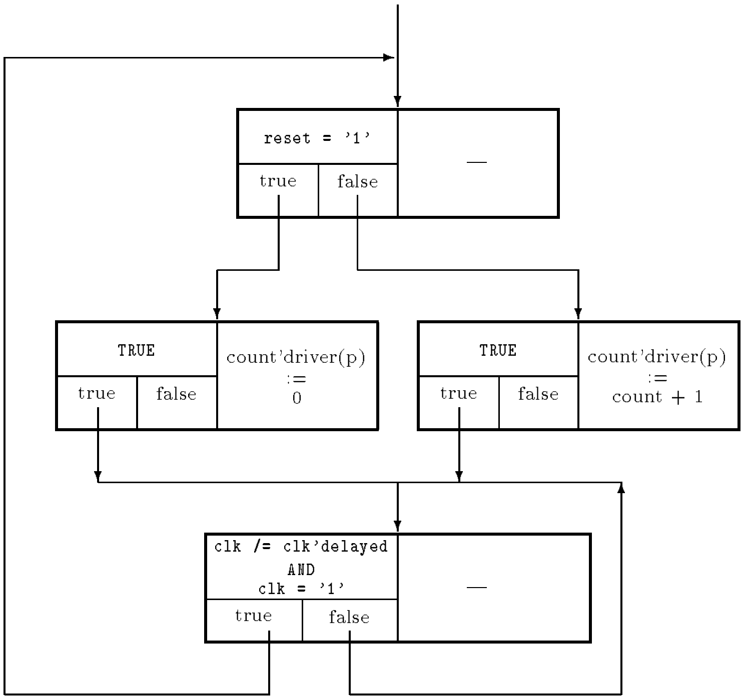

Figure 3: Control Graph of 8-Bit Counter.

12.6.3 [2] and ensures that the FSM's initial state corresponds to the VHDL model at simulation time 0.

Refer to Figure 2 for the VHDL design of an 8-bit counter. The counter has a `reset` pin, a `clock` input and an output port of mode `BUFFER`. The mode `BUFFER` allows the output to be read in the VHDL description. Figure 3 shows the control graph for this counter. Each control graph node's condition is `TRUE` when no branch is intended. The left hand side of each node shows the condition and targets of branches; the right hand side consists of a list of state transitions. Note, that both the condition and the transitions are represented as BDD's. In the control graph, the expression `clk'event` has been translated into `clk /= clk'delayed` which shows that the effective value of `clk` and its value delayed by a delta step are compared to check for an event. `clk'delayed` is a signal and corresponds to a state variable in the FSM. The transition `count'driver(p) := count + 1`[1] shows that the FSM state variable `count'driver(p)`, process `p`'s driver of `count`, is modified by the transition, *not* the effective value of `count` that is read at the right hand side of the assignment. The *effective* and *driving* values of signals are thus implemented according to Section 12.6 in the VHDL LRM [2].

These control graphs are subject to several transformations. Each of the transformations is a reduction rule yielding a smaller graph. For every node in the graph an applicable rule is applied until no further reduction is achieved.
The following reduction rules are applied:

Branch Optimization If a control graph node's condition is constant true (false), remove pointer to false (true) successor nodes.

Unreachable Node Removal
If a control graph node is no suc-

[1]The modulo-2^n-operation in Figure 2 just truncates to n bits.

cessor of any other node, it will be removed from the graph.

No-Op Removal If a control graph node's condition is constant and its set of state transitions is empty, it will be removed from the graph.

Basic Block Compaction If two nodes are part of a basic block: Node b is successor of node a and a has no other successors than b and b is not successor of any other node than a (Figure 4), then a and b will be merged into one node ab: The state transitions of a are substituted in the state transitions of b.

Branch Over Node
If exactly one node is only executed on a condition and it's successor is the same node that is executed when the condition is not met (Figure 4), then the condition will be merged into the transitions: The transitions take effect when the condition is met and otherwise do not modify the state.

2-Way Branch Over Nodes If two nodes are successors of a condition and both nodes have the same successor (Figure 4), then the condition and both nodes are combined to one transition: Dependent on the condition, either transitions take effect.

In the end the control graph usually consists of one node for each `wait`-statement. Only in cases where loops take a data-dependent number of iterations, there are more nodes left. In the end of the control graph transformations, all variables that do not occur in the final transitions are eliminated. These transformations are also applied to the control graphs resulting from calls to subprograms (functions, user-defined operators, procedures). This way, the control graphs of processes are kept small and the variables local to subprograms are eliminated as soon as possible.

The control graph in Figure 3 is reduced as follows: In the first step, the three

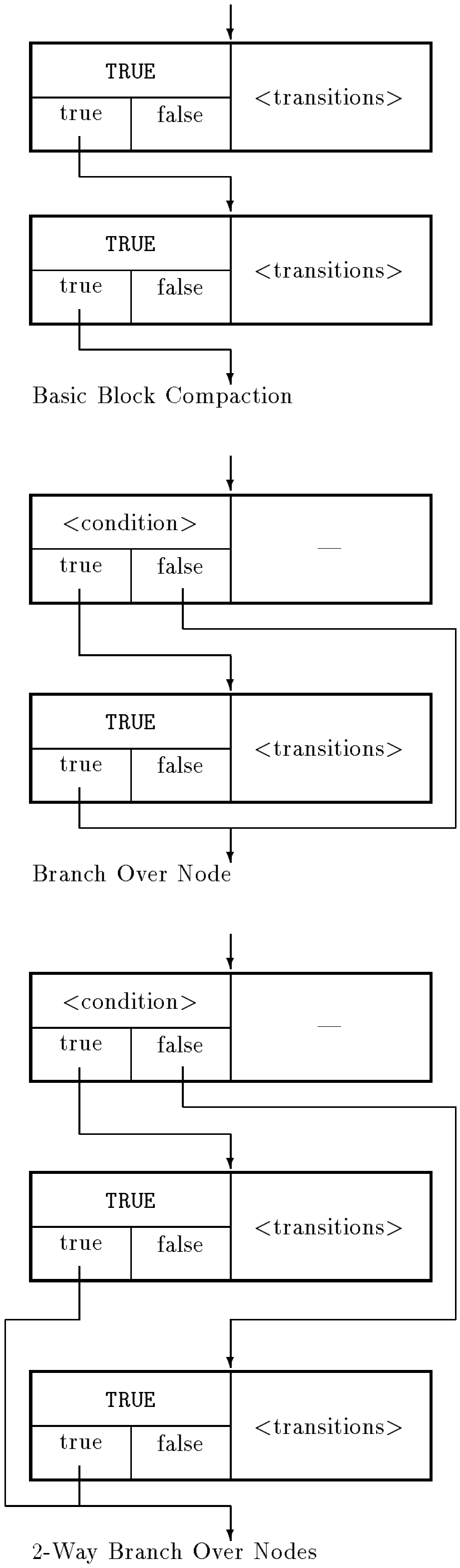

Figure 4: Control Graph Reduction Rules.

Design			VHDL	cpu time	`vhdl2fsm`	FSM		
name	I	O	LOC	in [5]	cpu time	S	$\mid \delta \mid$	$\mid \lambda \mid$
adc	3	9	55	7.0	0.2	15	55	47
count	8	4	38	3.0	0.2	5	52	52
prefetch	68	96	52	62.0	9.7	129	762	665
tlc	4	12	93	16.0	0.2	19	208	112

Table 1: Comparison with Results Published in [5].

nodes in the two upper rows are combined to one node by the 2-Way-Branch-Over-Nodes rule. In the next step, the resultant two nodes are combined to one node by the Branch-Over-Node rule. After this, there is only one node left that represents the entire process.

3.2 Translation of Control-Graphs into FSM's

In the second phase of the VHDL compilation the control graphs are combined to an FSM consisting of inputs, states, next-state functions and output functions. The state variables in the FSM consist of signals outside of processes and of the state variables belonging to processes. State variables belonging to processes are signal drivers, variables and signals local to processes, and the program counter whose values are the nodes in the control graph. Each process exports the process state, a property with values *suspended* or *running* dependent on whether the program counter points to the control graph node of a `wait`-statement or to another statement's control graph node. The FSM is constructed by combining all states and state-transition functions from the processes and adding primary inputs and outputs as well as other signals. State variables are introduced for the primary inputs and outputs. The inputs' states variables are read within the FSM and these state variables are updated from the primary inputs when all processes are suspended. In the same way, the outputs' state variables are updated from their signal drivers when all processes are suspended.

In VHDL, processes communicate via signals. The processes modify signal drivers and at the end of each delta cycle, the signal drivers' values are propagated to signals and may cause the same or another process to resume execution in the next delta cycle. No simulation time passes when processes are executed in delta time steps. For FSM's in formal verification, we are interested in values changing in simulation time, not for signal changes in delta time steps. We compute the final FSM using the current state transition functions that describe the behavior in delta time steps: The fixed point of the state transition functions is the result of process reinvocations. The fixed point is computed and represents state transitions in terms of simulation time. This process is called *macro-machine generation* [5]. The initial state is recomputed by evaluating the macro-machine's state transition function for the initial state and the inputs' initial values. The primary outputs are still delayed by a delta time step since there is a signal assignment to their state variables involved. In the output functions the next-state functions are substituted for the state variables in order to remove this delta delay.

In the macro machine, there are state variables that no output and no other variable's transition function depend on. Such state variables are removed from the FSM. Typically, all signal drivers' state variables and all primary output's state variables are removed by this optimization. State identification — the elimination of functionally equivalent state variables, described in [10] — is also applied to the FSM.

It should be noted that the resultant

Design			VHDL	EDIF	`edif2fsm`	`vhdl2fsm`	FSM		
name	I	O	LOC	cells	cpu time	cpu time	S	$\mid \delta \mid$	$\mid \lambda \mid$
strobe	6	18	161	127	0.9	2.0	48	178	63
busint	123	26	511	196	5.4	12.2	37	277	446
div	18	4	737	75	1.0	28.1	27	632	59
channel	8	22	1278	234	4.6	430.9	94	854	92

Table 2: VHDL vs. EDIF for Industrial VHDL Designs.

FSM exhibits exactly the same behavior as the VHDL simulation of the model. Even simultaneous changes of clock and data inputs — for which synthesized designs usually differ from their specifications – have the same effect in the FSM and in VHDL simulations. This is an important point since the FSM extraction is one step in the formal verification process whose results have to be equivalent to exhaustive simulation.

4 Results

The **`vhdl2fsm`** translator has been implemented using a commercial VHDL analyzer to parse VHDL and produce a parse tree annotated with type information. The FSM translator accesses this parse tree and translates entity/architecture pairs into FSM's. Structural VHDL is supported by compiling the instantiated entity and architecture with the given assignments to generics and then adding them to the FSM currently under construction. The variable ordering of the BDD data structure is adjusted by dynamic variable reordering [8]. Reordering is invoked whenever the number of BDD-nodes has doubled since the most recent reordering. The VHDL **`generics`**, also used in Figure 2, as well as **`generate`**-statements [2] provide for *scalability* which is crucial for BDD-based formal verification [6].

See Table 1 for the runtime required for the designs benchmarked in [5]. I the number of inputs, O the number of outputs, LOC is the number of lines of VHDL code excluding any code from packages. The cpu time published in [5] and the cpu time we measured are both reported in seconds on a Sparc2. For the FSM generated by **`vhdl2fsm`**, S is the number of state variables, $\mid \delta \mid$ and $\mid \lambda \mid$ the number of BDD-nodes for the state transition and output functions, respectively. Table 2 reports some results for industrial designs and compares the VHDL translation to the translation of the EDIF resulting from synthesizing the VHDL with a commercial tool. The number of EDIF cells is reported as well as the runtimes of both **`edif2fsm`** and **`vhdl2fsm`**. While the translation times of the VHDL models grow overlinearly with the size of the VHDL descriptions, the times to translate EDIF-netlists are about proportional to the number of cells. This comparison shows the price that is to pay for dealing with VHDL's semantics.

5 Conclusion and Future Work

Our approach allows to translate VHDL-designs that are common in industry into BDD-based finite state machines. Over [5] the size of VHDL that can be translated has been improved by about two orders of magnitude. The reason for this improvement is that the fixed point of the state transition function is not applied to the sequential statements of processes anymore but part of it is done by the control-graph modifications. Nevertheless, a fixed point computation is still the bottleneck of the VHDL translation. For the larger designs in Table 2, more than 90% of the CPU time is spent in fixed point computation which is still necessary to determine the results of process re-invocations.

In the future, a growing need to support timing can be anticipated since real-time extensions to symbolic model checking are subject of growing attention.

The similarity of our BDD-based control graphs to the data structures used in high-level synthesis is striking. It may be interesting to use our data structures (BDD's for data + graphs for control) or even the generated FSM's as basis for a synthesis tool.

References

[1] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation", *IEEE Transactions on Computers*, Volume C-35, Number 8, pp. 677–691, August 1986.

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

[3] K. S. Brace, R. L. Rudell and R. E. Bryant, "Efficient Implementation of a BDD Package", *27th Design Automation Conference*, pp. 40–45, June 1990.

[4] P. P. Hou, R. W. Owens, M. J. Irwin, "High-Level Specification and Synthesis of Sequential Logic Modules", *Proc. CHDL'91 - Computer Hardware Description Languages and their Application*, pp. 131–142, April 1991.

[5] T. Filkorn, M. Payer, P. Warkentin, "Symbolic Verification of High-Level Synthesis Results from CALLAS", *Proc. 6th International Workshop on High-Level Synthesis*, pp. 344–353, IEEE, November 1992.

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

[7] A. Debreil, P. Oddo, "Synchronous Designs in VHDL", *Proceedings Euro-DAC'93 with Euro-VHDL'93*, pp. 486–491, IEEE Computer Society Press, September 1993.

[8] R. L. Rudell, "Dynamic Variable Ordering for Ordered Binary Decision Diagrams", *Proc. IEEE ICCAD*, pp. 42–47, November 1993.

[9] G. Döhmen, "Petri Nets as Intermediate Representation between VHDL and Symbolic Transition Systems", *Proceedings Euro-DAC'94 with Euro-VHDL'94*, pp. 572–577, IEEE Computer Society Press, September 1994.

[10] M. Payer, J. Bormann, T. Filkorn, J. Lohse, G. Venzl, P. Warkentin, "CVE: An Industrial Formal Verification Environment", *Internal Report, 1994*.