

An Approach to Behavioral Synthesis from a Formal Model of VHDL

Rajesh K. Bawa[†] Pierre Guerrier[‡] Ludovic Jacomme[†] Pirouz Bazargan Sabet[†]

[†]Laboratoire MASI,
Université Pierre et Marie Curie,
4 place Jussieu,
75252 Paris Cedex 05, FRANCE

[‡]Laboratoire d'Informatique de l'X,
Ecole Polytechnique,
Route de Saclay,
91128 Palaiseau Cedex, FRANCE

e-mail: {Rajesh.Bawa, Pierre.Guerrier, Ludovic.Jacomme, Pirouz.Bazargan}@masi.ibp.fr

Abstract

In this paper we present a method for behavioral synthesis of hardware systems described in VHDL. Due to the simulation-based semantics of VHDL, most of the existing tools restrict the subset in order to facilitate the task of synthesis. The approach proposed here is based on an internal formal model in terms of Interpreted and Timed Petri Nets (ITPN). A set of equations can be extracted which allows us to perform the recognition of memorizing elements, a key step in the synthesis of behavioral VHDL, without imposing any cumbersome description style.

1. Introduction

Although VHDL [1] was initially designed for simulation purposes, it is more and more used as a specification language for behavioral synthesis. Because of an underlying event-driven simulation mechanism, it is difficult to synthesize directly the VHDL descriptions. Most of the existing tools, and notably the Synopsys VHDL Compiler [2], restrict the accepted VHDL subset to overcome the simulation semantics. These style restrictions allow to syntactically map VHDL constructs to the corresponding hardware (registers, latches, adders, etc ...).

[2, 3, 4, 5, 6, 7, 8] propose different methods for behavioral synthesis with more or less restricted subsets of VHDL using a template-based description style. We have formalized a large subset of sequential VHDL in terms of Interpreted and Timed Petri Nets [9] on which we perform model checking [10] and equivalence proof [11]. We propose to use this intermediate formal model to synthesize a more natural subset of sequential VHDL than that accepted by the existing tools, especially by alleviating the constraints on the programming style.

In the following paper, we will present our method for behavioral synthesis. Section 2 presents briefly the VHDL subset and the formal model used, based on Petri Nets. Section 3 deals with some of the main aspects of behavioral synthesis of VHDL descriptions after translation into the formal model and we conclude with the future prospects.

2. The Subset of VHDL Description and the Formal Model

The VHDL description we consider is supposed to be elaborated: this is a collection of processes communicating through signals. In particular, all concurrent statements other than processes are supposed to be replaced by their equivalent process. Variables and signals are of types bit, bit_vector, boolean or enumerated. This is indeed a practical limitation but it will be alleviated in the near future. Inside a process, we restrict ourselves to the following sequential statements:

```
sequential_statement ::=
variable_assignment   |   signal_assignment   |
wait_statement        |   if_/case_statement  |
loop_statement        |   next_/exit_statement |

signal_assignment ::= sig_name <= expression
wait_statement ::= wait on signal_list until condition
```

Notice that the temporal clauses are excluded both in signal assignment and wait statement. Furthermore, we assume that each execution path in the body of loop statements contains at least one wait statement, imposing the termination of the execution of each user-process.

VHDL semantic is informally expressed by means of its simulation engine. One has to develop a formal model to reason about a VHDL description. We chose the Petri Net formalism as it supports non determinism that will be necessary for VHDL'93, but it can also represent deterministic systems, such as VHDL'87. Various techniques of construction of the transition system representing all behaviors of the modeled system can be applied to this formalism.

The Petri Net [12] represents the control structure of VHDL processes and their synchronization reproducing the VHDL simulation semantics (by decomposing the simulation cycle into RESUME, EXECUTE and UPDATE phases which provide synchronization barriers for the processes). An external data part (P_{data}) of the Petri Net contains the data modified by the firing of transitions in the control part. The construction and behavior of the Petri Net are presented in [9].

Each process is composed of places and transitions. Places refer to the states of the process and transitions are fired to pass from one state to the next, representing the VHDL statement executed between these two states. Connections between places and transitions are expressed by Pre and Post matrices. $Pre(t,p) = 1$ indicates an arc from a place p to a transition t , and $Post(t,p) = 1$ indicates an arc from t to p . Transitions modeling processes are split into two disjoint sets. Those modeling VHDL *wait* statements belong to the RES set, they are only firable during the resumption phase of VHDL delta cycles (RESUME phase). All other VHDL statements are represented by EXE transitions, which are firable during the execution phase of VHDL delta cycles (EXECUTE phase).

Interactions between the control part and the data part occur while transitions are fired. These interactions are represented by means of attributes associated to each transition, t , of the Petri Net:

- $g(t)$ is the guard of transition t : t may fire only if its guard is true. $g(t)$ is a Boolean function of data contained in the data part.
- $ASG(t)$ is the set of data modified while firing transition t .
- $TRF(t)$ is the set of transformations applied to the data in $ASG(t)$. $TRF(t)$ is a set of couples $(d, trf_{d,t})$ where $d \in ASG(t)$ and $trf_{d,t}$ is a Boolean function $d_i \in P_{data}$.

In VHDL'87, all statements enclosed between two *wait* statements in a process are atomic: instead of being represented by a sequence of EXE transitions, each corresponding to a VHDL statement, they can be represented by a unique EXE transition grouping all the data transformations (reduction rules are defined in [9]).

The Figure 1 exhibits the VHDL description of a latch, the corresponding reduced Petri Net and the different attributes associated to this Petri Net, used for the synthesis in the next section. The reduced Petri Net only have two transitions whereas an unreduced Petri Net would have nine transitions (representing a full control graph of the VHDL description). Notice that *eff_sig* denotes the effective value of the signal *sig*, and *drv_sig* and *evt_sig* respectively denote the driven value and the event attribute of the signal *sig*. *var_P_V* denotes the variable *V* declared in the process *P*.

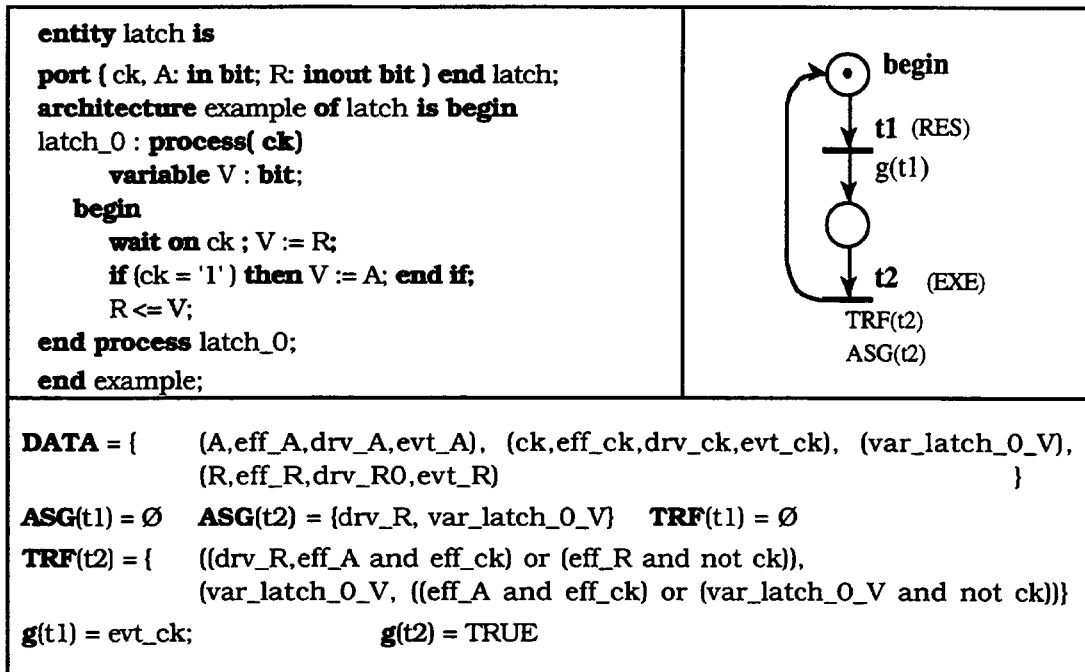


Figure 1. Petri Net representation, interacting data part (DATA) and attributes derived from the VHDL description of a latch.

3. Application to behavioral synthesis

We are able to extract from the formal model (ITPN) a set of equations, one for each assigned VHDL symbol (signal or variable), which summarizes all its possible execution paths during the VHDL simulation cycle. These equations are also used in formal verification environments, generating a stable state reachability graph for model checking or behavioral equivalence [13].

We have already shown that it is possible to use this state graph to synthesize the description as a finite state machine if appropriate requirements are met by the transition rules of the state graph [4]. This method is semantically very powerful and strictly simulation-compliant, but it does not match some industry needs, such as speed of synthesis (exploration of stable states is often time-consuming), and efficiency of the generated netlist (size and propagation delay).

Here we will discuss a novel approach, based on an independent synthesis of each VHDL symbol without considering any global property of the behavioral description: every assigned symbol will correspond to an equipotential in the generated netlist. This constraint makes the method semantically less powerful than [4], but can make it competitive with existing synthesis tools in terms of performance.

3.1. Automatic Recognition of Memorizing Elements for Synchronous Descriptions

Restrictions on VHDL constructs are often imposed by synthesis tools which use syntactic Pattern Matching of predefined templates. Since we have one formalism for all VHDL symbols, and our process for extracting transition equations from the VHDL behavioral description is not sensitive to the initial programming style, we do not impose the use of special templates in predefined combinations to specify memorizing elements. For example, the same register will be inferred equally from the following two descriptions:

<pre>process (ck) begin T <= R; if (ck'event and ck = '1') then R <= A; end if; end process;</pre>	<pre>process (ck) variable R: bit := '0'; begin if (ck'event and ck = '1') then R := A; end if; T <= R; end process;</pre>
--	---

Figure 2: Two equivalent VHDL description styles for a register

The VHDL code on the left hand side is an accepted and recommended template in the subset of [2], used to describe registers by means of "<edge> constructs". On the other hand,

the code on the right hand side, although equivalent, is not accepted in the same subset (where it is considered as illegal, because if a variable is assigned in an "<edge> construct", it is not allowed to read it later in the same process).

In the previous section we presented briefly the extraction of the set of transition equations for each VHDL description: the method will generate the same equations for these two descriptions. Let us go further and use this equation (1) as an example to introduce our algorithm for synchronous synthesis:

$$\text{var_R} \leq (\text{evt_ck and ck and A}) \text{ or } \neg(\text{evt_ck and ck}) \text{ and var_R} \quad (1)$$

$$\text{drv_T} \leq \text{var_R} \quad (2)$$

This algorithm applies whenever a VHDL process has only one signal (which we can refer to as "the clock") in its sensitivity list. In such cases, VHDL symbols assigned by this process *a priori* synthesize into D-flip-flops.

We perform a Shannon decomposition [14] of the assigned expression (figure 3), first using the `evt_ck` signal then eventually the `ck` signal itself for the variable ordering. Only the "evt_ck true" branch matters, because if the process has awaked, it means exactly that the clock changed.

This branch must not depend upon any other `evt_sig` attribute, because this implies testing absolute synchronicity between two events, which makes sense only within the discrete delta-cycle timing model of VHDL and not in physical hardware. This requirement is met in the above case, so we can carry on further.

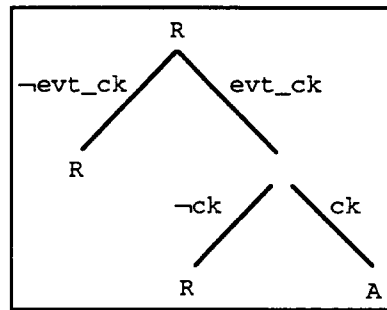


Figure 3: Shannon Decomposition

Depending on the value of the clock, this symbol has different assigned expressions. This means that the clock is read by the corresponding logic. This is an undesirable situation in many hardware technologies, but it can be allowed to synthesize into dual-phase logic. In our example, however, one of the sub expression has the nice property of being another instance of the symbol `R` itself. This will be the case whenever an "edge construct" or a similar construct is used to specify a register, but it is also possible if no special patterns are used, as a result of ITPN reductions. The reduction will automatically extract the property without constraints on the programming style.

Because of this property, we know the symbol `R` must be synthesized as a raising-edge sensitive D-flip-flop, sampling symbol `A`. Had `R` appeared in the other branch of the decomposition, we would have naturally synthesized as a falling-edge sensitive D-flip-flop. Since a D-flip-flop is a standard memorizing element, it makes a safe use of the clock in any hardware technology.

However, many symbols in a synchronous VHDL description, will be assigned outside of "edge constructs", and will not depend on the value of the clock. As we suppose that there is no trigger condition on WAIT statements (these have already been included in the assigned expressions, during the ITPN reduction phase), these symbols may update their values at any edge of the clock in a simulation of the VHDL process, implying dual-phase logic (figure 4 - leftmost). In this case, the logic to be synthesized depends on the use of these signals in the rest of the VHDL description. If such a signal S is used in an other process which is not synchronous, then we must generate dual-phase logic to be fully compliant with the simulation semantics of VHDL (figure 4 - rightmost).

<pre> process (ck) begin S <= expression; end process; </pre>	<pre> process -- relaxed begin -- compliance wait until ck = '1'; Output <= S; end process; </pre>	<pre> process (S) -- asynchronous: begin -- S must be Output <= S; -- correct at end process; -- all δ times </pre>
---	---	---

Figure 4: Signal implying either single- or dual-phase logic.

But in many cases, the symbols activate the outputs only through other synchronous processes, with logic sensitive to only one edge of the clock (figure 4 - middle). So we should only need to detect this edge and synthesize the appropriate D-flip-flop to get an equivalent behavior with a minimum number of memorizing elements.

3.2. Synthesis of combinatorial logic with feedbacks

The previous section dealt with the fully synchronous VHDL styles. On the opposite end of the spectrum, commercial synthesis tools also accept purely combinatorial sequential VHDL: that is, where processes are sensitive to all of the signals they read as inputs, and no explicit memory elements are specified by the usual constructs. If present, such constructs can be processed separately by an extension of the previous method provided that they do not imply testing absolute synchronicity (a requirement met by all patterns in subsets like [2]).

However, these tools impose again some programming constraints because they do not detect the possible glitches and race conditions that arise notably from feedbacks in the logic. Here, we propose an algorithm for recognizing and classifying such feedbacks, in order to generate and insert the appropriate memory elements in the feedback cycle, thus extracting synchronous behavior out of a data-flow-like VHDL descriptions.

This algorithm is based on differentiation properties of boolean expressions. Suppose, we have a signal x_1 which is computed with a feedback:

$$x_1 = f(x_1, x_2, \dots, x_n) = x_1 \cdot f_+(x_2, \dots, x_n) \oplus \overline{x_1} \cdot f_-(x_2, \dots, x_n)$$

then: $(\partial / \partial x_1) f(x_1, x_2, \dots, x_n) = f_+(x_2, \dots, x_n) \oplus f_-(x_2, \dots, x_n)$

We can define a sign for this differential, and see how this sign relates to the behaviour of f when x_1 changes:




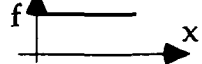
f_+	f_-	f	f	situation:
0	0	0		following
0	1	<0		oscillating
1	0	>0		latched
1	1	0		following

Figure 5: Differentiation of f

In the "oscillating" case, a negative feed back will cause the value of f to oscillate with δ -frequency as long as the $\{x_i, i>1\}$ remain in the negative-differential domain. This will freeze the simulation of the VHDL description, and it certainly is an undesirable feature in any hardware technology. Formal verification techniques can be used to check that this situation is unreachable, before any further analysis of the design.

In the "following" cases, the value of f does not depend upon the value of x_1 but only on those of the other variables: the feed back is not active, and in that domain of operation, f can be computed by a simple acyclic combinatorial circuit.

In the "latched" case, a positive feedback locks the value of f to the one it had before the $\{x_i, i>1\}$ entered the positive-differential domain: here we have a level-sensitive memorizing element, or a *latch* behaviour (triggered by the value of the test " f positive"). The latch is a standard, although low-level, component in all hardware technologies, thus apparently solving the problem of synthesis.

We propose to examine the application of this algorithm on a simple example, to see the problems that arise from practical considerations: The figure 6 shows two descriptions for an asynchronous Latch (L) with active low Set (S) and Reset (R), the Set being dominant.

<pre> process (S,R) begin if (S = '0') then L <= '1'; elsif (R = '0') then L <= '0'; end if; end process; </pre>	<pre> signal T : bit; -- auxiliary process (S,R,L,T) begin T <= R nand L; L <= S nand T; end process; </pre>
--	--

Figure 6: Two Styles for an Asynchronous Latch

The description on the left hand side of figure 6 uses partly specified assignments to imply a memory element (as required by [2]). The description on the right hand side is a

data-flow-like process with strictly identical behaviour. We extract the following equation for the output L , from our formal model:

$$L \leq S \text{ nand } (R \text{ nand } L) \tag{3}$$

First notice that the extraction of (3) required the propagation of the intermediate value T over one δ cycle to uncover the feedback: complex retro-actions can even take place across several processes and with more than one intermediate signal (and cycle). The search for all such complex retro-actions in a realistic VHDL description as yet needs to be performed more efficiently. Now we compute the differential of L with respect to itself:

$$L = F(L, \dots) = L \wedge F(L, \dots) |_{L=1} + \neg L \wedge F(L, \dots) |_{L=0}$$

As $L = L \wedge (\neg S + R) + \neg L \wedge (\neg S)$ we identify the two terms of the differential:

$$L^+ = F(L, \dots) |_{L=1} = \neg S + R \text{ and } L^- = F(L, \dots) |_{L=0} = \neg S$$

Hence, $L' = L^+ \oplus L^- = S \wedge R$

Analysis of the sign of the differential can be performed efficiently with BDDs [15]. L' is positive when it is non-zero, so we know formally that the description is always stable. When L' is zero, L is equivalent to $\neg S$, and the sampling condition is $S \wedge R$, so we can propose the equivalent circuit of figure 7.

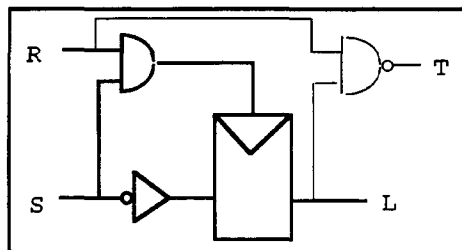


Figure 7: Synthesized Circuit

However, we still have three problems:

- The signal T has disappeared. If it were used elsewhere in the description, we have to re synthesize it with a NAND gate (the lighter part in the figure 7).
- The transistor count and the timing of the circuit may not be optimal, especially if we have to recreate extra hardware for intermediate signals. If we assume 2 transistors for an inverter, 4 for a 2-to-1 gate and 7 for a latch, then we generate 13 or 17 transistors, as compared to 8 for an optimal design.
- We may have introduced a race condition between the AND gate and the inverter. Correctness requires that the AND gate propagates changes to S at least as fast as the inverter. This is not warranted in all hardware technologies. A crude VHDL rewriting of the circuit would be correct because both paths would have the same delay of one δ .

We can solve all of these problems if we replace the latch-centered logic initially proposed, by a portable library cell that performs optimally and safely the same operations in all hardware technologies. To recognize the standard cell, we can use functional mapping

of the two BDDs of the sampled function and the trigger function, because these functions canonically define the behaviour of any level-sensitive memory element. Extra outputs such as \mathbb{T} above can also be matched against library patterns, to spare as much logic as possible.

However, no cell library provides the support required for this recognition, in the form of tabulated canonical functions for all memory elements. This trivial obstacle will have to be lifted to allow a wider range of research on our technique.

3.3. Actual limitations and future prospects

We have presented two situations for which we developed specific techniques to infer memorizing elements from behavioral descriptions. These situations cover a broader part of VHDL modeling styles than most synthesis subsets to date. But a wide array of conditions are still beyond our capabilities. This notably includes descriptions with multiple wait statements that have different sets of trigger signal, and descriptions with incomplete sensitivity lists, because of the inability to test strict synchronicity of events in hardware. This problem is avoided in existing tools, by an arbitrary completion of the sensitivity lists with missing signals, which obviously distorts the simulation semantics of VHDL in the synthesized hardware.

In the restrictive hypothesis of a "fundamental mode" where no two inputs can have strictly synchronous events in any simulation cycle, some of these descriptions can be synthesized using a XOR_n gate to summarize all the events of n trigger signals.

The actual implementation under development, at the MASI Laboratory, we have not been able so far to confront the temporal performances of our methodology with the existing tools. Future steps of this development will include an enlargement of the VHDL subset accepted by the existing translation tool [16], to cover the IEEE standard logic 1164. Our primary implementation target is the synchronous description styles of section 3.1.

Conclusion

We have presented an approach which exploits a formal model of VHDL in terms of Interpreted and Timed Petri Nets already used with success for formal methods such as behavioral equivalence and model checking. We are using this model to perform behavioral synthesis without imposing cumbersome description style. Our approach can not only synthesize existing subsets of VHDL for behavioral synthesis but also asynchronous VHDL descriptions with some restrictions. An implementation of our method is under development. We plan to alleviate its constraints with enhanced algorithms.

Bibliography

- [1] "IEEE Standard VHDL Language Reference Manual" IEEE Std 1076-1987.
- [2] Synopsys, Inc, "VHDL Compiler Reference Manual v3.0", part #1US00-10030, November 1992.
- [3] P. Eles, K. Kuchinski, Z. Peng, M. Minea, "Synthesis of VHDL Concurrent Processes", proceedings of the Euro-DAC'94.
- [4] A. Evans, E. Encrenaz, R. K. Bawa, L. Jacomme, "An Approach to the Synthesis of VHDL Concurrent Processes as a FSM", proceedings of IFIP WG 10.5 Workshop on Logic and Architecture Synthesis, INPG, Grenoble, France, December 1995.
- [5] P. Harper, S. Krolikoski, O. Levia, "Using VHDL as a Synthesis Language in the Honeywell VSYNTH System", proceedings of CHDL'90, J.A. Darringer, F.J. Rammig editors, North-Holland, 1990.
- [6] G. Mekenkamp, P. F. A. Middelhoek, B. E. Molenkamp, J. Hofstede, T. Krol, "A Syntax based VHDL to CDFG Translation Model for High-Level Synthesis", proceedings of VIUF Spring'96, Santa Clara, U.S.A., February-March 1996, pp 89-97.
- [7] A. Postula, "VHDL Specific Issues in High Level Synthesis", proceedings of Euro-VHDL'91.
- [8] J. Mirkowski, K. Bilinski, E.L. Dagless, "Petri Net Modelling of VHDL Simulation Cycle for High Level Synthesis Purposes", proceedings of VHDL User Forum in Europe SIG-VHDL Spring'96 working conference, Dresden, Germany, May 1996.
- [9] E. Encrenaz, "A Symbolic Relation for a Subset of VHDL'87 Descriptions and its Application to Symbolic Model Checking", proceedings of CHARME'95, Frankfurt, Germany, October 1995, LNCS 987.
- [10] R. K. Bawa, E. Encrenaz, "VMC: A tool for Model Checking VHDL descriptions", proceedings of VHDL User Forum in Europe SIG-VHDL Spring'96 working conference, Dresden, Germany, May 1996.
- [11] R. K. Bawa, E. Encrenaz, "Formal Verification of VHDL Descriptions by Symbolic State Space Exploration applied to Finite State Machines", proceedings of VIUF Spring'96, Santa Clara, U.S.A., February-March 1996.
- [12] T. Murata, "Petri Nets: Properties, Analysis and Applications", Proceedings IEEE, vol 77 n°4, April 1989, pp 541-580.
- [13] R. K. Bawa, E. Encrenaz, "A Platform for the Formal Verification of VHDL programs", 4th International Workshop on Symbolic Methods and Applications in Circuit Design, 10-11 October 1996, Leuven, Belgium.
- [14] C. E. Shannon, "A Symbolic Analysis of Relay and Switching Circuits", Transactions AIEE 57, pp. 305-316, 1938.
- [15] R.E. Bryant, "Graph Based Algorithms for Boolean Function Manipulation", IEEE Transactions on Computers, Vol C-35, pp. 677-691, 1986.
- [16] R. K. Bawa, E. Encrenaz, "A Tool for Translation of VHDL descriptions into a Formal Model and its Application to Formal Verification and Synthesis", 4th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, 9-13 September 1996, Uppsala, Sweden, in LNCS, Springer-Verlag.