Semantics of a Verification-Oriented Subset of VHDL

David Déharbe and Dominique Borrione

ARTEMIS-IMAG
BP 53
F-38041 Grenoble Cedex 9, France

Abstract. This paper gives operational semantics for a subset of VHDL in terms of abstract machines. Restrictions to the VHDL source code are the finiteness of data types, and the absence of quantitative timing informations. The abstract machine of a design unit is built by composition of the abstract machines for its embedded processes and blocks. The kernel process in our model is distributed among the composed machines. One transition of the final abstract machine models a VHDL delta cycle. This model can be used for symbolic model checking and equivalence verification.

1 Introduction

Giving a formal definition of the semantics of VHDL [7] is of highest importance for synthesis and formal verification. Many different approaches have been proposed to fulfill this need, see eg [1, 3, 4, 5, 8, 9, 10, 11]. A first conclusion can be drawn from a study of these works: one has to trade off the number of VHDL features modeled, and the practical usefulness of the semantics. VHDL is a very complex language and the models that capture all the features are almost inherently not applicable to produce design automation tools. On the other hand, if some aspects of the language are deliberately discarded, it becomes possible to deal with it efficiently, on a formal basis.

In the approach we present, we restrict ourselves to a subset of VHDL such that design descriptions can be mapped to finite state representations: objects must be of a finite type (no access nor file types, no unconstrained arrays, no generics) and quantitative timing informations are not accepted (no 'after' clauses in assignment statements, no 'for' clauses in wait statements). Under these restrictions, the elaboration of VHDL design entities in terms of our semantics produces abstract machines that are suitable for verification by symbolic methods (equivalence proof, model checking). By means of symbolic simulation techniques, a single execution of the generated abstract machines represents all possible executions of the corresponding VHDL script.

The research efforts that are closest to our work are the semantics presented by Damm et al. [2], that serve as a formal basis to feed VHDL designs to a model checker. Our approach mainly differs in the fact that a state in our model represents a point in the simulation where the current statement of all processes

is a wait statement, whereas in [2] a state represents a point where the current statement of each process is any statement. Using programming terminology, a program counter in our case goes from a wait statement to the next wait statement, whereas in [2] it goes from a sequential statement to the next sequential statement. Thus, the main advantage of our approach is that a single transition represents a whole simulation cycle, which is not the case in [2], where several transitions are needed.

Overview of the Article : Section 2 presents the target model towards which we shall map VHDL design entities. Section 3 gives the general principles that drive the different aspects of the semantics: the elaboration of models and their composition. These two aspects are presented in detail respectively in Sections 4 and 5. Section 6 gives a short conclusion and presents future developments of this work.

2 Abstract Machines

The VHDL semantics presented in this paper are expressed in terms of a class of models called abstract machines. These abstract machines are composed of a finite state machine and a boolean condition on the state space of this machine.

Definition 1 Abstract Machine. An *abstract machine* $\mathcal{M}$ is defined as a 7-tuple
$\langle I, S, O, s_0, NS, NO, sc \rangle$, where:

- I is a set of *input* variables of $\mathcal{M}$: $I = \{i_1, \ldots, i_{ni}\}$. Each input variable i_k has its value domain denoted $\mathcal{I}_k$. The input domain $\mathcal{I}$ is the Cartesian product $\mathcal{I}_1 \times \cdots \times \mathcal{I}_{ni}$;
- S is a set of *state* variables of $\mathcal{M}$: $S = \{s_1, \ldots, s_{ns}\}$. Each state variable s_k has its value domain denoted $\mathcal{S}_k$. The state domain $\mathcal{S}$ is the Cartesian product $\mathcal{S}_1 \times \cdots \times \mathcal{S}_{ns}$;
- O is a set of *output* variables of $\mathcal{M}$: $O = \{o_1, \ldots, o_{no}\}$. Each output variable o_k has its value domain denoted $\mathcal{O}_k$. The output domain is the Cartesian product $\mathcal{O}_1 \times \cdots \times \mathcal{O}_{no}$;
- $s_0 \in \mathcal{S}$ is the *initial state* of $\mathcal{M}$: s_0 is a valuation of the state variables S of $\mathcal{M}$;
- NS is the *state transition function* of $\mathcal{M}$: $NS = [NS_1, \ldots, NS_{ns}]$, and NS_k is the transition function of the state variable s_k : $NS_k : \mathcal{I} \times \mathcal{S} \rightarrow \mathcal{S}_k$;
- NO is the *output function* of $\mathcal{M}$: $NO = [NO_1, \ldots, NO_{no}]$, and NO_k is the output function of o_k : $NO_k : \mathcal{I} \times \mathcal{S} \rightarrow \mathcal{O}_k$.
- sc is a *stability condition* on the state space of $\mathcal{M}$[1] : $sc : \mathcal{I} \times \mathcal{S} \times \mathcal{O} \rightarrow \mathbb{B}$.

Informally, a state of the abstract machine represents a state of the VHDL design where all processes are suspended on one of their wait statements. Computations within the process statements that lead from one wait statement to another wait

[1] $\mathbb{B}$ denotes the boolean domain

statement are collapsed into a single transition, and intermediate computations are invisible in the abstract machine. The stability condition *sc* is true on the states of $\mathcal{M}$, that represent VHDL states where no process statement is active, hence all clauses of the respective current wait statement of the different processes are not satisfied.

3 General Principles

Given a VHDL design D, an abstract machine $\mathcal{A}$ is elaborated. The requirement put on $\mathcal{A}$ is that it has the same observable behavior as D. By observable behavior, we mean that the response of outputs of $\mathcal{A}$ to stimuli on its inputs should be the same as the response of the **out** ports of D to its assignments to its **in** ports. Both the behavior at the level of the delta cycle and the behavior at the level of stable states are considered.

3.1 Principles of Composition

The semantic rules give a formal definition to the translation of any VHDL design unit to the class of abstract machines. Since process statements are the atomic components of a design entity, *an abstract machine is associated to each process of the considered design unit.* The resulting machines are composed to obtain the semantics of the design unit, as illustrated in Fig. 1. These compositions also take into account the kernel activity: update of effective values, elaboration of implicit signals, etc.

Hence, one can identify three main parts to define these semantics:

- the semantic rules for elaboration, that associate an abstract machine to a VHDL process statement within some declaration environment,
- the semantic rules associated to the concurrent composition of concurrent statements, that corresponds to the parallel composition of abstract machines,
- the semantic rules associated to the declaration encapsulation mechanism that occurs in 'block' statements and 'architecture bodies', that corresponds to the hiding operator (or declaration encapsulation) on abstract machines.

As any VHDL concurrent statement that is not a block statement has a corresponding equivalent process statement, this discussion focuses on the semantics of process and block statements.

3.2 Principles of Elaboration

The process statement is the smallest piece of VHDL that can be simulated: consider an architecture body with a single process statement, and whose entity declaration **in** ports (resp. **out** ports) are the signals read (resp. assigned) within this process statement.

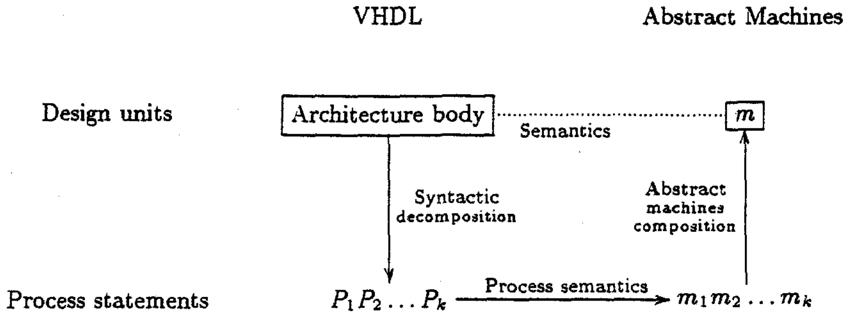


Fig. 1. Composition schema

It is quite clear that the input, output, and state variables of the abstract machine that corresponds to a process statement should be elaborated from the signals read, the signals assigned, and the variables declared in the process statement. State transition and output functions should be elaborated from the assignments to variables and signals. The initial state should correspond to the initial values of the process variables. And the stability condition should be true whenever the different clauses of the current wait statement are not satisfied.

However, there are practical problems to do this:

- In a process statement, assignment statements are executed sequentially, and thus are dependent of each other; the source expression of an assignment, as well as the condition of a conditional statement, depends on the previous variable assignments. But in the abstract machine, state transition and output functions occur simultaneously (in parallel), and are independent.
- A process statement may have several wait statements, and the elaboration of the stability condition has to be carefully performed to handle this.

The following section 4 provides a method to resolve these problems.

4 Elaboration

Throughout this section, the elaboration principles will be illustrated on a running example, shown in Fig. 2, where `F`, `N`, `M` are boolean signals. The elaboration is based on the analysis and the modification of the execution flow in the process statement. Thus, for sake of clarity, we base the discussion on a flow graph representation of the statement part of the process, as briefly depicted in section 4.1. In section 4.2, we present a method to merge wait statements as well as the subsequent modifications on the flow graph representation, such that the process behavior and reactivity to changes on signals' values are not modified. Following, section 4.3 relates to the problem of making the evaluation of expressions independent of the assignments that occured previously in the same simulation

cycle. A solution to this problem is given in section 4.4, where the flow graph is transformed into a tree and simplified. The obtained simplified execution tree is used to derive a decision diagram for each object, signal or variable, assigned within the process, as explained in section 4.5. Finally section 4.6 deals with the elaboration of the abstract machine. The method presented herein can be applied

```
P: process
   variable V : NATURAL := 0;
begin
   if N then
      if (V < 8) then;
         V := V + 1;
      end if;
   else
      V := V - 1;
      wait until (not M);
   end if;
   F <= (V = 8);
   wait until (N or M);
end process;
```

Fig. 2. P : A VHDL process statement example

to any process whose statement part is (or could be rewritten into) any combination of if statements, wait statements, and signal and variable assignment statements.

4.1 Representation of the Process Statement Part

The representation of the statement part of a process statement P is the flow-graph $\mathcal{G}(\mathtt{P})$ of the syntactic structure of the sequence of statements. Let P be a process statement, $\mathcal{G}(\mathtt{P}) = (V, A)$ is a directed graph such that:

- V is the set of vertices, each vertex corresponds to either an assignment statement, a wait statement or an if statement.
- $A \subseteq V \times V$ is the set of arcs, arcs are labeled with VHDL conditions.

For instance, the statement part of the process of Fig. 2 is represented by the graph of Fig. 3.

Non-accepted Process Statements: Let P be a process statement. If, in the graph $\mathcal{G}(\mathtt{P})$, there is a cycle such that there is no vertex carrying a wait statement, and such that the conjunction of the conditions that label the arcs is not equal to `false`, then we do not know how to elaborate an abstract machine. This characteristic arises in a statement part that contains an execution path without wait statement (also in a loop whose body does not contain a wait statement). These situations can lead to a non-terminating simulation cycle, and therefore cannot be handled in a finite state model.

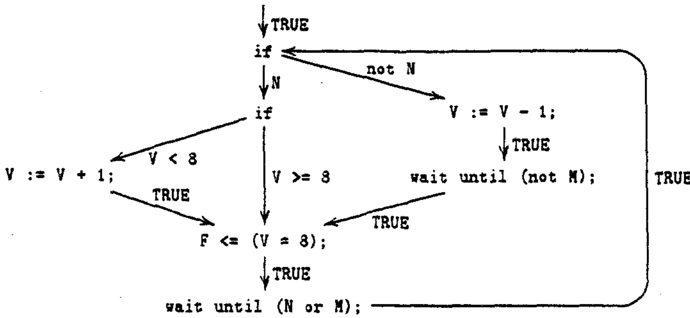


Fig. 3. Flow graph of the process statement P

4.2 Merging Wait Statements

Given the set of wait statements of a process, we replace all of them by a single wait statement that resumes if and only if the initial process resumes its activity from the current wait statement. From now on, we refer to this resulting wait statement as the *merged wait statement.*

Proposal 4.1 (Wait counter) *In order to transform a process statement* P *into a process statement* R *with a single wait statement, it is necessary to introduce a variable, named* CW *(for Current Wait), of type natural, that ranges from 0 to the number n of wait statements in* P, *and to label each wait statement of* P *with a different natural number between 1 and n. The value of* CW *is equal to 0 when* P *is at the beginning of a simulation, and to i ($1 \leq i \leq n$) if* P *is suspended at the ith wait statement.*

Suppose that the process has n wait statements, with (possibly implicit) sensitivity list s_i and condition clause c_i. The process resumes its activity on the ith wait statement if there is an event on one of the signals of s_i, and the condition c_i is valid. Hence, the condition under which the process resumes its execution is:

$$(\texttt{CW} = 0) \vee \bigvee_{i=1}^{n} \left((\texttt{CW} = i) \wedge c_i \wedge \left(\bigvee_{S \in s_i} \texttt{S'event} \right) \right)$$

This general expression is used to derive the condition clause of the merged wait statement, where an explicit sensitivity clause is no longer useful, since all the signals the different waits are sensitive to appear as primary in the condition clause. The condition CW = 0 exhibits the fact that the process is active at the beginning of the simulation.

In the example, the different wait statements are:

```
wait on M until (not M);     - - here, CW = 1
wait on N, M until (N or M); - - and there, CW = 2
```

Thus, the corresponding merged wait statement is:

```
wait until ( (CW = 0) or
             ((CW = 1) and (not M) and M'event) or
             ((CW = 2) and (N or M) (and N'event or M'event)));
```

The flow-graph representation has to be modified to replace the original wait statements by the merged wait statement, and to update the variable CW. An assignment to variable CW is inserted before each wait statement, the assigned value is the label of this wait statement. An if statement is added at the beginning of the process statement part and directs the execution flow to the appropriate set of statements, according to the current resumption condition. The graph representation of these sets of statements are the connected elements of the original flow graph, where the vertices carrying wait statements have been removed. Each one of these sets of connected elements is called a *zone*; all zones are inserted as the different branches of the added if statement. The ith zone, $1 \leq i \leq n$ (resp. 0), is the zone executed after the ith wait statement (resp. when the simulation starts).

The resulting if statement is followed by the merged wait statement that ends the process statement part. The new flow graph of the example, after performing these transformations, is depicted in Fig. 4, where:

<cond0> stands for `(CW = 0)`,
<cond1> stands for `(CW = 1) and M'event and (not M)`,
<cond2> stands for `(CW = 0) and (N'event or M'event) and (N or M)`

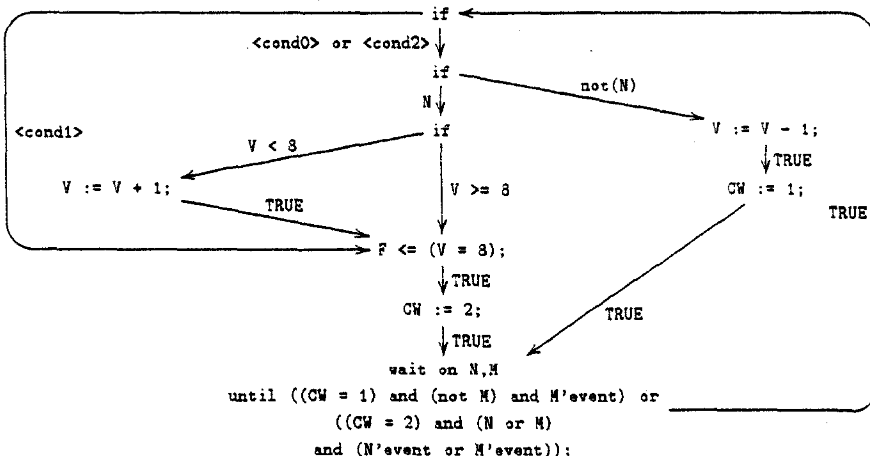


Fig. 4. Graph after merging wait statements

Finally, this transformation leads to a form of process statement that has the following features:

- the first statement is a conditional statement (on the current value of `CW`) and its branches do not hold wait statements, this statement is referred as the *transition statement*,
- the second and last statement is the unique wait statement of the process.

4.3 Resolving Dependencies

During simulation, an object that occurs in an expression is evaluated, and in the case of a variable, it evaluates to its current value. Thus, if this variable has been previously assigned in the same simulation cycle, its current value depends on this assignment. This dependency has to be eliminated in order to elaborate an abstract machine, where the atomic transitions must be independent of each other.

Thus, we need to distinguish the occurrences of variables in expressions that depend on some previous assignment executed in the same simulation cycle (these are called *reducible* occurrences) from the occurrences of variables whose current value has been assigned at a previous simulation cycle (these are called *irreducible* occurrences).

Proposal 4.2 (Reducible occurrences) *In a zone of statements, if a variable* `V` *is assigned an expression* `E`*, all following occurrences of* `V` *within expressions in the same zone can be replaced by* `E`*. Such variable occurrences within expressions are named* reducible occurrences.

The occurrence of a variable in a vertex statement (resp. in an arc condition) is reducible, if there is a path from the merged wait statement to this vertex (resp. arc), that passes through a vertex that carries an assignment to this variable. As a side effect, if all occurrences of a variable in a process are reducible, the variable is not necessary to compute the value of the other objects. It is therefore not useful to generate a state variable in the abstract machine to represent it.

Proposal 4.3 (Irreducible occurrences) *An occurrence of a variable* `V` *is said to be* irreducible *in a zone if there is an execution of the current zone such that there is no assignment to this variable.*

The occurrence of a variable in a vertex statement (resp. in an arc condition) is irreducible, if there is a path from the merged wait statement to this vertex (resp. arc), that does not pass through a vertex carrying an assignment to this variable.

The problem is that some occurrences happen to be both reducible and irreducible, as there might be several paths that go from the merged wait statement to a given vertex or arc (eg, in the assignment `F <= (V = 8);` of the example).

4.4 Derivation of the Execution Tree

In order to correctly process variable assignment statements and identify irreducible variables, we need a representation in which variable occurrences are either reducible or irreducible but never both at the same time for a particular execution path. Thus, we transform the graph of the process transition statement into an execution tree. The root of the execution tree is the initial vertex. Each vertex on a path is either an assignment or an if statement, and each path in this tree represents one possible execution of the statements in one zone. The execution tree is obtained by a simple unwinding of the graph of the transition statement. The leaves of the tree are implicitly followed by the merged wait statement. Figure 5 shows the execution tree that represents the transition statement of the example of Fig .2.

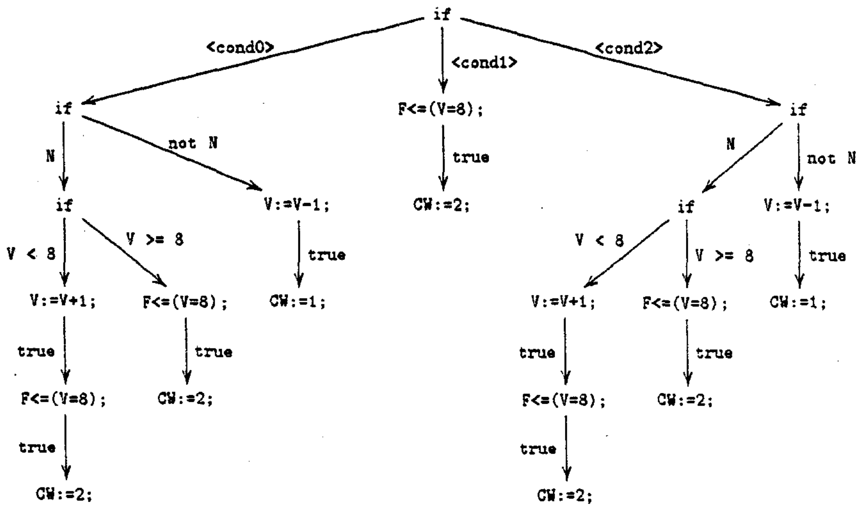
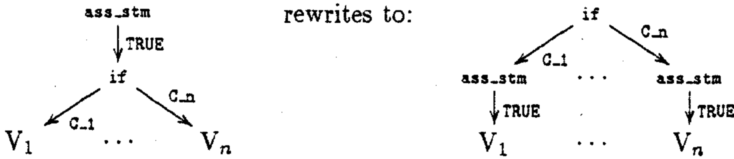


Fig. 5. Execution tree of the transition statement

On any path of the execution tree a variable occurrence is either reducible, or irreducible, but never both. This is a direct consequence of the fact that, in a tree, for every vertex and arc, there is a unique path that goes from the root to this vertex or arc. If a variable occurs in a vertex or in an arc label, and if there is an assignment statement to the variable on the path that leads to this expression, this occurrence is reducible, otherwise it is irreducible.

Simplification of an Execution Tree : The execution tree is simplified with two operations:

- reduction of reducible variable occurrences, by a simple traversal of the graph;
- pushing the 'if' vertices toward the root, and assignment statements towards the leafs of the execution tree. The algorithm is a rewrite system on the tree representation, schematically defined as:



This rewriting scheme pushes assignment statements towards the leaf of the execution tree, but preserves the relative order between assignment statements

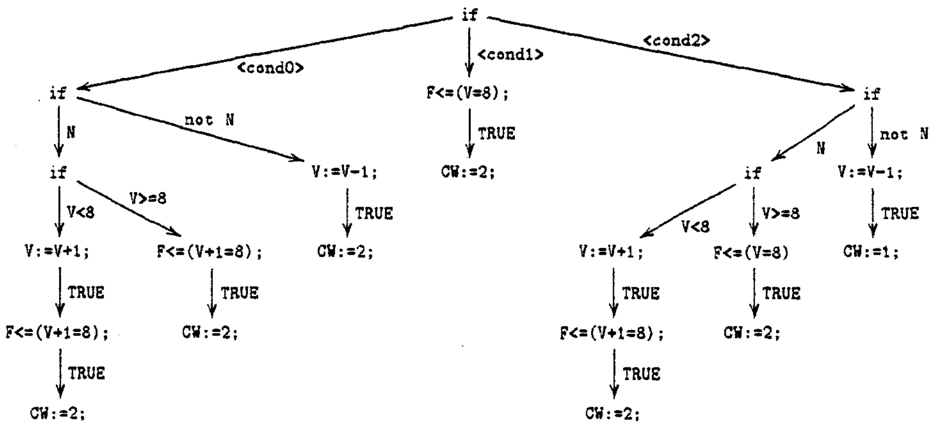


Fig. 6. Reduced execution tree of the transition statement

Fig. 6 presents the result of applying these reductions to the example. One can notice that, since reducible occurrences of variables have been replaced, there is no more dependency of expressions with respect to assignment statements. Now, when a variable occurs in an expression, it is evaluated to the value it had at the start of the zone, i.e. at the end of the previous simulation cycle, as it happens in the abstract machine.

4.5 Decision Diagrams for Assignments to Objects

In the simplified execution tree, assignments to objects (signals or variables) are independent of each other. Thus, in order to determine their characteristic

function, it is sufficient to create, for each assigned object O, a copy of this tree and to remove all vertices that contain assignments to other objects to get a simple decision diagram that gives the value assigned to O.

Fig. 7 shows the result of this operation on the example.

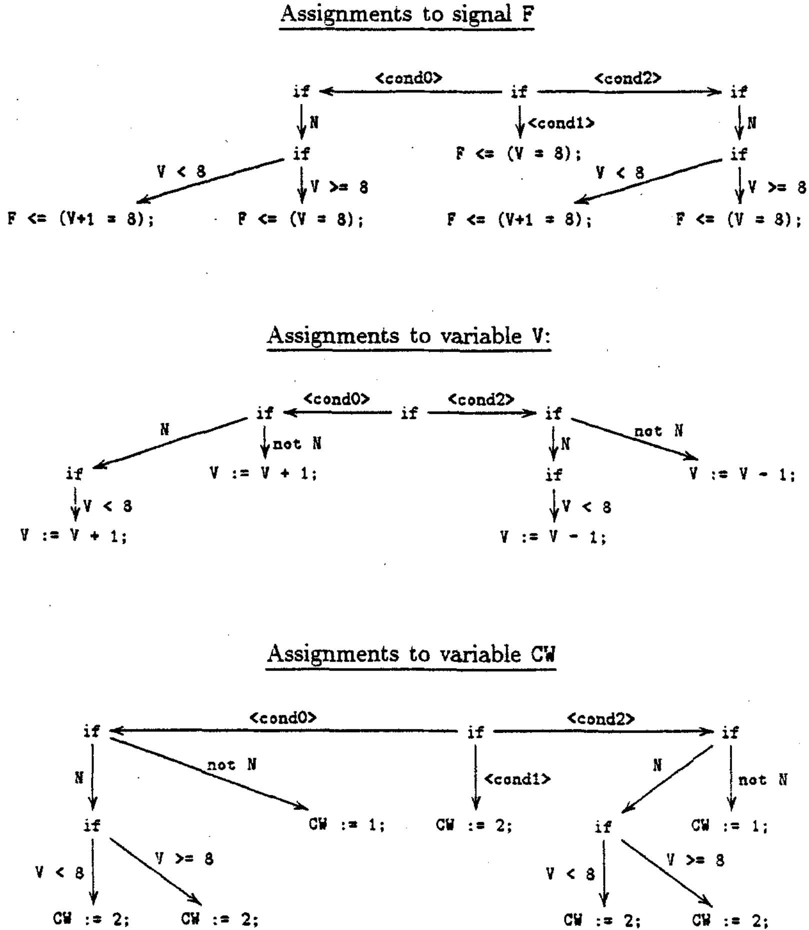


Fig. 7. Result decision diagrams.

4.6 Generation of the Abstract Machine

In order to elaborate the abstract machine, we suppose the existence of a morphism Φ (see e.g. [6]), from the VHDL value domains to the value domains of the abstract machine. To simplify this presentation, we use Φ on type names as

well as on expressions. In the following, types, operators and objects of VHDL are written using the `typewriter style`. For instance:

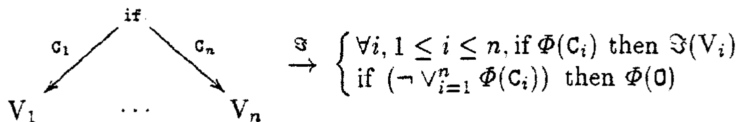
$$\Phi(\texttt{BOOLEAN}) = \mathbb{B},$$
$$\Phi(\texttt{A and not A}) = \Phi(\texttt{A}) \wedge \neg\Phi(\texttt{A}) = true,$$
$$\Phi(\texttt{NATURAL}) = \{n \in \mathbb{N} \mid 0 \leq n \leq \Phi(\texttt{NATURAL'HIGH})\},$$
$$\Phi(\texttt{4 - 2}) = 4 - 2 = 2$$

Given a process statement, we elaborate an abstract machine according the following general guidelines:

1. every signal `S` that appears in an expression within the process statement part elaborates an input variable $\Phi(\texttt{eff(S)})$ that carries its effective value, a signal that appears in a sensitivity list elaborates a boolean input variable $\Phi(\texttt{ev(S)})$, true whenever an event occurs on this signal,
2. every variable `V` of the process statement part, that has not been eliminated, elaborates a state variable $\Phi(\texttt{V})$ that carries its current value,
3. every signal `S` that appears as a target of a signal assignment statement within the process statement part elaborates $\Phi(\texttt{dr_val(P, S)})$, an output variable that carries the current value of its driver,
4. initial values of the elaborated variables (point 2) elaborate the initial state,
5. decision diagrams associated to the elaborated variables (point 2) elaborate the transition function,
6. decision diagrams associated to the elaborated outputs (point 3) elaborate the output function,
7. the condition clause of the unique wait statement elaborates the stability condition.

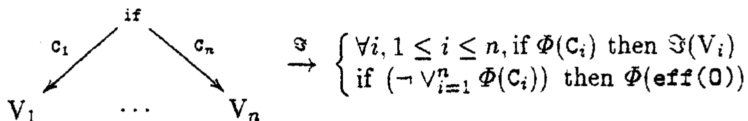
The elaboration of the transition functions is based on an interpretation $\Im$ of the decision diagrams of the assigned objects using the morphism Φ. $\Im$ takes into account the fact that when an object is not assigned, it keeps its previous value. $\Im$ interprets decision diagrams in terms of case-based function definitions. It is inductively defined in terms of the structure of the decision diagrams:

- the base case is the assignment statement vertex, its interpretation is defined by: `O := E;` $\xrightarrow{\Im} \Phi(\texttt{E})$
- the general case is an if vertex, its interpretation is defined by:
  1. if the object is a variable:



$$\xrightarrow{\Im} \begin{cases} \forall i, 1 \leq i \leq n, \text{if } \Phi(\texttt{C}_i) \text{ then } \Im(\text{V}_i) \\ \text{if } (\neg \bigvee_{i=1}^{n} \Phi(\texttt{C}_i)) \text{ then } \Phi(\texttt{O}) \end{cases}$$

  2. if the object is a signal:



$$\xrightarrow{\Im} \begin{cases} \forall i, 1 \leq i \leq n, \text{if } \Phi(\texttt{C}_i) \text{ then } \Im(\text{V}_i) \\ \text{if } (\neg \bigvee_{i=1}^{n} \Phi(\texttt{C}_i)) \text{ then } \Phi(\texttt{eff(O)}) \end{cases}$$

The stability condition sc is elaborated from the wait statement. The wait statement being `wait until cond`, the stability condition is $sc = \neg\Phi(\texttt{cond})$

5 Principles of Composition

We have presented the principles that guide the elaboration of abstract machines from a process statement. In this section, we give an informal presentation of the composition mechanism that applies to such elaborated abstract machines, and correspond to the following compositions of concurrent statements:

- *parallel composition* of abstract machines, corresponding to VHDL concurrent composition,
- *declaration encapsulation* mechanism that occurs in 'block' statements and 'architecture bodies',
- *hierarchy* with component instantiation statements.

These compositions are not associative: when elaborating a block statement or an architecture body, the whole instruction part must be elaborated (by means of parallel compositions) before the declaration encapsulation is done. We conclude this section by giving an evaluation of the complexity of the generated model.

5.1 Parallel Composition

Concurrent statement composition occurs in architecture and block statement parts. $\mathcal{M}_1 = \langle I_1, S_1, O_1, s_{01}, NS_1, NO_1, sc_1 \rangle$ and $\mathcal{M}_2 = \langle I_2, S_2, O_2, s_{02}, NS_2, NO_2, sc_2 \rangle$ being two abstract machines, the parallel composition of $\mathcal{M}_1$ and $\mathcal{M}_2$ is the abstract machine $\mathcal{M} = \langle I, S, O, s_0, NS, NO, sc \rangle$, such that:

1. $I = I_1 \cup I_2$, corresponds to the union of the variables that carry effective values of signals read in the composed concurrent statements.
2. $S = S_1 \cup S_2$, is the union of the variables that carry current values of VHDL variables. Since variables are objects local to a concurrent statement, $S_1 \cap S_2$ must be the empty set[2].
3. $O = O_1 \cup O_2$, is the union of the variables that carry current values of signals driven in the composed statements. The drivers being distinct, $O_1 \cap O_2$ must be the empty set.
4. $s_0 = s_{01} \circ s_{01}$, is the concatenation of the initial states of the composed abstract machines.
5. $NS = NS_1 \times NS_2$, the transition function of the composition is the product of the transition functions of the composed abstract machines.
6. $NO = NO_1 \times NO_2$, the output function of the composition is the product of the output functions of the composed abstract machines.
7. $sc = sc_1 \wedge sc_2$, states of the product machine are stable if the states of both machines are stable.

[2] To ensure that variables declared in different process statements are elaborated to distinct objects, the elaboration is done from a unique context-based VHDL name and not the local name of the variable.

5.2 Declaration Encapsulation

Declaration encapsulation of signals occurs in block statements and architecture bodies.

Block Statement : Let $\mathcal{M} = \langle I, S, O, s_0, NS, NO, sc\rangle$ be the abstract machine elaborated from the statement part of the block statement, S_d the set of signal declarations of its declarative part. The abstract machine elaborated from the block statement is $\mathcal{M}_e = \langle I_e, S_e, O_e, s_{0_e}, NS_e, NO_e, sc_e\rangle$. The variables in $\mathcal{M}$ that correspond to elements of S_d should not belong to the interface but to the internal variables of the result abstract machine $\mathcal{M}_e$. Furthermore, the kernel activity that corresponds to the update of the effective values of these local signals is modeled at the level of the block statement, by their transition function. We say that the kernel activity is *distributed* along the hierarchy in the design entities.

$\mathcal{M}_e$ is such that:

1. $I_e = I - (\{\Phi(\texttt{eff(s)}) \in I \mid \texttt{s} \in S_d\} \cup \{\Phi(\texttt{ev(s)}) \in I \mid \texttt{s} \in S_d\})$: inputs variables of $\mathcal{M}_e$ are variables that represent the effective values and the events on signals that are not in S_d,
2. $S_e = S \cup \{\Phi(\texttt{eff(s)}) \in I \mid \texttt{s} \in S_d\} \cup \{\Phi(\texttt{p_eff(s)}) \mid \texttt{s} \in S_d \wedge \Phi(\texttt{ev(s)}) \in I\}$: state variables are added to the result abstract machine to represent the effective values of all the signals declared locally and the previous effective values for the local signals which events affect the behavior of the local statements,
3. $O_e = O - \{\Phi(\texttt{dr_val(P,s)}) \in O \mid \texttt{s} \in S_d\}$: the outputs of the result abstract machine are variables that represent the current values of the drivers of signals that are not declared in the local declarative part,
4. the initial state s_{0_e} is equal to s_0 augmented with the initial values of the new variables introduced in 2,
5. the transition function NS_e is the product of NS with the atomic transition functions for the new state variables. The atomic transition function associated to a variable $\Phi(\texttt{p_eff(S)})$, for some local signal S, is defined by: $NS_{\Phi(\texttt{p_eff(S)})} = \Phi(\texttt{eff(S)})$. The atomic transition function associated to a variable $\Phi(\texttt{eff(S)})$, for some local unresolved signal S, which appeared as output O_i in $\mathcal{M}$ and whose associated output function was NO_i, is: $NS_{\Phi(\texttt{eff(S)})} = NO_i$. The atomic transition function associated to a variable $\Phi(\texttt{eff(S)})$, for some local resolved signal S, states that the next effective value is equal to the resolution function $\Re$ of S applied to the values of the n corresponding $\Phi(\texttt{dr_val(P,S)})$ output variables of $\mathcal{M}$:

$$NS_{\Phi(\texttt{eff(S)})}(I_e, S_e) = \Re(NO_1(I,S), \ldots NO_n(I,S)).$$

6. the output function NO_e is the function equal to NO where all the atomic output functions of the encapsulated output variables are removed.

Furthermore, the formula $\Phi(\texttt{eff(S)}) \neq \Phi(\texttt{p_eff(S)}))$ replaces each occurrence of a variable that represents $\Phi(\texttt{ev(S)})$ for some local signal S in the output and transition functions.

Architecture Body : In the case of an architecture body, the ports and signals of the corresponding entity declaration have also to be considered. The signal declarations of the entity declaration and the architecture body are encapsulated like the signal declarations that occur in a block declarative part (see above). Once these signal declarations have been encapsulated, the result abstract machine should have as remaining input and output variables those that correspond to the ports of the entity.

Let $\mathcal{M} = \langle I, S, O, s_0, NS, NO, sc\rangle$ be the abstract machine elaborated from the architecture statement part. Let P_{in}, P_{out}, P_{inout} be the set of ports of the entity declaration, according to their direction, and S_d the set of signals declared locally to the architecture and the entity. The encapsulation of all the above signals produces an abstract machine $\mathcal{M}_e = \langle I_e, S_e, O_e, s_{0e}, NS_e, NO_e, sc_e\rangle$, such that:

1. $I_e = I - \{\Phi(\texttt{eff(s)}) \in I \mid \texttt{s} \in S_d\} \cup \{\Phi(\texttt{ev(s)}) \in I \mid \texttt{s} \in S_d\})$,
   $\subseteq \{\Phi(\texttt{eff(s)}) \mid \texttt{s} \in P_{in} \cup P_{inout}\} \cup \{\Phi(\texttt{ev(s)}) \mid \texttt{s} \in P_{in} \cup P_{inout}\}$
   input variables of M_e represent the effective values and the events on the entity ports of mode **in** and **inout**, which are read in the architecture (ports which never appear in the architecture statement part are left out),
2. $S_e = S \cup \{\Phi(\texttt{eff(s)}) \in I \mid \texttt{s} \in S_d\} \cup \{\Phi(\texttt{p_eff(s)}) \mid \texttt{s} \in S_d \wedge \Phi(\texttt{ev(s)}) \in I\}$
   state variables are added to the result abstract machine to represent the effective values of all the signals declared locally and the previous effective values for the local signals whose events affect the behavior of the local statements,
3. $O_e = \{\Phi(\texttt{dr_val(A,s)}) \mid \texttt{s} \in P_{out} \cup P_{inout}\}$ the outputs of the result abstract machine are variables that represent the current values of the drivers of the entity ports of mode **out** and **inout**,
4. the initial state s_{0_e} is defined as for a block statement,
5. the transition function NS_e is defined as for a block statement,
6. the output function NO_e is the vector of the atomic output functions of the abstract machine outputs.
   For a port P, for which no output variable has been elaborated in machine $\mathcal{M}$ (the port is neither assigned nor the output of an internal component), the output function is $NO_{\Phi(\texttt{dr_val(A,P)})}(I_e, S_e) = \Phi(\texttt{E})$, where E is the initial value of P.
   For an unresolved port P, for which a single output variable has been elaborated in machine $\mathcal{M}$ and whose associated output function was NO_i, the output function in $\mathcal{M}_e$ is equal to NO_i.
   For a resolved port P, for which one or more output variables have been elaborated in machine $\mathcal{M}$ and whose associated output functions were NO_1,... ,NO_p, the output function in $\mathcal{M}_e$ is:

$$\Phi_{(\texttt{dr_val(A,P)})}(I_e, S_e) = \Re(NO_1(I,S), \ldots NO_n(I,S)).$$

As for block statements, the formula $\Phi(\texttt{eff(SIG)}) \neq \Phi(\texttt{p_eff(SIG)}))$ replaces each occurrence of a variable that represents ev(S) for some local signal S in the output and transition functions.

Component Instantiation Statements: The elaboration of a component instantiation statement directly derives from the previous section. The component is configured to an architecture A and an entity E. A copy of the abstract machine elaborated from design unit E(A) is brought to the statement part of the block or architecture in which the component is instantiated. All the local objects of the component instance are renamed by prefixing them with the label of the component instantiation statement.

Formal ports which are associated to an actual signal in the port map aspect are renamed with the name of the actual signal. Formal ports which are **open** in the port map aspect are renamed by prefixing the component formal name with the label of the component instantiation statement. For a formal port P of mode in which is associated to an expression E, all occurrences of the variable Φ(eff(P)) are replaced by the value Φ(E) of this expression, all occurrences of the variable Φ(ev(P)) are replaced by the value *false*, and these variables are removed from the inputs of the abstract machine.

Complexity of the Generated Abstract Machine: The kind of elaborated abstract machines is very likely to serve as a basis for BDD-based verification techniques. Therefore, a good measure to evaluate the complexity of the model is the number of boolean propositions needed to encode the different variables of the abstract machine.

A VHDL design unit with n in ports i_1,... i_n, p **out** ports o_1,... o_p, q internal signals s_1,... s_q, r process p_1,... p_r, each with v_i variable declarations p_i_v_1,... p_i_v_i and w_i wait statements, $(1 \leq i \leq r)$, elaborates a binary-encoded abstract machine such that[3]:

- The number of input variables is: $\sum_{k=1}^{n} \lceil \log |\texttt{i_k'TYPE}| \rceil + |\{\texttt{i_k} \mid trig(\texttt{i_k})\}|$. The first term of this sum represents the number of variables needed to encode the VHDL in ports, the second term is the number of variables needed to encode events on the in ports the processes of the design are sensitive to.
- the number of output variables is: $\sum_{k=1}^{p} \lceil \log |\texttt{o_k'TYPE}| \rceil$.
- the number of state variables is:

$$\sum_{k=1}^{q} \lceil \log |\texttt{s_k'TYPE}| \rceil + \sum_{trig(\texttt{s_k})} \lceil \log |\texttt{s_k'TYPE}| \rceil$$
$$+ \sum_{k=1}^{r} \lceil \log |w_k + 1| \rceil + \sum_{k=1}^{r} \sum_{j=1}^{v_k} \lceil \log |\texttt{p_k_v_j'TYPE}| \rceil .$$

The first term of this sum represents the number of variables needed to encode the VHDL local signals, the second term is the number of variables needed to encode the previous effective values of the signals to which the processes of the design are sensitive, the third term is the number of variables needed to encode wait counters, and the last term is the number of variables needed to encode the variables declared within process statements.

[3] We use the predicate $trig$(O) to say that a process statement is sensitive to an object O, the notation $|S|$ to represent the cardinality of the set S.

These are worst case figures. For instance, a process statement with a single final wait statement need not have a wait counter variable. This situation occurs quite frequently, since it corresponds to processes with sensitivity list and to concurrent signal assignments.

6 Conclusion

In this article, we have presented the semantics of a core VHDL that has the following theoretical restrictions with respect to the full language: types and executions must be finite, and no quantitative timing informations are allowed. Other restrictions in this paper (such as the non-inclusion of composite types) are not fundamental in nature and were made only to focus on the fundamental aspects of our approach. These semantics show how VHDL design units can be used to elaborate an abstract machine that can be considered as a symbolic model. This model is composed of a finite state machine where each transition captures one VHDL simulation cycle, and a stability condition that represents the states of the model that correspond to quiet states of the VHDL model. Under this form, the abstract machine behaves like a zero-delay reactive system. Hence, it is possible to verify strictly qualitative temporal properties (by means of symbolic model checking) or observational equivalence, at the level of delta cycles or at the level of quiet states. Under the current form, the model is applicable to synchronous as well as asynchronous circuits where propagation delays are abstracted to evaluation cycles.

In addition to an on-going realization of a symbolic model checker based on these semantics, it is our intention to investigate how this model can be used to study the reachability of stability conditions based on the stability of inputs in order to verify sequential circuits synchronized by one or more clocks. We have the intuition that a previous finite state machine model [4] which assumed strict restrictions on signals assigned under the condition of a clock pulse could be derived as a special case of the abstract machine model presented here. This could lead to a hierarchy of models where this model could serve as a validation of the simplifying assumptions made in the other one. More work is needed to bring formal justifications to the above statements.

Acknowledgements: This work has been partially supported by the CEC ESPRIT CHARME project N°3216 and the CHARME2 Working Group N°6018. The authors are grateful to their partners of CHARME for many enlightening discussions and fruitful co-operation over several years. They would also like to thank Prof. Ed Clarke and Marius Minea for useful comments on a previous presentation of this work.

References

1. C. Bayol, B. Soulas, F. Corno, P. Prinetto, and D. Borrione. A process algebra interpretation of a verification oriented overlanguage of VHDL. In *Euro-DAC*

with Euro-VHDL'94, pages 506–511, Grenoble, France, Sep. 1994. ACM/IEEE, IEEE Computer Society Press.

2. W. DAMM, B. JOSKO, and R. SCHLÖR. *Specification and Validation methods for Programming Languages and Systems*, chapter Specification and Verification of VHDL-based System-Level Hardware Designs, pages 331–410. Oxford University Press, 1995. E. Börger, editor.
3. K. DAVIS. A Denotational Definition of the VHDL Simulation Kernel. In *11th International Symposium on Computer Hardware Description Languages and their Applications*, pages 509–521, Ottawa, Canada, 1993.
4. A. DEBREIL and D. JAILLET. Synchronous description in VHDL for formal proof and resulting guidelines proposed by BULL. Advanced report, BULL Produits Systèmes Département Développement Assisté, rue Jean Jaures-B.P. 68-Les Clayes-sous-Bois-France, July 1992. BULL/92.0001 rev.A.
5. C. DELGADO KLOOS and P. BREUER, editors. *Formal Semantics for VHDL*, volume 307 of *Series in Engineering and Computer Science*. Kluwer Academic Publishers, Boston, March 1995.
6. R. HERRMANN and H. PARGMANN. Computing binary decision diagrams for VHDL data types. In *Euro-DAC with Euro-VHDL '94*, Grenoble, France, Sep. 1994.
7. IEEE. *IEEE Standard VHDL Language Reference Manual*, 1993. Std 1076-1993.
8. B. LEVY, I. FILIPPENKO, L. MARKUS, and T. MENAS. Using the State Delta Verification System for Hardware Description. In V. Stravidou, T. Melham, and R. Boute, editors, *Theorem Provers in Circuit Design*, pages 337–360, Nijmegen, Netherlands, June 1992. IFIP A-10, North Holland.
9. S. OLCOZ and J. COLON. A Petri net approach for the analysis of VHDL descriptions. In G. Milne and L. Pierre, editors, *Correct Hardware Design and Verification Methods*, volume 683 of *Lecture Notes in Computer Science*, pages 15–26, Arles, France, May 1993. ESPRIT WG 6018 and IFIP WG 10.2, Springer Verlag.
10. A. SALEM and D. BORRIONE. *VHDL for simulation, synthesis, and formal proofs of hardware*, chapter Formal semantics for VHDL timing constructs. Kluwer international series in engineering and computer science. Kluwer Academic Publishers, 1992. J. Mermet editor.
11. J. VAN TASSEL. A formalisation of the VHDL simulation cycle. Technical Report 249, University of Cambridge, Cambridge, March 1992.