Speeding up Test Pattern Generation
from Behavioral VHDL descriptions containing several processes

Loïc Vandeventer, Jean-François Santucci

Laboratoire d'Etude et de Recherche en Informatique
EERIE, Parc Scientifique G. BESSE, 30000 Nîmes -FRANCE
Tel : (33) 66 38 70 33, fax : (33) 66 84 05 06, Email : {vandeven, santucci}@eerie.fr

Abstract
In this paper, we focus on behavioral VHDL circuit descriptions modelled by several processes. Two test pattern generation approaches from such descriptions have recently been proposed in which a search algorithm leads to the use of a backtracking procedure. In order to speed up the search for test patterns in these approaches, we have developed two algorithmic improvement methods which reduce the number of backtracks and shorten the processing time between backtracks. Two pedagogical examples highlight the benefits of using each method.

1: Introduction
Nowadays, VLSI circuit are described by VHDL descriptions at different abstraction levels, according to either their structure or their behavior [1]. New test pattern generation approaches have been developed to deal with behavioral circuit descriptions [2,3,4,5,6,7]. Most of these approaches cannot deal with concurrency, are devoted to low abstraction levels or are not dedicated to VHDL.

For behavioral VHDL circuit descriptions containing several processes, two behavioral test pattern generation approaches have been proposed in the recent past [8,9]. These two approaches are based on the path sensitization method which allows test pattern generation to be decomposed into three phases: fault manifestation, fault effect propagation and constraint justification. The search for test patterns is performed at two levels in the VHDL descriptions:
- at a level internal to a process: when fault manifestation, fault effect propagation and constraint justification are performed on the statements inside a process,
- at a level external to the processes: when the process activation and the simulation cycle management are performed outside the processes, on the input/output signals of the process interconnection.

In the two behavioral test approaches, the search for test patterns is organized at the two levels by a depth-first algorithm which involves making choices. Because of these choices, conflicts may occur at the two levels, that is, inside a process and outside the processes. A backtracking mechanism is needed to make other choices in the case of conflicts.

An important property of the search procedure is that it is exhaustive: it guarantees finding a solution if one exists. During the search for a test pattern, problems are solved locally by reduction on the circuit modelling items. This leads to making choices and choices can become contradictory, which results in the emergence of conflicts. Because of the exhaustive nature of the search procedure, the number of operations performed to find a solution is an exponential function of the number of elements involved in the circuit description.

In order to avoid a combinational explosion, the managing of conflicts by backtracking imposes a backtrack limit which is a parameter of the test pattern generation results. Indeed, if the limit is too great large, a great number of faults will be aborted and if the limit is small, too much time will be needed to generate the test sequences. In order to accelerate the search for test patterns, it is necessary to reduce the number of backtracks and to shorten the processing time between backtracks [10,11], at both search levels. Two types of improvement methods exist: algorithmic and heuristic. Heuristic improvements speed up the search procedure in the majority of cases but in specific instances may never lead to a solution. Algorithmic improvements (also called deterministic) lead definitively and with certainty to a solution in all cases. Algorithmic improvements are therefore more valuable for test pattern generation. It would be wise however to dispose of powerful heuristic criteria in order to guide the test pattern generation[10,11].

In this paper, we only focus on speeding up the test pattern search level which performs fault manifestation, fault effect propagation and constraint justification inside a process. We then propose two algorithmic improvement methods. In order to reduce the number of backtracks, the first method is based on the classification of the conflicts which can occur inside a process. A conflict type is then chosen to be reduced first, according to the path sensitization generation principle and an improved generation principle is described which annihilates the occurrence of this conflict type. The second method, stemming from the gate level structural test domain, shortens the processing time between the backtracks inside a process. In order to illustrate these two improvement methods, we give for each a step-by-step
test pattern generation on two pedagogical examples which highlight the benefits of using the improvements.

The remainder of the paper is organized as follows. Section 2 presents the contribution of the two approaches concerning test pattern generation from behavioral VHDL descriptions containing several processes. Section 3 is devoted to the improvement which reduces the number of backtracks inside a VHDL process and concludes with a pedagogical example. Section 4 discusses the shortening of the processing time between backtracks inside a process and is illustrated by a second pedagogical example. Finally, a conclusion regarding our approach and a presentation of our future work concerning behavioral test pattern generation are given.

2: Behavioral test pattern generation for VHDL circuit descriptions containing several processes

In this section, we discuss the main concepts involved in the two works concerning behavioral test pattern generation from VHDL descriptions containing several processes [8,9]. We briefly evoke the contribution of each author in the areas of circuit and fault modelling and test pattern generation principle.

2.1: Circuit modelling

[8] uses the textual circuit description and converts it into Prolog predicates which represent the basic information of the description: statement types, object names, control and assignment clauses, and so on. [9] represents the descriptions with the help of two explicit models based on graph modelling concepts. The first is an input/output model which expresses the links which exists between the inputs/outputs of the different processes involved in the description and which uses the finite state machine representation for the processes. The second is an activation model, stemming from Petri nets, which expresses the different phases involved in the execution of processes, namely the initialization phase and the simulation phase. Moreover, an explicit representation of each process is made by associating a process with its inside behavior modelled by two graphs representing the control and data flows.

2.2: Fault modelling

In both approaches, the authors propose a fault modelling based on the failure modes of the basic elements of the description language. [8] introduces the term of perturbation model and lists the perturbations which may occur in a VHDL description. [9] refers to [12] who defined and validated a behavioral fault modelling scheme based on the basic elements of the programming language constructs.

2.3: Generation principle

The two authors use a test pattern generation method stemming from path sensitization according to a local resolution approach. A fault hypothesis is injected into one of the processes of the description, called the target process. Two phases are needed to build the sequence on primary inputs of the description in order to highlight a fault effect on a primary output. The first phase consists in solving within the target process the problems of fault manifestation, fault effect propagation to a target process output, and justification of the constraints set up during manifestation and propagation on target process inputs.

Once the first phase has been successful, a fault effect must be propagated to a primary output of the description and constraints satisfied on primary inputs. [9] uses the input/output graph in order to determine a propagation path and a justification path which both correspond to a set of processes to be crossed. This sets up the activation constraints on the selected processes.

Fault effect propagation and constraint justification are thus carried out at the two following levels:
- outside the processes, as far the selection and activation of the processes on the propagation and justification paths are concerned,
- inside a process, when a fault effect is located on a process input and must be propagated to a process output and when constraints are set up on the process outputs and must be satisfied on the process inputs.

[8] and [9] implement the generation principle with the help of AND/OR graphs, in which problems are reduced into sub-problems, in order to organize the search for the test sequences [13]. The search for a test pattern is performed by conventional depth-first algorithms which reduces the AND/OR graph to a solution tree. During the local resolution of problems, choices are made and conflicts may emerge. A backtracking procedure must be included to go back on these choices in case of failure. In order to avoid a combinational explosion, a backtrack limit must be imposed. Faults which have not been determined as detectable or undetectable are aborted.

The search for a test sequence is processed as follows: the reduction tree is initialized by the fault manifestation. The search procedure builds the AND/OR tree by reducing problems into sub-problems until one of the three following cases is encountered: (i) a solution is found, (ii) the backtrack limit is reached, (iii) all reductions have been performed within the backtrack limit but no solution is found. In the first case, a test sequence is found. In the second case, the fault is declared aborted. In the third case, the fault is declared redundant.

Both the approaches presented above allow test patterns to be generated for VHDL descriptions containing several processes. The generation principle used, stemming from path sensitization according to a local resolution approach, necessitates the management of conflicts by backtracking. In this paper, we propose two algorithmic improvement principles the aim of which are to speed up the fault effect propagation and the constraint justification resolution inside a process. The first one reduces with certainty the number of backtracks to be dealt with during the propagation and justification inside a VHDL process. The second one shortens the processing time between backtracks inside a process. The next two
sections are devoted to the presentation of these two improvement principles.

3: Reducing the number of backtracks

This section presents the improvement method the aim of which is to reduce the number of backtracks which may occur during fault effect propagation and constraint justification inside a VHDL process. We first present the methodology we followed to reduce the number of backtracks and then we explain the improvement principle. An improved generation principle can be set forth. At the end of the section, a pedagogical example is given to illustrate the efficiency of this improvement.

3.1: Methodology

In order to reduce the number of backtracks, it is necessary to reduce the number of conflicts the search procedure may encounter. Our approach is based on the classification of the conflict types which may occur during the propagation and justification inside a process. Afterwards, we determine a conflict type to be reduced first. We study the causes of this conflict type and develop an algorithmic method which will annihilate the number of conflicts of this type. This number is then reduced to zero. The number of conflicts the search procedure will have to deal with is then reduced and the backtracks will be made only in the case of conflicts of another type.

3.2: Improvement principle

In order to classify the conflicts, we first determine a classification of the variables involved inside a VHDL process. We here call a variable, any variable of the process or any signal of the description which is used inside the process. In a VHDL description, as in any programming representation, the language constructs determine control paths of the description. Some variables are present in the condition of these language constructs and the condition evaluation allows a control path to be selected and to be traversed. It must be noticed that from among all these control paths, some may never be traversed because they do not correspond to a consistent combination of values on the process inputs. We call such a path a non-executable path. Otherwise, when a path corresponds to a consistent combination of values on the process inputs, we speak of an executable path.

We call control variables those variables inside a process which are involved in the selection of a control path. Control variables are therefore those variables present in the conditions of the language constructs and those variables which are used in the assignments of the control variable present in the conditions. We call a data variable a variable which is not a control variable. It should be pointed out that a same variable can be a control variable of a path and a data variable of another path.

The conflict classification is made with the help of these two types of variables. Therefore, three conflict types exist: control conflicts, data conflicts and control-data conflicts. The first two correspond to the emergence of an inconsistency on a control variable and on a data variable. The third is much less frequent and may only occur on variables which are control variable of a path and data variables of another path.

By taking the test pattern generation principle described above, control conflicts must be dealt with first. Indeed, since the values of control variables are representative of a specific control path which is traversed, the instructions of data variables to be dealt with by the search procedure depend on the control variables. The causes of control conflicts are the following:

- during the search for a solution, the search procedure sets up control values which do not correspond to an executable control path. Consequently, a control conflict will appear.

- during the search for a solution, the control values set up by the search procedure do not correspond to a single path but to several. Consequently, these control values will become inconsistent.

In order to prevent control conflicts from emerging, it is necessary to first prevent the search procedure from traversing non-executable paths of the process and, then, to compel the search procedure to set up control values corresponding to the same path. These two requirements are met if the search procedure is forced to deal only with a given path of the process and if this path is executable.

3.3: Improved generation principle by annihilating the control conflicts

An improved generation principle can be set forth with takes into account the method of control conflict annihilation.

Control paths of the process must first be classified into executable and non-executable paths. It is necessary to determine the sets of control values which correspond to each executable path. This knowledge is then given to the search procedure which will perform the search for a solution on each executable path in the way we describe below. Two configurations may be encountered: either the process is the target process or it is not. In the first configuration, a fault is injected into the process and constraints must be satisfied on the process inputs in order to highlight a fault effect on one of the process outputs. The second configuration corresponds to a fault effect propagation problem or a constraint justification problem set up by the outside mechanism and to be performed inside the process under consideration.

In the first configuration, it is necessary to determine the subset of executable paths on which the injected fault is located. We call generation paths of a given fault this subset of executable paths. If a fault has no generation path, that is, the fault is located only on non-executable paths, a solution will never be found and the fault will be declared redundant. For the other faults, the search procedure will try the generation paths of the fault one after the other. On each generation path, the search procedure will determine if a solution has been found, if the backtrack limit has been reached or if all reductions have been performed within the backtrack limit without finding a solution. In the second and third cases, the
search procedure will continue trying paths. At the end, three cases are possible: (i) a solution is found on a generation path, (ii) all generation paths have been traversed within the backtrack limit and no solution has been found, (iii) all generation paths have been traversed, no solution has been found and the backtrack limit has been reached at least once. In the first case, a test has been found. In the second case, the fault is declared redundant and, in the third, the fault is aborted.

In the second configuration, when fault effect propagation or constraint justification must be performed inside a process which is not the target process, the improved generation principle consists in traversing the executable paths one by one until the propagation or the justification is successful through the process. If no executable paths allow the search to be successful, the mechanism which deals with the external search level (outside the processes) will have to manage the failure in the same way as it would if propagation and justification were unsuccessful without the improvement.

### 3.4: Pedagogical example

We now propose to illustrate the method of annihilation of control conflicts using an example of VHDL behavioral description, given in Figure 1.

```
Entity Example_1 IS
PORT(clk : IN bit;
e1,e2,e3 : IN bit;
e4,e5,e6: IN integer;
Sout : OUT integer);
END Entity Example_1;

ARCHITECTURE EX1 of Example_1 IS
BEGIN
process_1 : PROCESS
BEGIN
wait until clk='1';
a := e1 or e2 ; --#1
b := e1 and e3 ; --#2
if a='0' then
  if b='0' then z := e4 ; --#3
  else z := e5 ; end if; --#4
else c := e2 nand e3 ; --#5
if c='1' then z := e4 + e5 ; --#6
else z := e4 * e5 ; --#7
end if; --#8
s <= z * e6 ; --#8
END PROCESS process_1 ;

example : PROCESS
(a => e1 or e2;
 b => e1 and e3;
 c => e2 nand e3)
END Entity EX1;
END ARCHITECTURE;
```

Fig. 1. A VHDL description containing a non-executable path

This description contains two processes, but the process example only is described. In the process example, there are two if-then-else constructs imbricated which create 4 control paths. In Figure 2 gives the path predicates corresponding to each path, the corresponding assignment and the inputs values which allows the paths to be executed. Path p^n2 is not executable since it is not possible to have a input combination of values which gives simultaneously a=0' and b=1'. In contrast, path p^n1, 3 and 4 are executable and can be followed to generate test patterns. Moreover, the faults hypotheses which only concerns the statements located on path p^n2 are redundant since this path is non-executable.

```
Path p^n1     Path p^n2     Path p^n3     Path p^n4
a=0', b=0'   a=0', b=1'   a=1', c=1'   a=1',c=0'
a=e1 or e2   a=e1 or e2   a=e1 or e2   a=e1 or e2
b=e1 and e3   b=e1 and e3   b=e1 and e3   b=e1 and e3
c=e2 nand e3  c=e2 nand e3  c=e2 nand e3  c=e2 nand e3

Executable    Non-executable    Executable    Executable
```

Fig 2. Executable and non-executable paths

Once the executable paths of the process are determined, the improved generation principle can be used to speed up the propagation and the justification inside the process. We now propose to examine the case where the process example is the target process and the behavioral high-level fault "the signal Sout is permanently at the value 0". The three executable paths are generation paths of the faults. The three steps of path sensitization are solved as follows:

- **First step : Fault manifestation**: We generate a sequence on the process inputs in order to highlight a fault effect on the process output. The value 5 allows the fault to be manifested on s. The fault manifestation constraint is [Sout=5] and the fault effect to be propagated is [Sout=(5 0)]. Moreover, a process activation constraint on the signal clk would be set up. We don't care about since it is dealt with outside the process.

- **Second step : Fault effect propagation**: This step consists in propagating the fault effect to a process output. In this example, there is one process output, Sout. This step is solved since a fault effect has been set up on Sout by the fault manifestation.

- **Third step : Constraint justification**: This last step consists in solving the constraint set up during the two previous steps. In our example, only the constraint [Sout=5] must be justified on the process inputs. Thanks to the improvement, the search procedure chooses only one of the three executable paths of the process to justify the constraint [Sout=5]. Let us suppose the path p^n4 is chosen first. Control values corresponding to path p^n4 are \([e1='1',e2='1',e3='1',a='1',c=0']\) or \([e1='0',e2='1',e3='1',a='1',c=0']\). We suppose the first set is chosen first. On the assignment a', the constraint [Sout=5] can be solved in different ways. Let us suppose it is solved by the set of constraints \(z=e, e6=1\). In the if-then-else constructs, the assignment \(z=5\) is directly chosen according to path p^n4. The constraint [Sout=5] is then solved on \(z=5\). We obtain, for example, \(e4=0,e5=5\). These constraints are solved since e4, e5 and e6 are process inputs. Control values corresponding to path p^n4 are solved too.

Figure 3 shows a test pattern for the fault under consideration.

```
e1 e2 e3 e4 e5 e6 Sout
0 1 1 0 5 1 0 5 0
```

Fig. 3. Test pattern of the fault under consideration

We now present a second algorithmic improvement.

### 4: Shortening the processing time between backtracks

This section describes a second algorithmic improvement the aim of which is to shorten the processing time between backtracks. The improvement principle stems from the well known concept of "Headlines", defined by [10] in the field of test pattern generation for structural circuit descriptions at the gate level. We propose an analogy of this structural test improvement to the behavioral test domain in order to speed up the fault effect propagation and the constraint justification inside a VHDL process.

#### 4.1: Improvement principle

The aim of this improvement is to find items of a VHDL process description on which constraint
justification inside this process can be performed without backtracking. Constraint justification problems can be stopped on such items and postponed to a final step of the test pattern generation in order to first solve all the conflicts which causes backtracks. Such items will be called stop-nodes. This improvement can be described as the determination of the "parts" of the VHDL process description where constraint justification is performed without backtracks and of a "part" of the VHDL process where all backtracks are "located". The search procedure dwells in this latter "part" in order to deal with all conflicts to be solved by backtracking, until a consistent set of constraints is set up on a subset of process inputs and on stop-nodes. Afterwards, test sequences are obtained by justifying the constraints set up on the stop-nodes.

4.2: Stop-node determination in VHDL processes

In a behavioral description, conflicts are caused by the variables which generate a divergence. We say that a variable generates a divergence when this variable is used in the right part of two or more assignments in the description. Therefore, the stop-nodes are those variables of the description which proceed from no divergence. In a VHDL process, stop-nodes will be those variables which proceed from no divergence according to the assignments involved in the process.

In order to determine stop-nodes in a VHDL process, we define the notions of predecessors and successors of a variable. These definitions are based on the presence of the variables in the right and left parts of the VHDL process assignments. We call predecessors of a variable $V$ the set of variables which are in the right part of the instructions which assign the variable $V$. We note $\text{Pred}(V)$ this set. Consequently, the VHDL process inputs have no predecessors and can therefore be assimilated to stop-nodes. We call successors of a variable $V$ the set of variables for which the variable $V$ is present in the right part of the assignments of these variables. The process outputs thus have no successors. Therefore, a variable of the VHDL process is a stop-node if each of its predecessors is a stop-node having only one successor. The definition of stop-nodes is highlighted in Figure 4. Using this definition, the set SN of stop-nodes can easily be determined in the VHDL processes.

Let $V$ and $W$ be two variables and $I$ the set of process inputs

$$\text{SN} = \{ V / \forall W \in \text{Pred}(V), W \in \text{SN} \cup \text{I and Card(Succ(W)) = 1} \}$$

Fig. 4, Definition of stop-nodes

The VHDL language constructs determine the control paths of the description. Divergences caused by some variables depend on which control path of the process is traversed. It is therefore necessary to determine the stop-nodes corresponding to each path of the VHDL process. The definition of stop-nodes is then applied on the set of assignments located on the path under consideration.

4.3: Improved generation principle with the help of stop-nodes

Using the stop-nodes, a new improved generation principle can be set forth. This principle concerns the justification and propagation inside a target process as well as the justification or the propagation inside a process which is not a target process.

The new improved generation principle is the following: after the stop-nodes of the VHDL process under consideration have been determined, the search procedure may start in order to find a solution in fault effect propagation or constraint justification inside the VHDL process. When a constraint is set up on a stop-node, the search procedure will consider this constraint as solved, at first. The solution the search procedure must find is a consistent set of constraints set up on stop-nodes and on several inputs of the VHDL process. Backtracks will be performed until the solution is found. Three cases can be encountered: (i) a solution is found, (ii) the backtrack limit has been reached, (iii) all reductions have been performed within the backtrack limit but no solution is found. In the third case, the fault is declared redundant and, in the second case, it is aborted. In the first case, a test sequence is found for the fault by justifying the constraints set up on the stop-nodes.

We have a second algorithmic improvement for justification and propagation inside a VHDL process. Propagation is concerned insofar as it consists in setting fault effect propagation constraints which must be justified afterwards. This improvement shortens the time between backtracks.

4.4: Pedagogical example

As was the case for the first improvement presented in this paper, we propose to illustrate the second improvement by means of a pedagogical example. This is carried out on the VHDL description we wrote, which is given in Figure 5. In this VHDL description, only $\text{Process}_3$ is described. We determine the stop-nodes contained in $\text{Process}_3$. This is presented in Figure 6.

```vhdl
Entity Example_2 IS
  ... 
END Entity Example_2;

Process_3 : PROCESS
  ... 
END PROCESS Process_3;
```

Fig. 5, a VHDL description containing stop-nodes

$\text{Process}_3$ contains two variables which provide divergences, $d_3$ (a process input) and $data4$ (an internal variable). $\text{Process}_3$ also contains three stop-nodes: $data1$, $data3$ and $data4$.

Let us now consider the behavioral high-level fault "In the assignment #7, the signal $Sout$ is permanently at the value 0". The three generation steps are:

- **First step:** Fault manifestation: The value 1 allows the fault to be manifested. We then obtain a fault manifestation constraint [$Sout=1$] to be justified and a fault effect [$Sout = (1 0)$] to be propagated.
- **Second step:** Fault effect propagation: This second step is solved since $Sout$ is an output of $\text{Process}_3$. 

On the assignment, we obtain the justification step is successful.

In VHDL behavioral circuit descriptions containing improvements which speed up the search for test patterns.

### 5. Conclusion and future work

In this paper, we have presented two algorithmic improvements which speed up the search for test patterns in VHDL behavioral circuit descriptions containing several processes. The first improvement reduces the number of backtracks and the second shortens the processing time between backtracks. For each, we have presented an illustration using a pedagogical example on which the benefits of the improvements are highlighted.

These two algorithmic improvements have been implemented in an existing behavioral test pattern generator. Experiments have been carried out in order to validate our approach. (14,15).

Our future work will focus on the determination of algorithmic or heuristic improvements which will speed up test pattern generation at the level of the input/output signal interconnection of the description processes.

### References


---

**Fig. 6, Determination of stop-nodes**

- Third step: **Constraint justification**: Without the improvement, and for any given fault, the search procedure would have to set up constraints on the six process inputs. Since there are divergences in the process, conflicts may arise during the setting up of constraints on the internal variables. In this case, the search procedure has to backtrack until these six constraints are consistent, corresponding to twelve consistent constraints set up on the internal variables and the process inputs.

With the improvement, the constraint justification is stopped on the stop-nodes. For the fault under consideration, the first stop-nodes the search procedure will encounter are data1 and data4. Therefore, the search procedure must only set up three consistent constraints on data1, data4 and d3, which is the only process input which provides divergences. In order to set up these three consistent constraints, the search procedure must set up consistent constraints on the following variables: data5, data6, data4, data2 and data1. This corresponds to only six consistent constraints to be set up by the search procedure instead of twelve without the improvement.

We now justify the manifestation constraint [Sout=1] on each assignment of the process.

On the assignment #7, we obtain [data5=0, data6=1].

On the assignment #6, we obtain [data5=0, data2=1, data4=1].

On the assignment #5, the search process will try different constraint sets, such as [data4=0, d3=0], and backtracks until it sets up [data6=0, d3=1] which is consistent. We thus obtain [data4=1, data2=1, d3=1].

Since data4 is a stop-node, the assignments #4 and #3 are skipped in justification. The search procedure will perform the justification of data4 later. In this way, the search procedure quickly deals with the assignment #2 on which conflicts may arise since d3 provides divergences. Since the search procedure may deal with conflicts more quickly, the time between backtracks is thus shortened. On this assignment, the search procedure will try constraint sets until it finds [data1=1, d3=1]. We thus obtain [data1=1, d3=1, data4=1] and the justification step is successful.

We now justify the constraints set up on the stop-nodes. We obtain [d1=0, d2=1, d3=1, d4=1, d5=1, d6=0] which is a test pattern for the fault under consideration.

**5. Conclusion and future work**

In this paper, we have presented two algorithmic improvements which speed up the search for test patterns in VHDL behavioral circuit descriptions containing...