A Formal Model for Behavioral Test Generation

by

Chang Hyun Cho

Dissertation submitted to the Faculty of the

Virginia Polytechnic Institute and State University

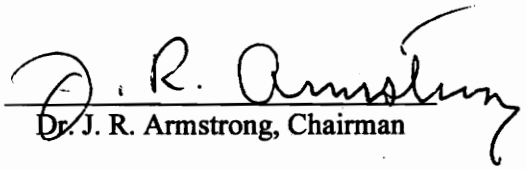
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

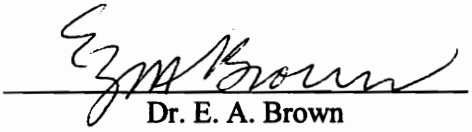
in

Electrical Engineering

APPROVED:

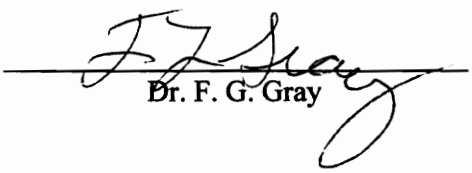


Dr. J. R. Armstrong, Chairman



Dr. E. A. Brown

Dr. D. S. Ha



Dr. F. G. Gray

Dr. J. G. Tront

February, 1994

Blacksburg, Virginia

A Formal Model for Behavioral Test Generation

by

Chang Hyun Cho

Dr. J. R. Armstrong, Chairman

Electrical Engineering

(ABSTRACT)

A formal behavioral test generation algorithm, called the B-algorithm, is presented together with a behavioral VHDL model and a realistic behavioral fault model. Using the behavioral VHDL model, a behavioral VHDL circuit description is represented as a set of equivalenet process statements and connections among them. The behavioral fault model consists of three types of behavioral faults (behavioral stuck-at faults, behavioral stuck-open faults, and micro-operation faults) which well represent faulty behaviors of a digital cicuit. The behavioral VHDL model and the behavioral fault model improve the efficiency of test generation by reducing the size of the domain searched during the test generation procedure. The B-algorithm generates tests directly from behavioral VHDL circuit descriptions using three basic test generation operations (activation, propagation, and justification), which are systematically executed by manipulating three data structures (B-frontier, J-frontier, and A-queue). Rules for the test generation operations are formally defined using the concepts of two-phase activation and two-phase propagation. The difference between simulation semantics and test generation semantics is discussed, and a method of efficiently assigning time periods without being affected by simulation semantics is proposed. A method of handling bus resolution functions, reconvergent fanout, and feedback loops during test generation is discussed. Two-phase testing, a testing strategy where a fault is detected using two consecutive test sequences, is introduced and is formally incorporated into the B-algorithm.

Acknowledgements

First and foremost, I wish to express my sincere thanks to my advisor Dr. James R. Armstrong for his continuous support and encouragement during the preparation of this dissertation. Many thanks are due to Dr. Dong S. Ha for his helpful suggestions on some important topics of this dissertation. I would also like to thank Dr. F. Gail Gray, Dr. Joseph G. Tront, and Dr. Ezra A. Brown for serving as members of my committee.

It has been a great pleasure to work with so many hard-working and friendly people in the computer research lab. Special thanks are due to Sandeep Shah and Bob Lineberry for their continuous encouragement.

I would like to dedicate this work to my wife Changsook and two sons Kyunghoon and Sunghoon who have shown constant patience throughout the duration of this dissertation. Their invaluable love and support are the most important reasons for all my achievements. I thank God for giving me strength and keeping my family happy and healthy.

Table of Contents

List of Illustrations

Chapter 1. Introduction

With the increasing complexity of VLSI circuits, test generation has become one of the most complicated and time-consuming problems in digital system design. Classical gate level test generation algorithms [1-16] cannot generate tests for complex VLSI circuits because gate level fault models and gate level circuit models create an enormously large search space during test generation. Hence, it has become important to develop an approach to modeling faults at a higher level of abstraction, and a method that can generate tests to detect these faults [17,18].

There have been several approaches which generate tests using behavioral fault models and behavioral circuit models. Levendel and Menon [19] proposed an approach to generating tests using behavioral faults defined by perturbing language constructs of computer hardware description languages (CHDL's). Stuck-at faults at inputs, outputs, and state variables, control faults, and general function faults are defined in this approach. Abraham *et al.* [20,21] defined functional faults for microprocessors using unique characteristics of microprocessor models. Faults are defined for decoding functions, control functions, and data functions. This approach is suitable for microprocessors, but cannot be directly applied to general behavioral models. Su *et al.* [22-24] proposed an approach to generating tests using register transfer (RT) level faults defined by perturbing language constructs of register transfer languages (RTL's). The RT level faults defined are jump faults, condition faults, data transfer faults, register selection faults, and operator selection faults. This approach is more general than Abraham *et al*'s approach [20,21], but has a drawback of generating tests using ill-structured control constructs of RTL's. Ghosh and

Chakraborty [25,26] proposed behavioral fault models based on the subset of the language constructs of C. Faults are defined for variables and control constructs such as *for*, *switch*, *if*, *while*, *assignment*, and *waitfor* (it is not a language construct of C, but was assumed to exist in C). Vector-wise stuck-at faults, not bit-wise stuck-at faults, are defined for variables.

Barclay and Armstrong [27,28] proposed an approach similar to Levendel and Menon's approach. This approach defines behavioral faults by perturbing language constructs of VHDL [29] (an IEEE standard HDL): control constructs such as *if-then-else*, *case-when*, and *signal assignment* and micro-operations such as *logic*, *relational*, and *arithmetic operations*. It generates tests using artificial intelligence techniques of goal trees and goal solving to represent and manipulate sensitization, justification, and propagation requirements. O'Neill *et al.* [30-32] developed the BTG (Behavioral Test Generator), enhancing this approach by adopting methods which reduce the search space created during test generation. Lam [33] improved BTG by presenting methods to handle reconvergent fanout and feedback loops. Baweja [34] incorporated a set of heuristics into BTG, which is useful for generating test vectors for micro-operations representing complex logic. Baweja showed that this approach helps to increase the equivalent gate level fault coverage. BTG successfully generates tests for behavioral VHDL models for some MSI/LSI circuits. However, it needs to be improved in some respects to be used for larger circuit models as discussed below.

First, the previous behavioral fault model used by Barclay *et al.* [27,28,30-34] generates a significant number of unnecessary behavioral faults because some types of faults are equivalent to others. In addition, the fault model generates vector-wise stuck-at faults from which only a limited number of test patterns can be generated (all 0's and all 1's).

Second, the test generation algorithm uses Prolog-oriented AND/OR goal trees for representing and solving test generation problems. The algorithm takes advantage of the built-in language features (such as backtracking mechanisms) of Prolog [35], but suffers from its inflexible control constructs. The language C or C++ provides flexible control constructs and various bit manipulation constructs which are essential for implementing a behavioral test generation algorithm. Programs written in these languages are known to run faster than those written in other high level languages. In order to implement the algorithm using C or C++, it is necessary to develop a more general and formal algorithm.

Third, the test generation rules defined so far are ad-hoc and hence it is not clear whether they cover every possible case that can occur during the test generation procedure or not. In addition, the interaction between the overall test generation procedure and the detailed test generation rules has not been clarified, which makes it difficult to determine the completeness of the algorithm.

Finally, the algorithm is incomplete or generates inefficient tests for some special cases:

1. Lam [33] used a special propagation scheme for handling reconvergent fanout, but there are special cases when it does not work properly.
2. The algorithm generates tests using the simulation semantics of VHDL. In an event-driven simulation environment, an event must occur for a statement to be executed. If this concept is directly used for the test generation algorithm, unnecessarily long tests could be generated.
3. The algorithm cannot handle bus resolution functions (BRF's) which can occur in VHDL models with multiple process statements.

A formal model for behavioral test generation is proposed in this dissertation which can solve the problems. First, a behavioral VHDL model consisting of equivalent process statements is

proposed. This model leads to efficient test generation since behavioral faults and propagation and justification rules are defined only for VHDL constructs implying process statements. Second, a new behavioral model consisting of only three types of behavioral faults (behavioral stuck-at faults, behavioral stuck-open faults, and micro-operation faults) is proposed. Using the new fault model, fewer number of equivalent faults are generated and fewer number of test generation rules need to be defined, which also reduces the complexity of the test generation algorithm. Third, a behavioral test generation algorithm, called the B-algorithm, is presented. In contrast to the previous approaches, the B-algorithm doesn't use the concepts of goals or goal trees for representing or solving problems. It generates tests using three test generation operations: activation, propagation, and justification. Rules for the test generation operations are formally defined using the concepts of two-phase activation and two-phase propagation. Special test generation rules for bus resolution functions [29] are also defined. The difference between simulation semantics and test generation semantics is discussed, and a method of efficiently assigning time periods is presented. A method of handling reconvergent fanout and feedback loops is discussed. The concept of two-phase testing is introduced and is formally incorporated into the B-algorithm. A method of systematically performing the test generation operations by manipulating three data structures (the B-frontier, the J-frontier, and the A-queue) is presented.

This dissertation is organized as follows. Chapter 2 reviews various gate level test generation techniques and behavioral test generation techniques. Chapter 3 presents a behavioral VHDL model of equivalent process statements. Chapter 4 presents a new behavioral fault model and discusses its characteristics. Chapter 5 presents a behavioral test generation algorithm, called the B-algorithm, which can generate tests directly from the behavioral VHDL model using the new behavioral fault model. Chapter 6 suggests further research area. Chapter 7 concludes the dissertation.

Chapter 2. Literature Review

2.1. Gate Level Test Generation Techniques

Gate level test generation algorithms generate tests from gate level circuit models using gate level fault models. Typical gate level fault models are stuck-at faults, bridging faults [36,37], and delay faults (or transition faults) [38-40]. In this section, only algorithms using a single stuck-at fault model (where only one stuck-at fault is defined at a time) are reviewed.

Algorithms for Combinational Circuits

D-algorithm [1], PODEM [2], and FAN [3] are widely-used gate level test generation algorithms which generate tests for purely combinational circuits using a single stuck-at fault model. For a given stuck-at fault, each gate level test generation algorithm creates a D or $\overline{D}$ at the fault site (D represents a signal which has the value 1 in the fault-free circuit and 0 in the faulty circuit, and $\overline{D}$ represents a signal which has the value 0 in the fault-free circuit and 1 in the faulty circuit).

D-algorithm is the first formal gate level test generation algorithm for combinational circuits. It generates tests by activating a fault by creating a D or $\overline{D}$ at the fault site, propagating it to a primary output, and justifying the values used for the activation and the propagation to primary inputs. Values may be assigned to internal signal lines during the test generation procedure. There may be several possible values to justify a line or several paths to propagate an error. If a selected value or path leads to a conflict, D-algorithm backtracks to a most recent decision point. In the

presence of reconvergent fanout, the D or $\overline{D}$ is propagated through all fanout paths toward the reconvergent point.

PODEM (Path-Oriented DEcision Making algorithm) was originally developed to improve the D-algorithm in generating tests for combinational logic circuits that implement error correction and translation (ECAT) type functions. In ECAT type circuits, conflicts frequently occur during the justification of logic values assigned to internal signal lines, and are usually detected only after most of the test generation procedure is performed. PODEM eliminates the justification procedure by assigning values to only primary inputs. Since it determines logic values of primary inputs first and observes if they can be a test, more signal values can be implied in an earlier stage of the test generation procedure, which helps to reduce the number of backtracks in comparison to D-algorithm.

FAN (fanout-oriented test generation algorithm) further reduces the number of possible backtracks during the test generation procedure. First, it stops backtracks at the *head lines* (a head line is a fanout stem which can be reached from a primary input without passing through any fanout point) instead of continuing backtracks to primary inputs. Since the line justification of the head lines can always be done without any backtracking, it can be postponed to a final stage of the test generation procedure. Second, it tries to determine as many signal lines as possible which can be uniquely determined. Third, it applies the unique sensitization technique when the D-frontier consists of a single gate. Finally, it reduces the test generation time by using a multiple backtrace (backtrace through multiple paths) procedure instead of a single backtrace (backtrace through a single path) procedure. FAN has been shown to be more efficient and has higher fault coverage than PODEM.

Algorithms for Sequential Circuits

Test generation algorithms for combinational circuits have been extended to handle sequential circuits [5-16]. Test generation algorithms for sequential circuits transform a synchronous sequential circuit into an iterative combinational array corresponding to multiple time frames [41]. In this transformation, a flip-flop is modeled as a combinational element having an additional input to represent its current state and an additional output to represent its next state. Hence, a stuck-at fault in an iterative array may have repeated fault effect at every time frame.

Putzolu and Roth [5] extended the D-algorithm to handle multiple fault sites for sequential circuits using the conventional five-valued logic model (0, 1, X, D, $\overline{D}$). As in the D-algorithm, their approach propagates a fault effect to a primary output first (forward process), and then justifies the values necessary for the propagation (backward process). Hence, the assignments of values which are made during the forward process (over multiple time periods) should be stored in order to be used later for the backward process. Putzolu and Roth's approach did not correctly take into account the repeated effects of a fault in a sequential model because it generates tests as if the circuit were combinational. Muth [6] proposed the nine-valued logic model (0/0, 1/1, 1/0, 0/1, 1/X, 0/X, X/1, X/0, X/X) to solve this problem. The nine-valued logic allows sensitized paths to be created during the backward process. Hence it only has to select a sensitized path during the forward process. Both approaches perform test generation both backward and forward in time.

Marlett [7,8] proposed an approach, called the EBT (Extended Backtrace), to perform test generation only in reverse time. EBT first selects a path from the fault site to a primary output (PO), then sensitizes the path starting from the PO. After the sensitization of the selected path succeeds, the fault is activated by justifying the needed value at the fault site. As a result, test

generation proceeds only reverse in time, i.e., the test vector generated first becomes the last test vector in a given test sequence. Since an output of a sequential element at the current time frame is determined only by the inputs at the current time frame and the states at the previous time frame, EBT only has to keep the circuit status at two adjacent time frames: the current time frame and the previous time frame. The disadvantage of EBT is that the preselected path is not always correct. Since the number of possible paths could be large, to try every path is impractical. Mallela and Wu [9] preconstructed a diagram composed of several possible paths in order to try more paths without excessive overhead. If one path fails, several paths can be pruned from the diagram by their topological relation to speed up the whole process. Cheng *et al.* [10-12] proposed an approach, called the Gentest, which improves EBT by preselecting a sensitized PO instead of a sensitized path from the fault site to a PO. Because the number of PO's in a circuit model is in general smaller than that of propagation paths, Gentest is more efficient than the previous approaches [7-9]. In order to accurately predict which primary output must have a sensitized value, this approach proposes a new testability measurement called drivability which corresponds to the observability measurement described in [42]. Gentest uses the SPLIT [10] circuit model, a 9-valued model, to speed up the justification process.

Ma *et al.* [13] and Ghosh *et al.* [14] proposed a PODEM-based algorithm which relies on the existence of a reset state and assumes that every test sequence starts from the reset state. It first extracts the part of the state transition graph (STG) of the finite state machine to be tested by systematically visiting states reachable from the reset state. It then generates test sequences using the partial STG in conjunction with algorithms for fault excitation and propagation and state justification.

Niermann and Patel [15] proposed HITEC, a PODEM-based algorithm which uses dominators and mandatory assignments of FAN [3] and SOCRATES [4]. It introduces three techniques. First, it uses a targeted D element technique which increases the number of possible mandatory assignments and reduces the over-specification of state variables. Second, it uses the state knowledge of previously generated test vectors for state justification without the cost of the memory overhead of the STG [13,14]. Third, for faults that were aborted during the first test generation pass, it uses the fault simulation knowledge during the second test generation pass in order to increase the fault coverage.

Kelsey *et al.* [16] proposed FASTEST, a PODEM-based algorithm with a 9-valued logic model. FASTEST has two unique features. First, it determines the number of time frames required to activate the fault under consideration and the number of time frames required to observe the activated fault. Correct determination of the number of time frames in which the fault should be activated and observed prevents the test generator from performing unnecessary search in the input space. Second, it operates exclusively in forward time and does not allow assignment of any state values that would need justification.

2.2. Behavioral Test Generation Techniques

Microprocessor Testing

Thatte and Abraham [20] proposed a graph-theoretic approach for functional testing of microprocessors. With this approach a microprocessor is modeled as a directed graph (called the S-graph) where a node represents a register and an edge represents an instruction which transforms information between the nodes. The microprocessor functions considered are register decoding,

instruction decoding and control, data storage, data transfer, and data manipulation. The faults for register and instruction decoding are failures to address the correct register or to execute the correct instruction. The faults specified for the data storage and transfer functions are conventional stuck-at and bridging faults. No specific model is proposed for the data manipulation function because of the wide variety in existing designs for functional units. The generated test sequences consist of a set of valid instructions which can be assembled to produce the test program. Brahme and Abraham [21] expanded Thatte and Abraham's work by developing a comprehensive model of the instruction execution process. They set up codewords so that any simple fault operation on a codeword can produce a non-codeword. These codewords are used to classify faults into types.

Test Generation Using Register Transfer Languages (RTL's)

Su *et al.* [22-24] proposed an approach to generating tests from a circuit model written in RTL's. This approach defines RT (register transfer) level faults such as jump faults, condition faults, data transfer faults, register selection faults, and operator selection faults by perturbing language constructs of RTL's. This approach is based on symbolic execution through paths called function submodules (FS's). An FS is a loop of path in an RTL model starting from the first RT-statement and going back to the first RT-statement. Symbolic execution is a kind of program execution technique which manipulates symbolic values instead of actual values during program execution. This approach generates tests by solving the inequalities between the results from the symbolic execution of the fault-free model and the ones from the symbolic execution of the faulty machine. One main disadvantage of this approach is the cost of deriving symbolic execution paths for all possible test cases and solving the symbolic inequalities.

Test Generation With Hardware Description Languages (HDL's)

Levendel and Menon [19] proposed an approach to generating tests from a circuit model written in (computer) hardware description languages. This approach defines stuck-at faults and behavioral faults such as control faults and general function faults by perturbing language constructs of HDL's. Using the D-cubes similar to the ones used in the D-algorithm, it derives D-propagation conditions for switching algebraic expressions, nonswitching algebraic expressions (such as shifting, addition, counting, decoding, and encoding), and control constructs. In case of HDL blocks, it performs the propagation from the inputs or state variables of such blocks to their outputs. The propagation method depends on the language type, i.e., procedural and nonprocedural. For a procedural language, the propagation of a value through an HDL block is performed by choosing an output variable and determining the input values which are required to sensitize a path toward the variable. For a nonprocedural language, it is performed by choosing an output variable and taking into account every construct inside the HDL block which can affect the output variable. One main disadvantage of this approach is the cost of deriving the D-cubes and manipulating the D-cubes for test generation.

Barclay and Armstrong [27,28] proposed an approach to generating tests from behavioral VHDL models. This approach defines behavioral faults by perturbing language constructs of VHDL: control constructs such as *if-then-else*, *case-when*, and *signal assignment* and micro-operations such as *logic*, *relational*, and *arithmetic operations*. It uses artificial intelligence techniques of AND/OR goal trees and goal solving to represent and manipulate sensitization, justification, and propagation requirements. An AND/OR goal tree is a data structure used to break a large problem into small pieces until each piece is small enough to be solved directly. The algorithm uses eleven types of low-level goals which represent the requirements during test generation. O'Neill [30] enhanced this approach by defining only three high-level goal types and using methods which can

reduce the search space created during test generation. Jani [32] adopted values R (rising edge) and F (falling edge), and used a timing model where R or F is represented in a single time period. Jani also made it possible to generate tests for VHDL models with multiple process statements. Lam [33] improved BTG by presenting methods to handle reconvergent fanout and feedback loops. Baweja [34] incorporated a set of heuristics into BTG, which is useful for generating test vectors for micro-operations representing complex logic.

Chapter 3. Behavioral VHDL Model for Test Generation

This chapter discusses behavioral VHDL models for test generation. First, the characteristics of behavioral models are discussed. Second, the types of behavioral models proper for test generation are discussed. Third, a subset of VHDL is defined for behavioral models for test generation. Fourth, the construction of behavioral models consisting of equivalent process statements is discussed. Fifth, a modeling method which can distinguish a clocked behavior from a non-clocked behavior is discussed. Finally, how different types of VHDL models affect test generation is discussed.

3.1. Behavioral Model

Traditionally, digital circuits have primarily been modeled at the gate level. A gate level model consists of primitive logic gates and their interconnections (wires). Faults are defined on wires (stuck-at faults or bridging faults), and test generation is performed by propagating the fault effects through logic gates and justifying logic values necessary for the propagation.

With the increasing complexity of VLSI circuits, test generation at the gate level has become time-consuming. Furthermore, some library cells used in VLSI circuits cannot be expressed with primitive logic gates [43]. This necessitates modeling of faults at a higher level of abstraction and generation of tests based on these faults.

By the definition in [44], a behavioral model is a model in which a component is modeled by defining its input/output response (behavior) by means of a procedure, not in terms of an interconnection of primitive components. The behavior of a digital component is usually defined in a function table or specified in natural languages. Both function tables and natural languages express *activities* which are performed when some *conditions* are *true*. Activities are primarily arithmetic, logic, and data transfer operations. Conditions represent the control under which activities occur. As an example, the following English specification is considered:

"When the *reset* input is LOW, the counter is incremented on a falling edge of the clock."

In this specification, the activity is "the counter is incremented" and the conditions are "when the *reset* input is LOW" and "on a falling edge of the clock".

Hardware description languages (HDL's) are a suitable tool for describing such activities and conditions. Typical HDL's [29,45,46] provide rich constructs which can describe various arithmetic, logic, and data transfer operations. They also provide block-structured control constructs such as *if* statements and *case* statements which are convenient for describing conditions. Using VHDL, an IEEE standard HDL [29], the above English specification can be described as follows:

```
if (RESET = '0') and (CLK = '0' and CLK'EVENT) then
  COUNT <= INC (COUNT);
end if;
```

In the VHDL code, the two conditions are expressed as (RESET = '0') and (CLK = '0' and CLK'EVENT), respectively. Since CLK'EVENT is TRUE when CLK has just changed its value, the expression (CLK = '0' and CLK'EVENT) correctly describes a falling edge of CLK. The

activity is described using the signal assignment statement including the function INC. Since the signal assignment statement is inside the *if* statement containing the conditions, it is easy to find that the activity is performed when the conditions are *true*.

3.2. Behavioral Models for Test Generation

Since HDL's provide rich constructs, there may be many different ways of modeling a circuit at the behavioral level. A behavioral model for test generation needs to be developed considering two aspects - the complexity of the overall test generation procedure and the quality of generated test vectors. The test generation procedure mainly consists of the activation of faults, the propagation of the fault effects through HDL constructs, and the justification of the values necessary for the activation and the propagation. Therefore, it is important to select the types of HDL constructs through which the activation, the propagation, and the justification are easily accomplished. In order to generate high quality test vectors which can detect high percentage of physical defects, a model which most closely describes real hardware is desirable. However, the complexity of test generation should be considered when such a model is developed.

The behavior of a circuit can be modeled using sequential statements such as variable assignment statements and loop statements. With this type of model, variables are often defined for temporary storage for arithmetic and logic operations or indices for loop statements. Figure 1 shows a VHDL model of a *ones counter* (adopted from [44]) which counts the number of ones in a 3-bit input vector and outputs the number in a 2-bit vector. This model consists of two parts. The first part counts the number of logic 1's using a loop statement which includes an *if* statement, which in turn includes a variable assignment statement. The result of the computation

is stored in an integer variable NUM. The second part converts the integer variable to a 2-bit vector.

```
entity ONES_CNT is
port (A: in BIT_VECTOR(0 to 2);
      C: out BIT_VECTOR(0 to 1));
end ONES_CNT;

architecture FIRST of  ONES_CNT is
begin
  process(A)
    variable NUM: INTEGER range 0 to 3;
  begin

    NUM := 0;
    for I in 0 to 2 loop
      if (A(I) = '1') then
         NUM := NUM + 1;
      end if;
    end loop;

    case NUM is
      when 0 => C <= "00";
      when 1 => C <= "01";
      when 2 => C <= "10";
      when 3 => C <= "11";
    end case;

  end process;
end FIRST;
```

Figure 1. A model of the ones counter with a loop statement and a case statement

It is usually complex to propagate a fault through a loop statement. It is even more complex to justify logic values necessary for the propagation using a loop statement. For example, let us consider a case when a decimal value 2 needs to be assigned to NUM during a test generation procedure. This can be accomplished by the following steps:

1. Since the initial value of NUM is 0 (it is known from the first statement NUM := 0;), the variable assignment statement (NUM := NUM + 1;) should be executed twice to obtain NUM = 2.
2. For the variable assignment statement to be executed twice, the condition expression (A(I) = '1') of the *if* statement should be satisfied twice during the execution of the loop statement.
3. Now, the remaining problem is to select two bit positions with a logic 1 from the three bit positions (0, 1, and 2). The set of possible pairs of bit positions is {(0,1), (0,2), (1,2)}. Hence, the set of possible values for A(0 to 2) is {110, 101, 011}.

This case is realtively simple because the range of the index I of the loop statement is small. However, it still requires a few steps of reasoning which would result in a complex test generation procedure.

In general, a behavioral model consisting of variable assignment statements and loop statements makes the test generation procedure prohibitively complex. In addition, the quality of the test vectors generated from this type of models would be low because the implementation detail necessary for obtaining high quality test vectors is hidden. For these reasons, the VHDL constructs *variables, variable assignment statements, and loop statements will not be considered for test generation in this dissertation.* This is not a major restriction because algorithmic models including these constructs can be translated into data flow models using high level synthesis tools [47].

The ones counter can also be modeled as shown in Figure 2 by removing the loop statement from the model in Figure 1. This model has a case statement which maps the input to the output using a set of *when* clauses and signal assignment statements. Since the relationship among the signals

is explicitly described in this type of model, the propagation and justification procedure is simpler than that in the previous model. For example, to justify a value 10 for C, the third clause of the case statement needs to be executed, which requires that the value of A should be 011, 101, or 110. This is a straightforward procedure compared to the one used for the loop statement in the previous model.

```
architecture SECOND of ONES_CNT is
begin
  process(A)
  begin
    case A is
      when "000" => C <= "00";
      when "001"|"010"|"100" => C <= "01";
      when "011"|"101"|"110" => C <= "10";
      when "111" => C <= "11";
    end case;
  end process;
end SECOND;
```

Figure 2. A model of the ones counter with a case statement using signals

The ones counter can also be modeled using boolean equations as shown in Figure 3. Since this model consists of only primitive logic operators (*not*, *and*, and *or*), the propagation and justification procedure is basically the same as the one used in gate level test generation. However, the test generation procedure will be prohibitively complex if every module in a VLSI circuit is modeled using primitive logic operators.

```
architecture of THIRD of ONES_CNT is
begin
 C(1) <= (A(1) and A(0)) or (A(2) and A(0)) or (A(2) and A(1));
 C(2) <= (A(2) and not A(1) and not A(0)) or  (not A(2) and A(1) and not A(0))
          or (not A(2) and not A(1) and A(0)) or (A(2) and A(1) and A(0)) ;

end THIRD;
```

Figure 3. A model of the ones counter with only signal assignment statements.

In general, to obtain high quality test vectors, a behavioral model should describe its underlying hardware in as much detail as possible. However, the top-down design procedure in VLSI circuit design may not provide the detailed information on hardware necessary for generating high quality test vectors. Even when this information is available, the high complexity of test generation may preclude the use of the detailed information on the hardware. Therefore, behavioral models for test generation should be created considering the tradeoff between the quality of test vectors and the complexity of the test generation procedure.

3.3. VHDL subset

VHDL is selected as a tool for developing behavioral models for test generation since VHDL is an IEEE standard HDL, and is widely used in industries for modeling digital circuits. VHDL has the richest constructs among available HDL's and provides variety and flexibility in modeling digital circuits [44,48]. However, as discussed in the previous section, variables, variable assignment statements, and loop statements will not used because they make the test generation procedure complex. In this section, a subset of VHDL which can be used for behavioral models for test generation is defined.

Objects

Signals are allowed. Ports are also allowed since a port is a signal declared in the interface list of an entity declaration. The modes for ports *in*, *inout*, and *out* are allowed. Literals (constant values) are allowed. Variables are not allowed for the reason discussed in the previous section.

Types

Bits and bit vectors are allowed. Slicing of bit vectors is allowed. Booleans and integers are not allowed. However, it should be noted that booleans can be transformed into bits, and integers into bit vectors. While boolean signals are not allowed, expressions whose result type is boolean are allowed. For example, a relational expression (A < B) is allowed as the condition expression of an *if* statement.

Arithmetic and Logic Operators

Logic operators *and*, *or*, *not*, *nand*, *nor*, and *xor* and relational operators =, /= (not equal), <, and <= (less than or equal) are allowed. Arithmetic operators ADD, SUB, INC, and DEC are allowed. As these are not built-in VHDL operators, they are assumed to be defined in VHDL functions.

Attributes

Only attributes 'STABLE and 'EVENT are used. As will be discussed in Section 3.5, these are used for modeling the behavior of clocks.

Concurrent Statements and Sequential Statements

VHDL models basically consist of concurrent statements which are executed asynchronously with respect to each other. The whole set of concurrent statements is defined in VHDL LRM [29] as follows:

```
concurrent_statement ::=
    block_statement
```

| process_statement
| concurrent_procedure_call
| concurrent_assertion_statement
| concurrent_signal_assignment_statement
| component_instantiation_statement
| generate_statement

Of these concurrent statements, only block statements, process statements, and concurrent signal assignment statements are used for test generation. The other four statements are not used for the following reasons:

- A concurrent procedure call which calls a procedure is not allowed because a procedure usually consists of variable assignment statements and loop statements.
- A concurrent assertion statement is used for reporting messages, not for describing the behavior of a circuit.
- A component instantiation statement is used for describing the structure of a component, not the behavior.
- A generate statement is not used because it is essentially the same as a loop statement.

Of the three concurrent statements selected for test generation, only the process statement can have sequential statements inside it. The whole set of sequential statements is defined as follows [29]:

sequential_statement ::=
wait_statement
| assertion_statement
| signal_assignment_statement
| variable_assignment_statement
| procedure_call_statement
| if_statement
| case_statement
| loop_statement
| next_statement

| exit_statement
| return_statement
| null_statement

Of these sequential statements, only signal assignment statements, *if* statements, and *case* statements are used. Other statements are not used for the following reasons:

- A wait statement is used to cause the suspension of a process statement or a procedure. Wait statements will not be used because the suspension makes the propagation and justification procedure too complex.
- Variable assignment statements are not used for the reason discussed in the previous section.
- An assertion statement and a procedure call statement are basically the same as a concurrent assertion statement and a concurrent procedure call, respectively.
- Next statements and exit statements are used with loop statements.
- Return statements are used in a procedure, which will not be used.
- A null statement has no effect other than to pass on to the next statement in a VHDL code.

3.4. Behavioral Models of Equivalent Process Statements

Although only a subset of VHDL is used for behavioral models, it still includes many different types of VHDL constructs. For gate level test generation, propagation and justification rules are relatively simple because rules are defined only for primitive logic operators. For example, a D at an input of an AND gate can be propagated to the output by assigning logic 1's to the other inputs. These rules are complex for behavioral test generation because different rules should be

defined for various VHDL constructs. The number of these rules can be reduced by using the equivalence among concurrent statements.

The three types of concurrent statements selected in the previous section are process statements or equivalent process statements. Hence, the test generation rules developed for process statements can be used for the equivalent process statements transformed from concurrent signal assignment statements and block statements. In this section, the basic structure of the process statement is first discussed and then the construction of equivalent process statements from other concurrent statements is discussed. Finally, various types of connections among process statements are defined.

3.4.1. The Process Statement

Shown below is the production rule for a process statement [29].

process_statement ::=
  [*process*_label :]
    process [(sensitivity_list)]
      { process_declarative_item }
    begin
      { sequential_statement }
    end process [*process*_label] ;

Two main parts of a process statement are a sensitivity list and a set of sequential statements. A sensitivity list consists of one or more signals. A set of sequential statements inside a process statement is executed when a signal in the sensitivity list changes its value. The types of sequential statements selected in the previous section are signal assignment statements, *if* statements, and *case* statements. Both *if* statements and *case* statements can recursively have any of the three types of statements inside them. However, the inner-most sequential statements

should always be signal assignment statements. *If* statements and *case* statements provide the conditions on which the signal assignment statements inside them are selected. Hence, the process statement is a proper construct to describe *activities* and *conditions* specified for functional modules in a digital circuit.

3.4.2. Construction of Equivalent Process Statements

In this section, how other concurrent statements are mapped into process statements is discussed.

From Concurrent Signal Assignment Statements

There are two types of concurrent signal assignment statements - conditional signal assignment statements and selected signal assignment statements. For a given concurrent signal assignment statement, there is an equivalent process statement [29].

The general form of a conditional signal assignment statement and its equivalent process statement is shown below. In each statement, T denotes a target signal, W's denote waveforms, and C's denote conditions. Two options *guarded* and *transport* can be used in a concurrent signal assignment statement. The option *guarded* cannot be used if the statement is not under a block statement with a guard expression. This situation will be explained later in this section. The sensitivity list of the equivalent process statement consists of signals which are inputs of the conditional signal assignment statement. The inputs are signals which appear in W's and C's.

```
T <= [ transport ] W1 when C1 else
                   W2 when C2 else
                        .
                        .
                   W(N-1) when C(N-1) else
                   W(N);
```

```
process (a set of signals in W's and C's)
begin
  if C1 then
     T <= [ transport ] W1;
  elsif C2 then
     T <= [ transport ] W2;
                .
                .
  elsif C(N-1) then
     T <= [ transport ] W(N-1);
  else
     T <= [ transport ] W(N);
  end if;
end process;
```

The general form of a selected signal assignment statement and its equivalent process statement is shown below. E denotes the selection expression of the selected signal assignment statement and is also shown in the *case* statement inside the equivalent process statement. As in the case of the conditional signal assignment statement, the sensitivity list of the equivalent process statement consists of signals which are inputs of the selected signal assignment statement. The inputs are signals which appear in E, W's, and C's.

```
with E select
  T <= [ transport ] W1 when C1,
                     W2 when C2,
                              .
                              .
                     W(N) when C(N);
```

```
process (a set of signals in E, W's, and C's)
begin
  case E is
    when C1 => T <= [ transport ] W1;
    when C2 => T <= [ transport ] W2;
               .
               .
    when C(N) => T <= [ transport ] W(N);
```

```
  end case;
end process;
```

From Block Statements

A block statement can have concurrent statements inside it. Of the three types of concurrent statements (block statements, concurrent signal assignment statements, and process statements), only concurrent signal assignment statements are allowed inside a block statement. This is because otherwise a block statement contains nested block statements or process statements, and thus implies a structural model. Under this assumption, the production rule for a block statement can be expressed as follows:

```
block_statement :==
    block_label :
    block [ ( guard_expression ) ]
        block_declarative part
    begin
        { concurrent_signal_assignment_statement }
    end block [ block_label ] ;
```

If a guard expression exists in a block statement, each concurrent signal assignment statement inside the block statement must have an option *guarded* and is controlled by the guard expression. Hence, the equivalent process statement of a concurrent signal assignment statement can be formed by imposing an additional condition (guard expression) in the beginning of the *if* statement (for a conditional signal assignment statement) or the *case* statement (for a selected signal assignment statement). However, the option *guarded* is not used any more in each signal assignment statement inside the process statement. Let G denote a guard expression. G is added to the sensitivity list of the equivalent process statement for each concurrent signal assignment

statement. The equivalent process statements for a conditional signal assignment statement and a selected signal assignment statement can be expressed as shown below.

```
process (a set of signals in G, W's, and C's)
begin
 if G then
   if C1 then
     T <= [ transport ] W1;
   elsif C2 then
     T <= [ transport ] W2;
                .
                .
   elsif C(N-1) then
     T <= [ transport ] W(N-1);
   else
     T <= [ transport ] W(N);
   end if;
 end if;
end process;
```

```
process (a set of signals in G, E, W's, and C's)
begin
 if G then
   case E is
     when C1 => T <= [ transport ] W1;
     when C2 => T <= [ transport ] W2;
                .
                .
     when C(N) => T <= [ transport ] W(N);
   end case;
 end if;
end process;
```

The following example shows a block statement with a guard expression which expresses the behavior of a JK flip-flop and its equivalent process statement.

```
  B: block (CLK = '1' and CLK'EVENT)
  begin
    Q <= guarded Q when J= '1' and K = '1' else
                      '1' when J = '1' and K = '0' else
```

```
                    '0' when J = '0' and K = '1' else
                    not Q;
end block B;


process (CLK, J, K,Q)
 begin
   if CLK = '1' and CLK'EVENT then
     if J = '1' and K = '1' then
       Q <= Q;
     elsif J = '1' and K = '0' then
       Q <= '1';
     elsif J = '0' and K = '1' then
       Q <= '0';
     else
       Q <= not Q;
     end if;
   end if;
end process;
```

3.4.3. Connectivity Among Process Statements

Any VHDL model consists of an entity declaration and an architecture body. The entity declaration specifies primary inputs and primary outputs of the model. The architecture body consists of concurrent statements which specify the behavior of the model. Since each concurrent statement other than the process statement has an equivalent process statement, an architecture body is essentially a set of process statements which are connected to each other with ports and signals.

Each process statement contains a sensitivity list and a set of sequential statements. The inputs to a process statement appear on the right-hand side expressions of signal assignment statements and the control expressions of *if* and *case* statements. Inputs can be divided into two groups depending on whether they appear in the sensitivity list or not. Hence, the set of signals in the

sensitivity list of a process statement is a subset of the set of input signals to the process statement. The outputs of a process statement appear on the targets of signal assignment statements. The following notation is used for defining the connectivity among process statements. For convenience, a process statement will be abbreviated as a *process*.

p: a process inside a behavioral model

I_p: a set of input signals to a process p

S_p: a set of input signals in the sensitivity list of a process p

O_p: a set of output signals from a process p

In fact, S_p is a subset of I_p for a given process p. An ordered pair (i, j) of two processes i and j is defined to be *connected* if a signal s in O_i is also included in I_j. If a signal s in O_i is also included in S_j, a value change (event) on s causes the execution of the process j. In this situation, (i, j) is defined to be *strongly connected*. If a signal s in O_i is included in I_j, but is not included in S_j, a value change (event) on s cannot cause the execution of j. In this situation, (i, j) is defined to be *weakly connected*. The connectivity among processes can be defined using set notation as follows:

Definition: An ordered pair (i, j) is *connected* if $O_i \cap I_j \neq \varnothing$.

Definition: An ordered pair (i, j) is *strongly connected* if $O_i \cap S_j \neq \varnothing$.

Definition: An ordered pair (i, j) is *weakly connected* if $O_i \cap I_j \neq \varnothing$ and $O_i \cap S_j = \varnothing$.

There are cases when an output of a process is used as an input for more than one process. This connection forms a fanout point, i.e., the output is a stem and the inputs are branches. Hence, for

the same signal name, at least three lines (one stem and at least two branches) are defined. This interpretation will be used in Chapter 4 for constructing the list of stuck-at faults.

Handling of Bus Resolution Functions

There are cases when a signal *s* in O_i is also included in O_j. In this case, the signal *s* has two drivers and hence implies a bus where the two drivers contend for the same destination. When a signal has multiple drivers, a bus resolution function (BRF) is required to resolve the values of the multiple drivers into a single value for the signal. The following four types of BRF's are considered:

1. The *one-hot* BRF
2. The wired-X BRF
2. The *wired-or* BRF
3. The *wired-and* BRF

The one-hot BRF is a function which returns a logic value V if one and only one driver has V and other drivers have Z. One possible implementation of the one-hot BRF for logic values ('0', '1', 'X', 'Z') is adopted from [44]:

```
type MVL4 is ('0', '1', 'X', 'Z');
type MVL4_VECTOR is array (NATURAL range <>) of MVL4;

function OneHot (V: MVL4_VECTOR) return MVL4 is
  variable result: MVL4 := 'Z';
  variable got_one: BOOLEAN := FALSE;
begin
  for I in V'RANGE loop
    next when V(i) = 'Z';
    if got_one then
```

```
      result := 'X';
      return result;
    end if;
    got_one := TRUE;
    result := V(i);
  end loop;
  return result;
end OneHot;
```

The wired-X BRF is a function which returns a logic value V if at least one driver has V and other drivers have Z. If a driver has a logic value V and another driver has $\overline{V}$, it returns a value X. The wired-or (wired-and) BRF is a function which returns the result of the logical *or* (*and*) function of the drivers. For both wired-or and wired-and, both 1's and 0's predominate over Z's. The wired-or is dominated by 1 and detection of 1 produces 1. In contrast, the wired-and is dominated by 0 and detection of 0 produces 0.

BRF's usually include variable assignment statements and loop statements, which are not included in the VHDL subset defined in Section 3.3. Therefore, test generation rules are not defined for the VHDL constructs inside BRF's. Instead, test generation rules are predefined in the test generation algorithm for each type of BRF's. The user specifies the type of each BRF, and when a BRF is encountered during the test generation procedure, the test generation algorithm selects and applies proper rules predefined for the type of the BRF. The test generation rules for BRF's will be discussed in Chapter 5.

3.5. Modeling of Clocked and Non-clocked Behavior

The behavior of digital circuits can be divided into two types, *clocked* and *non-clocked*. A clocked behavior is one whose activity is controlled by a clock signal. Hence, the synchronous

behavior of any sequential circuit is a clocked behavior. For example, the activity *increment* of a counter circuit is a clocked behavior since the counter is incremented on an edge of a clock signal. In contrast, a non-clocked behavior is one whose activity is not controlled by a clock signal. Hence, the behavior of any combinational circuit is a non-clocked behavior. An asynchronous behavior of a sequential circuit is also a non-clocked behavior. For example, the activity *reset* of a counter circuit is a non-clocked behavior because the counter is reset with a specified value of the reset signal regardless of the status of its clock signal.

A clock signal is easily modeled in VHDL using the attributes 'STABLE or 'EVENT. Let S denote a signal object. S'STABLE is FALSE when S has just changed. S'EVENT is TRUE when S has just changed. Hence, the expression (S = '1' and not S'STABLE) or (S = '1' and S'EVENT) represents a positive edge on the signal S. Similarly, (S = '0' and not S'STABLE) or (S = '0' and S'EVENT) represents a negative edge on S. Therefore, any statement which is controlled by one of these expressions indicates a clocked behavior. In a process statement, this type of expressions appears in the conditional expressions of *if* statements. Shown below is a process statement of a positive edge-triggered D flip-flop. Since the *if* statement contains the attribute 'EVENT, the behavior expressed by the *if* statement is considered a clocked behavior.

```
process(CLK)
begin
  if (CLK = '1' and CLK'EVENT) then
     Q <= D;
  end if;
end process;
```

In the above process statement, only one signal exists in the sensitivity list. For the simulation purpose, the process statement can be written without using the attribute 'EVENT as shown

below. Although this statement expresses the same intended behavior, this type of model should be avoided because a clocked behavior cannot be distinguished from a non-clocked behavior.

```
process (CLK)
begin
   if (CLK = '1') then
      Q <= D;
   end if;
end process;
```

Hence, *a clock signal must be modeled using the attribute 'STABLE or 'EVENT.* With this rule applied to any VHDL model for test generation, any signal accompanied by 'STABLE or 'EVENT is easily recognized as a clock signal. In the following VHDL code representing a positive edge-triggered D flip-flop with an active LOW reset input, the test generation algorithm easily recognizes RESET as a non-clock signal and CLK as a clock signal by checking the presence of 'EVENT.

```
process (RESET, CLK)
begin
  if (RESET = '0') then
    Q <= '0';
  else if (CLK = '1' and CLK'EVENT) then
          Q <= D;
       end if;
  end if;
end process;
```

3.6. Types of Behavioral Models vs. Test Generation

A behavioral model of a circuit can be developed in different ways depending on how much in detail the circuit is modeled and which VHDL constructs are used for the circuit model. In this section, how different types of VHDL models affect test generation is discussed.

3.6.1. Multiple Functional Modules with A Single Control

A digital circuit frequently has a set of modules which are controlled by a single control signal. This situation can be modeled in two ways. First, a single process statement is used and the activities of the modules are controlled by a single *case* statement which expresses the control. Second, multiple process statements are used and the activity of each module is controlled under a separate *case* statement existing in each process statement. As an example, two different types of models are introduced for a circuit where two 4 x 1 multiplexers are controlled by a single control signal. The following VHDL code illustrates the first modeling style.

```
process (S, A, B)
begin
  case S(0 to 1) is
     when "00" => X <= A(0);  Y <= B(0);
     when "01" => X <= A(1);  Y <= B(1);
     when "10" => X <= A(2);  Y <= B(2);
     when "11" => X <= A(3);  Y <= B(3);
   end case;
end process;
```

This model implies the hardware structure in Figure 4.a, where only one line exists for the control signal S. This model can be written as shown below using the second modeling style.

```
process (S, A)
begin
  case S is
     when "00" => X <= A(0);
     when "01" => X <= A(1);
     when "10" => X <= A(2);
     when "11" => X <= A(3);
   end case;
end process;

process (S, B)
begin
```

```
    case S is
        when "00" => Y <= B(0);
        when "01" => Y <= B(1);
        when "10" => Y <= B(2);
        when "11" => Y <= B(3);
    end case;
end process;
```

The second model implies the hardware structure shown in Figure 4.b, where a fanout point is formed for the control signal S. Since stuck-at faults are defined for fanout stems and fanout branches, more faults are defined in the two-process model than in the one-process model. Hence, tests generated from the two-process model can detect more gate level stuck-at faults than those generated from the one-process model.

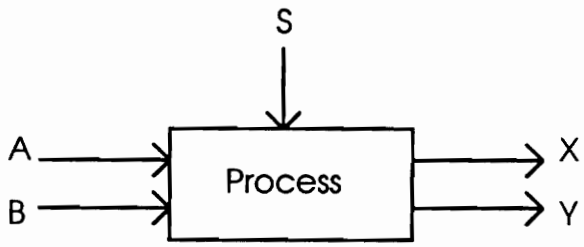


Figure 4.a. Hardware Structure for the one-process model (Multiplexer)

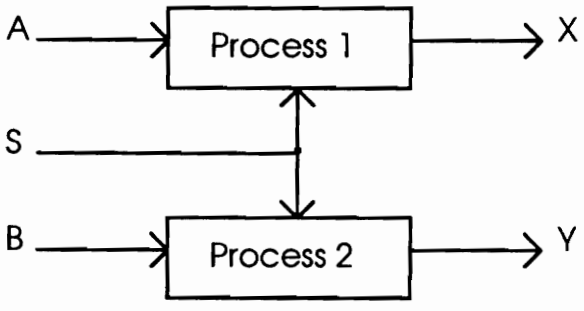


Figure 4.b. Hardware Structure for the two-process model (Multiplexer)

3.6.2. A Functional Module with Multiple Activities

The behavior of a functional module consists one or more *activities* performed on some conditions. When only one activity is defined for a functional module, it is in most cases modeled in a single process statement. However, when more than one activity is defined for a functional module, they can be modeled in either a single process statement or in two or more process statements. As an example, consider a positive edge-triggered flip-flop with a reset input (active LOW). This circuit has two activities: *transfer* and *reset*. If two process statements are used, this circuit could be modeled as follows:

```
A: process (RESET)
begin
   if RESET = '0' then
      Q <= '0';
   end if;
end process;

B: process(CLK)
begin
  if (RESET = '1') and (CLK = '1' and CLK'EVENT) then
     Q <= D;
  end if;
end process;
```

In the model, the process statements A and B express the activities *reset* and *transfer*, respectively. This model implies the hardware structure shown in Figure 5.a. This model makes the test generation complex for two reasons. First, since the output Q appears in both A and B, this model needs a bus resolution function which requires special test generation rules. Second, since the primary input RESET appears in both A and B, RESET forms a fanout point. Since

stuck-at faults are defined for every fanout stem and fanout branch, more faults are generated from this model. The circuit can also be modeled as a single process as shown below. This model implies the hardware structure shown in Figure 5.b. Test generation for this model is simpler than that for the previous model because no fanout point or bus resolution function is involved in test generation.

```
process (RESET, CLK)
begin
  if RESET = '0' then
    Q <= '0';
  else if (CLK = '1' and CLK'EVENT) then
          Q <= D;
        end if;
  end if;
end process;
```

In summary, *if a functional module has more than one activity, it makes the test generation procedure simpler to model the activities with a single process statement than to model the activities with two process statements.*

3.7. Summary

In this chapter, behavioral models proper for test generation have been discussed. Using VHDL constructs which are related to real hardware and the fact that every concurrent statement has its equivalent process statement, a behavioral model is expressed as a set of equivalent process statements. This type of behavioral models leads to efficient test generation since behavioral faults and propagation and justification rules are defined only for the VHDL constructs constituting process statements. The usage of attributes 'STABLE and 'EVENT for identifying a

clocked behavior from a non-clocked behavior has also been discussed. This feature is used in Chapter 5 for assigning time periods for sequential circuits. Finally, types of VHDL models which lead to an efficient test generation procedure have been discussed.

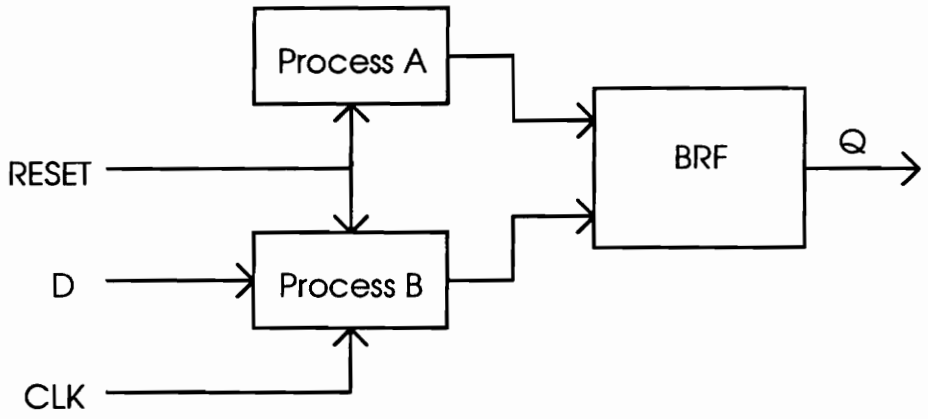


Figure 5.a. Hardware Structure for the two-process model (D flip-flop)

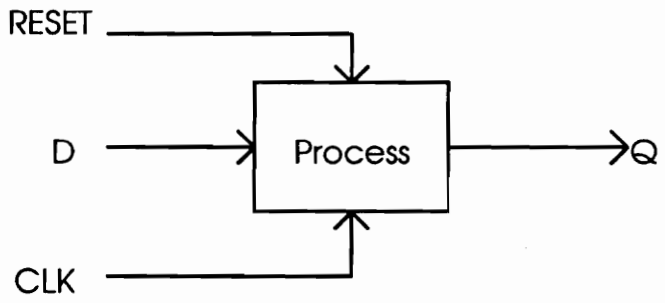


Figure 5.b. Hardware Structure for the one-process model (D flip-flop)

Chapter 4. Behavioral Fault Model

In contrast to gate level test generation where stuck-at faults are defined on signal lines, behavioral faults are defined by perturbing VHDL constructs used in behavioral models. In this chapter, the definitions of the previous behavioral fault model defined in [27,28,30-34] are reviewed, and their drawbacks are discussed. A new behavioral fault model consisting of behavioral stuck-at faults, behavioral stuck-open faults, and micro-operation faults is presented, and the characteristics of each fault type are discussed. Finally, an algorithm is presented which enumerates a list of behavioral faults from a behavioral VHDL model.

4.1. Model Perturbation

Traditional gate level test generation approaches [1-16] use gate level models consisting of primitive logic gates and signal lines (primary inputs, primary outputs, and internal lines connecting the primitive logic gates). Stuck-at faults are defined for all signal lines included in a gate level model. A stuck-at fault for a signal line l indicates that l is permanently stuck at a certain logic value regardless of the logic value transferred to l. Two types of stuck-at faults, stuck-at-0 (SA0) and stuck-at-1 (SA1), are assigned for every signal line in a gate level model.

Gate level stuck-at faults were closely tied to physical defects when TTL technologies were used to implement digital circuits. For example, when an input of a TTL gate is open, the value of the input remains at logic 1, which can be modeled as a stuck-at-1 fault on the input. In general, the stuck-at fault model has a close relationship to the physical defects encountered in systems built from SSI/MSI TTL devices mounted on printed circuit boards.

For today's circuits implemented with CMOS technologies, many physical defects cannot be modeled with gate level stuck-at faults. Physical defects are usually caused by process instabilities and contamination during the IC fabrication process. Physical defects can be translated into circuit level faults such as line stuck-at faults, transistor stuck-ON/OFF faults, floating line faults, and bridging faults [49]. It has been reported in [50] that only approximately 50% of circuit level faults can be modeled as gate level stuck-at faults. In the absence of a close tie to physical defects, the stuck-at fault model has, in essence, become a form of model perturbation.

As discussed in Section 3.1, a behavioral model describes a set of activities performed under some conditions. A faulty behavior is modeled by perturbing one of VHDL constructs describing activities and conditions to an erroneous one. Consequently, a set of behavioral faults is defined by perturbing VHDL constructs in a behavioral VHDL model. It has been shown in Section 3.4 that a behavioral model consisting of the subset of VHDL constructs (Section 3.3) can be converted into a model of equivalent process statements each of which can contain three types of sequential statements (signal assignment statements, *if* statements, and *case* statements). Hence, behavioral faults are defined on the VHDL constructs used for the three types of sequential statements, i.e., *if-then-else*, *case-when*, *assignment* (<=), signals, and operators.

4.2. The Previous Behavioral Fault Model

4.2.1. Definitions

An approach of modeling behavioral faults by perturbing VHDL constructs was first proposed in [27]. This approach uses a *single behavioral fault model*, i.e., only one behavioral fault is assumed to occur at a time. Five types of behavioral faults are defined by perturbing various VHDL constructs. Their definitions are given below.

Definition: A **stuck-then (stuck-else) fault** is a fault where the set of statements under the *then (else)* clause of an *if* statement is always executed regardless of the value of its condition expression.

Definition: A **dead-clause fault** is a fault where a clause of a *case* statement is selected, but the set of statements under the clause is not executed.

Definition: An **assignment control fault** is a fault where the value of the source expression (right-hand side expression) of a signal assignment statement is not correctly transferred to its target.

Definition: A **stuck-data fault** is a fault where all bits of a signal s on a data path are permanently stuck at the same logic value (1 or 0) regardless of the value transferred to s. Two stuck-data faults (stuck-at-1's and stuck-at-0's) are defined for each signal.

A micro-operation is defined before defining a micro-operation fault.

Definition: A **micro-operation (MOP)** is an operation where one or more signals are manipulated by a logic, a relational, or an arithmetic operator.

Definition: A **micro-operation fault** is a fault where an operator in a micro-operation is faulted to another operator.

The previous behavioral fault model is useful for comprehensively perturbing VHDL constructs in a behavioral model under test. However, this approach has the following drawbacks:

1. The previous fault model generates a significant number of unnecessary behavioral faults because some types of faults are equivalent to others. Hence, the fault equivalence between behavioral faults needs to be analyzed to exclude unnecessary faults. The relationship between behavioral faults is discussed in Section 4.2.2.

2. Since test generation rules are defined for each type of behavioral faults, the number of total test generation rules is proportional to the number of fault types. The number of fault types needs to be reduced, if possible, in order to make the test generation procedure simpler. Again, this can be accomplished by using the fault equivalence between behavioral faults.

3. Vector-wise stuck-data faults are defined considering the situation where every bit in a data bus or an address bus is faulted in the same way. Only limited patterns of logic values are generated from stuck-data faults. For example, the logic value 00000000 is selected to activate a stuck-at-11111111 fault for an 8-bit signal, and 11111111 is selected to activate a stuck-at-00000000 fault. These values are not sufficient for detecting possible physical defects on the 8-bit signal lines. This necessitates the usage of bit-wise stuck-at faults.

4. Stuck-data faults are defined only for signals on data paths. In case that a control expression of an *if* statement or a *case* statement is complex, bit-wise stuck-at faults need to be defined for the signals within the control expression to obtain high quality test vectors. If stuck-at faults are also defined for unnamed signals formed by control expressions, stuck-then/stuck-else faults can be removed from the fault list because they are equivalent to the stuck-at faults. This will be discussed in detail in the next sub-section.

4.2.2. Equivalent Faults

Stuck-then/Stuck-else Faults vs. Stuck-at Faults

Both a stuck-then fault and a stuck-else fault are defined for an *if-then-else* construct. The *then* (*else*) clause of an *if* statement is executed when the value of its condition expression is TRUE = logic 1 (FALSE = logic 0). Hence, "if-stuck-at-then" is equivalent to "expression-stuck-at-1". Similarly, "if-stuck-at-else" is equivalent to "expression-stuck-at-0". Hence, if stuck-at faults are also defined for an unnamed signal (this kind of signal is called a "virtual signal" and will be formally defined in Section 4.3.) formed by the condition expression of an *if* statement, the stuck-then (stuck-else) fault is equivalent to the stuck-at-1 (stuck-at-0) fault for the unnamed signal. As an example, let us consider the following VHDL code:

```
if (A = B) then
  Z <= C;
else
  Z <= D;
end if;
```

Let *u* denote the unnamed signal formed by the expression (A = B). Then, the stuck-then fault is equivalent to the *u*-stuck-at-1 fault and the stuck-else fault is equivalent to the *u*-stuck-at-0 fault. To conclude, *stuck-then/stuck-else faults can be removed from the behavioral fault list if stuck-at faults are defined for unnamed signals corresponding to condition expressions.*

Micro-operation Faults for Logic Operators vs. Stuck-at Faults

A heuristic was selected for perturbing logic operators: a logical dual is used to perturb a logic operator. The logical dual of a boolean function is obtained by replacing the variables of the function with their complements and complementing the results [51]. It was shown in [52] that this heuristic is as good as other micro-operation fault models for large circuits and it is better for small circuits. Logical duals of some logic operators used in the approach are shown in the following table, where EQV denotes an equivalence operation and BUF denotes no operation.

Logic Operation	Logical Dual
AND	OR
OR	AND
NAND	NOR
NOR	NAND
XOR	EQV
EQV	XOR
NOT	BUF

Of the seven micro-operation faults in the table, each of the three micro-operation faults (XOR -> EQV), (EQV -> XOR), and (NOT -> BUF) is detected by any test for stuck-at faults on the input(s) of the corresponding operator because each of the logic operators XOR, EQV, and NOT is

perturbed to its complement [51]. Hence, the three micro-operation faults can be removed from the behavioral fault list if bit-wise stuck-at faults are defined on the inputs.

For the remaining four micro-operation faults, it is shown that each micro-operation fault is detected by a test for a stuck-at fault on the inputs of the corresponding logic operator.

1. An (AND -> OR) fault is detected if at least one input is logic 0 and at least one input is logic 1. A stuck-at-1 fault on an input of an AND operator is detected if the input is set to logic 0 and all other inputs are set to logic 1. It is derived that a test for a stuck-at-1 fault on any input of an AND operator detects the corresponding (AND -> OR) fault. Hence, (AND -> OR) faults need not be defined.

2. An (OR -> AND) fault is detected if at least one input is logic 0 and at least one input is logic 1. A stuck-at-0 fault on an input of an OR operator is detected if the input is set to logic 1 and all other inputs are set to logic 0. It is derived that a test for a stuck-at-0 fault on any input of an OR operator detects the corresponding (OR -> AND) fault. Hence, (AND -> OR) faults need not be defined.

3. Similarly, (NAND -> NOR) and (NOR -> NAND) faults are detected by the tests for the stuck-at faults on the inputs of the logic operators NAND and NOR.

The same analysis can be applied to an operator having expressions instead of simple signal names as its arguments. The general form of a logic operation is shown as follows:

(expression) logic_operator (expression)

In this case, the micro-operation fault is detected by a test for a stuck-at fault for the unnamed signal formed by one of the expressions.

In general, *a micro-operation fault for a logic operator is detected by a test for a stuck-at fault on one of its arguments (a signal or an unnamed signal for an expression).* Hence, micro-operation faults for logic operators can be removed from the behavioral fault list if bit-wise stuck-at faults are defined for any signal or any unnamed signal for an expression.

Dead-clause Faults vs. Assignment Control Faults

A dead-clause fault is defined for each clause in a *case* statement. Thus, *n* dead-clause faults are defined for a *case* statement with *n* clauses. The inner-most statement of each clause is always a signal assignment statement. When a dead-clause fault occurs in a *case* statement, a set of signal assignment statements under the clause is not executed. If only one signal assignment statement exists under the selected clause, the dead-clause fault is equivalent to the assignment control fault defined for the statement. If more than one signal assignment statement exists, the dead-clause fault in fact causes more than one assignment control fault. A VHDL code illustrating the situation is shown below. In the code, **c**'s and **s**'s indicate clause numbers and statement numbers, respectively.

```
    case S(1 to 2) is
c1: when "00" => X <= A;  s1
                 Y <= E;  s2
c2: when "01" => X <= B;  s3
                 Y <= F;  s4
c3: when "10" => X <= C;  s5
                 Y <= G;  s6
c4: when "11" => X <= D;  s7
```

```
                    Y <= H;   s8
    end case;
```

Let us consider the dead-c2 fault. When this fault occurs, neither of the statements s3 and s4 is executed. As a result, two assignment control faults simultaneously occur. However, this contradicts to the assumption of a single behavioral fault model. Hence, this kind of faults will not be considered.

To conclude, *a dead-clause fault is equivalent to an assignment control fault under the assumption of a single behavioral fault model.* Therefore, dead-clause faults can be removed from the fault list.

4.3. The New Behavioral Fault Model

Since stuck-then/stuck-else faults, dead-clause faults, and micro-operation faults for logic operators can be removed from the fault list, the remaining fault types are stuck-at faults, assignment control faults, and micro-operation faults for arithmetic or relational operators. These will be renamed as behavioral stuck-at faults, behavioral stuck-open faults, and micro-operation faults, respectively. The new behavioral model is also *a single behavioral fault model*, i.e., it is assumed that only one behavioral fault occurs at a time.

Before defining the three behavioral faults, a *signal*, a *virtual signal*, a *source expression*, a *fanout point*, a *fanout stem*, and a *fanout branch* are defined.

Definition: A **signal** is a VHDL port or a VHDL signal.

Definition: A **virtual signal** is an unnamed signal formed by an expression which is not a simple signal name.

For example, let us consider the following VHDL code.

C <= (X and Y) or ((X xor Y) and Z);

From the above code, the following virtual signals are defined.

(X and Y)
(X xor Y)
((X xor Y) and Z)

When a VHDL model is translated into an intermediate form, VHDL expressions are usually converted to expression trees as in VTIP [53]. Hence, a virtual signal is assigned to every expression tree in an intermediate form.

Definition: A **source expression** is an expression whose value can be referenced during the execution of the statement where it resides.

By the above definition, the right-hand side expression of a signal assignment statement, the condition expression of an *if* statement, and the selection expression of a *case* statement are source expressions. The following VHDL code is considered to illustrate source expressions. In the example, COND, A, B, C, and D are assumed as primary inputs and X and Y are assumed as

primary outputs. In the VHDL code, the expressions COND = '1' (statement s1), A and B (s2), A or C (s3), and X xor D (s4) are source expressions.

```
  process (COND, A, B, C, D, X)
  begin
s1:  if COND = '1' then
s2:    X <= A and B;
     else
s3:    X <= A or C;
     end if;
s4:  Y <= X xor D;
  end process;
```

Definition: A **fanout point** is a point where a driving signal is connected to more than one signal line. When a fanout point is formed, the driving signal line is called a **fanout stem** and every other signal line is called a **fanout branch**.

By the above definition, it follows that there exists one fanout stem and at least two fanout branches for a fanout point. A signal forms a fanout point when it satisfies one of the following two conditions:

Condition 1 - A signal appears more than once in a source expression or appears in more than one source expression.

Condition 2 - A VHDL port of mode *inout* appears in at least one source expression. This condition represents a case when a primary output is used as an input to an expression.

For a given behavioral model, fanout points can be identified using Condition 1 and Condition 2. In the VHDL code shown above, the port A forms a fanout point because it satisfies Condition 1, i.e.,

it appears in more than one source expression (the source expression of s2 and the source expression of s3). The port X is also a fanout point because it satisfies Condition 2, i.e., the primary output appears in the source expression of s4.

Definition: A **behavioral stuck-at (BSA) fault** is a fault where a bit of a signal, a virtual signal, a fanout stem, or a fanout branch is permanently stuck at logic 1 or 0 regardless of the value transferred to the bit. Two stuck-at faults (stuck-at-1 and stuck-at-0) are defined for each bit.

Definition: A **behavioral stuck-open (BSO) fault** is a fault where the value of the source expression (right-hand side expression) of a signal assignment statement is not correctly transferred to its target. It is identical with an assignment control fault defined in the previous behavioral fault model.

A micro-operation fault is redefined for the new behavioral fault model as follows. In the definition, a relational operator is assumed to have multi-bit arguments. This is because a relational operator with single-bit arguments can be considered as a logic operator. For example, the expression (A(1) =/ B(1)) is equivalent to the expression (A(1) xor B(1)).

Definition: A **micro-operation (MOP) fault** is a fault where an arithmetic or a relational operator is faulted to another operator.

Three types of faults will be explained in detail in the following three sections.

4.4. Behavioral Stuck-at Faults

By definition, behavioral stuck-at (BSA) faults are defined for every possible signal line formed by signals in a behavioral model. If a signal is a bit vector, both SA1 and SA0 are defined for each bit of the vector. In general, for an *n*-bit signal, *2n* single BSA faults are defined. For example, for a bit vector A(1 to 2), four single BSA faults A(1)-SA1, A(1)-SA0, A(2)-SA1, and A(2)-SA0 are defined. Although BSA faults are defined in the same way as gate level stuck-at faults, the number of BSA faults is much smaller than that of gate level stuck-at faults because a behavioral VHDL model is described at a higher level of abstraction.

Behavioral Stuck-at Faults for Virtual Signals

BSA faults are defined for every possible virtual signal in a behavioral model. There are cases when BSA faults for a virtual signal are equivalent to BSA faults for a signal inside the expression corresponding to the virtual signal. This case occurs when a virtual signal is defined for an expression (S = C), where S is a single bit signal and C $\in$ {0,1}. Let V denote the virtual signal for this expression. Then, the stuck-at-1 fault for V is equivalent to the stuck-at-C fault for S. Similarly, the stuck-at-0 fault for V is equivalent to the stuck-at-$\overline{C}$ fault. Hence, BSA faults for S can be removed from the fault list.

BSA faults can also be defined for virtual signals for expressions within an expression ((S = C) and (S'EVENT)) where S denotes a single bit signal and C $\in$ {0, 1}. As discussed in Section 3.5, this type of expression indicates a clock. If C = 1, the expression indicates a rising edge ('R') of a clock S. If C = 0, it indicates a falling ('F') edge of S. In this case, two expressions (S = C) and (S'EVENT) are considered as inseparable expressions [33]. In other words, the whole expression is

converted to (S = C′) where C′ ∈ {'R', 'F'}. Let V denote the virtual signal for the whole expression. Then, BSA faults are defined only for S and V, not for the virtual signals corresponding to (S = C) and (S'EVENT). As an example, the following VHDL code is considered:

```
if (CLK = '1' and CLK'EVENT) then
   Q <= D;
end if;
```

The condition expression of the *if* statement is converted to (CLK = 'R'). Let V denote the virtual signal for (CLK = 'R'). Both a stuck-at-1 fault and a stuck-at-0 fault are defined for V. To activate a stuck-at-1 fault for V, logic 0 should be applied to V, which requires that one of the values in {'0', '1', 'F'} be applied to CLK. On the other hand, to activate a stuck-at-1 fault for CLK, one of the values in {'0', 'R', 'F'} should be applied to CLK. Hence, BSA faults for V are not equivalent to those for CLK in this case.

Effects of Behavioral Stuck-at Faults for Signals in Control Expressions

Definition: A **control expression** is an expression which controls the execution of a set of statements.

Hence, the condition expression of an *if* statement and the selection expression of a *case* statement are called control expressions.

Contrary to the previous behavioral fault model, BSA faults of the new behavioral fault model are also defined on signals in control expressions. A BSA fault on a signal or a virtual signal in a control expression perturbs the control intended by the control expression. As an example, let us consider the following VHDL code representing a 4-function ALU:

```
  process (A, B, FSEL)
  begin
s1:  case FSEL(1 to 2) is
s2:      when "00" => F <= A;
s3:      when "01" => F <= (not A);
s4:      when "10" => F <= A and B;
s5:      when "11" => F <= add (A,B);
     end case;
  end process;
```

In the example, if FSEL = 00 is applied in the presence of the FSEL(1)-stuck-at-1 fault, the statement s4 is selected when s2 is to be selected. Similarly, if FSEL = 01 is applied in the presence of FSEL(1)-stuck-at-1, the statement s5 is selected when s3 is to be selected. In other words, a different arithmetic or logic function is selected when a fault occurs. This shows how the instruction decoding faults and the data register decoding faults for micro-processors [20,21] are handled using the behavioral fault model.

4.5. Behavioral Stuck-open Faults

A behavioral stuck-open (BSO) fault is defined for an assignment operator (<=) in any signal assignment statement. The main difference between a BSO fault and a BSA fault for the target signal is that *the value of the target signal in the presence of a BSO fault is indeterminable if it is not initialized* while it is stuck at a constant logic value in the presence of a BSA fault. A BSO

fault in general represents an open circuit in a model. A BSO fault also represents additional functional faults when the assignment statement where the BSO fault resides is controlled by control expressions.

Case 1 - an assignment statement which is not controlled by any control expression

The right-hand side of the statement describes a combinational circuit ranging from a primitive logic gate to a complex functional block. A BSO fault in this case means that the ouput line of the combinational circuit is disconnected and that the value of the output is indeterminable (if this causes the line to be stuck at logic 1 or 0, it is modeled as a BSA fault).

Case 2 - an assignment statement controlled by a clock

This type of statement describes a synchronus behavior of a sequential circuit. A BSO fault in this case means that a data transfer is not correctly performed when a clock signal is applied.

Case 3 - an assignment statement controlled by a non-clock signal

This describes an asynchronous behavior of a sequential circuit or a combinational circuit controlled by a control signal. In the former case, a faulty behavior means that the corresponding sequential circuit is not correctly set or reset with an asynchronous *set* or *reset* input. In the latter case, a faulty behavior means that one of the sources in a multiplexer or a decoder is not correctly transferred to the target signal. The following example illustrates the latter case:

```
    if (S = '1') then
s1:   X <= A;
s2:   Y <= C;
    else
```

```
s3:   X <= B;
s4:   Y <= D;
    end if;
```

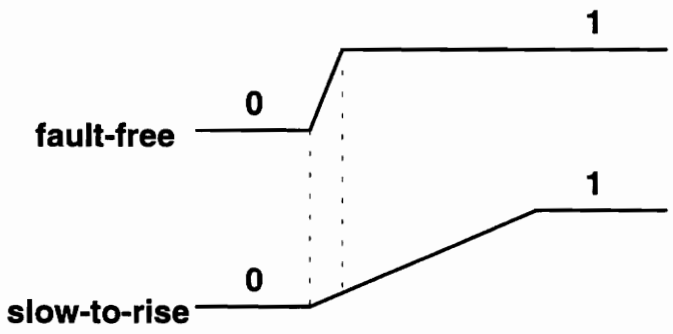
The BSO fault in the statement s1 could represent a faulty behavior that the source A is not correctly selected when S = 1.

Behavioral Stuck-open Faults vs. Gate Level Transition Faults

As mentioned before, about half of physical defects cannot be modeled with gate level stuck-at faults. There have been approaches [40,54] to using transition faults (or delay faults) at the gate level as a means of detecting physical faults which are not modeled with gate level stuck-at faults. A transition fault is defined as follows:

Definition: A **transition fault** is a defect that delays either a rising or falling transition. Two transition faults are defined for a signal line: slow-to-rise and slow-to-fall. A **slow--to-rise** (**slow-to-fall**) fault is a defect that delays a rising (falling) transition.

The following timing disgram shows a slow-to-rise fault.



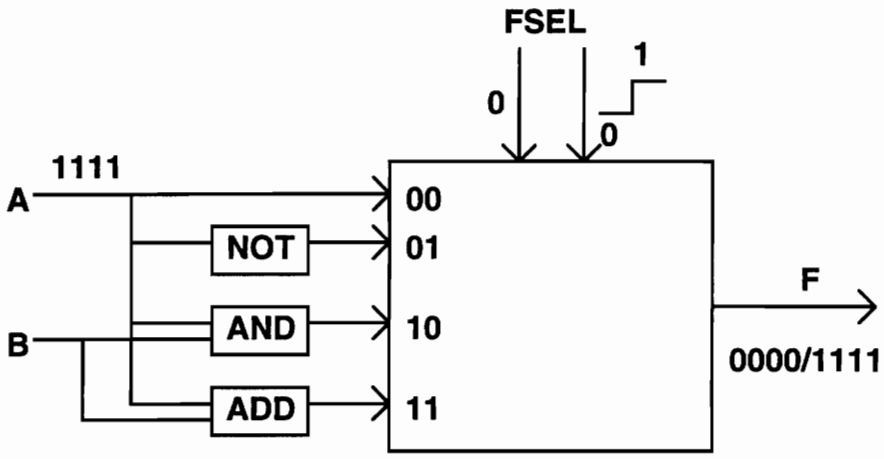
A test for a transition fault consists of two patterns: an initialization pattern and a transition propagation pattern [52]. An initialization pattern places the inital transition value at the point of the fault. The fault-free value and the faulty value will be the same after the initialization pattern. A transition propagation pattern places the final transition value at the point of the fault. After the transition propagation pattern is applied, the fault-free value is different from the faulty value within the period of the test.

The activation of a BSO fault also needs two steps because the inital value of the target signal is not determinable. The only difference between a BSO fault and a transition fault is that the fault-free value is permanently different from the faulty value for a BSO fault while both values are different for only a fixed amount of time for a transition fault. Hence, a test for a BSO fault can detect some gate level transition faults depending on time history.

Let us consider the VHDL code for a 4-function ALU shown in Section 4.4. Signals A, B, and F are assumed to be 4-bit vectors. A test which can detect the BSO fault for the statement s3 is shown below. The test consists of two patterns. The first pattern initializes the target F to 1111 using the statement s2. The second pattern attempts to transfer 0000 to F using the statement s3. As a result, 0000 is transferred to F for the fault-free case and 1111 for the faulty case.

A	B	FSEL	F
1111	x	00	x
1111	x	01	0000/1111

The above VHDL code implies a hardware structure shown below. As shown in the diagram, the above test also detects the transition fault on the signal line FSEL(2).



4.6. Micro-operation Faults

The new behavioral fault model defines a micro-operation (MOP) fault only for arithmetic and relational operators because the MOP fault for a logic operator can be detected by a test for a BSA fault on an input of the operator. Contrary to logic operators, an arithmetic operator or an relational operator represents a large block of logic.

An arithmetic operator or a relational operator is perturbed to another operator considering its operation. For example, the arithmetic operation ADD can be perturbed to XOR or SUB, considering the defects in the carry propagation path. This perturbation also represents the case when SUB is selected instead of ADD due to an ALU decoding fault in a micro-processor model. The following table shows the perturbation of arithmetic operators and relational operators. Since INC and DEC can be described using ADD and SUB functions, respectively, they are not included in the table.

Fault-free Operator	Faulty Operator
ADD	SUB, XOR
SUB	ADD, XOR
BVEQ	BVNEQ
BVNEQ	BVEQ
BVLT	BVGE
BVLE	BVGT

It is obvious that a single test for an MOP fault cannot detect majority of physical defects inside the functional block corresponding to the MOP. For example, a single test is not sufficient to test a 4-bit adder. A systematic method has been developed which generates more test vectors as the bit length of signals involved in an MOP increases. This will be discussed in Chapter 5.

If tests are available for an MOP representing a functional block of logic, they can be directly used without defining any MOP fault. In this case, each test should consist of an input test vector and a value pair (a value for a fault-free micro-operation and a value for a faulty micro-operation) at an output of a micro-operation.

4.7. Generation of A Behavioral Fault Set

Behavioral faults are generated by scanning the VHDL constructs in a behavioral model. A complete set of behavioral faults in a behavioral model is defined as follows:

Definition: A **complete set of behavioral faults** in a behavioral model is the set of BSA faults, BSO faults, and MOP faults under an assumption of a single behavioral fault.

Definition: A **fault identification (FID)** is a form

(FN, FT, POS, [DES]),

where FN = fault number

FT = fault type (*bso*, *mop*, and *bsa*)

POS = the position of the statement or the expression where the fault resides

[DES] = an optional description which details the fault.

Each fault type has a different form of FID:

1. BSO fault - (*fn*, *bso*, *sn*), where *sn* is a statement number.
2. MOP fault - (*fn*, *mop*, (*sn*, *rel_pos)*, (*op*, *fop)*), where *rel_pos* is a relative position of the expression where *op* exists, *op* a fault-free operator, and *fop* a faulty operator.
3. BSA fault - (*fn*, bsa, (*sn*, *rel_pos)*, (*st*, *val)*), where *rel_pos* is a relative position of a signal line for which the *bsa* is defined, *st* is a type of the signal line (nor: normal; vs: virtual signal; fs: fanout stem; fb: fanout branch), and *val* is logic 1 or 0.

Generation of BSO Faults

An FID is generated whenever an assignment statement is encountered. For example, (2, bso, s3) is the FID for a BSO fault for the assignment statement s3. FN = 2 indicates that it is the second behavioral fault generated from a given behavioral model.

Generation of MOP Faults

An FID for MOP faults is generated whenever an arithmetic or a relational operator with multi-bit arguments is encountered. The identification of the position of an MOP can be accomplished utilizing the rule that every expression is enclosed with parentheses in a behavioral model (section 4.3). Since every sub-expression in an expression is enclosed with parentheses, an expression can be mapped into a tree where each node (except a leaf node) represents a sub-expression. Each leaf node represents a signal name. Then, *rel_pos* of an MOP is defined as (*level*, *loc*), where *level* denotes a level in the tree and *loc* is the order of the operator at the level. As an example, let us consider the following VHDL code which is mapped into an expression tree shown in Figure 6. In the tree, rel_pos = (3,2) indicates the position of the second ADD operator at Level 3.

CO <= (A3 ADD B3) ADD ((A2 ADD B2) ADD (A1 ADD B1));

Substituting rel_pos = (*level*, *loc*), an FID for an MOP fault is defined as follows:

(*fn*, *mop*, (*sn*, (*level*, *loc*)), (*op*, f*op*)).

Generation of BSA Faults

An FID for BSA faults is generated whenever a signal, a virtual signal, a fanout stem, or a fanout branch is encountered in a behavioral model.A complete set of BSA faults is defined below.

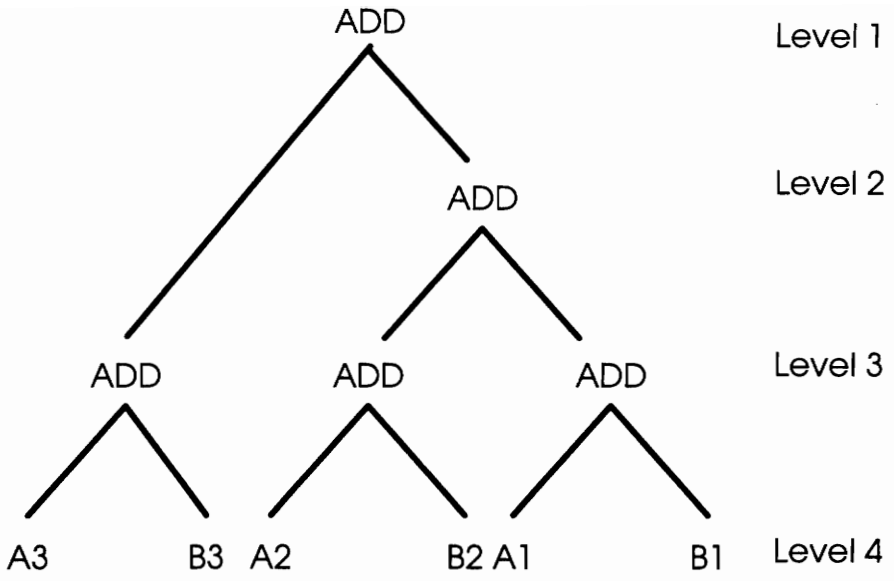


Figure 6. An Expression Tree

Definition: A **complete set of BSA faults** is the set of single BSA faults for all bits of signals, virtual signals, fanout stems, and fanout branches.

Signals, virtual signals, fanout stems, and fanout branches can be recognized using expression trees. The relative position (*rel_pos*) of each signal bit is defined with (*level*, *loc*, *bit*), where *bit* denotes the bit position of a signal. Substituting rel_pos = (level, loc, bit), an FID for a BSA fault is defined as follows:

(*fn*, *bsa*, (*sn*, (*level*, *loc*, *bit*)), (*st*, *val*)).

Each non-leaf node in an expression tree is identified as a virtual signal. Each leaf node in an expression tree represents a signal. If another leaf node in the same tree has the same signal name

or a leaf node in another expression tree has the same signal name, a fanout point is identified. Let S denote the signal which forms a fanout point and n the number of signals with the same name. Then, for each leaf node with the same signal name, a fanout branch S_i ($1 < i < n$) is defined. The fanout stem for S is defined for the first leaf node with the same name for convenience.

For a VHDL port of mode *out*, no fanout point is formed, and the position of the fault for the port is defined as (sn, (0, 0, bit)), where sn denotes the number of the first assignment statement whose target has the same name. For a port of mode *inout*, a fanout point is formed. In this case, a fanout branch is defined for the first target (of an assignment statement) with the same name.

Let BSA_T denote a set of BSA faults for a behavioral model. BSA_T can be divided into three sets BSA_{in}, BSA_{stem}, and BSA_{po} as follows:

1. BSA_{in} - a set of BSA faults for inputs of VHDL statements (including fanout branches for non- primary output signals).
2. BSA_{stem} - a set of BSA faults for fanout stems (including fanout stems formed by primary outputs).
3. BSA_{po} - a set of BSA faults for all primary outputs (including fanout branches formed by primary outputs).

An algorithm for constructing a set of BSA faults is shown below (function Make_BSA). It includes a function Set_Fanouts which finds a set of signals which form fanout points. The three sets BSA_{in}, BSA_{stem}, and BSA_{po} are created in the algorithm, in that order.

```
Make_BSA ()
{
  PO <- a set of primary outputs;
  ET <- a set of expression trees;
  FO <- Set_Fanouts (); /* a set of fanout points */
  /* Add stuck-at faults for inputs of VHDL statements --- BSA_in */
  while (there exists an unprocessed expression tree in ET) {
    select an expression tree;
    assign BSA faults for each bit of every non-leaf node;
    mark the faults with vs (virtual signal);
    assign BSA fault for each bit of every leaf node;
    while (there exists an unprocessed leaf node in the tree) {
    select a leaf node;
      if (the signal name for the node is in FO)
        mark the faults with fb (fanout branch);
      else
        mark the faults with nor (normal signal);
    }
  }
  /* Add stuck-at faults for fanout stems --- BSA_stem */
  assign BSA faults for each bit of every signal in FO;
  mark the faults with fs (fanout stem);
  /* Add BSA faults for PO's --- BSA_po */
  while (there exists an unprocessed primary output in PO) {
    select a primary output;
    assign BSA faults for each bit of every primary output in PO;
    if (the primary output is included in FO)
      mark the faults with fb (fanout branch);
    else
      mark the faults with nor (normal);
  }
}


  Set_Fanouts ()
  {
    PO  <-- a set of primary outputs;
    ET <- a set of expression trees;
    FO <- an empty set;
    /* Fanout points for inputs of VHDL statements */
    while (there exists an unprocessed node) {
      select a node;
      if (there exists a node with the same signal name in ET)
         add the name to FO;
    }
    /* Fanout points for primary outputs */
```

```
    while (there exists an unprocessed primary output in PO) {
      select a primary output;
      if (there exists a node with the same signal name in ET)
         add the name to FO;
    }
    return FO;
  }
```

Let us again consider the VHDL code for the 4-function ALU shown in Section 4.4. In the VHDL code, A, B, and FSEL are assumed to be 2-bit input ports and F a 2-bit output port. The function Set_Fanouts returns a set FO = {A, B}. The main function Make_BSA generates the set of behavioral faults shown in Figure 7. Faults 1-4 and 5-6 are BSO faults and MOP faults, respectively. Faults 7-10 are BSA faults for the signal FSEL. Faults 11-14, 15-18, 19-22, and 27-30 are for fanout branches of A, and faults 35-38 are for the fanout stem of A (defined on the first occurrence of A). Faults 23-26 and 31-34 are for fanout branches of B, and faults 39-42 are for the fanout stem of B (defined on the first occurrence of B). Faults 43-46 are BSA faults for the primary output F. It should be noted that s2 is the first statement whose target is F.

4.8. Summary

Three behavioral faults - behavioral stuck-at (BSA) faults, behavioral stuck-open (BSO) faults, and micro-operation faults (MOP) - have been defined by using the fault equivalence among the previous behavioral faults and defining bit-wise stuck-at faults on control signals as well as data signals. For BSA faults which are defined for signals, virtual signals, fanout stems, and fanout branches, the methods of recognizing virtual signals and fanout points have been presented. The relationship between BSO faults and gate level transition faults have been discussed. Issues on MOP faults have been addressed. Finally, an algorithm of enumerating behavioral faults from a VHDL model has been presented with an example.

(1, bso, s2)
(2, bso, s3)
(3, bso, s4)
(4, bso, s5)
(5, mop, (s5,(1,1)), (add, sub))
(6, mop, (s5,(1,1)), (add, xor))
(7, bsa, (s1, (1,1,1)), (nor, '0'))
(8, bsa, (s1, (1,1,1)), (nor, '1'))
(9, bsa, (s1, (1,1,2)), (nor, '0'))
(10, bsa, (s1, (1,1,2)), (nor, '1'))
(11, bsa, (s2, (1,1,1)), (fb, '0'))
(12, bsa, (s2, (1,1,1)), (fb, '1'))
(13, bsa, (s2, (1,1,2)), (fb, '0'))
(14, bsa, (s2, (1,1,2)), (fb, '1'))
(15, bsa, (s3, (2,1,1)), (fb, '0'))
(16, bsa, (s3, (2,1,1)), (fb, '1'))
(17, bsa, (s3, (2,1,2)), (fb, '0'))
(18, bsa, (s3, (2,1,2)), (fb, '1'))
(19, bsa, (s4, (2,1,1)), (fb, '0'))
(20, bsa, (s4, (2,1,1)), (fb, '1'))
(21, bsa, (s4, (2,1,2)), (fb, '0'))
(22, bsa, (s4, (2,1,2)), (fb, '1'))
(23, bsa, (s4, (2,2,1)), (fb, '0'))
(24, bsa, (s4, (2,2,1)), (fb, '1'))
(25, bsa, (s4, (2,2,2)), (fb, '0'))
(26, bsa, (s4, (2,2,2)), (fb, '1'))
(27, bsa, (s5, (2,1,1)), (fb, '0'))
(28, bsa, (s5, (2,1,1)), (fb, '1'))
(29, bsa, (s5, (2,1,2)), (fb, '0'))
(30, bsa, (s5, (2,1,2)), (fb, '1'))
(31, bsa, (s5, (2,2,1)), (fb, '0'))
(32, bsa, (s5, (2,2,1)), (fb, '1'))
(33, bsa, (s5, (2,2,2)), (fb, '0'))
(34, bsa, (s5, (2,2,2)), (fb, '1'))
(35, bsa, (s2, (1,1,1)), (fs, '0'))
(36, bsa, (s2, (1,1,1)), (fs, '1'))
(37, bsa, (s2, (1,1,2)), (fs, '0'))
(38, bsa, (s2, (1,1,2)), (fs, '1'))
(39, bsa, (s4, (2,2,1)), (fs, '0'))
(40, bsa, (s4, (2,2,1)), (fs, '1'))
(41, bsa, (s4, (2,2,2)), (fs, '0'))
(42, bsa, (s4, (2,2,2)), (fs, '1'))
(43, bsa, (s2, (0,0,1)), (nor, '0'))
(44, bsa, (s2, (0,0,1)), (nor, '1'))
(45, bsa, (s2, (0,0,2)), (nor, '0'))
(46, bsa, (s2, (0,0,2)), (nor, '1'))

Figure 7. A Complete Set of Behavioral Faults For a VHDL Code

Chapter 5. Behavioral Test Generation Algorithm

In this chapter, a test generation algorithm is discussed which generates test vectors from the behavioral VHDL model (defined in Chapter 3) assuming the behavioral fault model (defined in Chapter 4).

Section 5.1 compares the behavioral test generation to gate level test generation, reviews the previous approaches, and outlines the new approach. Section 5.2 presents rules for test generation operations *activation*, *propagation*, and *justification*. Section 5.3 discusses the assignment of time periods during test generation. Sections 5.4, 5.5, and 5.6 discuss how to handle bus resolution functions, reconvergent fanout, and feedback loops during test generation, respectively. Section 5.7 discusses *two-phase testing*, a method of generating tests when a signal on a fault propagation path cannot be initialized using another path. Section 5.8 shows the overall test generation algorithm with test generation examples. Section 5.9 summarizes this chapter.

5.1. Test Generation Approach

5.1.1 Gate Level Test Generation vs. Behavioral Test Generation

The D-algorithm [1], PODEM [2], and FAN [3] are widely-used test generation algorithms which use gate level circuit descriptions assuming a single stuck-at fault model. Stuck-at faults are defined for all signal lines (including fanout stems and fanout branches) in a circuit description.

For a given stuck-at fault, each gate level test generation algorithm creates a D or $\overline{D}$ at the fault site and tries to propagate it to a primary output. Propagation and justification through a primitive gate can be accomplished using simple rules. However, the overall test generation procedure for VLSI circuits containing hundreds of thousands of gates would be prohibitively expensive because the complexity of a test generation procedure exponentially increases with the number of gates [55].

Behavioral test generation approaches use behavioral circuit descriptions assuming a behavioral fault model. The number of elements in a behavioral circuit description is usually much smaller than that in a gate level circuit description. For example, a single micro-operation ADD(A,B) in a behavioral circuit description may be represented with tens to hundreds of primitive gates in a gate level circuit description. It is obvious that the propagation and justification of a value through a single micro-operation is much faster than that through a number of gates. This is the main advantage of behavioral test generation over gate level test generation. However, behavioral test generation algorithms become complex because different propagation and justification rules must be defined for different VHDL constructs.

While only D or $\overline{D}$ is created at the fault site in gate level test generation, various value pairs can be created at the fault site in behavioral test generation. This is because the whole vector instead of a single bit of a signal is used for the activation of behavioral faults. For example, one of logic patterns that can activate an MOP fault (ADD -> SUB) for a 3-bit micro-operation ADD(A,B) is A = XX1 and B = XX1. If A = 001 and B = 001 are assigned, the value of the micro-operation is 010 for the fault-free case (ADD), and 000 for the faulty case (SUB). The following table shows possible value pairs that can be generated at the fault site using the logic pattern (A = XX1 and B = XX1). Each of these value pairs is defined as a good/bad value pair as shown below.

A	B	ADD	SUB
001	001	010	000
001	011	100	110
001	101	110	100
001	111	000	010
011	001	100	010
011	011	110	000
011	101	000	110
011	111	010	100
101	001	110	100
101	011	000	010
101	101	010	000
101	111	100	110
111	001	000	110
111	011	010	100
111	101	100	010
111	111	110	000

Definition: A **good value** (denoted as G) is a logic value of an expression for a fault-free circuit model. A **bad value** (denoted as B) is a logic value of an expression for a faulty model. A **good/bad value pair** is a value pair G/B where the expression on the fault site has G for a fault-free model and B for a faulty model.

5.1.2. Previous Approaches

The automatic test generation algorithm for behavioral VHDL models was initially developed by Barclay and Armstrong [27]. The algorithm basically applies the D-algorithm to behavioral VHDL models using the value pair G/B instead of D or $\overline{D}$. As in the D-algorithm, a test for a given fault is generated by creating a G/B at the fault site, propagating the G/B to a primary output, and justifying all values necessary for the creation and the propagation of the G/B to primary inputs.

The algorithm uses a set of logic values similar to the 9-valued logic used in the 9-V algorithm [56]. When a G/B is assigned to a single bit signal, it is either a D or $\overline{D}$. When a G/B is assigned to a vector signal, it could be one of many different values because the only requirement is G $\neq$B. The value pair G/X is used when only a good value G is known. Similarly, X/B is used when only a bad value B is known. The following table compares the value systems for the D-algorithm, the 9-V algorithm, and the behavioral test generation algorithm (denoted as the B-algorithm):

D-algorithm	9-value	B-algorithm
0	0/0	0
1	1/1	1
D	1/0	G/B
$\overline{D}$	0/1	
X	X/X	X/X
	0/X	G/X
	1/X	
	X/0	X/B
	X/1	

Barclay's algorithm was implemented using the artificial intelligence techniques of AND/OR goal trees and rule databases. An AND/OR goal tree is a data structure used to break a large problem into small pieces until each piece is small enough to be solved directly. The algorithm was written in Prolog which has built-in language features [35] suitable for AND/OR goal trees and rule databases. The algorithm uses eleven types of goals which represent the requirements during the test generation procedure:

VIO - Need **value** in **object** at **time**.
VIE - Need **expression** equal to **value** at **time**.
EXEC - Execute **statement** at **time**.
DNE - Don't execute **statement** at **time**.

EXG - Execute **statement** at **time** given that **set of statements** execute.
OBSOBJ - Observe value in **object** at **time** for **good value** or **bad value**.
OBSEXPR - Observe **(sub)expression** at **time** for **good value** or **bad value**.
OBSEXEC - Observe execution of clauses of **statement** at **time**, expecting **good** or **bad** clauses to execute.
DND2 - Preserve **value** in **object** loaded at **time1** until **time2**.
DND1 - Preserve **value** in **object** at period **time**.
TR - Specify time relation **time1** $\leq$ **time2** or **time1** = **time2**.

VIO and VIE goals are used to assign values to signals (including virtual signals). EXEC and EXG are used to execute VHDL statements. OBSOBJ and OBSEXPR are used to determine where the original fault effect (G/B) is currently propagated. OBSEXEC is used to detect a fault-induced difference of execution of statements. TR represents a constraint on when to assign values to signals (including virtual signals) or propagate values through VHDL constructs. DND2, DND1, and DNE ensure consistency by requiring that assigned values should not be overwritten.

A set of initial goals are specified for each behavioral fault before the goal-solving begins. The goal-solving procedure begins by examining the set of initial goals. An initial goal is selected and then solved by determining that it is primitive (e.g., an assignment of a value to a primary input) or by breaking it into an appropriate set of subgoals. These subgoals are added to the *unsolved goal list* which contains all goals that must be solved. The unsolved goal list is sorted based on heuristics whenever a new goal is added to the list or a goal is removed from the list. The main purpose of the heuristics is to detect conflicts quickly by solving goals which lead to conflicts first. Hence, goals which involve no choices are solved first, and *related* goals (goals in the same time period) are considered next. This was implemented by assigning a weight to each goal and then sorting by this weight. The weight of a goal is computed as an estimate of the number of ways to solve it. Hence, goals are first grouped by the time period, and then sorted by their weights. TR

goals involve two time periods, and provide constraints on the actual ordering of time periods. TR goals are solved as the final step in the goal-solving procedure.

Barclay's approach is basically a best-first search method which attempts to minimize the number of conflicts between goal solutions. However, it has two drawbacks:

1. It represents the goal tree explicitly, storing and manipulating the set of goals to be solved. Whenever a goal is solved, the *unsolved goal list* is resorted and checked for new conflicts among goals. A significant amount of overhead is incurred in handling the unsolved goal list as the length of the list grows. The best-first search method also suffers from Prolog's blind backtracking mechanism: when a goal fails, the last solved goal is undone first without attempting to determine which previously solved goal is related to the conflict. Prolog's backtracking mechanism works best when related goals are solved in sequence. Since the unsolved goal list is resorted whenever a goal is solved, there are cases when related goals are not solved in sequence. This causes the backtracking to travel through a number of unrelated goals before reaching one that is related to the conflict, which significantly increases the test generation time.
2. It assigns relative time periods during the goal-solving procedure. The actual ordering of time periods are determined in the last step of the goal-solving procedure by solving TR goals. This strategy is used to avoid as many conflicts as possible during the goal-solving procedure. However, as the list of test events and the corresponding TR goals grows longer, the procedure of finding a suitable time ordering requires more and more backtracking, which significantly increases the test generation time.

O'Neill [30] improved Barclay's approach by using higher level goal types, a depth-first search method, and a different time-handling. The three high level goals are characterized as follows:

Justification: Given an expression and a desired value, *justification* determines the set of input values which place the value in the expression. A justification goal can require solving justification and execution subgoals.

Propagation: Given a good/bad value pair (G/B) and the initial location in the VHDL model, *propagation* selects a path to the output, and specifies all values necessary to move the G/B along this path. A propagation goal can require solving propagation, justification, and execution subgoals.

Execution: Given a statement, *execution* determines the sequence of control statements which govern the execution of the statement and determines the values required for the expressions in the control statements. An execution goal can require solving execution and justification subgoals.

O'Neill's approach has the following advantages over Barclay's approach:

1. While Barclay uses eleven goal types all of which are explicitly represented in a goal tree, O'Neill uses only three explicit goal types (justification, propagation, and execution). A set of rules define how each goal is to be solved under the various circumstances that arise during the test generation procedure. Since the other goals of Barclay's approach are represented implicitly in the flow of program execution, the overhead incurred in handling the unsolved goal list is reduced.

2. O'Neill uses a depth-first search method which enables related goals to be solved in sequence. Hence, when a backtracking occurs, the source of the conflict is detected without undoing many unnecessary goals.
3. O'Neill handles time in absolute terms during the goal-solving procedure while Barclay does it in relative terms. An event queue is maintained, and test events are inserted into the time queue as soon as they arise. Each event in the queue is assigned a time tag. Thus, when the goal-solving procedure is complete, the events in the queue have a fixed time ordering, and no further processing is required.

Both Barclay and O'Neill used a timing model where primary inputs are set to values at the beginning of each time period, primary outputs are sampled at the end of each period, and clocks are represented using two adjacent time periods. Using this timing model, a rising (falling) edge of a clock is represented in two time periods: one with a 0 (1) immediately followed by one with a 1 (0). The first part of Figure 8 shows the timing diagram for a D flip-flop (input: D, output: Q, and clock: CLK) represented by this timing model. Jani [32] adopted values R (rising edge) and F (falling edge), and used a timing model where R or F is represented in a single time period. Since there is no way of splitting a clock signal during the test generation procedure, this timing model reduces the overhead associated with clock signals represented in two time periods. It is still assumed that all data values are available before an edge of a clock signal. The second part of Figure 8 shows the timing diagram for a D flip-flop represented by this timing model.

Barclay and O'Neill handled only VHDL models with a single process statement. Jani made it possible to generate tests for VHDL models with multiple process statements. This is a big improvement because non-trivial VHDL models usually include multiple process statements. However, Jani's approach as well as Barclay's and O'Neill's cannot handle reconvergent fanout

because the test generation algorithm uses a single path propagation scheme for propagating faults. Lam [33] solved this problem by using a special propagation scheme where a single path propagation scheme is used until a conflict is detected, and a multiple path propagation is used when a conflict is detected.

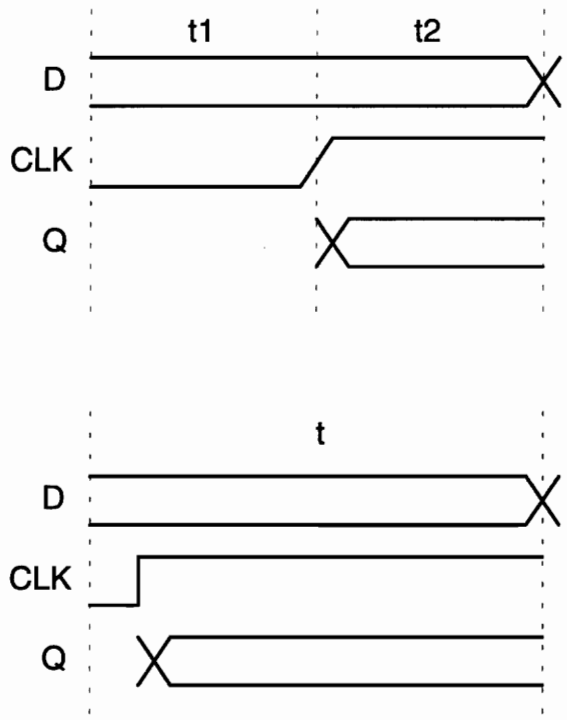


Figure 8. Timing diagrams for two timing models

Lam further improved the test generation algorithm by adding the capability of initializing internal signals to certain logic values. This is useful for generating tests for circuit models that include imbedded sequential circuits. Users are supposed to provide a logic value and the number of the assignment statement whose target signal needs to be initialized. The test generation algorithm selects a path (a set of VHDL statements) which can initialize the target signal, and initializes the

signal to the specified logic value. This procedure is called a *circuit initialization*, and is performed before the test generation procedure begins. This is a big advantage of behavioral test generation over gate level test generation.

Baweja [34] attached a heuristic test generator to the test generation program. The heuristic test generator includes a set of heuristics which is useful for generating test vectors for big micro-operation faults discussed in Chapter 4. Whenever the test generation algorithm encounters a big micro-operation fault, it calls the heuristic test generator, which generates a set of test vectors for the given fault using the set of heuristics. The number of test vectors increases as the number of bits of the big micro-operation increases. This strategy was shown to increase the equivalent gate-level fault coverages [34,57,58].

Figure 9 shows the overall system of the behavioral test generator (BTG) that has been developed since Barclay first started. The function of each module in the BTG is explained as follows:

Preprocessor: It consists of three parts, the translator, the intermediate form extractor (IFE), and the fault list extractor (FLE). The translator converts a VHDL model to Prolog predicates. It was originally implemented in C for an early version of the algorithm, but is currently performed manually. The IFE converts Prolog predicates to an intermediate form (Prolog facts) that represents the knowledge of the circuit, i.e., controllability and observability information. The FLE generates a list of behavioral faults. The IFE and the FLE are written in Prolog.

Basic Test Generator: It contains the rules for the three high level goals - justification, propagation, and execution. It generates tests for each fault in the fault list using the Prolog

predicates, the intermediate form, the fault list, and the test generation rules. It interacts with users when some internal signals need to be initialized. The basic test generator is written in Prolog.

Heuristic Test Generator: It contains the test generation heuristics for big micro-operation faults. It obtains the information about a big micro-operation fault from the basic test generator, applies appropriate heuristics, and returns a set of test vectors local to the big micro-operation to the basic test generator. The heuristic test generator is written in Prolog.

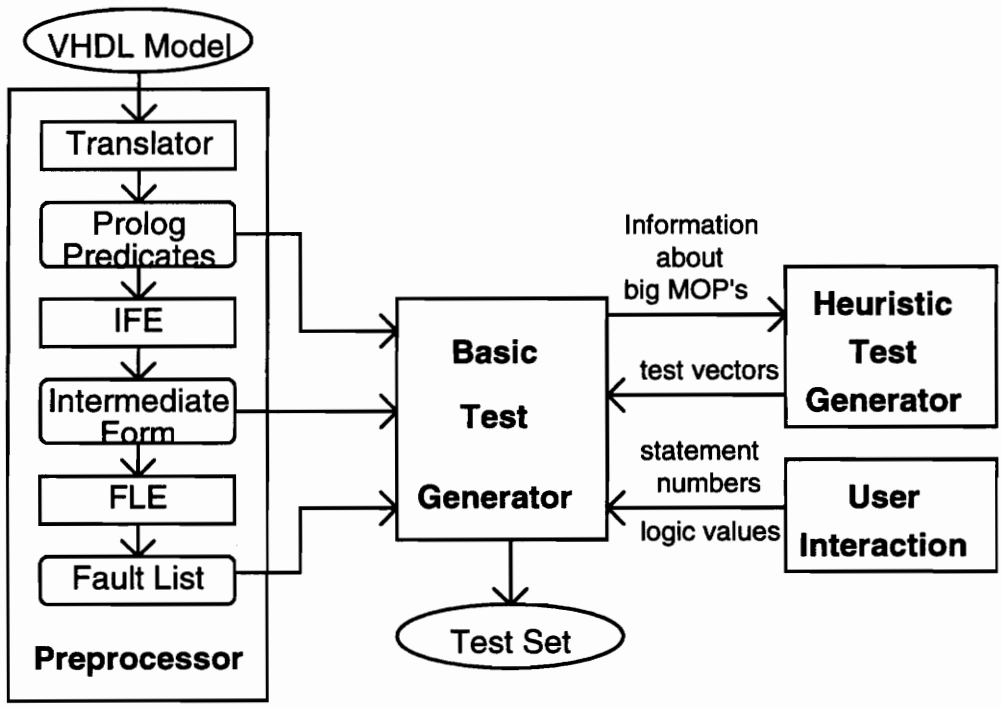


Figure 9. The Behavioral Test Generator

5.1.3. A New Approach - Formalization of the Algorithm

Problems in The Previous Approaches

The behavioral test generation algorithm developed by Barclay *et al.* [27,28,30-34] successfully generates tests for behavioral VHDL models for some MSI/LSI circuits. However, the algorithm needs to be improved in some respects to be used for larger circuit models.

First, the algorithm uses Prolog-oriented AND/OR goal trees for representing and solving the problems. The algorithm takes advantage of the built-in language features (such as backtracking mechanisms) of Prolog, but also suffers from its inflexible control constructs. The language C or C++ provides flexible control constructs and various bit manipulation constructs which are essential for implementing logic operations. Programs written in these languages are known to run faster than those written in other high level languages. In order to implement the algorithm using C or C++, it is necessary to develop a more general and formal algorithm.

Second, the test generation rules defined so far are ad-hoc and thus it is not clear whether they cover every possible case that can occur during the test generation procedure or not. In addition, the interaction between the overall test generation procedure and the detailed test generation rules has not been clarified, which makes it difficult to determine the completeness of the algorithm.

Third, the algorithm is incomplete or generates inefficient tests for some special cases:

1. Lam used a special propagation scheme for handling reconvergent fanouts, but there are special cases when it does not work properly. Further improvement of the propagation scheme will be discussed in Section 5.5.

2. The algorithm generates tests using the simulation semantics of VHDL. In an event-driven simulation environment, an event (a change on a signal value) must occur for a statement to be executed. If this concept is directly used for the test generation algorithm, unnecessarily long tests could be generated. Lam partially solved this problem by ignoring the presence of process statements when generating tests. This problem will be further investigated in Section 5.3.
3. The algorithm cannot handle bus resolution functions (BRF's) which can occur in VHDL models with multiple process statements. Since a typical BRF consists of loop statements and variable assignment statements which are not included in the VHDL subset, special test generation rules for BRF's need to be developed. This will be discussed in Section 5.4.

Outline of The New Approach

The main task of the behavioral test generation algorithm (it will be denoted as the B-algorithm from now on) is to generate a test sequence for a given fault. A test sequence consists of one or more test vectors. In contrast to the previous approaches, the new approach doesn't use the concepts of goals or goal trees for representing or solving problems. The B-algorithm consists of two sub-tasks, *circuit initialization* and *test generation*. Circuit initialization is used for initializing internal signals in sequential circuit models, and is optional, as the user can choose to initialize them or not to. Circuit initialization justifies the values necessary for the initialization to primary inputs (PI's). This procedure is accomplished using justification operations defined below. Test generation activates the given fault by creating a value pair G/B at the fault site, propagates the G/B to a primary output (PO), and justifies the values necessary for the activation and the propagation to PI's. This procedure is accomplished using three test generation operations - *activation*, *propagation*, and *justification* - which are defined below. Figure 10 shows the relationship among the main tasks, the sub-tasks, and the test generation operations.

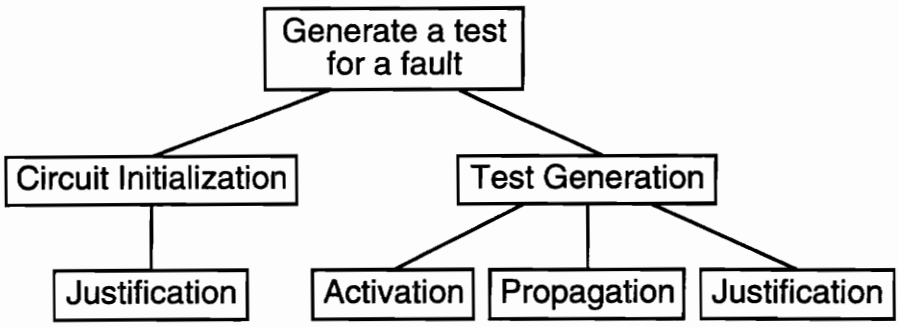


Figure 10. The Test Generation Procedure

Definition: **Activation** is an assignment of values to signals or virtual signals so as to create a value pair G/B ($G \neq B$) at the fault site.

Activation is performed only once for each fault during the execution of the B-algorithm. The assignment of values resulting from the execution of an activation is represented as one or more justification operations. The position of the fault site depends on the type of the given fault. The fault site is the specified signal for a BSA fault, the target signal of the specified assignment statement for a BSO fault, the virtual signal corresponding to the specified MOP for an MOP fault. How to activate a fault depends on its type, and thus different activation rules must be defined for different fault types.

Definition: **Propagation** is an assignment of values to inputs of a VHDL construct (an expression or a statement) so as to propagate a value pair G/B ($G \neq B$) at a location L through the VHDL construct to another location L′. The value pair G/B at L may be propagated as another value pair G′/B′ ($G' \neq B'$) to L′.

A good/bad value pair is initially located at the fault site. If the current position of the good/bad pair is a PO, no propagation is executed. Otherwise, a propagation is executed using a propagation rule. The assignment of values resulting from the execution of a propagation is represented as justification operations. How to and where to propagate a good/bad value pair depends on its current position in a VHDL model, and thus different propagation rules must be defined for different positions.

Definition: **Justification** is an assignment of values to inputs of an expression or a statement so that the expression or a target signal within the statement can obtain a desired value.

Justification is performed when a value is required for a signal or an expression during the test generation procedure. If a value is required at a PI, no justification is executed. Otherwise, a justification is executed using a justification rule. How to justify a value at a signal or an expression depends on the type of the VHDL construct whose output is the signal or the expression. Hence, different justification rules must be defined for different VHDL constructs.

In order to justify a value at the target of an assignment statement which is included in one or more control statements, it is necessary to assign values to the control expression(s) of the control statement(s) as well as to assign values to the right-hand side (RHS) expression of the assignment statement. O'Neill [30] defined the *execution* goal which specifies which statements to execute.

Since the task of an execution goal is essentially to assign proper values to control expressions, an execution goal can be represented as a set of justifications. Hence, the execution is not separately defined as an operation in the new approach. Instead, additional rules are defined to handle this type of justifications. These will be discussed in Section 5.2.

The B-algorithm calls propagations and justifications recursively until a G/B at the fault site is propagated to a PO and all values are justified to PI's. For systematic execution of this procedure, the B-frontier and the J-frontier are defined for propagations and justifications, respectively. The B-frontier is analogous to the D-frontier defined in the D-algorithm. The function of the J-frontier is basically the same as that of the J-frontier defined in the D-algorithm.

B-frontier: It contains a set of expressions or statements whose output value (the value of an expression or the value of the target signal of an assignment statement) is currently x, but have one or more G/B's on their inputs.

An entry is inserted into the B-frontier for the following cases:

1. If a G/B exists in any input of an expression and the value of the expression is x, the expression is inserted.
2. If a G/B is located at the whole right-hand side expression of an assignment statement and the value of the target signal is currently x, the assignment statement and the control statements (if any exists) which govern its execution are inserted.
3. If a G/B is located at the condition expression of an *if* statement and the value of the target signal of any assignment statement within the *if* statement is x, the *if* statement and the assignment statement are inserted.

4. If a G/B is located at the selection expression of a *case* statement and the value of the target signal of any assignment statement within the *case* statement is x, the *case* statement and the assignment statement are inserted.

Hence, a propagation operation consists of selecting one entry from the B-frontier and assigning values to the unspecified inputs of the entry so that the output can have a good/bad value pair. As a result of the execution of a propagation operation, one or more new entries may be inserted into the B-frontier. If the B-frontier is empty during the execution of the B-algorithm, test generation for the fault under consideration fails because the fault effect cannot be propagated to a PO.

J-Frontier: It contains a set of expressions or statements whose output value is known, but is not uniquely decided by its input values.

An entry is inserted into the J-frontier for the following cases:

1. If the value of an expression is to be justified, the expression is inserted.
2. If the value of a target signal is to be justified, all assignment statements with the target signal and the control statements (if any exists) which govern their execution are inserted.

Hence, a justification operation consists of selecting one entry from the J-frontier and assigning values to the unspecified inputs of the entry so that its output can have the desired value. As a result of the execution of an activation or a propagation, one or more new entries may be inserted into the J-frontier. If a good/bad value pair is set up at a primary output and the J-frontier is empty during the execution of the B-algorithm, the test generation procedure is successful.

The assignments of values resulting from the execution of a test generation operation sometimes uniquely determine other values. The process of computing these values and checking for their consistency with the previously determined ones is called *implication*. It is important to perform as many implications as possible after the execution of each test generation operation. This is because it helps to detect any conflict as soon as possible and hence to minimize the number of possible incorrect decisions.

Figure 11 shows the outline of the B-algorithm. The main function *B-alg* consists of functions *Initialize*, *Activate*, and *Prop_Just*. *Initialize* performs the circuit initialization by applying a justification rule to each entry in the J-frontier. This function is recursive because it continues to execute justifications until PI's are reached. *Activate* creates a G/B at the fault site by applying a proper activation rule to a given fault. It is noteworthy that functions *Two_phase_activate* and *Htg* (heuristic test generator) are called for activating a BSO fault and an MOP fault, respectively. These two functions will be explained in detail in Section 5.8. *Prop_Just* propagates the G/B at the fault site to a PO by applying a propagation rule to each entry in the B-frontier. If the B-frontier is empty during the execution of *Prop_Just*, it returns *Failure*. *Prop_Just* also justifies the values necessary for the activation and the propagation to PI's by applying a justification rule to each entry in the J-frontier. If a good/bad value pair is set up at a PO and the J-frontier is empty during the execution of the Prop_Just, it returns *Success*. *Prop_Just* is also recursive because it continues to execute propagations until a PO is reached and continues to execute justifications until PI's are reached.

Each of the functions *Initialize* and *Prop_Just* includes the function *Imply_and_check*, which performs *implication* and maintains the D-frontier and the J-frontier. *Implication* can be implemented using an *assignment queue* [41] which contains all values to be assigned to signals or

expressions. The assignment queue will be denoted as the A-queue. An entry in the A-queue has the form (e, v', t, d), where e is an expression (including a signal), v' is the value to be assigned to e, t is the time period when v' is assigned to e, and $d \in \{forward, backward\}$. *Imply_and_check* processes in turn every entry in the A-queue. The value v' to be assigned to e at time t is compared with the current value v of e at time t. A conflict is detected if $v \neq$ x and $v' \neq v$ (it should be noted that all values are initialized to x). *Imply_and_check* returns *Failure* when it detects a conflict. *Imply_and_check* will be explained in detail in Section 5.8.

```
B-alg(fault_id)
{
  set all values to x;
  Initialize;
  Activate(fault_id);
  if (Prop_Just() = Success)
    report Success;
  else
    report Failure;
}
```

```
Initialize()
{
  /* Implication, Detection of conflict, Maintenance of the J-frontier */
  if (Imply_and_check () = Failure)
    return Failure;
  /* Justification */
  if (J-frontier = ∅)
    return Success;
  select an entry from the J-frontier;
  while (there exists an unspecified input of the entry) {
    select an unspecified input of the entry;
    while (there exists an untried choice for the value of the input) {
      assign a value to the input using a justification rule;
      if (Prop_Just() = Success)
        return Success;
    }
  }
  return Failure;
}
```

```
Activate(fault_id)
{
  switch (fault_id) {
    case BSA : generate G/B;
    case BSO : Two_phase_activate(fault_id);
    case MOP : Htg(fault_id);
  }
}


Prop_Just()
{
  if (Imply_and_check () = Failure)
    return Failure;

  if (a good/bad value pair is not set up at a PO) {
    if (B-frontier = ∅)
      return Failure;
    while (there exists an untried entry in the B-frontier) {
      select an untried entry from the B-frontier;
      apply a propagation rule to the entry;
      }
      if (Prop_Just() = Success)
        return Success;
    }
    return Failure;
  }

  /* A good/bad value is set up at a PO - Justification starts */
  if (J-frontier = ∅)
    return Success;
  select an entry from the J-frontier;
  while (there exists an unspecified input of the entry) {
    select an unspecified input of the entry;
    while (there exists an untried choice for the value of the input) {
      assign a value to the input using a justification rule;
      if (Prop_Just() = Success)
        return Success;
    }
  }
  return Failure;
}
```

Figure 11. Outline of the B-algorithm

5.2. Test Generation Rules

5.2.1. Activation Rules

Different activation rules are defined for different fault types. BSA faults are activated in the same way that gate level stuck-at faults are activated. BSO faults are activated in two steps because the initial value of the target signal of each assignment statement is not known. MOP faults are activated using heuristics predefined in the B-algorithm. A set of input values which can create a good/bad value pair at the output of an MOP is generated using the heuristics.

BSA Fault

For each bit of a signal or a virtual signal in a VHDL model, two types of BSA faults, BSA-0 (stuck-at-0) and BSA-1 (stuck-at-1), are defined. A BSA-v fault ($v \in \{0, 1\}$) for a bit *b* of a signal is activated by assigning a value $\overline{v}$ to *b*. As a result, a good/bad value pair $\overline{v}/v$ is set up at *b*. If the signal is a virtual signal defined for a control expression, the value pair has the effect of executing different sets of assignment statements. As an example, the following VHDL code which represents a dual 2 x 1 multiplexer is considered:

```
s1:  if (SEL = '1') then
s2:      X <= E1;
s3:      Y <= F1;
     else
s4:      X <= E2;
s5:      Y <= F2;
     end if;
```

Let us consider a BSA-0 fault on the virtual signal V defined for the expression (SEL = '1'). The fault can be activated by assigning a logic 1 to V and thus creating 1/0 on V. For the fault-free

case (V = 1), the set of statements {s2, s3} under the *then* clause of the *if* statement is executed. For the faulty case (V = 0), the set of statements {s4, s5} under the *else* clause is executed.

BSO Fault

A BSO fault is defined for the assignment operator in each assignment statement in a VHDL model. Let S denote an assignment statement, e the right-hand side (RHS) expression of S, and t the target signal of S. When S is fault-free, the value of e is transferred to t. When S is faulty, the value of t remains as its initial value since the value of e is not transferred to t. The initial value of t is unknown because any signal in a VHDL model is initialized to x in the beginning of the B-algorithm. Hence, the value of t needs to be initialized using another statement (if any exists) which has t as its target signal. If no such statement exists, a special scheme is used for determining the initial value of t.

Case 1 - When there exists another assignment statement with the same target

Let S′ denote another assignment statement with the same target t and the RHS expression e′. The B-algorithm first chooses a good value G and a bad value B ($G \neq B$). How to select G and B will be discussed later in this section. Since different values cannot be transferred to a signal in the same time period, the activation of the BSO fault on S should be performed in two time periods, and thus is called the *two-phase activation*:

1. Load the value B into t using S′. Since S′ is fault-free (a single behavioral fault model is assumed), the value of t will be the same regardless of the presence of the BSO fault on S.

2. Load the value G into t using S. Since G is transferred into t for the fault-free case, the value of t is updated to G. Since G is not transferred into t for the faulty case, the value of t remains as B. Hence, a good/bad value pair G/B ($G \neq B$) is created at the fault site t.

If S is an assignment statement which is not governed by any control statement, the good value G is loaded into t by only assigning G to e. If S is governed by one or more control statements, G is loaded into t by assigning proper values to the control expressions of the control statements as well as assigning G to e. Similarly, if S′ is an assignment statement which is not governed by any control statement, the bad value B is loaded into t by assigning B to e′. If S′ is governed by one or more control statements, B is loaded into t by assigning proper values to the control expressions of the control statements as well as assigning B to e′. After the execution of the activation operation, the new assignments of values together with the expressions are added to the J-frontier.

As an example, let us consider the BSO fault on the statement s2 in the 2 x 1 multiplexer model shown above. Signals E1, E2, F1, F2, X, and Y in the model are assumed to be 2-bit vectors. First, two values G and B are selected. Let G = 00 and B = 11. Since s4 has the same target (X) as s2, s4 is used for loading B into X. B (11) is loaded into X by assigning SEL = 0 and E2 = 11. For the fault-free case, G (00) is loaded into X by assigning SEL = 1 and E1 = 00. For the faulty case, X holds its initial value 11. As a result, a good/bad value pair 00/11 is set up at the target X.

A good/bad value pair G/B is selected using the following rules:

1. If both e and e′ are constant values, two cases can happen. If the constant values e and e′ are the same, the BSO fault cannot be activated because there is no method of creating a G/B at t. Otherwise, use the constant e as G and e′ as B.

2. If either e or e′ (but not both) is a constant value, two cases can happen. If e is a constant, use e as G and select a B which is not equal to G. If e′ is a constant, use e′ as B and select a G which is not equal to B.
3. If neither e nor e′ is a constant value, the only constraint for selecting G and B is that the value of e is not equal to the value of e′.

Case 2 - When there is no other assignment statement with the same target

In this case, there is no method of initializing t to a certain logic value because S, the only statement with the target t, is faulty. Hence, there should be a special way of determining the initial value of t using only S. This can be accomplished under the assumption that the value of PO's is measured twice for a given test sequence during the testing of the device corresponding to the model. This will be discussed in detail in Section 5.7.

MOP Fault

Before discussing the activation of MOP faults, the following notation is defined:

O: an operator

O_f: a faulty operator perturbed from O

arg1, arg2, ..., argn: n arguments used for both O and O_f

M: a fault-free MOP which contains O and its arguments arg1, arg2, ..., argn

M_f: a faulty MOP which contains O_f and its arguments arg1, arg2, ..., argn

An MOP fault where an operator O is perturbed to O_f is activated by assigning the arguments to appropriate values such that $M \neq M_f$, i.e., $O(arg1, arg2, ..., argn) \neq O_f(arg1, arg2, ..., argn)$. As

mentioned in Chapter 4, the number of test vectors should increase as the number of bits of the arguments increases. Some heuristics which satisfy this rule have been developed and implemented in the heuristic test generator (HTG) [34,57,58].

As an example, let us consider an MOP fault where an operator ADD is perturbed to SUB. Let A and B be two n-bit arguments for ADD. ADD(A, B) can be perturbed in two different ways, SUB(A, B) and SUB(B, A). For the activation of the fault ADD(A, B) -> SUB(A, B), two conditions $B \neq 2^{n-1}$ and $B \neq 0$ are derived from the relation ADD(A, B) ≠ SUB(A, B). For the activation of the fault ADD(A, B) -> SUB(B, A), two conditions $A \neq 2^{n-1}$ and $A \neq 0$ are derived from the relation ADD(A, B) ≠ SUB(B, A). Assuming that both A and B are 4-bit vectors, the following six logic patterns are generated for A and B using the two sets of conditions:

A	B
XXXX	XXX1
XXXX	XX1X
XXXX	X1XX
XXX1	XXXX
XX1X	XXXX
X1XX	XXXX

The next problem is to assign logic values to the don't care bits of each logic pattern. To detect as many defects in adders as possible, all possible value pairs need to be created for each corresponding don't care bit of A and B [58]. It helps to create various logic values on carry propagation lines in adders and hence to detect more faults on carry propagation lines. Since the number of possible value pairs is four (00, 01, 10, and 11), each row in the above table produces

four test vectors if all don't care bits in a logic pattern of an argument are assigned the same logic value.

The following table shows the 15 test vectors generated from the six logic patterns using the heuristic. Nine out of 24 possible test vectors are not included in the table because the same vectors were already generated. Eight vectors are generated from the first half of the patterns and seven vectors from the second half (the test vector of all 1's is not included because it was already generated). The first pattern of the first half generates four vectors and each of the remaining two patterns generates two vectors. For an n-bit ADD, the first half of the patterns generates 4 + 2(n-2) (= 2n) test vectors and the second half generates 2n-1 vectors. Hence, 4n-1 test vectors are generated using the heuristic.

A	B	A	B
XXXX	XXX1	0000	0001
		0000	1111
		1111	0001
		1111	1111
XXXX	XX1X	0000	0010
		1111	0010
XXXX	X1XX	0000	0100
		1111	0100
XXX1	XXXX	0001	0000
		1111	0000
		0001	1111
XX1X	XXXX	0010	0000
		0010	1111
X1XX	XXXX	0100	0000
		0100	1111

5.2.2. Propagation Rules

Different propagation rules are defined for different positions of a good/bad value pair in a VHDL model. During the execution of the B-algorithm, each entry in the B-frontier is processed in turn using a proper propagation rule. Each entry in the B-frontier has a good/bad value pair at one of the four positions: 1) an inner expression, 2) the outermost RHS expression of an assignment statement, 3) the outermost condition expression of an *if* statement, and 4) the selection expression of a *case* statement.

A good/bad value pair at an inner expression

An expression in a VHDL model can be represented as O(arg1, arg2, ..., argn) where O is an operator and arg's are its arguments. A good/bad value pair at an argument of an operator can be propagated by assigning proper values to the other arguments. For primitive logic gates, the propagation rules used for gate level test generation algorithms can be used. For example, to propagate a D at an input of an *and* gate, the other inputs should be set to 1. Similar rules are defined for arithmetic and relational operators as follows:

ADD(arg1, arg2) - A good/bad value pair G/B at one argument can be propagated to the output by assigning any value to the other argument. A G/B at one argument is propagated to the output *unchanged* by assigning all 0's to the other argument. It is propagated *changed* by assigning values other than all 0's to the other argument.

SUB(arg1, arg2) - A G/B at one argument can be propagated to the output by assigning any value to the other argument. A G/B at arg1 is propagated to the output *unchanged* by assigning all 0's to arg2. A G/B at arg2 is propagated to the output *changed* regardless of the value of arg1.

In the following discussion, EQ, NEQ, LT, and LE denote VHDL operators =, =/, <, and <=, respectively.

EQ(arg1, arg2) and NEQ(arg1, arg2) - A G/B at an argument can be propagated to the output by assigning any value to the other argument. If the other argument is set to G, G/B is propagated to the output as 1/0 for EQ and as 0/1 for NEQ. If the other argument is set to B, G/B is propagated to the output as 0/1 for EQ and as 1/0 for NEQ.

LT(arg1, arg2) - A G/B at arg1 is propagated as 1/0 (0/1) by assigning arg2 such that $G < arg2 \leq B$ ($B < arg2 \leq G$). A G/B at arg2 is propagated as 1/0 (0/1) by assigning arg1 such that $B \leq arg1 < G$ ($G \leq arg1 < B$).

LE(arg1, arg2) - A G/B at arg1 is propagated as 1/0 (0/1) by assigning arg2 such that $G < arg2 < B$ ($B < arg2 < G$). A G/B at arg2 is propagated as 1/0 (0/1) by assigning arg1 such that $B < arg1 < G$ ($G < arg1 < B$).

Rules for GT and GE are not separately defined because LT(arg1, arg2) is equivalent to GT(arg2, arg1), and LE(arg1, arg2) is equivalent to GE(arg2, arg1).

A good/bad value pair at the outermost RHS expression of an assignment statement

An assignment statement may or may not be governed by one or more control statements. If an assignment statement is not governed by any control statement, a good/bad value at the RHS expression is automatically propagated to the target. Otherwise, the control expression(s) must be

set to proper values so that the assignment statement can be executed. The expressions together with the values are inserted into the J-frontier. This procedure replaces the role of the execution goal defined by O'Neill [30]. As an example, a VHDL code representing a positive edge-triggered D flip-flop with an asynchronous reset input (active LOW) is considered. In this example, a good/bad value G/B at the input D can be propagated to the output Q by setting RESET = 1 and CLK = R (rising edge).

```
process(CLK, RESET)
begin
  if (RESET = '0') then
    Q <= '0';
  elsif (CLK = '1' and CLK'EVENT) then
    Q <= D;
  end if;
end process;
```

A good/bad value pair at the outermost condition expression of an *if* statement

An *if* statement can have two clauses, the *then* clause and the *else* clause. Two types of *if* statements are defined depending on the existence of the *else* clause.

Definition: An *if* statement is **balanced** if it has both the *then* clause and the *else* clause.

Definition: An *if* statement is **unbalanced** if it has only the *then* clause.

A good/bad value G/B at a condition expression or a selection expression can affect the execution of statements under the clauses selected by G and B. A *good clause*, a *bad clause*, a *good statement*, and a *bad statement* are defined as follows:

Definition: A **good (bad) clause** is a clause which is selected when the control expression is set to a good (bad) value.

Definition: A **good (bad) statement** is a statement under a good (bad) clause.

The following notation will be used for the above definitions:

C_g: a good clause

C_b: a bad clause

S_g: a good statement under C_g

S_b: a bad statement under C_b

e_g: the RHS expression of S_g

e_b: the RHS expression of S_b

t_g: the target of S_g

t_b: the target of S_b

An *if* statement can have any combinations of *if*, *case*, and assignment statements inside it. However, the innermost statements must be assignment statements. The inputs of an *if* statement are the control expressions of control statements and the RHS expressions of assignment statements inside it. The outputs of an *if* statement are the targets of assignment statements inside it. A good/bad value pair at the condition expression of an *if* statement can be propagated to one of the targets by assigning proper values to the source expressions (control expressions and the RHS expressions) inside it. If any control statement encloses the *if* statement, the control expression of the control statement should be assigned proper values to select the *if* statement. Since a condition expression is a boolean expression, possible good/bad value pairs are 1/0 (D) and 0/1 ($\overline{D}$). The

propagation of a D or $\overline{D}$ at a condition expression to a target will be discussed for three different cases.

Case 1 - Two assignment statements S_g and S_b have the same target t.

Since S_g and S_b exist in different clauses, this case can occur only for a balanced *if* statement. A D or $\overline{D}$ at a condition expression is propagated to the common target t by assigning a good value G′ to the RHS expression e_g of S_g and a bad value B′ to e_b of S_b (the rules for selecting the values G′ and B′ were discussed in Section 5.2.1). If S_g (S_b) is controlled by other control statements within the *if* statement, the corresponding control expressions must be set to proper values so that S_g (S_b) can be executed. Since S_g is executed for the fault-free case and S_b is executed for the faulty case, a new good/bad value pair G′/B′ is created at t. The assignments e_g = G′ and e_b = B′ can be made in parallel in a single time period because only one of the statements S_g and S_b can be executed in a single time period depending on the value of the condition expression.

As an example, let us again consider the dual 2 x 1 multiplexer model shown in Section 5.2.1. A D at the expression (SEL = '1') can be propagated to X by assigning a G′ to E1 and a B′ to E2. As a result, a new value pair G′/B′ is set up at X. The D can also be propagated to Y as G′/B′ by assigning a G′ to F1 and a B′ to F2.

Case 2 - No two statements S_g and S_b have the same target, but a target inside a given *if* statement is used as a target outside it.

In this case, either a good value or a bad value, but not both, can be propagated through the *if* statement to a target (either t_g or t_b) because only one assignment statement (either S_g or S_b) inside the *if* statement can be used for the propagation. Hence, another assignment statement

outside the *if* statement should be used for initializing the target before the propagation. Let S′ denote an assignment statement (outside the *if* statement) which has the target t′ and the RHS expression e′. If $t' = t_g$, a bad value should be loaded into t′ using S′. If $t' = t_b$, a good value should be loaded into t′using S′. The propagation of a D or $\overline{D}$ at a condition expression to a target for these two cases is discussed below. In contrast to Case 1 when the propagation can be performed in a single time period, this procedure requires two time periods because two different values cannot be loaded into the same target in a single time period. Hence, this propagation method is called the *two-phase propagation*.

a) $t' = t_g$

1. Load a bad value B′ into t_g using S′. This can be accomplished by assigning B′ to e′ and a proper value to each control expression (if any exists) which governs the control of S′.
2. Assign a good value G′ to e_g. Since G′ is loaded into t_g only for the fault-free case, a new good/bad value pair G′/B′ is created at t_g.

b) $t' = t_b$

1. Load a good value G′ into t_b using S′. This can be accomplished by assigning G′ to e′ and a proper value to each control expression (if any exists) which governs the control of S′.
2. Assign a bad value B′ to e_b. Since B′ is loaded into t_b only for the faulty case, a new good/bad value pair G′/B′ is created at t_b.

As an example, let us consider the propagation of a D (1/0) at the condition expression (F2 = F3) of the *if* statement s2 to the target Z in the following VHDL code:

```
process (F1, F2, F3, X, Y)
begin
s1:  if (F1 = '1') then
s2:      if (F2 = F3) then
s3:          Z <= X(1 to 2);
         end if;
     else
s4:      Z <= Y(1 to 2);
     end if;
end process;
```

Since a D is currently set up at the condition expression, the good clause C_g of s2 is the *then* clause and the bad clause C_b is the *else* clause. C_g contains the assignment statement s3 which has the same target as s4. First, a good value G′ and a bad value B′ need to be selected. Let G′ = 00 and B′ = 11. Second, B′ = 11 is assigned to Y and the condition expression of s1 is set to 0, i.e., F1 = 0. As a result, B′ = 11 is loaded into Z. Third, G′ = 00 is assigned to X and the condition expression of s1 is set to 1, i.e., F1 = 1. As a result, a D at (F2 = F3) has been propagated to Z as 00/11.

In the above example, another assignment statement with the same target (s4) is outside the *if* statement (s2), but is still inside the same process statement. There are cases when another statement with the same target exists in a different process statement. In this situation, a bus resolution function (BRF) is required to resolve the values of the two drivers created by the two assignment statements with the same target. The propagation rules for BRF's will be discussed in detail in Section 5.4.

Case 3 - No target inside a given *if* statement is used as a target inside or outside the *if* statement.

As in Case 2, either a good value or a bad value, but not both, can be propagated to the target (t_g or t_b) because only one statement (either S_g or S_b) inside the *if* statement is used for the propagation. Since there is no other assignment statement with the same target in the given VHDL model, there is no method of initializing the target before the propagation. Hence, there should be a special method of determining the initial value of the target. This is the same situation encountered in the second case of the activation of BSO faults. This will be discussed in detail in Section 5.7.

A good/bad value pair at the selection expression of a case statement

A *case* statement can have any combinations of *if*, *case*, and assignment statements inside it. However, the innermost statements must be assignment statements. The inputs of a *case* statement are the control expressions of control statements and the RHS expressions of assignment statements inside it. The outputs of a *case* statement are the targets of assignment statements inside it. Hence, a good/bad value pair at the selection expression of a *case* statement should be propagated to one of the targets by assigning proper values to the source expressions. If any control statement encloses the *case* statement, the control expression of the control statement should be assigned proper values to select the *case* statement.

Since a selection expression is a signal name, various good/bad value pairs can be set up at a selection expression while only a D or $\overline{D}$ is set up for a condition expression. Therefore, the propagation through a *case* statement is a generalization of the propagation through an *if* statement. The propagation through a case statement is also defined for three different cases. Since the basic approach is the same as in the propagation through an *if* statement, only Case 1 will be discussed in detail.

Case 1 - Two assignment statements S_g and S_b have the same target t.

As in the Case 1 of the propagation through the *if* statement, a good/bad value pair G/B at a condition expression is propagated to the common target t by assigning a good value G′ to the RHS expression e_g of S_g and a bad value B′ to e_b of S_b. If S_g or S_b is controlled by other control statements within the *case* statement, the corresponding control expressions must be set to proper values so that S_g or S_b can be executed. Since S_g is executed for the fault-free case and S_b is executed for the faulty case, a new good/bad value pair G′/B′ is created at t. The assignments e_g = G′ and e_b = B′ can be made in parallel in a single time period because only one of the statements S_g and S_b can be executed in a single time period.

As an example, let us consider the propagation of a good/bad value at the signal SEL to Z in the following *case* statement.

```
  case SEL(1 to 2) is
s1:  when "00" => Z <= X1(1 to 2);
s2:  when "01" => Z <= X2(1 to 2);
s3:  when "10" => Z <= X3(1 to 2);
s4:  when "11" => Z <= X4(1 to 2);
  end case;
```

Suppose that a good/bad value pair 00/10 is currently set up at the signal SEL. Then, S_g = s1 and S_b = s3. This value pair can be propagated to Z by assigning a G′ to X1 and a B′ to X3. Let G′ = 00 and B′ = 11. Then, a new value pair 00/11 is set up at Z.

The propagation through a *case* statement for the following cases is similar to the propagation through an *if* statement.

Case 2 - No two statements S_g and S_b have the same target, but a target inside a given *case* statement is used as a target outside it.

Case 3 - No target inside a given *case* statement is used as a target inside or outside the *case* statement.

5.2.3. Justification Rules

As mentioned in Section 5.1, an entry in the J-frontier contains an expression *e* and a logic value *v*. There exist two types of expressions in the J-frontier, target signals of assignment statements and virtual signals corresponding to logic, arithmetic, and relational operations. Different justification rules are defined for different types of expressions.

A value at the target signal of an assignment statement

A signal t can be used as a target of one or more assignment statements. The B-algorithm first selects one of the assignment statements with the target t. The selected assignment statement may or may not be governed by one or more control statements. Let S be an assignment statement with the target t and an RHS expression e. Let v be a value that was assigned to t. If S is not governed by any control statement, the justification is performed by assigning v to e. If e is not a primary input, e and v will be added to the J-frontier. If S is governed by any control statement, the justification is performed by assigning v to e and proper values to control expressions so that S can be executed. The control expressions together with logic values are newly added to the J-frontier.

There are cases when the target t of an assignment statement also appears in its RHS expression. This situation can be expressed as follows:

```
t <= O(t, arg1, arg2, ..., argn-1);
```

This situation implies a sequential behavior because t cannot be assigned two different values in a single time period. It is assumed that this type of assignment statements is controlled by at least one control expression including a clock signal. This is a reasonable assumption because the real circuits behave in this way. As an example, a VHDL code representing a 4-bit counter is considered.

```
  process (CLK, CLEAR)
  begin
    if (CLEAR = '0') then
s1:       COUNT <= "0000";
    elsif (CLK = '1' and CLK'EVENT) then
s2:       COUNT <= ADD (COUNT, "0001");
    end if;
  end process;
```

If an entry of the J-frontier contains a signal COUNT and a value "1111", the value can be obtained by executing the statement s1 once and the statement s2 (2^4-1) times. In general, the B-algorithm handles the justification in the above-mentioned situation (a target also appears in the RHS expression of an assignment statement controlled by a clock signal) as follows:

1. Find another assignment statement with a target t and an RHS expression c, a constant logic value (if no such assignment statement exists, i.e., no initialization logic exists, no test can be generated.). Assign proper values to the control expressions which govern the assignment statement so that the constant value c can be transferred to t. Use c as a reference for the justification of the value, say v, of the target t. If $v = c$, the justification is done. Otherwise, go to Step 2.
2. Find the assignment statement which has t in its RHS expression and is controlled by a clock signal. Set the target t to the value v in the current time period. Assign proper values to the

control expressions which govern the assignment statement so that the assignment statement can be executed. Determine the value, say v', of t in the previous time period using the operator O. If $v' = c$, the justification is done. Otherwise, set t to v' and repeat the justification procedure. This procedure continues until the target t is set to v'.

A value at a virtual signal corresponding to an operation

The value of a virtual signal corresponding to an operation is justified by assigning values to its arguments. If an operation includes any argument which is also an operation, the value of the outer operation is justified by assigning a proper logic value to the inner operation. If a value is assigned to the inner operation and the input values of the inner operation are not uniquely determined by an implication procedure, the inner operation is in turn inserted into the J-frontier with its value. This procedure continues until all arguments of an operation are simple signal names.

1) A value at the output of a logic operator

The justification of a logic value at the output of a logic operator is performed using the rules defined for gate level test generation algorithms [41]. For example, to justify a logic 1 at the output of an *and* gate, all inputs of the *and* gate are set to 1.

2) A value at the output of an arithmetic operator

Two arithmetic operators, ADD(arg1, arg2) and SUB(arg1, arg2), are included in the subset defined in Chapter 3. Many different value pairs of arg1 and arg2 can be used to satisfy a value assigned to one of the two arithmetic operators. In order to assign a logic value to an n-bit adder ADD(arg1, arg2), 2^n different value pairs of arg1 and arg2 can be selected. For example, 2^{16}

combinations are possible for a 16-bit adder. It is not practical to allow every possible combination to be tried during the execution of the B-algorithm. Two heuristic methods are used to solve this problem. First, a wide vector is divided into small pieces, and the justification is performed for each piece. For example, if a 16-bit vector can be divided into four 4-bit vectors, 2^{16} combinations can be reduced to 4 x 2^4 combinations. Second, the maximum number of combinations is limited by the B-algorithm.

3) A value at the output of a relational operator

The value to be assigned to a relational operator is always a logic 1 or 0. However, as in arithmetic operators, the number of possible combinations for the values of arguments is very large for a large n, the number of bits. For example, to justify a value 1 for the relational operator EQ(arg1, arg2), the constraint arg1 = arg2 is used for selecting the values of arg1 and arg2. To justify a value 0, the constraint arg1 $\neq$ arg2 is used. The justification of values for other relational operators NEQ, LT, and LE is similar to the justification of a value for EQ.

A special method is used for justifying a value of a relational operation which includes an attribute 'EVENT or 'STABLE. As discussed in Chapter 3, an expression (S = C and S'EVENT) where S is a signal and C $\in$ {0, 1} is converted to an expression (S = C′) where C′ $\in$ {R, F}. If the converted expression is (S = R), S is set to R to justify a 1 at the expression and is set to one of the values {F, 1, 0} to justify a 0 at the expression. If the converted expression is (S = F), S is set to F to justify a 1 and is set to one of the values {R, 1, 0} to justify a 0.

5.3. Assignment of Time Periods

Since the behavioral test generation algorithm handles VHDL models representing sequential behaviors as well as combinational behaviors, the assignment of time periods is an important issue. The basic rule for the assignment of time periods is to assign a single time period whenever a clock transition is required during the test generation procedure. Section 5.3.1 discusses how this rule is applied for generating tests from VHDL models which are written based on the *event-driven* simulation semantics. Section 5.3.2 and 5.3.3 discuss exceptional cases when this rule cannot be directly applied. Section 5.3.2 discusses the assignment of time periods for *two-phase activation*. Section 5.3.3 discusses the assignment of time periods for *two-phase propagation*.

5.3.1. Simulation Semantics vs. Test Generation Semantics

In digital logic simulation, an event is defined as a change on a signal value. An event-driven simulation is a type of digital logic simulation which proceeds only when an event occurs. An initial event triggers simulation and may cause an event or events, which may in turn cause more events. Simulation continues until no event occurs. VHDL is a typical event-driven simulation language. The process statement is one of the VHDL constructs which represent the concept of event-driven simulation. A set of statements inside a process statement is executed only when a signal in its sensitivity list changes its value. This approach is very efficient for digital logic simulation. However, if this concept is directly used for test generation, unnecessarily long test sequences can be generated because a signal change is always necessary for the justification and the propagation operations through a process statement [59].

As discussed in [59], the test generation algorithm can overcome this problem by ignoring the sensitivity lists of process statements during the propagation and the justification operations. Once

the sensitivity list of a process statement is ignored, the execution of signal assignment statements is governed by the control expressions of control statements inside the process statement. As discussed in Section 3.5, a control expression may or may not have an attribute 'EVENT or 'STABLE. A clocked behavior can be distinguished from a non-clocked behavior by checking the existence of an attribute 'EVENT or 'STABLE in a control expression. Shown below are the rules for the assignment of a value to a signal in a control expression. This method is efficient for test generation because unnecessary signal transitions for invoking processes are avoided.

1. If a signal in a control expression is accompanied by an attribute 'EVENT or 'STABLE, consider it as a clock signal and generate a signal transition (R or F) for it.
2. If a signal in a control expression is not accompanied by either 'EVENT or 'STABLE, consider it as a non-clock signal and generate a signal level (1 or 0) for it.

The B-algorithm assigns a time period whenever a signal transition R or F is generated for a clock signal. Other signal levels generated during the test generation procedure are put into the same time period until another signal transition is generated. It is assumed that all signal levels in the same time period are stable before a signal transition occurs. This is exactly what happens in the real hardware.

Example

Figure 12 shows a VHDL model for a circuit consisting of a 2 x 1 multiplexer (the process p1), a 4-bit register with an asynchronous reset (p2), and an output buffer (p3). The process p1 selects one of the input data depending on the value of SEL. P2 resets the register REG when CLEAR = 0, and loads DATA into REG when CLEAR = 1 and CLK = R (rising edge). P3 outputs the value of REG to DO when EN = 1, and all 1's when EN = 0. Let us assume a BSA-0 fault at the condition

expression (CLEAR = '0'). This fault can be activated by applying a logic 1 to the condition expression and thus creating a good/bad value 1/0 (D) at the condition expression.

Case 1 - The sensitivity lists are not ignored.

The good value (1) at the condition expression can be justified by setting CLEAR to 0. The propagation of 1/0 (D) at the condition expression through the *if* statement s4 to REG can be performed by selecting two statements s5 and s6, selecting a new good value G′ (0000: the value of the RHS expression of s5) and a new bad value B′, assigning B′ to the RHS expression (DATA) of s6, and setting the condition for executing s6 for the faulty case. Let us select B′ = 1111. With CLK = R, s5 is executed for the fault-free case (CLEAR = 0) and s6 is executed for the faulty case (CLEAR = 1). As a result, a new good/bad value pair G′/B′ (0000/1111) is set up at DATA. A signal transition R at CLK can invoke the process p2 and thus no additional signal transition is necessary for invoking p2. The value (1111) of DATA can be justified by setting D1 = 1111 and SEL = 1 (or setting D2 = 1111 and SEL = 0). However, a signal level at D1 (1111) or SEL (1) cannot invoke the process p1. In order to invoke the process p2, a signal transition R instead of a signal level 1 is assigned to SEL (as an alternative, a signal transition can be created at D1). As a result, 1/0 at the condition expression is propagated to REG as 0000/1111. This new good/bad value pair can be propagated to a primary output DO by setting EN = 1. A signal transition R instead of a signal level 1 is assigned to EN in order to invoke the process p3. From this procedure, the following test sequence is generated for the given BSA-0 fault:

time	D1	D2	SEL	CLEAR	CLK	EN	DO
t1	1111	x	R	x	x	x	-
t2	x	x	x	0	R	x	-
t3	x	x	x	x	x	R	0000/1111

```
entity MUX_REG_BUF (
       SEL, CLEAR, CLK, EN in bit;
       D1, D2: in bit_vector(0 to 3);
       DO: out bit_vector(0 to 3)) is
end MUX_REG_BUF;


architecture DATA_FLOW of MUX_REG_BUF is
   signal DATA, REG: bit_vector(0 to 3);
begin

   p1: process(SEL, D1, D2)
   begin
s1    if (SEL = '1') then
s2       DATA <= D1;
      else
s3       DATA <= D2;
      end if;
   end process p1;

   p2: process(CLEAR, CLK)
   begin
s4    if (CLEAR = '0') then
s5       REG <= "0000";
      elsif (CLK = '1' and CLK'STABLE) then
s6       REG <= DATA;
      end if;
   end process p2;

   p3: process(REG, EN)
   begin
s7    if (EN = '1') then
s8       DO <= REG;
      else
s9       DO <= "1111";
      end if;
   end process p3;

end DATA_FLOW;
```

Figure 12. A VHDL model consisting of a multiplexer, a register, and a buffer.

Case 2 - The sensitivity lists are ignored.

In this case, the signal levels assigned during the propagation and the justification operations need not be modified to signal transitions to invoke processes. The test generation procedure is the same as in Case 1 except the modification of values. Since unnecessary signal transitions for invoking processes are not generated, only one time period is necessary for this test sequence while three time periods were necessary for the one generated in Case 1. From this procedure, the following test sequence is generated for the given fault:

time	D1	D2	SEL	CLEAR	CLK	EN	DO
t1	1111	x	1	0	R	1	0000/1111

5.3.2. Assignment of Time Periods for Two-phase Activation

As discussed in Section 5.2.1, a BSO fault on an assignment statement can be activated using a special scheme called *two-phase activation*, which consists of two phases: *preloading* and *transfer*. Preloading loads a logic value to a given target using a fault-free assignment statement. Transfer attempts to load a different value to the target using the faulty assignment statement. Since preloading must precede transfer, preloading and transfer cannot be performed in a single time period and therefore each phase requires at least one time period. In general, preloading or transfer may generate values spanning more than one time period. For each phase, time periods are assigned based on the *one time period for one clock transition* rule discussed in Section 5.3.1. All time periods assigned for preloading must precede all time periods assigned for transfer. Let t_i denote the initial time period assigned for preloading. In general, every value assignment for preloading must be made before t_i or at t_i, and every assignment for transfer must be made at t_{i+1} or after t_{i+1}.

As an example, let us consider the BSO fault on the statement s6 in the VHDL model shown in Figure 12. This fault can be activated using the following procedure. The test sequence generated from this procedure is shown in the table below.

1. Select the assignment statement s5 which has the same target (REG) as s6.
2. Select a good value G and a bad value B. G = 1111 and B = 0000 (the value of the RHS expression of s5) are selected.
3. In order to execute s5, assign CLEAR = 0 with a time tag t_i. As a result, B = 0000 is loaded into REG at t_i. Since CLEAR is a PI, no more justification is necessary.
4. Assign G (1111) to DATA with a time tag t_{i+1}. In order to execute s6, assign CLEAR = 1 and CLK = R with t_{i+1}. As a result, a good/bad value pair 1111/0000 is set up at REG.
5. Since CLEAR and CLK are PI's, no more justification is necessary. To justify DATA = 1111, assign SEL = 1 and D1 = 1111 with t_{i+1}.

time	D1	D2	SEL	CLEAR	CLK	EN	DO
t_i	x	x	x	0	x	x	-
t_{i+1}	1111	x	1	1	R	x	-

The above procedure generates two time periods although the attribute 'EVENT appears only once in the VHDL model. In fact, the generated test sequence first resets the register and then loads a data value to the register. This is a proper test sequence which can detect the behavior of a register whose reset function is fault-free, but whose data load function is faulty. It should be noted that the B-algorithm can automatically generate this type of test sequences without a special knowledge or an experience of an expert.

5.3.3. Assignment of Time Periods for Two-phase Propagation

As discussed in Section 5.2.2, the propagation of a good/bad value at the control expression of a control statement needs a special scheme called *two-phase propagation* when no two assignment statements inside the control statement have the same target, but a target inside the control statement is used as a target outside it. If no target inside the control statement is not used anywhere in the model, two-phase testing is required, which will be discussed in detail in Section 5.7. Two-phase propagation is similar to two-phase activation because it also creates a new good/bad value pair at a target. Two-phase propagation also consists of two phases: preloading and transfer. Preloading loads a logic value to a given target using an assignment statement outside the control statement. Transfer attempts to load a different value to the target using an assignment statement inside the control statement. As in two-phase activation, all time periods assigned for preloading must precede all time periods assigned for transfer.

As an example, let us consider the VHDL code shown below. Suppose that C1, C2, and DI are PI's and that DO and CO are PO's.

```
    process (C1, C2)
    begin
s1    if (C1 = '1' and C1'EVENT) then
s2       DO <= DI;
s3       CO <= '1';
       end if;
s4    if (C2 = '1' and C2'EVENT) then
s5       CO <= '0';
       end if;
    end process;
```

Suppose that a D (1/0) is currently set up at the condition expression of s1. To propagate the D to CO, the following procedure is used:

1. Select two statements s3 and s5.
2. Select a new good value 1 (the RHS expression of s3) and a new bad value 0 (the RHS expression of s5).
3. To load the new bad value to CO, the statement s4 needs to be executed. Set C2 = R with a time tag t_i.
4. To load the new good value to CO, the statement s1 needs to be executed. Set C1 = R with a time tag t_{i+1}. As a result, a new good/bad value pair 1/0 with a time tag t_{i+1} is set up at CO.

The following table shows the test sequence generated from the above procedure:

time	C1	C2	DI	CO	DO
t_i	R	x	x	-	-
t_{i+1}	x	R	x	1/0	-

5.4. Handling of Bus Resolution Functions

As discussed in Section 3.4.1, a bus resolution function (BRF) is required if a signal appears as a target of an assignment statement in two or more process statements. Four types of BRF's (*one-hot*, wired-X, *wired-or*, and *wired-and*) introduced in Section 3.4.1 are considered for test generation. Test generation rules cannot be applied to every construct in BRF's because they usually consist of variable assignment statements and loop statements which are not included in the VHDL subset defined in Chapter 3. Instead, test generation rules are defined for each type of BRF's considering each BRF as a single entity. The user is supposed to specify the type of a BRF and the name of the signal whose value will be resolved by the BRF. Whenever the B-algorithm encounters a BRF during the test generation procedure, it performs test generation operations on

the BRF using the test generation rules defined for it. The rules for each type of BRF's will be discussed below.

While non-BRF operations use three logic values {0, 1, X}, BRF's use four logic values {0, 1, X, Z}, where Z denotes a high impedance state. Figure 13 shows logic operations for the four BRF's when only two drivers are involved.

one-hot

	0	1	X	Z
0	X	X	X	0
1	X	X	X	1
X	X	X	X	X
Z	0	1	X	Z

wired-X

	0	1	X	Z
0	0	X	X	0
1	X	1	X	1
X	X	X	X	X
Z	0	1	X	Z

wired-and

	0	1	X	Z
0	0	0	0	0
1	0	1	X	1
X	0	X	X	X
Z	0	1	X	Z

wired-or

	0	1	X	Z
0	0	1	X	0
1	1	1	1	1
X	X	1	X	X
Z	0	1	X	Z

Figure 13. Logic Operations for BRF's

Let s_r denote a resolved value of a signal s. The value s_r of a BRF with *n* drivers are determined as shown below.

<u>One-hot</u>

1. If a driver has a logic value V ($\in$ {0, 1}) and other drivers have Z's, s_r = V.
2. If all drivers have Z's, s_r = Z.
3. For other cases, s_r = X.

<u>Wired-X</u>

1. If at least one driver has a logic value V ($\in$ {0, 1}) and other drivers have Z's, s_r = V.

2. If all drivers have Z's, $s_r = Z$.

3. For other cases, $s_r = X$.

Wired-and

1. If a driver has a logic value 0, $s_r = 0$.

2. If at least one driver has a 1 and other drivers have Z's, $s_r = 1$.

3. If all drivers have Z's, $s_r = Z$.

4. For other cases, $s_r = X$.

Wired-or

1. If a driver has a logic value 1, $s_r = 1$.

2. If at least one driver has a 0 and other drivers have Z's, $s_r = 0$.

3. If all drivers have Z's, $s_r = Z$.

4. For other cases, $s_r = X$.

Propagation

A good/bad value pair at a driver of a BRF can be propagated to its output by assigning values to other drivers as follows:

One-hot - other drivers are assigned to Z's.

Wired-X - other drivers are assigned to Z's.

Wired-and - other drivers are assigned to 1's or Z's.

Wired-or - other drivers are assigned to 0's or Z's.

Example

Let us consider three processes which contain a common target as shown below. Suppose that the signal t is resolved by a *wired-and* BRF. A good/bad value pair at e1 can be propagated to t by assigning a 1 or a Z to e2 and e3. For a wired-or BRF, the value pair can be propagated by assigning a 0 or a Z to e2 and e3. For a *one-hot* BRF and a *wired-X* BRF, both e2 and e3 must be set to Z to propagate a good/bad value pair at e1 to t.

```
p1: process(sensitivity list)
begin
  .
  t <= e1;
  .
end process p1;

p2: process(sensitivity list)
begin
  .
  t <= e2;
  .
end process p2;

p3: process(sensitivity list)
begin
  .
  t <= e3;
  .
end process p3;
```

Justification

A value s_r at a signal with multiple drivers is justified using the following rules:

One-hot

1. If $s_r = 1$, set a driver to 1 and others to Z's.

2. If $s_r = 0$, set a driver to 0 and others to Z's.
3. If $s_r = Z$, set all drivers to Z's.

Wired-X

1. If $s_r = 1$, set at least one driver to 1 and others to Z's.
2. If $s_r = 0$, set at least one driver to 0 and others to Z's.
3. If $s_r = Z$, set all drivers to Z's.

Wired-and

1. If $s_r = 1$, set at least one driver to 1 and others to Z's.
2. If $s_r = 0$, set a driver to 0.
3. If $s_r = Z$, set all drivers to Z's.

Wired-or

1. If $s_r = 1$, set a driver to 1.
2. If $s_r = 0$, set at least one driver to 0 and others to Z's.
3. If $s_r = Z$, set all drivers to Z's.

Example

Suppose that the target t in the previous three-process model is resolved by a *wired-and* BRF. A value 1 at t can be justified by setting a driver to 1 and others to Z's. For example, e1 = 1, e2 = Z, and e3 = Z justify the value 1 at t. A value 0 at t can be justified by setting a driver to 0. For example, e2 = 0 justifies the value 0 at t.

5.5. Handling of Reconvergent Fanout

Reconvergent fanout is a topology where branches of a fanout point converge on a path from the fanout point to a PO. Reconvergent fanout in a circuit model often creates conflict in signal value assignment during test generation [41]. In the presence of reconvergent fanout, a *sensitized value* (a good/bad value pair) can be propagated through one or more reconvergent paths. A *single path propagation* is a scheme where only a single path is selected to propagate a sensitized value. It is known that a single path propagation is incomplete when the five-valued logic (0, 1, D, $\overline{D}$, and X) is used for test generation [56]. D-algorithm solves this problem by using a *multiple path propagation* where more than one paths are simultaneously selected to propagate a sensitized value. If there are n possible propagation paths, D-algorithm could try up to 2^n-1 combinations of paths (the combination of no path is excluded from 2^n possible combinations).

Lam [33] developed a multiple path propagation scheme which includes a procedure for detecting reconvergent fanout:

1. Propagate a fault through a path toward a PO. Record signals which are on the propagation path.
2. During justifications for the propagation, check if one of the signals that need to be justified is on the propagation path.
3. If none of the signals that need to be justified is on the propagation path, then there is no reconvergent fanout.
4. If one of the signals that need to be justified is on the propagation path, try to justify the desired value without altering the good/bad value pair assigned to the signal. If this can be done, justification is a success. Otherwise, go to Step 5.

5. Propagate the fault through a second path as well as the first path by justifying the signals which are not on the propagation paths.

Lam's approach can handle reconvergent with two paths, but needs to be generalized to handle reconvergent fanout with more than two paths. If Step 5 in the above procedure is not successful and a third path exists, the procedure has to try the third path. This procedure should continue until propagation is successful or no more path exists. If the number of reconvergent paths is large and/or reconvergent paths are nested, the procedure of detecting reconvergent fanout will cause a significant overhead to the test generation algorithm.

Using the B-frontier, the procedure for detecting reconvergent fanout is not required for a multiple path propagation. Whenever a good/bad value pair is assigned at a fanout stem, each propagation path corresponding to a branch at the fanout point is inserted into the B-frontier. The B-algorithm processes each entry in the B-frontier, which may in turn add more entries to the B-frontier. This procedure continues until the good/bad value pair reaches at a PO (success) or the B-frontier is empty (failure). During the procedure, the B-algorithm neither records signals on propagation paths nor checks if the signals that need to be justified are on the propagation paths.

Example

A VHDL model which contains reconvergent fanout is shown below. As shown in the model, the signal SEL forms a fanout point because it appears in more than one source expression. Let SEL_0 denote the fanout stem of SEL, SEL_1 the fanout branch of SEL used in the *if* statement s1, and SEL_2 the fanout branch of SEL used in s2. Each fanout branch controls a 2x1 multiplexer. The

outputs of the two multiplexers are converged through an *and* operator in the process p3. Hence, reconvergent fanout is formed in the model.

```
entity RCVG_FO (
        SEL: in bit;
        A1, A2, B1, B2: in bit_vector(0 to 3);
        Z: out bit_vector(0 to 3)) is
end RCVG_FO;

architecture DATA_FLOW of RCVG_FO is
  signal X, Y: bit_vector(0 to 3);
begin
  p1: process(SEL, A1, A2)
  begin
  s1: if (SEL = '1') then
        X <= A1;
      else
        X <= A2;
      end if;
  end process p1;

  p2: process(SEL, B1, B2)
  begin
  s2: if (SEL = '1') then
        Y <= B1;
      else
        Y <= B2;
      end if;
  end process p2;

  p3: process(X, Y)
  begin
  s3:  Z <= X and Y;
  end process p3;
end DATA_FLOW;
```

Let us consider a BSA-0 fault at SEL_0. It can be activated by applying a logic 1 to SEL_0. As a result, a D (1/0) is created at SEL_0. The D at SEL_0 is assigned to SEL_1 and SEL_2 through an *implication* procedure, i.e., SEL_1 = D and SEL_2 = D. The B-frontier currently has two

entries, the *if* statements s1 and s2, i.e., B-frontier = {s1, s2}. Let us first select s1. The D at SEL_1 can be propagated to X by assigning a good value G′ to A1 and assigning a bad value B′ to A2. As a result, a new good/bad value pair G′/B′ is set up at X. Now, s1 is removed from the B-frontier and s3 is added to the B-frontier. Hence, B-frontier = {s2, s3}. Let us select s3 from the B-frontier. The value pair G′/B′ at X can be propagated to Z by assigning 1111 to Y. To justify (Y = 1111), either (SEL_2 = 1 and B1 = 1111) or (SEL_2 = 0 and B2 = 1111) can be selected. However, both assignments conflict to the previous assignment (SEL_2 = D). Let us select s2 from the B-frontier. The D at SEL_2 is propagated to Y by assigning G′ to B1 and assigning B′ to B2. As a result, a good/bad value pair G′/B′ is set up at Y. The D at SEL_1 has already been propagated to X as G′/B′. Since (G′ and G′ = G′) and (B′ and B′ = B′), a good/bad value pair G′/B′ is set up at Z (by an *implication* procedure). Figure 14 schematically shows the multiple path propagation discussed above.

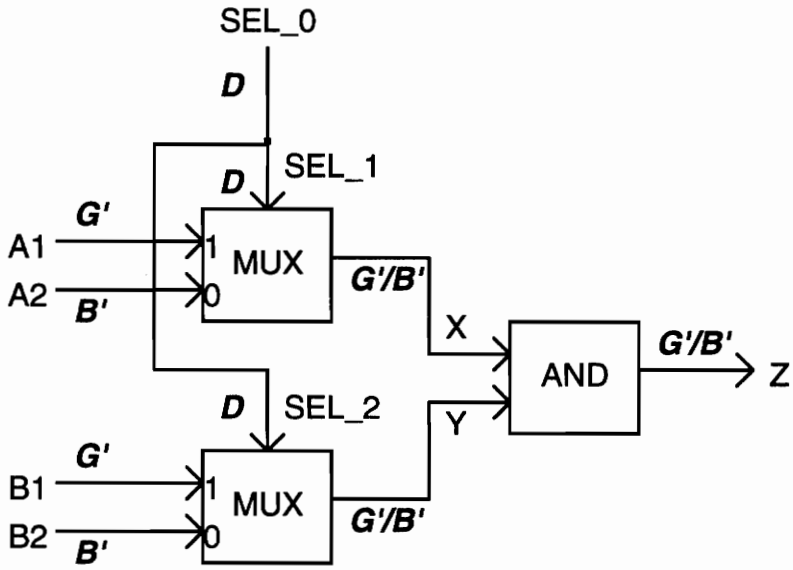


Figure 14. A Multiple Path Propagation

In the above procedure, a single value (Y = 1111) is tried to propagate G′/B′ at X to Z. A conflict is detected because a single value is to be assigned to SEL_1 which was already assigned to a good/bad value pair D (1/0). If a good/bad value pair is assigned to Y, i.e., Y = 1111/XXXX, G′/B′ at X can be propagated to Z without a conflict. This can be accomplished by assigning 1/X to SEL_2 and 1111/XXXX to B1. In other words, a single path propagation can be used in this case by using a good/bad value pair for every signal in the model. This is the same method used by the 9-V algorithm [56] which uses a 9-valued logic shown in Section 5.1.1. Although this method allows a single path propagation, it is not adopted by the B-algorithm because the overhead in manipulating good/bad value pairs for every test generation operation would be significant.

5.6. Handling of Feedback Loops

Let S be an assignment statement with a target t and a RHS expression e. Let c_1, c_2, ..., c_n be control expressions (if any) which control S. A *feedback loop* is formed when the target t of S is directly or indirectly (through other statements) used in e or c's. The target t is called a *feedback signal*. If t is used in e, the feedback loop is called a *self-loop*. The statement s2 in the following VHDL code illustrates a self-loop.

```
process(CLEAR, CLK)
begin
     if (CLEAR = '0') then
s1:     COUNT <= "00";
     elsif (CLK = '1' and CLK'EVENT) then
s2:     COUNT <= ADD (COUNT, "01");
     end if;
end process;
```

A case when a target t is indirectly used in e or c's is illustrated in the two-process model shown below. The target signal COUNT of the statement s6 in the model is used in the control expression

of s1 which controls s2 and s3. The target signal LOAD of s2 and s3 is used in a control expression of s4 which controls s6. Hence, a feedback loop (COUNT -> LOAD -> COUNT) is formed in the model. The statement s6 also forms a self-loop because COUNT is used in its RHS expression. The target signals LOAD and COUNT are feedback signals.

```
process(LIM, COUNT)
begin
s1:  if (LIM =/ COUNT) then
s2:     LOAD <= '1';
     else
s3:     LOAD <= '0';
     end if;
end process;

process(CLEAR, CLK)
begin
s4:  if (CLEAR = '0') then
s5:     COUNT <= "00";
     elsif ((LOAD = '1') and (CLK = '1' and CLK'EVENT)) then
s6:     COUNT <= ADD(COUNT, "01");
     end if;
end process;
```

A synchronous sequential behavior is represented as a set of assignment statements controlled by a clock signal which is represented by an attribute 'EVENT or 'STABLE. With a transition on the clock signal, the RHS expression of each assignment statement is transferred to its target. If a target is a feedback signal and is controlled by a clock, its current state is dependent on its previous state. If it is on a self-loop, a value v_i on a feedback signal at a time period t_i can be justified by assigning a proper value v_{i-1} to the signal at t_{i-1}. The value v_{i-1} in turn can be justified by v_{i-2}. If the initial value of the signal is known, this procedure will continue until the justification can be accomplished using the initial value. However, if the initial value is unknown, this procedure will continue infinitely. Hence, the feedback signal needs to be initialized for the justification. If it is not on a self-loop, its current state is also dependent on the state of the other signals in the feedback

loop. If the state of the other feedback signals is unknown, its current state cannot be determined even when its previous state is known. For example, if the value of LOAD in the above two-process model is unknown, the current state of COUNT cannot be determined even with the knowledge of its previous state. Hence, the other feedback signals also need to be initialized for the justification.

The design of a sequential circuit always includes logic which can initialize the circuit to a certain state in any time period. For example, in the above single-process model, the signal COUNT can be initialized to 00 with CLEAR = 0 at any time. Hence, it is assumed that any VHDL code which represents a synchronous sequential behavior has an initialization logic. Whenever a feedback signal controlled by a clock is assigned a certain logic value, the initialization logic is referred to see if it can justify the logic value. For example, in the single-process model, whenever the signal COUNT is assigned 00 during test generation, the statement s1 is used to justify the value.

Let S be an assignment statement which has a target t on a feedback loop and a RHS expression e, and is controlled by a clock. Since an initialization logic is assumed to exist, there exists another statement S′ which has the same target t and a constant value c as its RHS expression. Suppose that the target t is assigned a value v during test generation. The B-algorithm first compares two values v and c. If v = c, the justification is successful. Otherwise, the B-algorithm first sets the target t to a value v with a time tag t_i. The value v of t can be justified by assigning proper values to other control expressions which controls S, generating a clock signal, and assigning the value v to e. The value v' of t at the time period t_{i-1} can be determined using the value v and the expression e. If v' = c, the justification is successful. Otherwise, the new value v' is assigned to the target and the same justification procedure is repeated. This procedure continues until the value of t reaches c.

A feedback signal controlled by a clock can be initialized only if the corresponding initialization logic is fault-free. If a fault exists in the initialization logic, there is no way of initializing the feedback signal. For example, let us consider a BSO fault on the statement s5 in the two-process model. Two-phase activation is required for activating this fault. First, a bad value is assigned to the RHS of the statement s6 which has the same target. However, there is no way of justifying this value because the initialization logic (the statement s5) is faulty.

5.7. Two-phase Testing

In Section 5.2, *two-phase activation* and *two-phase propagation* were discussed. Two-phase activation is used for activating a BSO fault. Two-phase propagation is used for propagating a good/bad value pair through an *if* or *case* statement. The basic idea of two-phase activation and two-phase propagation is to *initialize* a selected target to a certain logic value using an assignment statement (the first phase) and then *transfer* a different value to the target using another statement (the second phase). The target obtains a logic value after the first phase and a good/bad value pair after the second phase. However, if a selected target appears in only one assignment statement, both initialization and transfer should be performed using the single assignment statement. In this special situation, there is no method of initializing the target because the assignment statement is affected by the fault under consideration. Hence, there should be a method of determining the initial value of the target without initializing it. This can be accomplished by propagating the value of the target to a PO in each phase and measuring the PO at the end of each phase during the hardware testing, i.e., during the testing of a digital device described by the VHDL model. This method is called *two-phase testing* because PO's are measured twice for detecting a fault.

5.7.1. Two-phase Testing for Two-phase Activation

Let S be an assignment statement with a target t and a RHS expression e. Two-phase testing is required for testing the BSO fault on S if the target t is not used in any other statement in the model. The B-algorithm generates a test sequence for the BSO fault assuming that PO's are measured at the end of each phase during the hardware testing. The B-algorithm first chooses a good value G and a bad value B ($G \neq B$) for t and then performs the following procedure:

1. The RHS expression e is assigned the value B and all condition expressions (if any) controlling S are assigned proper values so that S can be executed. For the fault-free case, B is loaded into t. For the faulty case, the value of t is unknown (denoted as X). As a result, a good/bad value pair B/X is set up at t. The value pair is propagated to a PO. Suppose that a value pair B′/X is set up at the PO. During the hardware testing, the test vectors generated from this step are applied to the device under test (DUT). If the measured value of the PO of the DUT is equal to B′, the initial value of t is determined to be B. Whether the DUT is fault-free or faulty can be determined only after the test vectors that will be generated from Step 2 are applied to the DUT. If the measured value of the PO is not equal to B′, the DUT is determined to be faulty and the test vectors from Step 2 need not be applied to it.
2. The RHS expression e is assigned the value G and all condition expressions (if any) controlling S are assigned proper values so that S can be executed. For the fault-free case, the value G is loaded into t. For the faulty case, t holds its initial value which was determined to be B in Step 1. As a result, a good/bad value pair G/B is set up at t. This value is propagated to the same PO. Suppose that a value pair G′/B′ is set up at the PO. During the hardware testing, the test vectors generated from this step are applied to the DUT. If the measured value of the PO is

equal to G′, the DUT is determined to be fault-free. Otherwise, the DUT is determined to be faulty. Figure 15 schematically shows the test generation procedure.

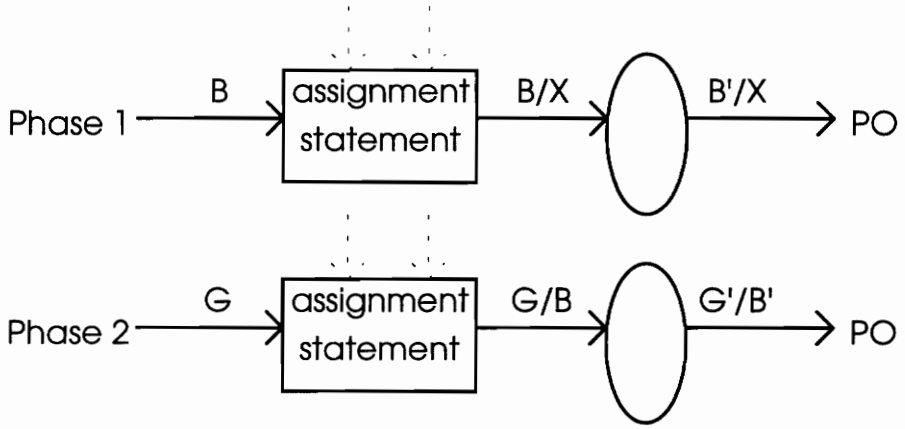


Figure 15. Two-phase Testing for Two-phase Activation

Example

```
process(C, D, E, F)
begin
s1:  F <= C and D;
s2:  Z <= E nor F;
end process;
```

Let us consider the BSO fault on the statement s1 in the above VHDL code. All signals used in the code are assumed to be two-bit vectors. Since s1 is the only assignment statement which has the target F in the code, the B-algorithm generates a test sequence for this fault using the concept of two-phase testing. The B-algorithm first chooses a good value G and a bad value B. Assume that G = 00 and B = 11 are chosen. The B-algorithm generates a test sequence using the following procedure:

1. The bad value 11 is assigned to the RHS expression of s1. This can be justified by assigning C = 11 and D = 11. For the fault-free case, 11 is loaded into F. For the faulty-case, the value of F is unknown. As a result, a value pair 11/XX is set up at F, which is propagated to Z as 00/XX by assigning E = 00. During the hardware testing, if the measured value of Z is equal to 00, the initial value of F is determined as 11. Otherwise, the DUT is determined to be faulty.
2. The good value 00 is assigned to the RHS expression of s1. This can be justified by assigning C = 00 or D = 00. Assume that C = 00 is chosen. For the fault-free case, 00 is loaded into F. For the faulty case, F holds its initial value 11 determined in Step 1. As a result, a value pair 00/11 is set up at F, which is propagated to Z as 11/00 by assigning E = 00. During the hardware testing, if the measured value of Z is equal to 11, the DUT is determined to be fault-free. Otherwise, the DUT is determined to be faulty. The following table shows the test sequence generated from Step 1 and Step 2:

time	C	D	E	Z
t1	11	11	00	00/XX
t2	00	x	00	11/00

5.7.2. Two-phase Testing for Two-phase Propagation

Suppose that a good/bad value pair is currently set up at a control expression and that the target of any assignment statement controlled by the control expression is not used in any other assignment statement in the model. Two cases can be considered depending on the existence of a good clause. If a good clause does not exist, i.e., a good statement does not exist, there is no method of loading a value into the target of any assignment statement controlled by the control expression when the

model is fault-free. Hence, the good value at the control expression cannot be propagated to a PO, which makes test generation impossible. This case is illustrated in Example 1. If a good clause exists, the good/bad value pair at the control expression can be propagated to a PO using the concept of two-phase testing. Let S be a good statement with a target t and a RHS expression e. The B-algorithm first chooses a new good value G′ and a new bad value B′ (G′ ≠ B′) for t and then performs the following procedure:

1. The RHS expression e is assigned the value B′ and other condition expressions (if any) controlling S are assigned proper values so that S can be executed when the model is fault-free. For the fault-free case, B′ is loaded into t. For the faulty case, the value of t is unknown (denoted as X). As a result, a good/bad value pair B′/X is set up at t. The value pair is propagated to a PO. Suppose that a value pair B″/X is set up at the PO. During the hardware testing, the test vectors generated from this step are applied to the DUT. If the measured value of the PO of the DUT is equal to B″, the initial value of t is determined to be B′. Whether the DUT is fault-free or faulty can be determined only after the test vectors that will be generated from Step 2 are applied to the DUT. If the measured value of the PO is not equal to B″, the DUT is determined to be faulty and the test vectors from Step 2 need not be applied to it.
2. The RHS expression e is assigned the value G′ and other condition expressions (if any) controlling S are assigned proper values so that S can be executed when the model is fault-free. For the fault-free case, the value G′ is loaded into t. For the faulty case, t holds its initial value which was determined to be B′ in Step 1. As a result, a good/bad value pair G′/B′ is set up at t. This value is propagated to the same PO. Suppose that a value pair G″/B″ is set up at the PO. During the hardware testing, the test vectors generated from this step are applied to the DUT. If the measured value of the PO is equal to G″, the DUT is determined to be fault-free.

Otherwise, the DUT is determined to be faulty. Figure 16 schematically shows the test generation procedure.

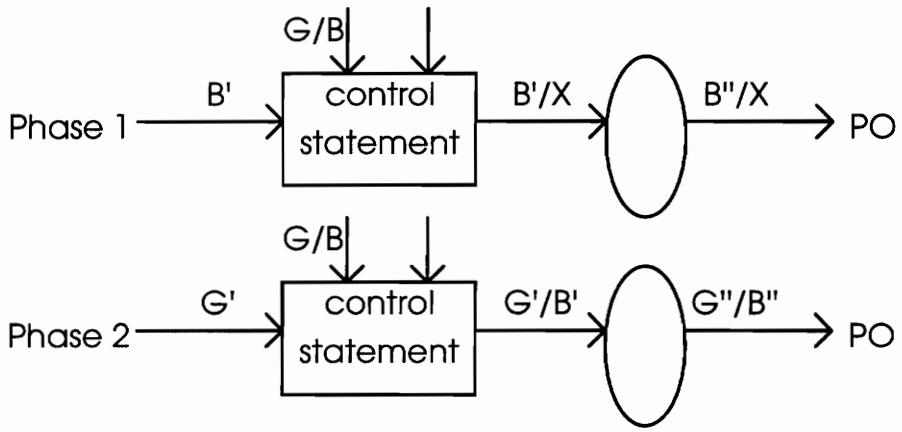


Figure 16. Two-phase Testing for Two-phase Propagation

Example 1

```
if (A = '1') then
  Z <= X;
end if;
```

The above *if* statement does not have an *else* clause. Suppose that a $\overline{D}$ (0/1) is currently set up at the condition expression (A = '1'). When the circuit is fault-free, the *else* clause (empty) is selected and thus no value can be loaded into Z. Since the propagation of the $\overline{D}$ is impossible, no test sequence can be generated in this situation.

Example 2

```
case DSEL(0 to 1) is
s1:  when "00" => Z1 <= I1(0 to 1);
s2:  when "01" => Z2 <= I2(0 to 1);
s3:  when "10" => Z3 <= I3(0 to 1);
s4:  when "11" => Z4 <= I4(0 to 1);
end case;
```

Suppose that a value pair 10/11 is currently set up at DSEL. The statement s3 is selected for the fault-free case and s4 is selected for the faulty case. The *good* statement s3 is chosen to propagate the value pair. The B-algorithm first chooses a new good value G and a new bad value B. Assume that G′ = 00 and B′ = 11 are chosen. The test sequence is generated using the following procedure:

1. The new bad value 11 is assigned to the RHS expression of s3, i.e., I3. For the fault-free case, 11 is loaded into Z3. For the faulty-case, the value of Z3 is unknown. As a result, a value pair 11/XX is set up at Z3. During the hardware testing, if the measured value of Z3 is equal to 11, the initial value of Z3 is determined as 11. Otherwise, the DUT is determined to be faulty.
2. The new good value 00 is assigned to I3. For the fault-free case, 00 is loaded into Z3. For the faulty case, Z3 holds its initial value 11 determined in Step 1. As a result, a value pair 00/11 is set up at Z3. During the hardware testing, if the measured value of Z3 is equal to 00, the DUT is determined to be fault-free. Otherwise, the DUT is determined to be faulty. The following table shows the test sequence generated from Step 1 and Step 2:

time	DSEL	I1	I2	I3	I4	Z3
t1	10/11	x	x	11	x	11/XX
t2	10/11	x	x	00	x	00/11

Recursive Two-phase Testing

Let I/J be a good/bad value pair created during a phase of a two-phase testing procedure. The value pair I/J is partially defined in the first test phase and fully defined in the second phase. If a good/bad value pair I/J has to be propagated to the control expression of a control statement which does not include any target which is used as a target in any other assignment statement, another two-phase testing is required for the propagation. In the first test phase of the second two-phase testing procedure, another partially defined good/bad value pair needs to be created and propagated to a primary output. The values assigned for the activation and the propagation need to be justified to primary inputs before the second test phase starts. In the second test phase, the same procedure needs to be performed.

Recursive two-phase testing causes two problems. First, it makes test generation too complex. Second, it makes the hardware testing inefficient because it requires multiple number of measurements at PO's. For these reasons, the B-algorithm does not allow any recursive two-phase testing. Due to this restriction, there may be cases when a test, which can be generated if recursive two-phase testing is allowed, cannot be generated. However, this situation does not occur frequently for two reasons: 1) a control statement, in most cases, includes two assignment statements with the same target (typical examples are multiplexers and registers with an asynchronous input.). 2) recursive two-phase testing is required only when a good/bad value pair is propagated through two or more control statements which do not include any target which is also used in some other place in the model.

5.8. Overall Test Generation Algorithm

5.8.1. The B-algorithm

The B-algorithm was outlined in Figure 11 (Section 5.1). It basically generates a test for a given fault by activating the fault by creating a good/bad value pair at the fault site, propagating the good/bad value pair to a PO, and justifying the values necessary for the activation and the propagation to PI's. In general, this procedure is performed only once to generate a test for a given fault. However, if two-phase testing is required, this procedure needs to be performed twice. In addition, for an MOP fault, a set of test sequences, not a single test sequence, needs to be generated. The number of test sequences to be generated increases with the number of bits of the arguments of MOP. Hence, after the activation of an MOP fault, the remaining part of the test generation procedure needs to be performed the same number of times as the number of test sequences to be generated.

Figure 17 shows the main function *B-alg* and the function *Multiple_test* which can also handle test generation for MOP faults and two-phase testing. *B-alg* consists of three parts: *the initialization phase*, *the first test phase*, and *the second test phase*. In the initialization phase, B-alg sets all signals to x, resets the *two-phase flag*, and (optionally) performs the circuit initialization using the function *Initialize*. The two-phase flag indicates that two-phase testing is required during test generation. The first test phase consists of functions *Activate* and *Prop_Just*. *Activate* creates a good/bad value pair by applying a proper activation rule depending on the type of a given fault under consideration. After the activation of the given fault, *B-alg* performs the remaining part of the algorithm depending on the type of the fault. For an MOP fault, it calls the function *Multiple_test*, which performs its remaining part the same number of times as the number of tests to be generated. In fact, the code inside the *for* loop of *Multiple_test* is exactly the same as the

remaining part of *B-alg*. For a BSA fault and a BSO fault, the remaining part of *B-alg* is performed only once. The good/bad value pair is propagated by the function *Prop_Just*. The two-phase flag is set if two-phase testing is required during the execution of the functions *Activate* and *Prop_Just*, which will be shown in Figures 18 and 19. If the first test phase is not successful, *B-alg* reports *Failure* and stops. If the first test phase is successful and two-phase testing is not required (i.e., the two-phase flag is not set), *B-alg* reports *Success* and stops. If the first test phase is successful but two-phase testing is required (i.e., the two-phase flag is set), *B-alg* performs the second test phase, which consists of the functions *Second_phase_activate* and *Prop_Just*. *Second_phase_activate* creates a new good/bad value pair assuming that the test sequence generated from the first test phase is applied to the circuit during the hardware testing and that the expected value is measured at the specified PO. The new good/bad value pair is propagated again by the function *Prop_Just*. If the second test phase is successful, *B-alg* reports *Success*. Otherwise, it reports *Failure*.

Activation

Activation is performed by the function *Activate* shown in Figure 18. *Activate* creates a good/bad value at the fault site by applying a proper activation rule to a given fault. For a BSA-v fault, it generates a value pair $\overline{v}/v$ by applying $\overline{v}$ to the signal bit. For a BSO fault, it calls the function *Two_phase_activate*, which activates the BSO fault in two steps: *preloading* and *transfer*. If the target of the assignment statement with the BSO fault is used in any other assignment statement in the model, activation is completed in the first test phase. As a result, a completely defined good/bad value pair G/B is set up at the fault site. If the target is not used anywhere in the model, only a bad value B is loaded in the first test phase. As a result, a partially defined value pair B/X is set up at the fault site. Since two-phase testing is required in this case, the two-phase flag is set. In addition,

the target name and the values G and B are stored in preparation for the second test phase. In the second test phase, the function *Second_phase_activate* first sets the target to the value B assuming that its initial value is B. It then loads the value G into the target. As a result, a value pair G/B is set up at the fault site. For an MOP fault, *Activate* calls the function *Htg*, which generates a set of tests local to the MOP using the corresponding heuristics discussed in Section 5.2. Each local test consists of an input value (local to the MOP) and a good/bad value pair. Each good/bad value pair is propagated to a PO and each input value is justified to PI's using the function *Multiple_test*.

Propagation and Justification

Propagation and justification are performed by the function *Prop_Just* shown in Figure 19. It calls the function *Imply_and_check* in the beginning. *Imply_and_check* computes all values that can be uniquely determined by implication, detects any conflict, and maintains the B-frontier and the J-frontier. The detailed operation of *Imply_and_check* will be discussed in the next section. After the execution of *Imply_and_check*, *Prop_Just* propagates a good/bad value pair at the fault site to a PO by applying a propagation rule to each entry in the B-frontier. If a good/bad value is not yet set up at a PO and the B-frontier is empty, it returns *Failure*. If two-phase testing is not required by the specific propagation rule, the propagation is completed in the first test phase. If two-phase testing is required, it first checks the status of the two-phase flag. If the two-phase flag is already set, it returns *Failure* because recursive two-phase testing is not allowed. Otherwise, it sets the two-phase flag and stores the target name, a new good value G, and a new bad value B in preparation for the second test phase. This whole procedure is recursively performed because *Prop_Just* continues to execute propagations until a good/bad value is set up at a PO.

Prop_Just also justifies the values necessary for the activation and the propagation to PI's by applying a justification rule to each entry in the J-frontier. If a good/bad value pair is set up at a PO and the J-frontier is empty during the execution of *Prop_Just*, it returns *Success*. If any entry in the J-frontier cannot be justified, it returns *Failure*. This procedure is also recursively performed because *Prop_Just* continues to execute justifications until PI's are reached.

5.8.2. Handling of the B-frontier, the J-frontier, and the A-queue

The B-algorithm in fact performs the three test generation operations (activation, propagation, and justification) by manipulating the three data structures: the B-frontier, the J-frontier, and the A-queue (assignment queue). An entry in the A-queue has the form (e, v', t, d), where e is an expression (including a signal), v' is the value to be assigned to e, t is the time period when v' is assigned to e, and d (direction)$\in$ {*forward*, *backward*}. The test generation procedure starts with activation, which creates a good/bad value pair at the fault site by assigning values to expressions using proper activation rules. The newly assigned values and the good/bad value pair are first inserted into the A-queue. The direction of the newly assigned values is backward, and that of the good/bad value pair is forward.

Figure 20 shows the function *Imply_and_check* which handles the data structures. As shown in Figure 19, *Imply_and_check* is called whenever an entry in the B-frontier and the J-frontier is processed. The value of each entry in the A-queue is first compared with the previously determined value, i.e., the value v' to be assigned to e at time t is compared with the current value v of e at time t. A conflict is detected if $v \neq$ x and $v' \neq v$ (it should be noted that all values are initialized to x). If a conflict is detected, *Imply_and_check* returns *Failure*. If no conflict is detected, the entry is further processed according to its direction (forward or backward) for implying values or

determining new entries to be added to the J-frontier and the B-frontier. If the direction is backward, *Imply_and_check* first calls the function *Backward_implication*, which attempts to determine input values of an element when its output value is known. If the direction is backward and the expression is a fanout branch, it also attempts to imply (*forward*) the value of the entry through other fanout branches connected to the fanout point. Hence, it also calls the function *Forward_implication*, which attempts to determine the output value of an element using its input values. If the direction is forward, *Imply_and_check* calls the function *Forward_implication*. If the direction is forward and the expression is a fanout stem, *Forward_implication* is executed for every fanout branch connected to it. After *Forward_implication* or *Backward_implication* is performed, any newly implied values are in turn inserted into the A-queue. This procedure continues until the A-queue is empty. Backward implication and forward implication are performed as follows:

Backward Implication

The backward implication procedure is similar to the justification procedure. If a value is assigned *backward* to an output of an element (an expression or a statement), the function *Backward_implication* attempts to imply the values of the inputs of the element.

If a value is assigned to an output of an expression, the values of its arguments are determined based on the operation involved in the expression. If one of its arguments is a virtual signal and a logic value can be implied for the virtual signal, the virtual signal together with its value and direction (backward) is newly added to the A-queue. This is in fact performed in the last step of the function *Imply_and_check*. This procedure continues until all arguments are simple signals and values are uniquely determined for the simple signals. If a value is assigned to a target signal of an

assignment statement, the forward implication procedure attempts to determine the values of its RHS expression and control expressions (if any) by which it is controlled. If an output of an element is assigned a value and some of its input values still cannot be uniquely determined, the element is inserted into the J-frontier.

Backward implication through an operation is one of the most difficult parts of the B-algorithm. For an operation which has an inverse function [51], its input values is determined by its output value and its inverse function. For example, if an operation ADD(A, B) currently has a value V, the value of B can be determined by computing its inverse function SUB(V, A). For an operation which does not have an inverse function, look-up tables are constructed for backward implication. To use look-up tables for backward implication costs significant memory space and CPU time.

Forward Implication

Forward implication is basically the same as logic simulation. If a value is assigned *forward* to an input of an element (an expression or a statement), the B-algorithm attempts to imply the value of the output of the element. If a value is assigned to an input of an (inner) expression, its output value is determined based on the operation involved in the expression. If a value is assigned to a control expression, the value of the RHS expression of each assignment statement controlled by it is transferred to the corresponding target. If a value is assigned to the RHS expression of an assignment statement which is not controlled by any control expression, the value is automatically transferred to its target. If the assignment statement is controlled by any control expression, the value is transferred to its target only when the control expressions are TRUE. If an input of an element is assigned a good/bad value pair and its output value still cannot be determined, the element is inserted into the B-frontier.

```
B-alg(fault_id)
{
  /* Initialization Phase */
  set all values to x;
  reset the two-phase flag;
  Initialize();
  /* The First Test Phase */
  Activate(fault_id);
  /* For an MOP fault, the remaining part is executed multiple times */
  if (the fault is an MOP) {
    Multiple_test();
    exit;
  }
  /* For a BSA fault and a BSO fault, the remaining part is executed only once */
  if (Prop_Just() = Success)
    if (the two-phase flag is not set) /* Two-phase testing is not required */
      report Success;
    else {  /* The Second Test Phase */
      set all values to x;
      Second_phase_activate();
      if (Prop_Just() = Success)
        report Success;
      else
        report Failure;
    }
  else /* failed in the first test phase */
    report Failure;
}

Multiple_test()
{
  for (each test local to the MOP) {
    if (Prop_Just() = Success)
      if (the two-phase flag is not set) /* Two-phase testing is not required */
        report Success;
      else {  /* The Second Test Phase */
        set all values to x;
        Second_phase_activate();
        if (Prop_Just() = Success) report Success;
        else report Failure;
       }
    else /* failed in the first test phase */
      report Failure;
  } /* for */
}
```

Figure 17. The main function B-alg and the function Multiple_test

```
Activate(fault_id)
{
  switch (fault_id) {
    case BSA : generate G/B;
    case BSO : Two_phase_activate(fault_id);
    case MOP : Htg(fault_id);
  }
}


Two_phase_activate(fault_id)
{
  if (there exists another assignment statement S' with the same target) {
    select a good value G and a bad value B;
    load B into the target using S';
    load G into the target using the statement with the BSO fault;
  }
  else /* there is no other assignment statement with the same target */ {
    select a good value G and a bad value B;
    load B into the target using the statement with the BSO fault;
    set the two-phase flag;
    store the target name and the values G and B;
  }
}


Second_phase_activate(fault_id)
{
  set the initial value of the target to B;
  load G into the target; /* As a result, a good/bad value G/B is set up at the target */
}


Htg(fault_id)
{
  generate a set of tests local to the MOP using the corresponding heuristics;
  store the set of input values local to the MOP;
  store the set of good/bad value pairs;
}
```

Figure 18. Functions for Activation

```
Prop_Just()
{
  /* Implication, Detection of Conflict, Maintenance of the J-frontier and the B-frontier */
  if (Imply_and_check () = Failure)
    return Failure;

  /* Propagation */
  if (a good/bad value pair is not set up at a PO) {
    if (B-frontier = ∅)
      return Failure;
    while (there exists an untried entry in the B-frontier) {
      select an untried entry from the B-frontier;
      apply a propagation rule to the entry;
      /* Handling of Two-phase Testing */
      if (two-phase testing is required by the propagation rule) {
        if (the two-phase flag is already set)
          return Failure; /* recursive two-phase testing is not allowed */
       set the two-phase flag;
       store the target name,  the new good value G, and the new bad value B;
      }
      if (Prop_Just() = Success)
        return Success;
    }
    return Failure; /* The B-frontier is empty */
  }

  /* A good/bad value is set up at a PO - Justification starts */
  if (J-frontier = ∅) return Success;
  select an entry from the J-frontier;
  while (there exists an unspecified input of the entry) {
    select an unspecified input of the entry;
    while (there exists an untried choice for the value of the input) {
      assign a value to the input using a justification rule;
      if (Prop_Just() = Success)
        return Success;
    }
  }
  return Failure;
}
```

Figure 19. The function Prop_Just

```
Imply_and_check()
{
  while (A-queue ≠ ∅) {
    select an entry (exp, val, time, dir) from the A-queue;
    if (conflict is detected)
      return Failure;
    if (dir = backward) {
      Backward_implication();
      if (exp is a fanout branch)
        for (each of other fanout branches connected to the fanout point)
          Forward_implication();
     }
    else /* dir = forward */
      if (exp is a fanout stem)
        for (each fanout branch connected to the fanout point)
          Forward_implication();
      else
        Forward_implication();
    insert newly determined values into the A-queue;
  }
}

Backward_implication()
{
  if (exp is an output of an expression)
    determine the values of arguments using the operation involved in the expression;
  else
    if (exp is a target signal of an assignment statement)
      determine the values of its RHS expression and control expressions;
  if (input values cannot be uniquely determined)
    add it to the J-frontier;
}

Forward_implication()
{
  if (exp is an input of an inner expression)
    determine the output value using the operation involved in the expression;
  else
    if (exp is a control expression)
      for (each assignment statement controlled by the control expression)
        determine the target value using the corresponding RHS expression;
    else /* exp is a RHS expression of an assignment statement */
      determine the target value depending on the values of control expressions;
  if (val is a good/bad value pair and the output value cannot be implied)
    add it to the B-frontier;
}
```

Figure 20. The function Imply_and_check

Maintenance of the A-queue

As mentioned before, each entry in the A-queue is assigned a time tag *t* which indicates that a given value needs to be assigned at a time period *t*. The entries in the A-queue are sorted on the time tags assigned to them. More than one entry can be assigned the same time tag if no conflict is detected among the values assigned to the entries. If a new entry is assigned an existing time period in the A-queue, a check is made to see if any conflict is detected among the values in the same time period. If a conflict is detected, the B-algorithm backtracks to assign a different time period. If a new entry needs to be inserted between existing time periods, existing time tags in the queue are renumbered.

As discussed in Section 5.3, a new time period is created whenever an attribute 'EVENT or 'STABLE is encountered. However, there are some restrictions on the creation of new time periods:

1. All time periods created in the initialization phase must precede the ones created in the first test phase. Similarly, all time periods created in the first test phase must precede the ones created in the second test phase.
2. As discussed in Section 5.3, all time periods created in the *preloading* phase during two-phase activation or two-phase propagation must precede the ones created in the *transfer* phase. No entry irrelevant to two-phase activation can be assigned the time periods created during two-phase activation. Similarly, no entry irrelevant to two-phase propagation can be assigned the time periods created during two-phase propagation.

Using the above rules, the B-algorithm can reduce possible conflicts on the assignment of time periods in advance.

Maintenance of the B-frontier and the J-frontier

The most difficult problem involved in the B-frontier and the J-frontier is to determine the order in which an entry is selected from them. As in gate level test generation [42], controllability and observability are measured to determine the order in which an entry is selected.

Definition: The controllability of an expression is the ability to set the expression (including a signal) to a logic value using PI's.

Definition: The observability of an expression is the ability to observe a good/bad value pair at the expression (including a signal) using PO's.

The B-algorithm measures the controllability as the *distance* from PI's to a given expression and the observability as the *distance* from PO's to a given expression. Since each entry in the J-frontier needs to be justified until PI's are reached, an entry with the lowest controllability is selected from the J-frontier. Similarly, since each entry in the B-frontier needs to be propagated until a PO is reached, an entry with the lowest observability is selected from the B-frontier.

The B-algorithm measures the *distance* by adding the numbers (weights) assigned to VHDL constructs between a given expression and a PI (for controllability) or a PO (for observability). This is performed during the preprocessing of the VHDL model. Different weights are assigned for different types of VHDL constructs. This is because propagation or justification through some constructs is much more difficult than that through other constructs. In general, a VHDL statement can be represented as shown in Figure 21.

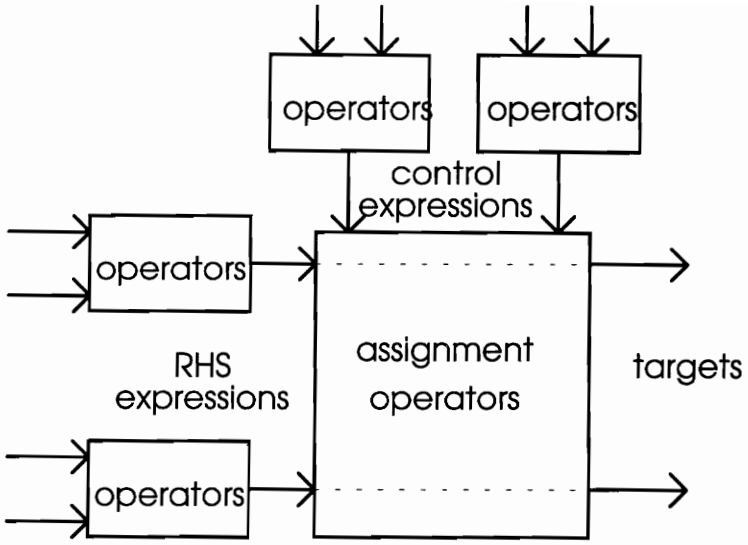


Figure 21. A general form of a VHDL statement

There are three different types of paths for propagation or justification: 1) through operators, 2) through an assignment operator (from a RHS expression to the corresponding target), 3) from a control expression to a target. The easiest path is an assignment operator when it is not controlled by any control expression. The most difficult path is the one from a control expression to a target. The easiness of propagation or justification through operators is in general in the following order: a logic operator, a relational operator, and an arithmetic operator. Considering this, the B-algorithm assigns weights to VHDL constructs in an increasing order as the following table. Although the weights may not be optimal, they help to select a proper entry from the B-frontier or the J-frontier.

Type of Constructs	weight
an assignment operator not controlled by control expressions	1
a logic operator	2
a relational operator	3
an arithmetic operator	4
an assignment operator controlled by control expressions	5
a path from a control expression to a target	6

5.8.3. Test Generation Examples

Example 1 - Test Generation for a BSA fault

Let us again consider the VHDL model shown in Figure 12 (Section 5.3) which consists of a multiplexer, a register, and a buffer. The test generation procedure for generating a test sequence for a BSA-0 fault at the condition expression (CLK = '1' and CLK'STABLE) is shown below.

Activation - The BSA-0 fault can be activated by assigning a logic 1 to the condition expression. As a result, a good/bad value pair 1/0 is set up at the condition expression.

Propagation - Two-phase propagation is required to propagate the good/bad value pair 1/0 at the condition expression to the target REG. First, two assignment statements s5 and s6 which have REG as a target are selected. The RHS expression of s5 is a constant value (0000) while that of s6 is a simple signal DATA. Let us select a new good value G' and a new bad value B'. According to the propagation rules shown in Section 5.2, the RHS expression (0000) of s5 is used as B'. Fifteen different values (0001 - 1111) can be used as G'. Let us arbitrarily select G' = 1111. If this selection is not successful, the test generation procedure backtracks to this selection point. In the preloading phase, assign a logic 1 to the condition expression (CLEAR = '0') so that the value B' can be transferred to REG. In the transfer phase, assign the value G' to DATA, a logic 1 to the condition expression (CLK = '1' and CLK'STABLE), and a logic 0 to the condition expression (CLEAR = '0') so that the value G' can be transferred to REG. As a result, a new good/bad value pair 1111/0000 is set up at REG. The new good/bad value pair can be propagated to DO by

assigning a logic 1 at the condition expression (EN = '1'). As a result, a good/bad value pair 1111/0000 is set up at the primary output DO.

Justification - The values assigned during the preloading phase are justified first. To justify a logic 1 at (CLEAR = '0'), CLEAR = 0 is assigned. Since CLEAR is a PI, no further justification is necessary. The values assigned during the transfer phase are justified next. There are two ways of justifying 1111 at DATA: 1) assign SEL = 1 and D1 = 1111 and 2) assign SEL = 0 and D2 = 1111. Let us arbitrarily choose the former one. Since SEL and D1 are PI's, no further justification is necessary. To justify a logic 0 at (CLEAR = '0'), CLEAR = 1 is assigned. Finally, the value used for propagating the good/bad value pair through the statement s7 is justified. To justify a logic 1 at (EN = '1'), EN = 1 is assigned. The following table shows the test sequence generated from the above procedure.

Time	D1	D2	SEL	CLEAR	CLK	EN	REG	DO
t_1	x	x	x	0	x	x	0000	-
t_2	1111	x	1	1	R	1	1111/0000	1111/0000

Example 2 - Test Generation for an MOP fault

The following VHDL model consists of three processes each of which represents a 3-bit adder. Let us consider an MOP fault (ADD -> SUB) in the statement s2.

```
entity ADD_TREE (
      A, B, C, D: in bit_vector(0 to 2);
      G: out bit_vector(0 to 2)) is
end ADD_TREE;
```

```
architecture DATA_FLOW of ADD_TREE is
  signal E, F: bit_vector(0 to 2);
begin
    p1: process(A, B)
    begin
s1    E <= ADD(A, B);
    end process p1;

    p2: process(C, E)
    begin
s2    F <= ADD(C, E)
    end process p2;

    p3: process(D, F)
    begin
s3    G <= ADD(D, F)
    end process p3;
end ADD_TREE;
```

Activation - Using the heuristic rules in Section 5.2.1, the following eleven vectors local to ADD(C, E) are generated perturbing it to SUB(C, E) and SUB(E, C). The first six vectors are generated from the perturbation ADD(C, E) -> SUB(C, E) and the last five vectors are generated from ADD(C, E) -> SUB(E, C). For each vector in the table, a good/bad value pair is set up at F.

C	E	C	E
XXX	XX1	000	001
		000	111
		111	001
		111	111
XXX	X1X	000	010
		111	010
XX1	XXX	001	000
		111	000
		001	111
X1X	XXX	010	000
		010	111

Propagation - The good/bad value pair value F can be propagated to G by assigning any value to D. Let us select D = 000.

Justification - There are many combinations of values of A and B which can justify each value of E in the above table. The easiest way is to set an argument of ADD(A, B) to all 0's and the other argument to the value of E. The following table shows the test vectors generated from this procedure.

Time	A	B	C	D	E	G (= F)	Fault type
t1	000	001	000	000	001	001/111	SUB(C, E)
t2	000	111	000	000	111	111/001	
t3	000	001	111	000	001	000/110	
t4	000	111	111	000	111	110/000	
t5	000	010	000	000	010	010/110	
t6	000	010	111	000	010	001/101	
t7	000	000	001	000	000	001/111	SUB(E, C)
t8	000	000	111	000	000	111/001	
t9	000	111	001	000	111	000/110	
t10	000	000	010	000	000	010/110	
t11	000	111	010	000	111	001/101	

Example 3 - Test Generation for A BSO Fault

Figure 22 shows a VHDL model of a 2-bit controlled counter with an asynchronous reset input (active HIGH). It counts up or down until a limit (LIM) is reached. A 2-bit input CON is decoded into a 4-bit register CONSIG on a rising edge of an input STRB. Each bit of CONSIG is used for controlling the counter as shown in the table below. A 2-bit input DATA is loaded into a 2-bit register LIM on a falling edge of STRB when CONSIG(0) = 1. The 2-bit register LIM is used for setting the limit for counting. The counter is reset with CLEAR = 1. It increments or decrements on a rising edge of a clock signal CLK.

CON	CONSIG	Function
00	1000	Load LIM
01	0100	Reset
10	0010	Count Up
11	0001	Count Down

Let us consider a BSO fault on the assignment statement s15, which indicates that the increment function of the counter doesn't work properly while other three functions work well.

Activation - Two-phase activation is required for activating the BSO fault. First, another statement s11 with the same target (COUNT) is selected. Second, a good value G and a bad value B are selected. Since the RHS expression of s11 is a constant value, the value B is automatically set to 00. Let G = 10. In the preloading phase, the control expression of s10 is set to TRUE so that the value B can be loaded into the target using s10. In the transfer phase, the RHS expression of s15 is set to G (10), and the control expression of s10, s12, s13, and s14 are set to FALSE, TRUE, TRUE, and TRUE, respectively. The time period created during the transfer phase is different from the one created during the preloading phase. As a result, a good/bad value pair 10/00 is set up at the target COUNT.

Propagation - Since COUNT is a PO, no more propagation is necessary.

Justification - Logic values assigned during the preloading phase are justified first. To justify the value (TRUE) at the control expression of s10, CONSIG(1) = 1 is assigned. To justify this value, CON = 01 and STRB = R (a rising edge) are assigned. Since CON and STRB are PI's, no more justification is necessary. The values assigned during the transfer phase are justified next. To justify the value (10) of the RHS expression of s15, the micro-operation ADD needs to be executed twice because the current value of COUNT is 00. Hence, all control expressions are set to the

specified values (s10: FALSE, s12: TRUE, s13: TRUE, and s14: TRUE) for two time periods. As discussed in Section 5.1 (Figure 8), all non-clock signals should be stable before a clock signal is set. To justify the value of the control expression of s13, the value of LIM should be assigned a value not equal to the current value (00) of COUNT. Three values 01, 10, and 11 satisfying this condition can be assigned to LIM. Let us arbitrarily assign 11 to LIM (if this choice is not successful, the test generation procedure backtracks to this point and chooses another value.). This value is justified by assigning DATA = 11, CONSIG(0) = 1, and STRB = F (a falling edge). Since DATA and STRB are PI's, no more justification is necessary for the values of DATA and STRB. The value (1) of CONSIG(0) is justified by assigning CON = 00 and STRB = R. To justify the values of the control expressions of s10 and s14, CONSIG(1) = 0 and CONSIG(2) = 1 are assigned. The value (X01X) of CONSIG is in turn justified by assigning CON = 10 and STRB = R. After all other conditions are set up, the clock signal CLK is set to R for two consecutive time periods to change the current value (00) of COUNT to the good value G (10). The following table shows the test sequence generated from the above procedure.

Time	STRB	CON	DATA	CLK	CONSIG	LIM	COUNT
t1	R	01	x	x	0100	-	00
t2	R	00	x	x	1000	-	00
t3	F	x	11	x	1000	11	00
t4	R	10	x	x	0010	11	00
t5	x	x	x	R	0010	11	01/00
t6	x	x	x	R	0010	11	10/00

The values of internal signals CONSIG and LIM as well as PI's and PO's are included in the table to show the signal flows during test generation. The test sequence consists of six time periods. It should be noted that the number of time periods can be reduced to five if the good value G is set to 01 instead of 10 during the activation of the BSO fault.

```
entity CONTROLLED_CTR (
        CLK, STRB: in BIT;
        CON, DATA: in BIT_VECTOR(0 to 1);
        COUNT: inout BIT_VECTOR(0 to 1)) is
end CONTROLLED_CTR;

architecture DATA_FLOW of CONTROLLED_CTR is
  signal LIM: BIT_VECTOR(0 to 1);
  signal CONSIG: BIT_VECTOR(0 to 3);
begin
  DECODE:
  process (STRB)
  begin
s1    if (STRB = '1' and STRB'EVENT) then
s2      case CON is
s3        when "00" => CONSIG <= "1000";
s4        when "01" => CONSIG <= "0100";
s5        when "10" => CONSIG <= "0010";
s6        when "11" => CONSIG <= "0001";
        end case;
      end if;
  end process DECODE;
  LOAD_LIMIT:
  process (STRB)
  begin
s7    if (STRB = '0' and STRB'EVENT) then
s8      if (CONSIG(0) = '1')) then
s9        LIM <= DATA;
        end if;
      end if;
  end process LOAD_LIMIT;
  COUNTER:
  process (CLK, CONSIG(1))
  begin
s10   if (CONSIG(1) = '1') then
s11     COUNT <= "00";
s12   elsif (CLK = '1' and CLK'EVENT) then
s13     if (COUNT =/ LIM) then
s14       if (CONSIG(2) = '1') then
s15         COUNT <= ADD(COUNT, "01");
s16       elsif (CONSIG(3) = '1') then
s17         COUNT <= SUB(COUNT, "01");
          end if;
        end if;
      end if;
  end process COUNTER;
end DATA_FLOW;
```

Figure 22. A VHDL model of the 2-bit Controlled Counter

5.9. Summary

A new approach to behavioral test generation has been presented. The new approach, called the B-algorithm, generates tests by performing three test generation operations (activation, propagation, and justification). The rules for the test generation operations have been defined adopting the concepts of two-phase activation and two-phase propagation. Special test generation rules for four bus resolution functions (BRF's) have also been defined. The difference between simulation semantics and test generation semantics have been addressed, and a method of efficiently assigning time periods have been discussed. As special cases, the assignment of time periods for two-phase activation and two-phase propagation has been discussed. A method of performing multiple propagation in the presence of reconvergent fanout has been discussed. A method of recognizing a feedback loop and initializing a feedback signal has been discussed. The concept of two-phase testing has been introduced and formally incorporated into the B-algorithm. A method of systematically performing the test generation operations by manipulating three data structures (the B-frontier, the J-frontier, and the A-queue) has been presented. The overall test generation algorithm has been presented with test generation examples.

Chapter 6. Further Research Area

6.1. Implementation of the B-algorithm

Since the old behavioral test generation algorithm [27,28,30-34] is based on AND/OR goal trees and rule databases, it was implemented using Prolog which has built-in backtracking and rule inference mechanisms. However, Prolog lacks flexible control constructs and bit manipulation constructs which are essential for implementing an efficient test generation algorithm. The B-algorithm has been constructed without the concepts of AND/OR goal trees and rule databases such that it can be easily implemented using a general-purpose high level language such as C or C++. Two advantages can be taken by implementing the B-algorithm using C or C++ instead of Prolog. First, the speed of the implemented test generator can be improved using more flexible control constructs and assembly-language style bit manipulation constructs of C or C++. Second, since the test generator can use data structures or intermediate forms commonly used by commercial VHDL tools, the test generator can be easily integrated into a comprehensive CAD system.

Figure 23 shows a possible test generation system for the B-algorithm. Let us call it the BTG2 (Behavioral Test Generator 2) for convenience. BTG2 consists of the preprocessor and the B-algorithm. The preprocessor consists of the translator, the knowledge extractor, and the fault list generator. The translator converts a VHDL model into data structures which can be directly used by the B-algorithm. A commercial tool such as VTIP [53] can be used for this purpose. From the data structures, the knowledge extractor generates test generation information such as

controllability and observability and topological paths, and the fault list generator generates a comprehensive list of behavioral faults using the method discussed in Chapter 4.

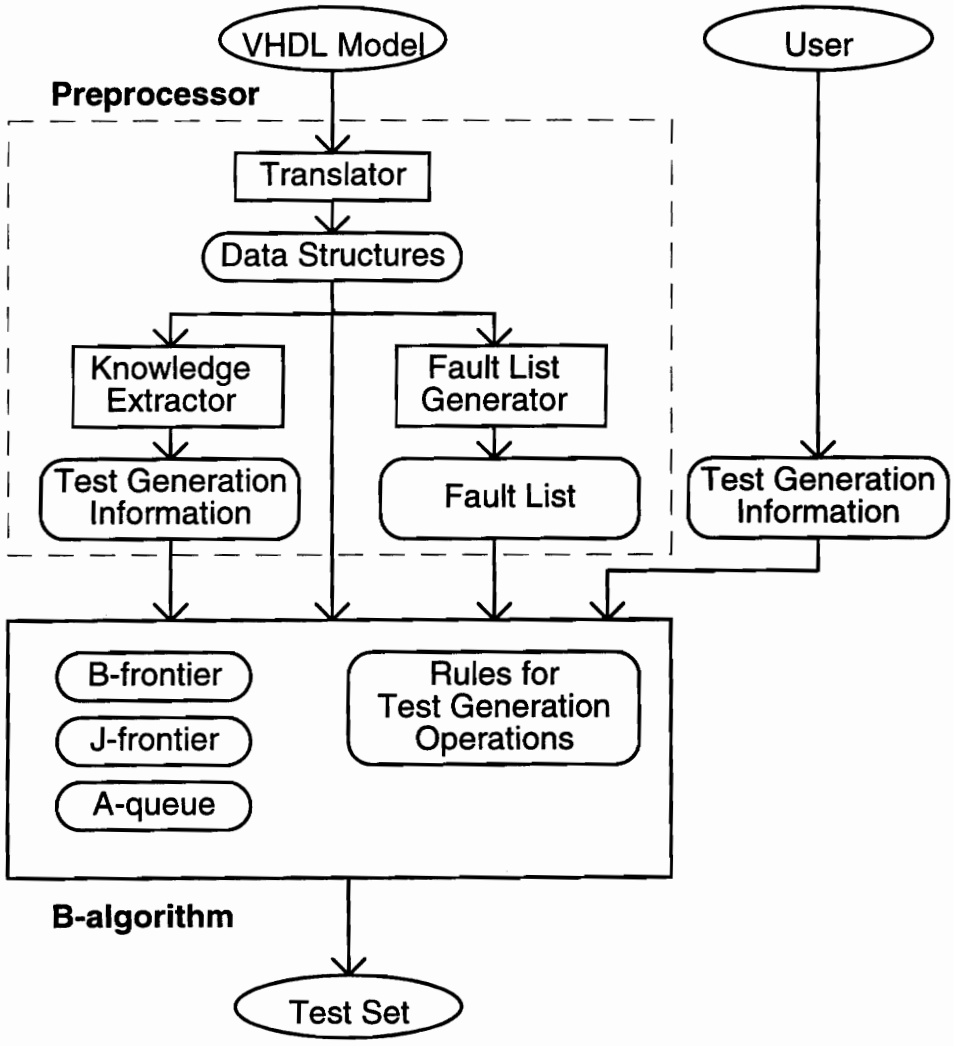


Figure 23. The Behavioral Test Generator 2 (BTG2)

The B-algorithm combines the basic test generator and the heuristic test generator shown in Figure 9. It keeps the rules for activation, propagation, and justification, and maintains the data structures the B-frontier, the J-frontier, and the A-queue. It can directly access the data structures and the test generation information generated by the preprocessor. It also collects test generation information, the types of bus resolution functions and the information (values and names) about internal signals to be initialized, directly from the user. BTG2 generates a test sequence for each behavioral fault in the fault list using the data structures and the test generation information. In BTG, each test sequence is generated in a separate file called a waveform file. However, this approach becomes impractical if thousands of test sequences are generated from a VHDL model. Therefore, BTG2 should be able to generate test sequences into a single waveform file.

6.2. Development of A Behavioral Fault Simulator

The B-algorithm generates a test sequence for each fault in the fault list. Hence, for n behavioral faults, the B-algorithm has to perform the test generation procedure n times. Although the number of behavioral faults is in general much smaller than that of gate level faults, the whole test generation procedure could be expensive for large n. One method of reducing CPU time required for test generation is to use a fault simulator during test generation. Whenever a test sequence is generated for a fault, the fault simulator determines whether the test sequence can detect any of the remaining faults in the fault list, and removes any faults that can be detected from the fault list. This procedure continues until the fault list is empty. The CPU time required for the whole test generation procedure can be significantly reduced using this procedure.

Ward and Armstrong [60] implemented a behavioral fault simulator which generates n faulty VHDL models from a fault-free VHDL model and a list of n behavioral faults. The behavioral fault simulator creates a faulty VHDL model corresponding to a behavioral fault by injecting the behavioral fault into the fault-free VHDL model. Once a faulty model is created, a given test sequence is applied to both the fault-free model and the faulty model, and the output values of the two models are compared to each other. If the output values are different, the behavioral fault injected into the faulty model is determined to be detected by the test sequence. This procedure needs to be applied to each faulty model created.

Ward and Armstrong's behavioral fault simulator injects faults into the source code of a fault-free VHDL model. Hence, each faulty model generated from the fault-free model needs to be stored and analyzed before it is used for fault simulation. This causes a big overhead for fault simulation if n is very large. This problem can be solved if faults are directly injected into the data structures (see Figure 23) translated from the VHDL source code. To achieve this, the fault simulator should be able to directly handle the data structures. Using this method, the behavioral fault simulator can perform fault simulation without keeping n faulty models.

Figure 24 shows a possible behavioral ATPG (automatic test pattern generation) system which combines the BTG2 (Figure 23) and a behavioral fault simulator (BFS). The user interaction part of the BTG2 is eliminated from the figure for convenience. Whenever a test sequence is generated from BTG2, BFS directly applies it to the data structures where faults are injected and performs fault simulation. Efficient fault simulation methods such as parallel fault simulation, deductive fault simulation, and concurrent fault simulation have been developed for gate level simulation [41]. Whether these methods can be applied to behavioral fault simulation has not been reported yet and should be studied for further improving a behavioral fault simulator.

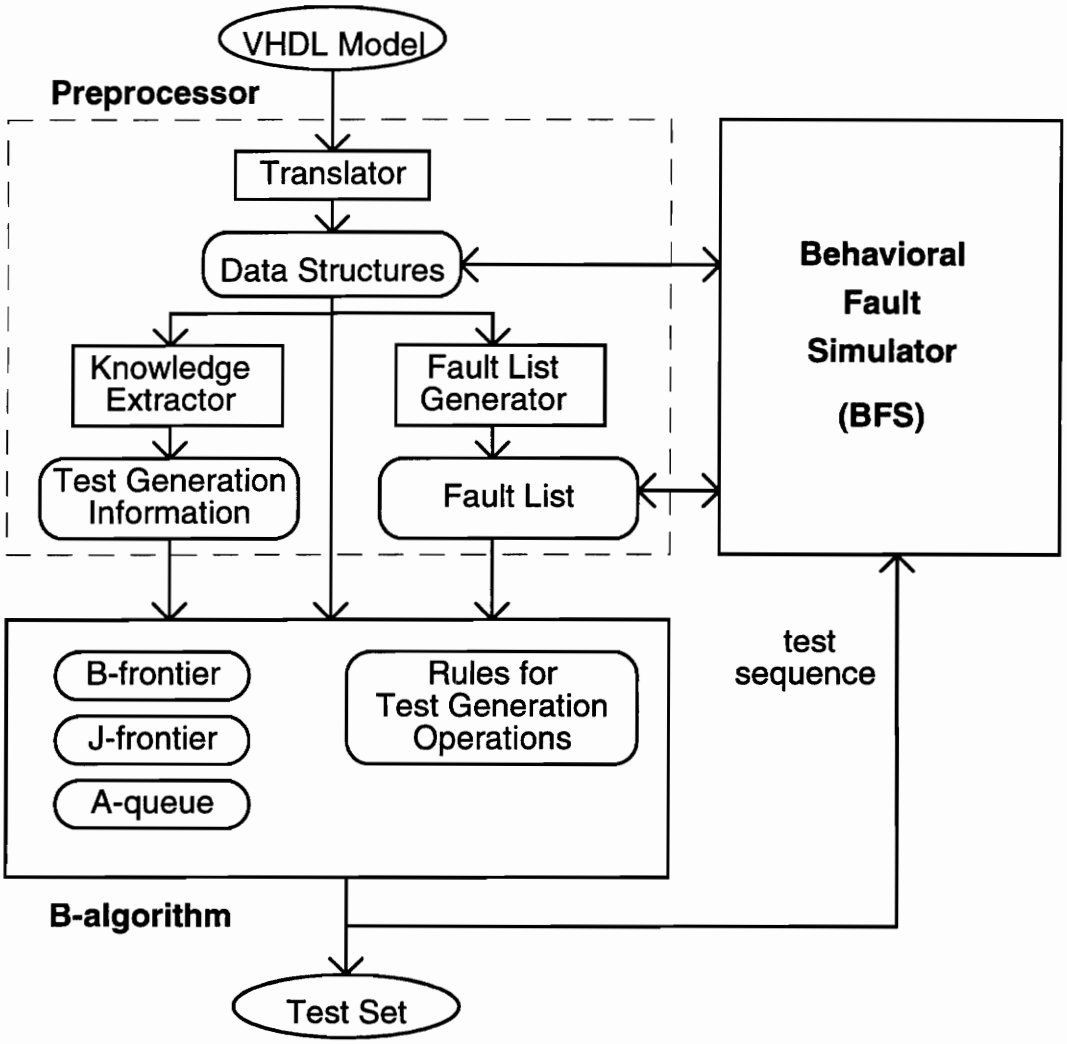


Figure 24. A Behavioral ATPG System

Chapter 7. Conclusions

A formal model for behavioral test generation has been presented. A behavioral VHDL model of equivalent process statements and a new behavioral fault model with a fewer number of fault types improve the efficiency of test generation by reducing the size of the domain searched during the test generation procedure. A behavioral test generation algorithm, called the B-algorithm, which generates tests using the behavioral VHDL model and the new behavioral fault model has been presented. In contrast to the previous approaches [27,28,30-34] where goals or goal trees are used for representing and solving test generation problems, the B-algorithm systematically performs the three basic test generation operations using the three data structures (B-frontier, J-frontier, and A-queue). Rules for the test generation operations have formally been defined using the concepts of two-phase activation and two-phase propagation. A method of efficiently assigning time periods without being affected by simulation semantics has been proposed, and a method of handling bus resolution functions, reconvergent fanout, and feedback loops during test generation has been discussed.

The B-algorithm has two unique features. First, the B-algorithm can generate tests for BSO faults, which can detect gate-level transition faults as well as gate level stuck-at faults. Second, it can generate tests for some types of faults, which without the concept of two-phase testing cannot be detected. The B-algorithm is suitable for being implemented using a general-purpose high level language such as C or C++, and hence can improve the test generation speed compared to the previous behavioral test generation algorithm [27,28,30-34] implemented using Prolog.

Bibliography

1. J. P. Roth, "Diagnosis of Automata Failures: A Calculus and a Method", IBM J. Res. Develop., vol. 10, pp. 278-291, July, 1964.

2. P. Goel, "An Implicit Enumeration Algorithm To Generate Tests For Combinational Logic Circuits", IEEE Trans. on Computers, vol. C-30, pp. 215-222, March, 1981.

3. H. Fujiwara and T. Shimono, "On the Acceleration of Test Generation Algorithms", IEEE Trans. on Computers, vol. C-32, pp. 1137-1144, Dec. 1983.

4. M. H. Schulz, E. Trischler, and T. M. Sarfert, "SOCRATES: A Highly Efficient Automatic Test Pattern Generation System", IEEE Trans. on Computer-Aided Design, vol. 7, no. 1, Jan., 1988.

5. G. R. Putzolu and J. P. Roth, "A Heuristic Algorithm for The Testing of Asynchronous Circuits", IEEE Trans. on Computers", vol. C-20, no. 6, pp. 639-647, June, 1971.

6. P. Muth, "A Nine-valued Circuit Model for Test Generation", IEEE Trans. on Computers, vol. C-25, no. 6, pp. 630-636, June, 1976.

7. R. Marlett, "EBT: A Comprehensive Test Generation Technique for Highly Sequential Circuits", 15th Design Automation Conference, pp. 335-339, June, 1978.

8. R. Marlett, "An Efficient Test Generation System for Sequential Circuits", 23rd Design Automation Conference, pp. 250-256, June, 1986.

9. S. Mallela and S. Wu, "A Sequential Circuit Test Generation System", IEEE International Test Conference, pp. 57-61, Nov., 1985.

10. W.-T. Cheng, "Split Circuit Model for Test Generation", 25th Design Automation Conference, pp. 96-101, June, 1988.

11. W.-T. Cheng, "The Back Algorithm for Sequential Test Generation", ICCD, pp. 66-69, Oct., 1988.

12. W.-T. Cheng, T. J. Chakraborty, "Gentest - An Automatic Test-Generation System for Sequential Circuits", vol. 22, no. 4, Computer, pp. 43-49, April, 1989.

13. H.-K. T. Ma, S. Devadas, A. R. Newton, and A. Sangiovanni-Vincentelli, "Test Generation for Sequential Ciruits", IEEE Trans. Compter-Aided Design, vol. 7, no. 10, pp. 1081-1093, Oct., 1988.

14. A. Ghosh, S. Devadas, and A. R. Newton, "Test Generation and Verification for Highly Sequential Circuits", IEEE Trans. on Computers, vol. 10, no. 5, pp. 652-667, May, 1991.

15. T. Niermann and J. H. Patel, "HITEC: A Test Generation Package For Sequential Circuits", Europian Design Automation Conference, Feb., 1991.

16. T. P. Kelsey, K. K. Saluja, and S. Y. Lee, "An Efficient Algorithm for Sequential Circuit Test Generation", IEEE Trans. on Computers, vol. 42, no. 11, pp. 1361-1371, November, 1993.

17. J. R. Armstrong, "Chip-level Modeling of LSI devices", IEEE Trans. on Computer-Aided Design, vol. CAD-3, pp. 288-297, Oct., 1984.

18. J. R. Armstrong, "Chip Level Modeling with HDLs", IEEE Design and Test of Computers, vol. 5, no. 1, pp. 8-18, Feb., 1988.

19. Y. H. Levendel and P. R. Menon, "Test Generation Algorithms for Computer Hardware Description Languages", IEEE Trans. on Computers, vol. C-31, pp. 577-588, July, 1982.

20. S. M. Thatte and J. A. Abraham, "Test Generation for Microprocessors", IEEE Trans. on Computers, vol. C-29, pp. 429-441, June, 1980.

21. D. Brahme and J. A. Abraham, "Functional Testing of Microprocessors", IEEE Trans. on Computers, vol. C-33, pp. 475-485, June, 1984.

22. S. Y. H. Su and Y. Hsieh, "Testing Functional Faults in Digital Systems Described by Register Transfer Language", IEEE International Test Conf., pp. 447-457, Oct. 1981.

23. T. Lin and S. Y. H. Su, "Functional Test Generation of Digital LSI/VLSI Systems Using Machine Symbolic Execution Techniques", IEEE International Test Conf., pp. 660-668, Oct., 1984.

24. T. Lin and S. Y. H. Su, "The S-Algorithm: A Promising Solution for Systematic Functional Test Generation", IEEE Trans. on Computer-Aided Design, vol. CAD-4, pp. 250-263, July, 1985.

25. T. J. Chakraborty and S. Ghosh, "On Behavior Fault Modeling for Combinational Digital Designs", IEEE International Test Conf., pp.593-600, Sep. 1988.

26. S. Ghosh and T. J. Chakraborty, "On Behavior Fault Modeling for Digital Designs", Journal of Electronic Testing: Theory and Applications, 2, pp. 135-151, 1991.

27. D. S. Barclay and J. R. Armstrong, "A Heuristic Chip-Level Test Generation Algorithm", 23rd Design Automation Conference, pp. 257-262, June, 1986.

28. D. S. Barclay, An Automatic Test Generation Method For Chip-level Circuit Descriptions, Master's Thesis, Dept. of Electrical Engr., Virginia Polytechnic Institute and State University, Feb.., 1987.

29. IEEE Standard VHDL Language Reference Manual, IEEE, Inc., March, 1988.

30. M. D. O'Neill, Improved Chip-level Test Generation Algorithm, Master's Thesis, Dept. of Electrical Engr., Virginia Polytechnic Institute and State University, Jan., 1988.

31. M. D. O'Neill, D. D. Jani, C. H. Cho, and J. R. Armstrong, "BTG: A Behavioral Test Generator", 9th International Symp. on CHDLs and Their Applications, pp. 347-361, June, 1989.

32. D. D. Jani, An Efficient Test Generation Algorithm For Behavioral Descriptions of Digital Devices, Master's Thesis, Dept. of Electrical Engr., Virginia Polytechnic Institute and State University, Dec., 1988.

33. F.-S. Lam, Test generation for Behavioral Models with Reconvergent Fanouts and Feedback, Master's Thesis, Dept. of Electrical Engr., Virginia Polytechnic Institute and State University, Sep., 1989.

34. G. Baweja, Gate Level Coverage of A Behavioral Test Generator, Master's Thesis, Dept. of Electrical Engr., Virginia Polytechnic Institute and State University, Mar., 1993.

35. W. F. Clocksin and C. S. Mellish, Programming in Prolog, 3rd Ed., Springer-Verlag, 1987.

36. M. D. Friedman, "Diagnosis of Short-Circuit Faults in Combinational Logic Circuits", IEEE Trans. on Computers, vol. C-23, no. 7, pp. 746-752, July, 1974.

37. M. Abramovici and P. R. Menon, "A Practical Approach to Fault Simulation and Test Generation for Bridging Faults", IEEE Trans. on Computers, vol. C-34, no. 7, pp.658-663, July, 1985.

38. P. Heish, "Delay Test Generation", 14th Design Automation Conference, pp. 486-491, 1977.

39. G. L. Smith, "Model for Dealy Faults Based on Paths", IEEE International Test Conference, pp. 342-349, 1985.

40. J. A. Waicukauski and E. Lindbloom, "Transition Fault Simulation by Parallel Pattern Single Fault Propagation, IEEE International Test Conference, pp. 542-549, Sep., 1986.

41. M. Abramovici, M. A. Breuer, and A. D. Friedman, Digital Systems Testing and Testable Design, Computer Science Press, 1990.

42. L.H. Goldstein and E. L. Thigpen, "SCOAP: Sandia Controllability/Observability Analysis Program", 17th Design Automation Conference, pp. 190-196, June, 1980.

43. DesignWare Databook, Version 3.0, SYNOPSYS, Dec., 1992.

44 J. R. Armstrong and F. G. Gray, Structured Logic Design With VHDL, PTR Prentice Hall Inc., Englewood Cliffs, NJ, 1993.

45 D. D. Hill, ADLIB Users Manual, Computer Systems Lab., Stanford University, CA, 1979.

46 Verilog-XL Reference Manual, Cadence Design Systems, Inc., 1991.

47. VHDL Compiler Reference Manual, Version 3.0, SYNOPSYS, Nov., 1992.

48. J. R. Armstrong, Chip Level Modeling with VHDL, Prentice Hall, Inc., Aug., 1988.

49 J. P. Shen and F. J. Ferguson, "Inductive Fault Analysis of MOS Integrated Circuits", IEEE Design and Test of Computers, vol. 2, no.2, pp. 13-26, Dec., 1985.

50 F. J. Ferguson and J. P. Shen, "Extraction and Simulation of Realistic CMOS Faults Using Inductive Fault Analysis", IEEE International Test Conference, pp. 475-484, 1988.

51. H. S. Stone, Discrete Mathematical Structures and Their Applications, Science Research Associates, Inc., 1973.

52. C. Chao and F. G. Gray, "Micro-operation Perturbations in Chip Level Fault Modeling", 25th Design Automation Conference, pp. 579-582, June, 1988.

53. VHDL Tool Integration Platform Installation Guide, CAD Language Systems, Inc., Feb., 1993.

54. U. Glaser, U. Hubner, and H. T. Vierhaus, "Mixed Level Hierarchical Test Generation for Transition Faults and Overcurrent Related Defects", IEEE International Test Conference, pp. 21-29, 1992.

55. O. H. Ibarra and S. Sahni, "Polynomially Complete Fault Detection Problems", IEEE Trans. on Computers, vol. C-24, no. 3, pp. 242-249, March, 1975.

56. C. W. Cha, W. E. Donath, and F. Ozguner, "9-V Algorithm for Test Pattern Generation of Combinational Digital Circuits", IEEE Trans. on Computers, vol. C-27, no. 3, pp. 193-200, March, 1978.

57. C. H. Cho, "A Chip Level Fault Coverage Experiment", Research Report, Dept. of Electrical Engr., Virginia Polytechnic Institute and State University, Aug., 1986.

58. C. H. Cho and J. R. Armstrong, "Heuristic Test Generation for Chip Level Circuit Descriptions", IEEE VLSI Test Workshop, March, 1988.

59. C. H. Cho and J. R. Armstrong, "VHDL Semantics for Behavioral Test Generation", 10th International Symp. on CHDLs and Their Applications, pp. 395-412, April, 1991.

60. P. C. Ward and J. R. Armstrong, "Behavioral Fault Simulation in VHDL", 27th Design Automation Conference, pp. 587-593, June, 1990.

Vita

Chang H. Cho was born in Seoul, Korea on November 18, 1954. He entered Seoul National University in March 1972, graduating with a Bachelor of Science degree in Electronics Engineering in February 1976. He attended graduate school at Virginia Polytechnic Institute and State University from September 1983 to February 1994, receiving a Master of Science degree in Electrical Engineering in March 1986. He is currently a Ph. D. candidate in electrical engineering.

Chang worked for the Agency for Defense Development in Korea as a research engineer from March 1976 to July 1983, participating in the development of a guidance and control system for a guided vehicle.

His research interests include modeling, simulation, synthesis, and computer-aided testing of VLSI circuits.