Use of Constraint Solving in order to Generate Test Vectors for Behavioral Validation

Christophe Paoli Marie-Laure Nivet Jean-François Santucci
SPE – Systèmes Physiques pour l’Environnement, URA CNRS 2053, University of Corsica, Quartier Grossetti, Bat 018, BP 52, 20250 Corté, France.
{cpaoli,nivet,santucci}@univ-corse.fr

Abstract

Validation of VHDL descriptions at the early phases of the microelectronic design is one of the most time consuming task design. This paper presents a test vector generation method for behavioral VHDL design. This method analyzes control and dependence flow of VHDL program. We use the cyclomatic complexity, that is a software metric based on a graph associated with the control part of software: the control flow graph (CFG). Significant control flow paths are selected using a powerful algorithm: the Poole’s algorithm. The execution of this set of paths satisfies the coverage of each decision outcome of the VHDL program. Any additional test path would be a linear combination of the basis paths already tested and therefore considered to be redundant. By considering the selected paths as a group of constraints, test data are generated and solved using constraint programming. These data form the test bench that test the VHDL description.

1. Introduction

This paper deals with the validation of VHDL[1] descriptions at the early phases of the microelectronic design. The aim is to verify that a description has the intended behavior. We assume that the designer generates behavioral test patterns at a high abstraction level to verify the correctness of a corresponding RTL description. This task can be accomplished through two approaches: formal verification and simulation based validation. Formal verification that proposes to mathematically prove the correctness of a description is only feasible for small descriptions. In simulation based validation, one of the main problems is to generate test vectors in order to verify design specifications. Whereas exhaustive simulation is infeasible for actual design, random vector generation is relatively easy but does not guarantee the verification of all the functionalities of the design. A different way to generate test vectors is based on metrics derived from software engineering concepts[2]. Several works on this topic have been published. In [3,4] test pattern generation considering a fault model is proposed. This approach is based on bit coverage instead of selective path[5,6,7] or branch coverage[8]. These methods suffer to the fact that it is not possible to measure the degree of confidence of the fault model.

We propose a method for generating design verification tests in order to execute control flow paths selected with software testing concepts[9,10,11]. We decided to use software engineering methods because a VHDL description is a software program. In addition, hardware description languages and conventional languages such as C or ADA are supported by common concepts[9]. We use a path coverage based criterion because it is more rigorous and more effective[9] at detecting errors than other common test coverage criteria (statement or branch coverage). Furthermore contrary to [12,13], the number of paths that has to be executed is limited according to the path coverage based criterion issued from [9].

We describe in the Figure 1 a framework for deriving test benches. The grayed box represents the test data generator based on software testing concepts presented in this paper. The proposed method is based on VHDL program control and dependence analysis, path generation, constraint generation and constraint solving. The test generation process presented can be divided in three sections.

* A part of this research is founded by EOARD (European Office of Aerospace Research and Development) under the contract number F61775-00-C0002 ; Project 00-4005 “Validation of VHDL descriptions” (2000-2003).
Analysis of the control of VHDL program and generation of control flow paths are presented in section 2. Analysis of the dependence between statements is discussed in section 3. Section 4 is dedicated to constraints generation and constraints solving for test vectors generation.

2. Analysis of the control of VHDL program and generation of control flow paths

In this part we briefly present how the control flow part of a VHDL description is modeled using graph concepts. These concepts are used: (i) to represent the sequencing of operations involved in a VHDL description and (ii) to generate the different execution paths using software techniques.

2.1. Graphical representation of control flow

Control flow graphs (CFG) describe the control structure of software programs. Each flow graph consists of nodes and edges. The nodes in the CFG are program’s statements (assignment and control) while the directed edges represent the transfer of control between nodes. This graphical description helps to understand complex algorithms.

The Figure 2 shows the control flow graph of a VHDL program where nodes are circles and edges are arrows. The node noted process node models the concurrent statements (communicating processes or signal assignments) encountered in VHDL programs. This node owns as much outgoing edge than there are processes plus one edge that allows to model the suspension of all the processes.

CFG uses the elements shown in Figure 3 which are control flow sub-graphs of control structures (assignment instruction, selective or repetitive instructions) involved in VHDL descriptions.

2.2. Control flow paths generation

The cyclomatic complexity[10] is a software metric based on the CFG. This complexity represents the number of linearly independent paths of the CFG. It is a function of the number of nodes and edges. This can be calculated as equal to e-n+2, where e is the number of edges and n is the number of nodes. A path is defined as a selection of traversed edges from the first to the last node. Path construction is defined as adding or subtracting the number of time each edge is traversed. The structured testing criterion defined by McCabe[9] limits the number of paths that have to be executed in order to test the control part of software. This criterion is simply stated: “Test a basis set of paths through the CFG”. A basis set is defined as a set of paths through the CFG that are linearly independent. These paths can be used to construct any other existing paths. Obviously, several basis sets can be obtained. The execution of this set of paths satisfies the coverage of each decision outcome of the VHDL program. Any additional test path would be a linear
combination of the basis paths already tested and therefore considered to be redundant.

In our approach, the generation of a basis set of test paths is based on a powerful algorithm: the Poole’s algorithm[11]. This algorithm is a modified depth-first search algorithm. The search starts at the source node and recursively descends down all possible outgoing paths. A source node is a node which has no incoming edges. The basis generated with such an algorithm is called primary basis in the following. The Figure 4 shows the results obtained by the algorithm on the VHDL program of the Figure 2. The edges are numerated from 1 to 10 and the paths are noted from P1 to P4.

Cyclomatic complexity = e - n + 2
10 - 8 + 2 = 4

Primary basis:
P1 - 1.10
P2 - 1.2.3.9.10
P3 - 1.4.5.6.8.9.10
P4 - 1.4.7.8.9.10

Figure 4. Cyclomatic complexity and Poole’s algorithm

3. Analysis of the dependence and test paths generation

Behavioral VHDL programs contain multiple communicating concurrent processes, concurrent signal assignments which are not found in traditional software programming languages. These VHDL features imply that a given path of the primary basis can be infeasible for one of these two reasons:

- The description implies strongly connected processes[14]. Two processes are strongly connected if all the signals of the sensitive list of the first process are outputs of the other process. The VHDL program of the Figure 2 illustrates this case.

- A variable or an internal signal involved in the control statement of the given path is assigned in a statement executed in an other path. The Figure 5 illustrates this case.

We propose in this section to solve the problem of infeasible paths building and analyzing a flow dependence graph (FDG).

3.1. Construction of the FDG

The FDG is defined on the same nodes as the CFG. The edges of the FDG represent the dependencies among the program’s statements. The meaning of the dependence is different the one that is used in the subsection 2.2. Here we are talking about dependence tied to variables and signals affectionations while in part 2.2 the dependence has familiarity with linear algebra. We use the following definition. A node A is dependent on an other node B if the object (a variable or a signal) assigned in B can be used in the node A. In the figures 5, the dependence between two nodes is illustrated with a dashed arrow. The signal a affected in the node a <= in1 is used in the node if a='0'.

![Figure 5. CFG and FDG of a VHDL program](image)

3.2. Analysis of the FDG

As seen before control flow paths can be infeasible for two reasons. We propose in this sub-part to solve the problem of infeasible paths analyzing the FDG. An infeasible path is a path which owns a control node that is dependent on an assignment node. The analysis of the FDG allows to know which path corresponds to the execution of this assignment. This path allows the execution of the infeasible path either by changing the primary basis (case of strongly connected processes) or by scheduling the paths (other cases).

3.2.1. Generation of the final basis. In the case of two strongly connected processes, we have to replace each infeasible path of the primary basis. The new basis is achieved according to the following process:

a) We analyze the FDG in order to find a path P of the basis which corresponds to the assignment tied to the infeasible path.

b) We construct a new path as a linear combination of the path P and the infeasible path.

c) We replace in the primary basis the infeasible path by the path obtained in the previous step. The new basis is called final basis.
Obviously, if there is no strongly connected processes in the VHDL description the primary basis remains unchanged.

3.2.2. Path scheduling. This case corresponds to VHDL descriptions without strongly connected processes. We have to find a path of the basis that allows the assignation to the right value of the signal or variable involved in the control statement of the infeasible path. This path will have to be executed before the infeasible path. An example of scheduling conditions of paths is shown in the figure 6.

<table>
<thead>
<tr>
<th>n° Path of the final basis</th>
<th>n° Path to execute before scheduling conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>-</td>
</tr>
<tr>
<td>P2</td>
<td>-</td>
</tr>
<tr>
<td>P3</td>
<td>P2</td>
</tr>
<tr>
<td>P4</td>
<td>P2</td>
</tr>
<tr>
<td>P5</td>
<td>P3</td>
</tr>
<tr>
<td>P6</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure 6. Scheduling conditions

In this example the first column corresponds to the final basis obtained in the step before. In front of each path of this first column there is the path detected with the FDG that has to be executed before. The path noted P5 must be preceded by the path P3, which one must be also preceded by the path P2.

For each infeasible path of the basis, we have to find the list of paths which have to be executed before. It is obvious that for feasible path, there is no scheduling condition. The scheduled path lists are obtained according to the following process:

a) For each path of the basis, we create a list composed of the path itself.

b) For each path we look at the scheduling conditions. We put the path that has to be executed before at the beginning of the list.

c) If this path is a infeasible we repeat the previous step, else the scheduled path list is generated.

The following scheduled path lists correspond to the example of the Figure 6:

((P1) / (P2) / (P2 / P3) / (P2 / P4) / (P2 / P3 / P5) / (P6))

4. Constraints generation and constraints solving for test vectors generation

In this section, we present how we use constraint generation and constraint solving in order to generate test vectors. We have been interested in expressing each statement of each path of the final basis in terms of constraints respecting the GNU prolog constraint programming language[15]. This process is presented in sub-section 4.1. The sub-section 4.2 is dedicated to the determination of the input data and their resulting values using the constraint solver over finite domains included in [15].

4.1. Constraints generation

To carry out the constraints generation we proposed to use quite the same method presented in[16]. This method treats spatial and temporal incarnation of VHDL signals and variables as unique constraint variable. Constraint generation process takes each path of the final basis (represented as a succession of edges) as its input and generates a list of constraints in the language of the constraint solver. They are two steps to generate constraints files. The first step is to defined the domain constraints and the second one is to treat relational constraints.

- Writing domain constraints consists in translating each type declaration of VHDL signals and variables into the suitable domain constraint variable declaration. So any declaration in VHDL results in at least one domain constraint. This domain contains all possible values for the variable. The figure below illustrates this process.

Entity example is

```
port (in1, in2 : in bit;
out1 : out bit);
end example
architecture behavior of example is
signal a:bit;
```


in1 ∈ \{0,1\}  
in2 ∈ \{0,1\}  
out1 ∈ \{0,1\}  
a ∈ \{0,1\}

```
```

Figure 7. Writing domain constraints

- Relational constraints correspond to all VHDL statements that define each path of the final basis. These statements could be variable or signal initializations, assignments, Boolean expressions or VHDL specific statements like process statement.

We don’t treat variables as signals. As matter of fact variables and signals have not the same behavior. Variables in VHDL are updated immediately as in other languages like Pascal or C, whereas signal assignments are delayed by a delta cycle.

Furthermore we have to be heedful of not considering VHDL program variables or signals as constraint variables. Indeed constraints have to be considered as relationship among constraint variables, which represent values of the program variables rather than the program variables themselves.
As an example consider the following VHDL variable assignments:

\[ \begin{align*}
    u &= v \\
    \ldots
    u &= \text{not } v
\end{align*} \]

Intuitively we would expect the generation of these constraints:

\[ \begin{align*}
    u &= v \\
    \ldots
    \text{u} &= \text{not } v
\end{align*} \]

Unfortunately these two constraints are inconsistent. So in order to respect the program statement we have to rename the constraint variables each time there is a new program variable assignment. Following this rule the previous statements will be translated as:

\[ \begin{align*}
    u &= v_n \\
    \ldots
    u &= \text{not } v_n
\end{align*} \]

This process is known as spatial incarnation in [16]. It is also used for signal assignments. Furthermore in order to take into account the signals assignments delay, Vemuri[16] added the notion of temporal incarnation. Indeed if we consider the signal assignment statement: \( a \leftarrow b \); the effect is that the current value of signal \( b \) is projected to be the value of \( a \) in the next simulation cycle. The corresponding constraint will be written as shown right below:

\[ a \leftarrow b; \quad a_{n+1} = b_n \]

The Figure 8 illustrates the constraint generation process for the VHDL description of the Figure 2. The statements encountered for the path P1 are presented in the left of the figure, the corresponding constraints are written on the right.

P1: 1.2.3.9.10
P1 statements
in1: bit; in1 \in \{0,1\}
in2 : bit; in2 \in \{0,1\}
out1 : bit; out1 \in \{0,1\}
a : bit; ae \in \{0,1\}
process(in1, in2)
  \((in1_0 \neq in1_{n-1}) \lor (in2_0 \neq in2_{n-1}),
  a \leftarrow in1 \text{ and in2;}
  a_n = in1_n \land in2_{n-1}.

Figure 8. Example of constraint generation

4.2. Constraint solving and test generation

Once all statements involved in each path of the final basis have been translated in the corresponding constraints, the next step is to generate the set of constraints respecting the scheduled paths lists introduced in sub-section 3.2.2. This is achieved by grouping in a file the constraints obtained for each path of the scheduled paths list.

We choose the constraint solver over finite domains included in GNU Prolog[15] in order to find a solution (input signal values) respecting this set of constraints.

5. Conclusion

The Figure 9 resumes our validation approach. We start from a VHDL description. A parser/analyser in Prolog allows to produce CFG and FDG expressed in a LISP format. From CFG, the primary basis is generated using Poole’s algorithm written in LISP language. Analyzing the FDG, scheduling conditions are obtained and the primary basis is transformed into final basis. The scheduled path list is generated from scheduling condition using a LISP software. From this basis and according to the scheduled path list, constraints are generated owing to a C language parser in a format supported by the constraint solver. Finally, a constraint solver generates final test vectors values.

Figure 9. Validation approach.

We have presented in this paper how to apply software testing concepts in order to verify behavioral VHDL program. We are implementing the test data generator. Our aim is to validate this approach on VHDL descriptions of ITC benchmarks[17] and to describe test vectors generated with the constraint solver, using
WAVES IEEE standard[18,19]. The test bench will apply a stimulus to the VHDL description and will check the VHDL description’s response using the waves dataset of the WAVES standard.

6. References


