Test Generation from VHDL Behavioral Models

Wencheng Li
Vitesse Semiconductor Corporation
741 Calle Plano
Camarillo, CA 93012

J. R. Armstrong
Department of Electrical Engineering
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061

Abstract

This paper presents a system for generating tests from a VHDL behavioral model. The tests can be used to thoroughly exercise the VHDL model, and also detect the faults in the equivalent gate level circuit of the model. The VHDL model is developed with the help of the Modeler's Assistant and represented as a Process Model Graph (PMG). A set of VHDL functions have been constructed to help develop VHDL models. Two algorithms are proposed to implement the test generation. **P-Algorithm** is used to generate tests at the process level. For each process a symbolic test set and the corresponding fixed valued test packages (FVTPs) are generated. Synthesis-related FVTP generation algorithms for the VHDL functions are derived to support the P-algorithm. **E-Algorithm** is employed to generate the entity level tests. The symbolic entity level tests are generated first and then the final fixed valued entity level tests are obtained by evaluating the symbolic expressions. The Synopsys synthesis tools are used to get the equivalent gate level circuit of a VHDL model. The gate level fault coverage is obtained by using the HILO fault simulator.

1 Introduction

With hardware description languages such as VHDL, designs can be described in either a top-down or bottom-up fashion through varying levels of abstraction. However, there is a corresponding need for an accompanying set of test generation techniques that can work throughout the abstraction hierarchy. Traditional gate level test generation methods [1, 2, 3] are time-consuming. The wealth of the rich behavioral information contained in a VHDL description cannot be used by such methods, but the behavioral information is potentially very helpful in easing the test generation problem and reducing the test generation time. Therefore, developing a behavioral level test generation method is an important research topic.

One approach to generating tests at the behavioral level is using behavioral fault models. Thatte and Abraham [4] developed functional fault models for a microprocessor based on its instruction set and the functions performed. Various fault models are presented corresponding to different functions present in a microprocessor such as register decoding function, control function, data storage function and data manipulation function. Levendel and Menon [5] extended the D-algorithm to generating tests from computer hardware description languages (CHDLs). The fault models considered are function variables stuck at logic 0 or 1, control faults, and function faults with user-specified faulty behaviors. Cho and Armstrong [6] proposed the B-algorithm to generate tests directly from behavioral VHDL circuit descriptions. Through perturbing VHDL constructs, they defined three types of behavioral faults: behavioral stuck-at faults, behavioral stuck-open faults, and micro-operation faults. In general, behavioral level test generation methods with specific behavioral fault models do not use any gate level information. Therefore it is likely that the tests generated by such methods will have low gate level fault coverage if the behavioral fault models cannot properly reflect the physical faults in a gate level circuit.

Another approach to develop behavioral level test generation methods is using both behavioral level information and gate level information. Behavioral information can be used to construct sensitization paths and resolve conflicts. Gate level information usually is combined into some kinds of high level primitives. Abradir and Breuer [7] introduced the I-path (I for identity) which can transfer data from one place in the circuit to another place without modification. Freeman [8] weakened the conditions for the definition of an I-path and defined the F-path (fault path) and S-path (S for stimulus) for propagating the fault effects to the primitive outputs and applying tests on the primitive inputs. Su and Kime [9] proposed the Hpath for multiple path sensitization in circuits with hierarchical descriptions. They also used a PODEM-like algorithm for local path sensitization of gate level hierarchical circuits. Lee and Patel [10, 11] combined gate level algorithms with high level approaches in ARTEST. Vishakantaiah, Abraham and Saab [12] introduced an approach which exploits a top-down design methodology and applies high level test knowledge to a low level test generator. Murray and Hayes [13] proposed a hierarchical test generation method in which circuits are constructed by small high level functional modules and the test data for such modules are stored as predefined stimulus/response packages. Kunda, Narain, Abraham and Rathi [14] introduced a methodology to speed up the test generation process for circuits with high-level primitives. Sarfert, Markgraf, Trischler and Schulz [15] also used high-level primitives in their test generation system.

Armstrong *et al* [16, 17, 18] introduced a VHDL behavioral model development tool and test bench generation system. The test generation algorithm is based on I/O path sensitization. It does not use a fault model, but merely tries to exercise all input/output paths through the VHDL behavioral model. It needs to be improved to generate tests for more complex models. Moreover the system does not deal with generating tests for targeting gate level faults. However it does provide an effective method to acquire test information contained in a VHDL behavioral model. Such information is used by the test generation method presented here.

The goals of our test generation method are generating tests directly from a VHDL behavioral description, using the tests to thoroughly exercise the VHDL model (test bench), and utilizing the high level tests to detect low level (stuck-at) faults and get high fault coverage. The test generation algorithms are developed based on the I/O path sensitization approach discussed above. But there exist several distinguished features of our approach. First, VHDL functions are defined to implement data path operators such as addition, comparison, increment, multiplication, and shift. These functions act as a set of general logic operators, which makes circuit modeling more flexible. Moreover, like high level primitives in [15], these functions contain some low level information. Secondly, various algorithms are developed to generate fixed value tests for the functions. Applying the tests to the synthesized gate level circuits employing such functions can get high fault (stuck at faults) coverage. No gate level test generator is needed but the same effects are achieved. Thirdly, symbolic tests are used to speed up test generation. Finally, some methods are developed to increase the testability for a VHDL behavioral model.

This paper is organized as follows. In section 2 we present the test generation system and fault coverage evaluation system. Some basic concepts are introduced. In section 3 we describe the process level test generation. The entity level test generation is presented in section 4. In section 5 we give the experimental results. And we conclude in section 6.

2 System Overview

Figure 1 shows the test generation system. The VHDL behavioral model for a digital system is developed by a CAD tool called the Modeler's Assistant [18]. It employs a graphical representation of a VHDL behavioral model called a Process Model Graph (PMG) shown in Figure 2. The PMG is used as the base for our test generation approach. During model construction, we use a VHDL subset which is synthesizable and simple. This makes test generation easy without loosing generality.

Here we call a VHDL behavioral model the MUT (model under test).

The VHDL model represented by a PMG contains a single entity consisting of multiple processes (or a single process). For each process, the process level test generation program (**ptg**), which employs the P-Algorithm, generates a *symbolic test set* (STS) that is a stimulus/response test set using a symbolic notation for different kinds of values. A symbolic test in the symbolic test set may have a corresponding *fixed valued test package* (FVTP) that is a set of the specific values assigned to the symbolic test according to specific algorithms. The FVTPs of all the symbolic tests of a process form a *fixed valued test set* (FVTS) for the process. Applying the FVTS to the equivalent gate level circuit of the process will generate high fault coverage.

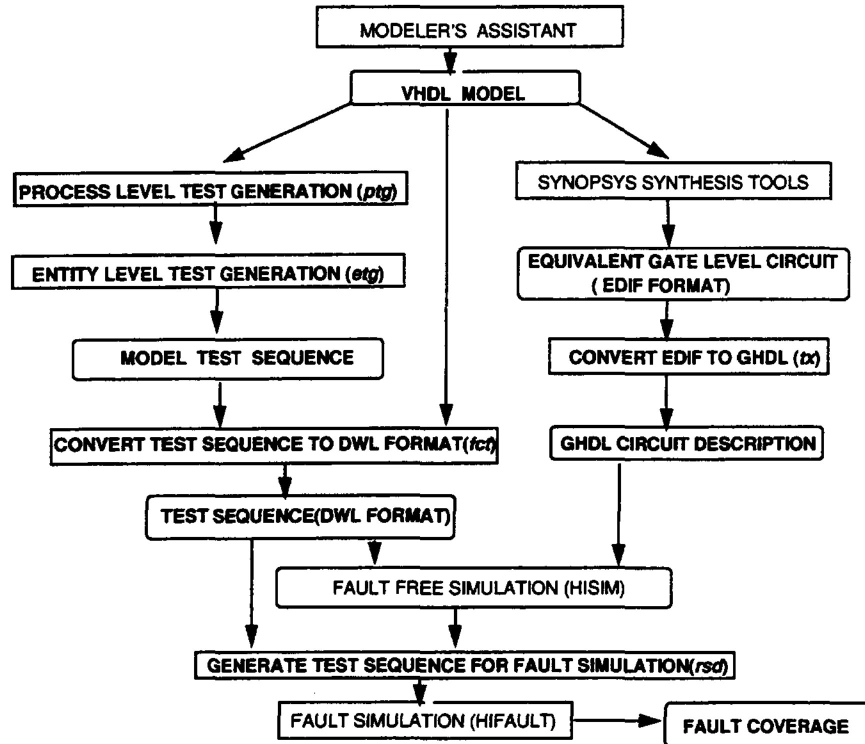

Figure 1. Test Generation System.

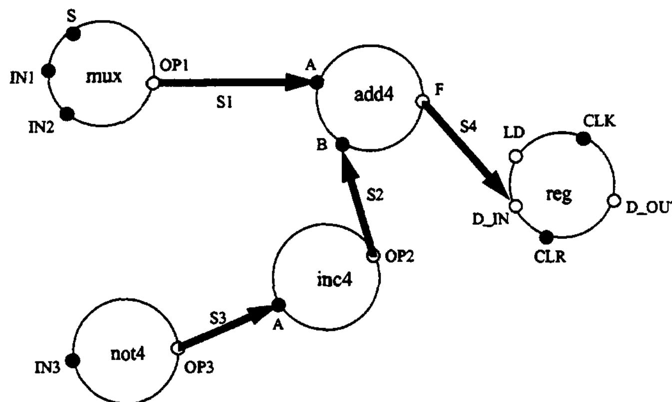

Figure 2. PMG of Model incadd.

Given the STSs and FVTSs of the processes of a MUT and the connection information among the processes, the entity level test generation program (**etg**), which employs E-Algorithm, tries to apply all the FVTSs by constructing forward propagation and backward justification

paths using the STSs. The final test sequences generated are converted to HILO DWL format by program **fct** to test the synthesized gate level circuit corresponding to the MUT.

The Synopsys synthesis tools are used to generate the gate level circuits (EDIF format) from the VHDL description of a MUT. Such EDIF format circuits are passed to HILO TX_EDIF which gives a preliminary translation result. The program **tx** takes the preliminary result and generates the final GHDL format required by HILO. Because there exist some don't care values and other uncertain values in the test sequences, fault free simulation (HISIM) is performed, and its outputs are parsed by program **rsd** to decide these values. Finally fault coverage is obtained through fault simulation (HIFAULT).

In summary, the whole system can be divided into three parts: VHDL Behavioral Model Development completed by using the Modeler's Assistant, Test Generation composed of programs **ptg** and **etg**, and Test Evaluation supported by several commercial CAD systems and programs **fct**, **tx**, and **rsd**. The major contributions of this paper are the above programs and the corresponding algorithms.

3 Process Level Test Generation

In this section, several concepts are introduced first. Then the P-algorithm is presented.

Control Data Flow Graph

A *Control Flow Graph* (CFG) is a directed graph $G(N_c, E_c)$. It consists of a set of nodes N_c and a set of directed edges E_c such that $e_{ij} \in E_c$ denotes a directed edge from the head node $n_i \in N_c$ to the tail node $n_j \in N_c$.

A node n_i is defined as an element in CFG that represents a control statement or an assignment statement within a VHDL process. A directed edge e_{ij} is defined as a pair of nodes (n_i, n_j) such that the node n_i proceeds the node n_j during the execution of the CFG and the edge has a property value (True or False) to indicate which following node is selected. The control statements (also called *control nodes*) are *case statements, if statements* and *for loop statements.* An assignment statement (*assignment node*) is a *signal assignment statement* or a *variable assignment statement*.

A *Data Flow Graph* (DFG) is also a directed graph $G(N_d, E_d)$. Here, N_d is a set of nodes that represent the data (signals and variables) and the operations (shown as circles). E_d is a set of directed edges that indicate the direction of the data flow among the nodes.

A *Control Data Flow Graph* (CDFG) incorporates the CFG and DFGs of a process to represent the data flow and sequencing of the DFGs. The CDFG of a VHDL process can show the data dependencies and control sequences within the process.

In a CFG, each node except for first one (first statement), has one input edge. An assignment statement node has one output edge or no output edge. A control statement node has one or two output edges. An *end node* is an assignment node with no output edge or a control statement node with one empty branch. Along a path from the first node to an end node in a CFG, the control signal values can be specified according to the edge property values. The data relation among the input/output signals can be determined by analyzing the DFG corresponding to the path.

Besides the CDFG, the process level test generation algorithm needs other information such as the declarations of ports, and the type and range of a signal. Therefore two additional nodes, the *declaration node* and the *signal/variable node*, are defined to complement the information stored in CDFG nodes, and thus give a complete *Directed Model Graph* (DMG) representing the information to be used for process level test generation [17].

Symbolic Test Modes

The *mode* of *a symbolic test* in a symbolic test set (STS) reflects the specific type of relation between the input signals and an output signal

of a process. The E-algorithm uses modes to select appropriate symbolic tests. There are four kinds of modes.

(1) *Initialization mode* (I-mode)
An I-mode test assigns the output a constant. Therefore it can be used to initialize a process representing a sequential circuit.

(2) *Hold mode* (H-mode)
An H-mode test makes the output of a process keep its previous value. In VHDL, this can be explicitly indicated (like *A<=A;*) or implied by a conditional structure. For example, an incompletely specified *if statement* in a combinational process will imply a latch in the synthesized gate level circuit. This will correspond to an H-mode test.

(3) *Propagation mode* (P-mode)
A P-mode test establishes a one-to-one and onto mapping between a data input and a data output of a process. It can be shown that the input and the output must have the same bit width in order to maintain such a mapping. A P-mode test is constructed by keeping an input as a symbolic variable and assigning all the other inputs as fixed values. For example, a P-mode test for an adder can be obtained by assigning one input as D_1 and the other input as a constant. A P-mode test is suitable for both forward propagation and backward justification.

(4) *Activation mode* (A-mode)
An A-mode test is a symbolic test that does not belong to the above three modes. It usually has more than one input. No one-to-one and onto mapping exists. But there may exist a one-to-one or onto mapping. Such a test is used to activate a process and also can be a candidate for propagation and justification.

In above discussion, we pay special attention to the properties of the mappings between the inputs and the outputs of a process. This is because such properties can be used to select an appropriate data propagation path or justification path during test generation. For example, in order to propagate the fault effects of preceding processes to the primary outputs through a process, different input values of the process should result in different output values. This means the mapping should be one-to-one. For justification, each output should have a corresponding input. That means the mapping should be onto. Therefore a one-to-one mapping can be used for propagation and an onto mapping used for justification.

Table 1 Symbolic Tests for Functions and Logic Operations.

	op	A	B	F	mode
F<= A op B;	AND	DC1	D1	D1	P
		D1	DC1	D1	P
		D1	D2	a(D1, D2)	A
	OR	DC0	D1	D1	P
		D1	DC0	D1	P
		D1	D2	o(D1, D2)	A
	XOR	DC0	D1	D1	P
		D1	DC0	D1	P
		D1	D2	e(D1, D2)	A
F<=op(A);	NOT	D1		nD1	P
	INC	D1		iD1	P
	DEC	D1		dD1	P
	SHIFTA	D1		tD1	A
F<= op(A,B);	SHIFTR	D1	D2	r(D1, D2)	A
	SHIFTL	D1	D2	l(D1, D2)	A
	ADD	DC0	D1	D1	P
		D1	DC0	D1	P
		D1	D2	s(D1, D2)	A
	MULT	D1	D2	m(D1, D2)	A
	COMP	D1	D2	c(D1, D2)	A

Symbolic notation

We use D_i $(i=1,2,3,...)$ to represent data inputs and outputs, R (0->1) and F (1->0) to represent the events on control signals that usually are clock signals, P to indicate the previous value of a signal in a sequential process, *X* to indicate the don't care, and *Z* to represent the high-impedance state. DC_0 (all *0s*) and DC_1 (all *1s*) are used to represent the preassigned data inputs.

Table 1 gives the symbolic tests for logic operators AND, OR, XOR, NOT and functions INC, DEC, ADD, MULT, COMP, SHIFTL, SHIFTR, SHIFTA. The tests for NAND (NOR) can be obtained by complementing the output values of AND (OR). There are four unary operators: *n* (not), *i* (increment), *d* (decrement), and *t* (arithmetic right shift) and eight binary operators: *a* (and), *o* (or), *e* (xor), *s* (addition), *c* (comparison), *m* (multiply), *l* (left shift), and *r* (right shift). We also use " ' " to represent inverse operations. For example, operator *i'* means inverse operation of operator *i*. That is, *i'* implements a decrement operation. From this table we can know that some operations have both P-mode and A-mode tests. However some operations only have A-mode tests, so it may be difficult to propagate or justify a value through such operations.

Function test generation

Each function has symbolic tests to allow propagation and justification and the corresponding fixed value tests to give good gate level fault (stuck-at) coverage. The symbolic tests are generated based on the functionality of a function. The FVTP generation algorithm for the function is obtained after analyzing the synthesized gate level circuits equivalent to the function. Therefore such an algorithm is synthesis-related. Here only the algorithm for the function ADD is presented. The function ADD adds two operands A and B and returns the sum. It is implemented by the ripple carry method. For simplicity, we assume that no carry input and carry output exist. Suppose that all the data inputs are n-bit BIT_VECTORs. We use a pair $(0^m1^{n-m}, 1^k0^{n-k})$ to represent the values on inputs A and B. Here $A=0^m1^{n-m}$ (*m 0s* followed by *n-m 1s*); $B=1^k0^{n-k}$ (*k 1s* followed by *n-k 0s*). 1^0 or 0^0 means that no *1* or *0* exists.

```
function ADD(A,B: in BIT_VECTOR) return
BIT_VECTOR is
  variable SUMV,AV,BV:
BIT_VECTOR(A'LENGTH-1 downto 0);
  variable CARRY: BIT;
begin
  AV := A;
  BV := B;
  CARRY := '0';
  for I in 0 to SUMV'HIGH loop
    SUMV(I) := AV(I) xor BV(I) xor CARRY;
    CARRY := (AV(I) and BV(I)) or (AV(I) and
              CARRY) or (BV(I) and CARRY);
  end loop;
  return SUMV;
end ADD;
```

The values on one input can be propagated to the output by assigning any specific value to the other input. For simplicity, we choose DC_0 (all *0s*) as such a specific value and get two P-mode tests. An A-mode test is also constructed. It can be used to handle multiple path propagation and generate new P-mode tests by assigning the specific values to one input.

Compared to the symbolic test generation of a function, the corresponding FVTP generation is much more difficult. This is because the symbolic tests are decided by the functionality of the function but the FVTPs are dependent on the implementation of the function. We can always generate some kinds of symbolic tests for a function according to its functionality. But when we generate the FVTPs, we should know how the function is implemented. For example, the Synopsys synthesis tools can give three implementations for the "+" (addition) operator: ripple carry, carry look-ahead, and fast carry look-ahead [26]. In our system, we implemented a function by providing the predefined VHDL code. The code can be converted to a gate level circuit by using certain kinds of synthesis tools. Different synthesis tools will generate different gate level circuits. The FVTP generation algorithm for a function is derived by analyzing the VHDL implementation and the gate level circuit. Therefore, the algorithms derived are synthesis related. Because we use the Synopsys

synthesis tools during deriving the algorithms, all the algorithms are meaningful only when the Synopsys synthesis tools are used. However, the needed algorithms can be developed for other synthesis tools.

Algorithm 1 generates fixed valued tests for ADD. They are composed of two parts. The first part is the tests for data paths which have the effect of exhaustive tests (explained in next section). Each test in the second part will generate a carry within the carry chain. The experimental results have shown that the tests generated by such an algorithm have high gate level fault coverage. The other five algorithms are derived based on the same idea.

Algorithm 1 (test generation for ADD(A,B))

(1). test vector generation for data paths
put $(0^n,0^n)$, $(0^n,1^n)$, $(1^n,0^n)$, and $(1^n,1^n)$ into the test set.

(2). test vector generation for the carry chain
for $k=0$ to $n-1$, do{put $(0^{n-k-1}10^k, 0^{n-k-1}10^k)$ and $(0^{n-k-1}10^k,1^n)$ into the test set}

P-Algorithm

Based on the basic symbolic test set and the basic FVTP generation algorithms and rules, a process level test generation algorithm, called **P-algorithm**, is developed. It contains following five parts:

1. Construct a control flow graph (CFG) for the process
2. Activate each CFG path by selecting values for control signals
3. Generate a data flow graph (DFG) for each CFG path
4. Generate symbolic test set (STS) for the DFG
5. Generate fixed valued test packages (FVTPs) from the symbolic tests

While generating symbolic tests for a DFG, the algorithm does symbolic value propagation and justification. The symbolic tests in Table 1 establish the base for such operations. The final symbolic tests for a process are obtained by combining the results from the step 2 and 4. Each symbolic test is assigned a test mode.

The algorithm will generate a fixed valued test packages for each symbolic test. If a symbolic test contains an operator of a function, then the FVTP of the symbolic test will cover the FVTP of this function. That means assigning the specific values to the symbolic values according to the fixed valued tests of the function. If only one Di exists in a P-mode symbolic test which contains no function operator, the FVTP can be obtained by assigning DC0 and DC1 as the values of the Di. This is because there exist bitwise symmetric structures in the process with such a P-mode test. Each bit of the signal with Di has the same properties. So applying all 1s and all 0s to the signal is equivalent to applying both 1 and 0 to each bit of the signal. Therefore the effect of exhaustive testing is obtained.

Examples

Figure 3 shows the symbolic test sets (left columns) and fixed valued test packages (right columns) for the five processes in Figure 2. For example, there are three symbolic tests in "add4" : $(D_6\ D_6\ DC_0)$, $(D_7\ DC_0\ D_7)$, and $(s(D_8, D_9)\ D_8\ D_9)$. The first two are P-mode. The third one is A-mode. The time frame for each test is 1. Each FVTP of the first two symbolic tests contains 2 fixed valued tests, but the FVTP of the third symbolic test contains 12 fixed valued tests.

portorder: F B A	F	B	A
1 —(number of frames)	1010	1010	0000
P —(mode)	0101	0101	0000
2 —(number of FVTP)	1010	0000	1010
D6 D6 DC0 —(symbolic test)	0101	0000	0101
1	0010	0001	0001
P	0100	0010	0010
2	1000	0100	0100
D7 DC0 D7	0000	1000	1000

1
A
12
s(D8,D9) D8 D9

0000	0001	1111
0001	0010	1111
0011	0100	1111
0111	1000	1111
0000	0000	0000
1111	1111	0000
1111	0000	1111
1110	1111	1111

(a) add4

portorder: OP2 A
1
P
6
iD4 D4

OP2	A
0001	0000
0010	0001
0100	0011
1000	0111
0000	1111
1111	1110

(b) inc4

portorder: OP1 IN2 IN1 S
1
P
2
D1 D1 X 1
1
P
2
D2 X D2 0

OP1	IN2	IN1	S
1111	1111	X	1
0000	0000	X	1
1111	X	1111	0
0000	X	0000	0

(c) mux

portorder: OP3 IN3
1
P
2
nD3 D3

OP3	IN3
0000	1111
1111	0000

(d) not4

portorder: LD D_IN D_OUT CLK CLR
1
P
2
1 D5 D5 R 0
1
H
1
0 X P R 0
1
I
1
0 X D0 0 R

LD	D_IN	D_OUT	CLK	CLR
1	1111	1111	R	0
1	0000	0000	R	0
0	X	P	R	0
0	X	0000	0	R

(e) reg

Figure 3. STs and FVTPs of Five Processes for the MUT incadd.

4 Entity Level Test Generation

E-Algorithm

The objective of the E-algorithm is to generate final test patterns for a MUT that will contain all the fixed value tests for each process. The algorithm is shown as follows.

1. Initialize processes
2. Select a process to test
3. Get an FVTP and the corresponding symbolic test
4. Propagate and justify the symbolic test to generate a symbolic entity test
5. Apply the FVTP to the symbolic entity test to form fixed valued entity test patterns
6. Repeat until all FVTPs of the processes in the MUT are applied

At the beginning the sequential processes need to be initialized. The E-algorithm selects an I-mode test for each sequential process to do initialization. If the input signals of the process are not PIs, then the algorithm does justification.

Some processes such as buffers are not necessary to test because synthesis tools will map them to wires. The tests for a complex process may cover the tests for other processes along the same data path. So the algorithm only selects the processes needed to test.

For each symbolic test of a process that has a corresponding FVTP, the algorithm first constructs a symbolic data path to generate symbolic entity tests. To construct the data path, the algorithm constructs a forward path first by propagating the output signal to a PO and then constructs backward paths by justifying the input signals to PIs. After getting the symbolic tests, the algorithm expands them into fixed valued test patterns by evaluating symbolic expressions.

There may exist path conflicts if a MUT has fanout reconvergency. There are two cases that will create fanout branches. One is that a signal from a process is connected to more than one process. The other is that there are two or more outputs in a process and there exists a fanout point inside the process that will control at least two outputs. The E-algorithm tries to resolve the conflicts by backtracking to select another path or performing multiple path propagation. Multiplexers and registers in a model can help to do so by providing different paths or keeping previous values.

Example

Figure 4 gives the symbolic entity tests for the MUT *incadd* in Figure 2. The frames indicate the test sequence for each test. Here frame 0 is for initialization. Frames 1 and 2 are obtained after propagating and justifying the symbolic test $(iD_4\ D_4)$ of process *inc4*. Blank positions will be filled as previous values. Frames 3 and 4 are for testing process *add4*. Frames 5 and 6 give the tests for applying the test $(D_2\ \ X\ \ D_2\ \ 0)$ of process *mux*. Each test is assigned an *"id"* number which is used to distinguish the test from other tests. The other symbolic test of the process *mux* and the symbolic tests of process *not4* and *reg* are not necessary to apply because they have be covered by the symbolic entity tests for other processes.

Each symbolic entity test for the MUT contains Di belonging to the symbolic test of a given process. By applying the corresponding FVTP, several final test patterns can be obtained. For example, a total of 12 final test patterns shown in Figure 5 are obtained from one symbolic test (frame 3 and 4). Now let's see how to apply the first fixed valued test *(0010 0001 0001)* of the A-mode test $(s(D_8, D_9)\ D_8\ D_9)$ in Figure 3 (a) to the symbolic test (frame 3 and 4) in Figure 4. The E-algorithm calculates the value of IN3: $ni'D_8 = ni'(0001) = n(0000) = 1111$. Other PIs can be assigned directly.

Testability enhancement

In order to propagate or justify a signal, we hope to get at least one P-mode test for each process. For some processes, such a test may not exist. Sometimes it is possible to get a P-mode test by inserting additional statements. For example, the function MULT does not have P-mode test. We can modify a multiplier process as follows.

```
mux_mult: process(SEL,B,A)
begin
 if SEL='1' then
  D_OUT<=A&B;
```

```
 else
  D_OUT<=MULT(A,B);
 end if;
end process mux_mult;
```

The operation & merges two signals to a single one. We can get a P-mode test *(OUT, A, B, SEL)* = *(D_1&D_2, D_1, D_2, 1)* from the & operation. At entity level, we can also insert a process like multiplexer to increase the testability of the whole model.

frame	id	OP3	IN3	OP	OP1	IN2	IN1	S	F	LD	D_OUT	CLK	CLR
0	0								X	0	D0	0	1
1	1	D4	nD4	iD4	DC0	DC0	X	1	iD4				
2	1								iD4	1	iD4	R	0
3	2	i'D8	ni'D8	D8	D9	D9	X	1	s(D8,D9)				
4	2								s(D8,D9)	1	s(D8,D9)	R	0
5	3	i'DC0	ni'DC0	DC0	D2	X	D2	0	D2				
6	3								D2	1	D2	R	0

Figure 4. Symbolic Tests for the Model incadd.

frame	id	OP3	IN3	OP	OP1	IN2	IN1	S	F	LD	D_OUT	CLK	CLR
3	2	0000	1111	0001	0001	0001	X	1	0010				
4	2								0010	1	0010	R	0
3	2	0001	1110	0010	0010	0010	X	1	0100				
4	2								0100	1	0100	R	0
3	2	0011	1100	0100	0100	0100	X	1	1000				
4	2								1000	1	1000	R	0
3	2	0111	1000	1000	1000	1000	X	1	0000				
4	2								0000	1	0000	R	0
3	2	0000	1111	0001	1111	1111	X	1	0000				
4	2								0000	1	0000	R	0
3	2	0001	1110	0010	1111	1111	X	1	0001				
4	2								0001	1	0001	R	0
3	2	0011	1100	0100	1111	1111	X	1	0011				
4	2								0011	1	0011	R	0
3	2	0111	1000	1000	1111	1111	X	1	0111				
4	2								0111	1	0111	R	0
3	2	1111	0000	0000	0000	0000	X	1	0000				
4	2								0000	1	0000	R	0
3	2	1110	0001	1111	0000	0000	X	1	1111				
4	2								1111	1	1111	R	0
3	2	1111	0000	0000	1111	1111	X	1	1111				
4	2								1111	1	1111	R	0
3	2	1110	0001	1111	1111	1111	X	1	1110				
4	2								1110	1	1110	R	0

Figure 5 Final Test Patterns of the Model incadd.

5 Coverage

The effectiveness of the algorithms presented in the previous sections has been evaluated by using various VHDL behavioral models. This section will present the test results for the models with a single process and the models with multiple processes.

One type of single process models we used contain only a single VHDL function. The test patterns generated for such a model were applied to the different equivalent circuits of the same model. The equivalent circuits of the models with a same function but different bit width (4-bit, 8-bit, 16-bit inputs) were also tested. The fault coverage was found to be in the range of 95% to 100%. Table 2 shows the test results after applying the test patterns to the different bit width adders. From the tables we can see that the fault coverage for smallest implementations are high but the fault coverage for fastest implementations decreases with increasing word width. After analyzing the gate level circuits, we found the following possible reasons. The smallest implementation of a model usually has a circuit structure which well matches its VHDL model. For example, we can see the different stages in the smallest implementation of a ripple adder. But the fastest implementation usually has less logic levels and more components because the synthesis tools flatten the original logic structure and construct the new one. This shows that the function FVTP generation algorithms are synthesis related.

Table 2 Test Results for Adders.
(Note: S--smallest implementation F--fastest implementation)

Model	Bit Width	Type	Gate Number	Total Faults (Top-Level)	Fault Coverages(%)
add4	4	S	18	46	100.0
		F	25	56	100.0
add8	8	S	49	98	100.0
		F	73	146	97.3
add16	16	S	109	204	100.0
		F	157	332	95.8

Tables 3 gives some other test results. All the different multiplier models were constructed by using the function MULT and assigning different bit width (4-bit, 8-bit, 16-bit, 32-bit) to the input signals. Only one gate level circuit for each model was evaluated. But the fault simulation was performed at both top-level and all-level. At top-level, the faults within a sub-circuit such as flip-flop, latch, and multiplexer will not be activated. At all-level, all the faults in a circuit will be activated.

Table 3 Test Results for Multipliers.

Model	Bit Width	Level	Comt. Number	Total Faults	Fault Cov.(%)	
					dropd	det'd
mult4	4	Top	22	152	100.0	100.0
		All	105	226	98.7	98.7
mult8	8	Top	109	636	99.8	99.4
		All	510	1052	98.7	97.9
mult16	16	Top	491	2662	100.0	100.0
		All	2267	4598	99.7	99.7
mult32	32	Top	2004	10950	100.0	99.2
		All	9388	18904	100.0	99.3

Table 4 shows the test results for more general single process models. Among them, **upcnt4** is a 4-bit up counter. **downcnt8** is an 8-bit down counter. **COUNTER** is a controlled counter

which can count up or down. **alu32** is an 8-function 32-bit ALU. **SHIFTREG** is a 4-bit shift register. **mux4_1** is a 32-bit 4-to-1 multiplexer. Finally, **demux1_4** is an 8-bit 1-to-4 demultiplexer. Both dropped faults and detected faults are evaluated.

From the above test results, we can know that the P-algorithm is effective on moderately complicated processes.

Table 4 Test Results for Single Process Models.

Model	Subcircuit Number	Gate Number	Total Faults (Top Level)	Fault Coverages(%)	
				dropd	det'd
upcnt4	9	52	60	100.0	93.3
downcnt8	23	108	122	100.0	98.4
COUNTER	14	83	108	98.1	83.3
alu32	314	1211	1616	99.6	99.6
SHIFTREG	15	68	90	96.7	86.7
mux4_1	64	230	464	98.7	98.7
demux1_4	0	46	112	100.0	100.0

The test results for six multiple process models are shown in Table 5. Among these models, **amd2910c** is a modified version of amd2910 micro-controller [23]. The amd2910c contains an instruction decoder, a stack and its pointer, a multiplexer, a controlled up-counter, and a controlled down-counter. Depending on the instructions supplied and the internal state signal values, the instruction decoder generates various control signals to control the actions of other components. **mpu2** is a small CPU which uses a counter to select the different functions of its ALU. **CKA** is a model that is used to illustrate path conflict resolution. **csi** is an example containing a 4-function shifter process. **muxadd** performs additions from multiple sources. Both combinational and sequential circuits exist among these models. Only one implementation is used for each model. The fault simulations were performed at top-level. The high fault coverage for these models shows that the E-algorithm is effective for multi-process circuits.

Table 5 Test Results for Multiple Process Models.

Model	Proc. No.	Level	Comt. Number	Total Faults	Fault Cov.(%)	
					dropd	det'd
amd2910c	10	Top	373	1386	95.2	95.1
		All	1732	3508	93.0	92.9
mpu2	6	Top	145	618	98.2	97.6
		All	565	1244	97.5	97.2
CKA	3	Top	16	88	98.9	96.6
		All	89	200	93.5	92.5
csi	4	Top	13	70	98.6	95.7
		All	60	138	99.3	97.8
incadd	5	Top	19	110	99.1	97.3
		All	87	206	98.5	97.6
muxadd	7	Top	646	3732	99.1	99.1
		All	2699	6430	98.0	98.0

6 Conclusions

A behavioral level test generation system for VHDL models was presented in this paper. The test patterns generated for a VHDL model give high fault coverage for the equivalent gate level circuit. Such a circuit can be a combinational circuit or clocked circuit, which random test generation cannot handle. There exist slight differences in the fault coverage when the test patterns were applied to the different equivalent gate level circuits of the same model. But most of them reach a desired high level of coverage. All the circuits were obtained by using the Synopsys synthesis tools. The function test algorithms were derived after analyzing such circuits. Therefore they are currently limited to the Synopsys synthesis tools but can be developed for other specific synthesis tools.

Our future work includes developing more VHDL models of complex digital circuits to help improve the algorithms and programs. A more complete VHDL function set and the corresponding algorithms need to be constructed. Such algorithms can be obtained by using the same approach introduced in this paper or provided by users of specific synthesis algorithms or tools. If the test generation algorithms can be combined with synthesis algorithms or tools, more perspective and practical results can be obtained. Speed comparison between our test system with other test systems will be done. We will also consider the possible application areas of our test generation approach in behavioral level testability analysis, design validation, and connection to a general ATPG system where our system serves as a preliminary test set generator to give a test set with certain coverage (for example, 95 percent) and the ATPG system generates tests for other hard-detected faults.

References

[1] J. P. Roth, "Diagnosis of Automata Failures: A calculus and a method," *IBM J. Res. Develop.*,Vol. 10, pp. 278-291, July 1966.

[2] P. Goel, "An implicit enumeration algorithm to generate tests for combinational logic circuits," *IEEE Trans. Comput.*, Vol. c-30, pp.215-222, March 1981.

[3] H. Fujiwara and T. Shimono, "On the acceleration of test generation algorithms," *IEEE Trans. Comput.*, Vol. c-32, pp.1137-1144, Dec. 1983.

[4] S.M.Thatte and J.Abraham, "Test generation for microprocessors," *IEEE Trans. Comput.*, vol. c-29, no.6, pp.429-441, June 1980.

[5] Y.H.Levendel and P.R.Menon, "Test generation algorithm for computer hardware description languages," *IEEE Trans. Comput.*, Vol. c-31, pp.577-588, July 1982.

[6] C.H.Cho and J.R.Armstrong, "The B algorithm: A behavioral test generation algorithm," *Proc. Int. Test Conf.*, 1994, pp.968-979.

[7] M.S.Abadir and M.A.Breuer, "A knowledge-based system for designing testable VLSI chips," *IEEE Design Test*, Aug. 1985, pp. 56-68.

[8] S.Freeman, "Test generation for data-path logic: the *F*-path method," *IEEE J.Solid -State Circuit*, Vol.. 23, Apr. 1988, pp.421-427.

[9] C-C.Su and C.R.Kime, "Multiple path sensitization for hierarchical circuit testing," *Proc. Int. Test Conf.*, 1988, pp. 152-161.

[10] J.Lee and J.H.Patel, "An architectural level test generator for hierarchical design environment," *21th Symposium on Fault-Tolerant Computing*, 1991, pp. 44-51.

[11] J.Lee and J.H.Patel, "ARTEST: an architectural level test generator for data path faults and control faults," *Proc. Int. Test Conf.*, 1991, pp. 729-738.

[12] P.Vishakantaiah, J.A.Abraham, and D.G.Saab, "CHEETA: composition of hierarchical sequential tests using ATKET," *Proc. Int. Test Conf.*, 1993, pp.606-615.

[13] B.T.Murray and J.P.Hayes, "Hierarchical test generation using precomputed tests for modules," *IEEE Trans. CAD*, vol.9, no.6, pp.594-603, June 1990.

[14] R.P.Kunda, P.Narain, J.A.Abraham, and B.D.Rathi, "Speed up of test generation using high-level primitives," *Proc. 27th Design Automation Conf.*, 1990, pp.594-599.

[15] T.M.Sarfert, R.Markgraf, E.Trischler and M.H.Schulz, "Hierarchical test pattern generation based on high-level primitives," *Proc. Int. Test Conf.*, 1989, pp. 470-479.

[16] S.Rao, B.Pan, and J.R.Armstrong, "Hierarchical test generation for VHDL behavioral models," *Proc. European Design Automation Conf.*, Feb.22-24, 1993, pp.175-183.

[17] S.Kapoor, J.R.Armstrong, and S.R.Rao, "An automatic test bench generation system," *Proc. VHDL Int. Users Forum*, 1994, pp.8-17.

[18] B.Singh, J.Wicks, P.Wright, and J.R.Armstrong, "The Modeler's Assistant: A CAD tool for behavioral model development," 14 pages, Proc. CHDL 93, April 1993, Ottawa, Canada.

[19] M.Abramovici, M.A.Breuer, and A.D.Friedman, *Digital systems testing and testable design*, Computer Science Press, 1990.

[20] J.R.Armstrong and F.G.Gray, *Structured logic design with VHDL*, PTR Prentice Hall, 1993.

[21] W. Li and J.R.Armstrong, "Behavioral Test Generation with Fault Coverage Enhancement, " *The Second Workshop on Hierarchical Test Generation*, Duisburg, Germany, Sept. 25-26, 1995.

[22] P.C.Maxwell and R.C.Aitken, "Test set and reject rates: All fault coverages are not created equal," *IEEE Design & Test of Computers*, Vol.10, pp.42-51, March 1993.

[23] K.C.Chang and E.J.Olson, "Tutorial C : VHDL synthesis for digital ASIC designs," *VHDL International Users'Forum*, May 1994.

[24] *VHDL compiler reference manual, Version 3.0*, Synopsys Inc., Nov. 1992.

[25] *Design compiler reference manual, Version 3.0*, Synopsys Inc., Dec. 1992.

[26] *DesignWare Databook, Version 3.0*, Synopsys Inc., Dec. 1992.

[27] *System HILO system reference manual*, GenRad Limited, 1991.

[28] *VHDL tool integration platform (VTIP) / Design library system (DLS) manual*, CAD language systems, Inc., 1993.

[29] *IEEE standard VHDL language reference manual*, IEEE, Inc., March 1988.

[30] W. Li, *A test generation system for behaviorally modeled digital circuits*, PhD's Dissertation , VPI&SU, June 1996.