Behavioral Fault Simulation in VHDL[1]

P. C. Ward
Robertshaw Controls Co.
Richmond VA.

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

Abstract

This paper presents two tools which facilitate the fault simulation of behavioral models described using VHDL. The first tool is the Behavioral Fault Mapper (BFM). The BFM algorithm accepts a fault-free VHDL model and a fault list of N faults from which it produces N faulty models. The process of mapping the faults in the fault list onto copies of the original VHDL model is automated. The N faulty models are immediately suitable for fault simulation. The second tool presented is the Test Bench Generator (TBG). The TBG algorithm creates the VHDL TestBench and all other files necessary to complete a batch-mode fault simulation of the N faulty models.

Introduction

For small applications, including some early LSI devices, the gate-level view is a viable approach to test generation and fault simulation. However, the fact that the problem is NP-complete [1] dictates that for very large combinational circuits the problem will quickly become intractable.

In functional testing, the magnitudes of the fault testing and fault simulation problems are reduced by collapsing the effects of individual faults and sets of faults into single functional faults. At this level of abstraction, the circuit models are typically described in Hardware Description Languages (HDLs) such as the VHSIC HDL (VHDL) [2-3] employed in this research. Automatic test-pattern generation (ATPG) techniques at this level have been developed [4-5] which are high level variations on the classical D-algorithm. The preferred method of validating a proposed VHDL-based ATPG algorithm is fault simulation. However, the VHDL model may be entirely behavioral; suggesting no structure and, therefore, no specific gate-level representation.

Traditional fault simulation algorithms [6-8] are ill equipped to simulate models which are behavioral in nature. This is due to the requirement that the circuit description input to these simulators must be resolvable into a predefined set of primitives. The *Whistle* system described in [9] provides a partial solution to simulation at various levels of abstraction. However, while the interconnections of the functional blocks of this system may be described in VHDL, the functional blocks themselves (as they exist in the design library) must be gate-level implementation descriptions.

System Overview

The Behavioral Fault Mapper (BFM) and Test Bench Generator (TBG) algorithms were designed to provide maximum utilty in the VHDL-based ATPG/Fault Simulation environment under development at Virginia Tech. This section discusses the use of the BFM and TBG in both the ATPG Validation and standard fault simulation environments.

ATPG Validation

Figure 1 shows the relationship of the BFM and TBG tools to the other components of the VHDL-based ATPG/Fault Simulation environment. The notation used in the figure is as follows: Curly braces ({,}) indicate a single file. Thus, $\{C_1,C_2\}$ would indicate a single file composed of the two components C_1 and C_2. The components may be any amount of ASCII text, including none. Square brackets ([,]) indicate a collection of related files. Thus, $[F_1,F_2,F_3]$ would indicate a collection of three related files, F_1, F_2, and F_3. The names of software modules are printed in boldface type.

The important elements of the figure are as follows:

- The *VHDL source file*, $\{F_0\}$, is any fault free VHDL part description (both Entity and Architecture) written in a subset of VHDL acceptable to all of the software modules.
- The **Preprocessor** selects and numbers the lines of the source file which will be the sites of behavioral faults. The Preprocessor also constructs any necessary internal representation (e.g. Prolog predicates) required by the Test Generation Algorithm and appends this representation to the text of the source file. The file output by the Preprocessor is denoted as $\{F'_0\}$.
- The **Test Generation Algorithm** is considered to be the Unit Under Test (UUT) in this system, since the effectiveness of the ATPG is that which is to be determined. The expected output of the Test Generation Algorithm is a file containing the fault list and a file containing the test vectors. The Test Generation Algorithm block is actually composed of separate modules for fault list extraction and for further processing of the internal representation in addition to the actual ATPG algorithm.
- The **Behavioral Fault Mapper** (BFM) accepts the preprocessed source file, $\{F'_0\}$, from the preprocessor and

[1] The research described here was supported in part by Mitre Grant B-44230.

the fault list, $\{f_1,f_2,...,f_N\}$, from the Test Generation Algorithm and produces the N faulty source files, $\{F_1,F_2,...,F_N]$. Any line numbering introduced into $\{ F'_0\}$ by the **Preprocessor** is removed from the $[F_1,F_2,...,F_N]$ files by the BFM.

- The **Test Bench Generator** (TBG) accepts as input a copy of the source file, $\{F_0\}$, and the test vectors, $\{t_1,t_2,...,t_N\}$, from which it produces the Test Bench file, $\{T_0\}$, and the collection of Simulation Support Files ([SSF]). The Test Bench is the top-level design unit required by the simulation facility. The *Simulation Support Files* are those files required for simulation in addition to the VHDL source file, the N faulty files and the Test Bench. The SSF will always include one or more files related to generating the simulation report. Optionally, the SSF include a command file which automates the fault simulation procedure. Other files are present as needed.
- The *Files Required for Simulation* block typically represents a sub-directory on the host machine. The output of this block consists of all of the files necessary to perform the fault simulation. The grouping of the simulation files, $[T_0,F_0,F_1,...,F_N,[SSF]]$, into a sub-directory is required if batch processing will be performed using the SSF batch control file supplied by the Test Bench Generator.
- The **VHDL Simulation** block represents all of the procedures necessary for executing the Intermetrics VHDL 1076 Support Environment. The software includes the VHDL Library System (VLS), the Analyzer, and the Simulator. The functions of the components of the Intermetrics VHDL 1076 Support Environment are fully described in [10].

The result of simulation is the *Fault Simulation Report.* If the simulation is run using the Simulation Support Files, then the fault coverage achieved by the ATPG for a particular run may be determined by inspection of the fault simulation report(s). Thus, a software facility for developing and evaluating ATPG algorithms is made available.

Fault Simulation

The BFM and TBG tools are not restricted to operation within the ATPG validation system described in the previous section. Both tools are eminently useful in standard fault simulation. For example, the BFM requires only a numbered VHDL source model and a fault list to create the N faulty models. (A numbered source model is simply a model in which those statements or language constructs that are to be altered by the BFM have been preceded by a unique integer followed by a colon.) Furthermore, the ATPG software will typically include a separate Fault List Extractor (FLE) module which may be used to automatically generate the fault list. The FLE typically requires the same numbered source file as does the BFM. In the absence of an FLE, the fault list may be generated manually.

To perform its function, the TBG requires the entity declaration portion of the source model and a test set. The test set is a file containing collections of input vectors. There will be one test set file per fault simulation, and one collection of input vectors per faulty model within the file. The output of the TBG is a TestBench and the set of Simulation Support Files previously discussed. The fault simulation may be performed in batch-mode using the batch control file provided by the TBG.

Behavioral Fault Mapper

The purpose of the Behavioral Fault Mapper (BFM) is to automate the mapping of faults from a fault list onto copies of a VHDL source model. Each faulty model generated by the BFM will contain a single behavioral level fault. The faulty models will be immediately suitable for fault simulation.

Fault Classes

The behavioral level faults mapped by the BFM are divided into eight categories known as *fault classes.* These eight fault classes constitute the fault model used in this research. This behavioral fault model is based on a form of *model perturbation* [4] which has been shown to provide good equivalent gate-level coverage [11]. The eight fault classes are:

1. Stuck-Then
2. Stuck-Else
3. Assignment Control
4. Dead Process
5. Dead Clause
6. Micro-operation
7. Local Stuck-data
8. Global Stuck-data

Fault Mappings

Discussion of each of the fault mappings will include the type of failure represented and how the failure is mapped into a device description. A criterion which greatly influenced the selection of the mappings for each of the faults was that the source code be minimally altered. Given a set of valid source code modifications, the one chosen will always be the one which modifies the source model the least while achieving the desired faulting effect.

Stuck-Then

The Stuck-Then fault represents a failure of the **if-then-else** construct to ever execute the **else** statements. In the following example the signal A will always be assigned the value '1' in the presence of a Stuck-Then fault, regardless of the value of *logical_expression.*

Example:

```
if (logical_expression) then
     A <= '1'; else
     A <= '0'; end if;
```

The Stuck-Then fault is mapped by replacing the logical expression between the **if** and **then** keywords with the Boolean value TRUE. (TRUE and FALSE are defined in Package STANDARD [2].) The BFM would alter the **if** statement in the previous example to read:

```
if ( TRUE ) then
     A <= '1'; else
     A <= '0'; end if;
```

This modification insures that the *then* portion of the **if-then** statement will always be selected.

Stuck-Else

The Stuck-Else fault represents a failure of the **if-then-else** construct to ever execute the **then** statements. It is the dual of the Stuck-Then fault and is mapped by replacing the logical expression between the **if** and **then** keywords with the Boolean value FALSE.

Assignment Control

The Assignment Control fault represents a failure of the VHDL assignment operator to assign a new value to a signal. In the following example the value of the signal A would never be altered if an Assignment Control fault were associated with this statement.

Example:
A < = *new_value_expression*;

The Assignment Control fault is mapped by replacing the expression to the right of the assignment operator, the *new_value_expression*, with the signal name to the left of the operator unless the signal is of type **out**. Signals of type **out** can not be read, that is, they can not appear on the right hand side of an assignment statement or in an expression. Therefore, the assignment control fault for **out** signals is mapped by placing the comment symbol ("–") in front of the assignment statement. This has the desired effect of preventing the output signal from being altered by this statement. The BFM would modify the statement in the previous example to read:

A < = A;

If signal A was of type **out**, the mapping would be:

– A < = *new_value_expression*;

Dead Process

The Dead Process fault is a failure of the statements within a process construct to execute. In the following example, the statements within the process would never be executed in the presence of a Dead Process fault.

Example:

```
process(A,B,C)
  begin
     statement_1
     statement_2

     statement_n
     end process;
```

The Dead Process fault is mapped by replacing the *sensitivity list* of the process statement with the reserved signal STATIC_BIT. The signal value of STATIC_BIT never changes. Consequently, if STATIC_BIT is the only signal in the sensitivity list of the process statement, the process will never be triggered. (The sensitivity list is the list of signals which trigger evaluation of the statements within the process() construct.) In the preceding example, the sensitivity list is composed of the three signals A,B, and C. The mapping algorithm declares the signal STATIC_BIT in the architecture portion of the device model and uses it to replace A,B, and C. The signal name STATIC_BIT is reserved by the BFM for mapping Dead Process faults and must not be assigned a value by any activity within the source model. The BFM would modify the previous example to read:

```
process( STATIC_BIT )
  begin
     statement_1
     statement_2

     statement_n
     end process;
```

Dead Clause

The Dead Clause fault is a failure of the VHDL CASE construct to execute one of the alternative sequences of statements (clauses). In the following example, none of the statements in the third of the four alternative sequences would be executed in the presence of a deadclause fault specified as: *S*, DEADCLAUSE, *sn*, bv, 10. where *S* is the fault serial number, DEADCLAUSE is the fault class, *sn* is the statement number of the CASE construct within the numbered source file, *bv* is a keyword indicating that the quantity which follows should be interpreted as a bit-vector, and 10 is the length two bit-vector which denotes the faulted clause. In the example, CON is a bit-vector of length two.

Example:

```
case CON is
   when "00" =>
      CONSIG <= "1000";
      ENIT  <= '0';
   when "01" =>
      CONSIG <= "0100";
      ENIT  <= '0';
   when "10" =>
      CONSIG <= "0010";
      ENIT  <= '1';
   when "11" =>
      CONSIG <= "0001";
      ENIT  <= '1';
   when others =>
      null;
end case;
```

The BFM maps the dead clause fault by requesting the Assignment Control fault for all assignment statements within the specified clause. Thus, the BFM would modify the when "10" clause in the previous example to read:

```
when "10" =>
   CONSIG <= CONSIG;
   ENIT  <= ENIT;
```

The Assignment Control fault mappings requested following ***when* *"10"*** = > prevent the assignment of values to any of the signal objects within the clause.

Micro-operation

A Micro-operation fault is the failure of an operator to perform its intended function. The operator may fail to any other operator in its class. For example, in the assignment statement A < = B XOR C; the XOR may fail to any operator in the logical operator class (**and, or, nand, nor, xor**). Other operator classes in use are relational (=, /=, <, < =), and miscellaneous (**not**). Additionally, the bit-vector functions ADD and SUB are recognized as a class of interchangeable operators even though they are not operators by definition. The ADD and SUB functions are implemented in VHDL as subprograms.

The micro-operation fault is mapped by replacing the target operator with the replacement operator. The replacement operator is expected to be supplied in the fault description.

Example:

```
Fault: BITAND → BITOR
ENBLD < = DS1 and not NDS2; --good
ENBLD < = DS1 or not NDS2; --faulty
```

Local Stuck-data

The Local Stuck-data fault is the failure of a signal or variable object to have the correct value. The local stuck-data fault is restricted to the expression into which it is mapped. That is, a signal or object, A, will only be "stuck" in one expression of the device model. All other occurences of A in the model will retain their expected values. An example of a physical defect modeled by this fault is an open circuit at the input to a TTL gate. Assume the input signal is A and that A is normally connected to pin D of the TTL gate. The open circuit between A and pin D will cause pin D to "float" to a logical 1 (or possibly become indeterminate.) However, the signal A is unaffected in the remainder of the circuit since pin D does not drive signal A.

Example:

```
Fault: stuckdata, bit, 0;
if (STRB = '1') then  --good
if (STRB = '0') then  --faulty
```

Global Stuck-data

The Global Stuck-data fault is the failure of a signal or variable to change value within the device model. The global stuck-data fault is similar to the local stuck-data fault except that a global stuck-data fault is not restricted to faulting a single line of the device model. The global stuck-data fault will inhibit transitions on a given signal or variable caused by statements within a specified range of the source model. The range over which the signal or variable is stuck is specified in the fault description. The range may vary from a single numbered line of the device description to all numbered lines of the device description. An example of a physical defect modeled by this fault is a stuck-at fault on a primary circuit input.

In the following example, no statements would be allowed to modify signal DO in the presence of a global stuck-data fault specified as: *S*, GSTUCKDATA, s1, s7, DO. where *S* is the fault serial number, s1 and s7 are the beginning and ending line numbers (inclusive) over which the fault is to be mapped, and DO is the target. (Note that the signal DO appears on the left-hand side of assignment statements at numbered statements 6 and 7.)

Example:

```
    entity REGISTER is
     port (DI : in BIT_VECTOR(1 to 8);
           STRB, ENBLD : in BIT;
           DO : out BIT_VECTOR(1 to 8));
    end REGISTER;

    architecture BEHAVIOR of REGISTER is
     signal DID: BIT_VECTOR(1 to 8);
    begin
1:  process(STRB)
    begin
2:     if (STRB ='1') then
3:        DID <= DI;
       end if;
    end process;
4:  process(DID,ENBLD)
    begin
5:     if (ENBLD ='1') then
6:        DO <= DID;
       else
7:        DO <= "11111111";
       end if;
    end process;
    end BEHAVIOR;
```

The BFM maps the global stuck-data fault by requesting the Assignment Control fault at all numbered lines within the specified range which contain the target. The BFM would modify the previous statements 6 and 7 in the example to read:

```
if (ENBLD ='1') then
   DO <= DO;
else
   DO <= DO;
end if;
```

Test Bench Generator

The purpose of the Test Bench Generator (TBG) is to automate generation of the files required to perform fault simulation within the Intermetrics VHDL 1076 Support Environment [10]. The test bench is the top-level design unit required to perform simulation in the Support Environment. Conceptually, the test bench is a software implemented breadboard. As with traditional breadbords, circuit components are wired together, stimuli applied and response data collected. All of the flexibility of the traditional breadboard is present. The two types of test benches produced by the TBG are the ATPG Validation test bench and the Fault Simulation test bench.

The ATPG Validation Test Bench

The ATPG validation test bench is designed to check the validity of a set of test vectors generated by a behavioral-level ATPG algorithm. Figure 2 shows the simple configuration used to determine whether or not the set of test vectors (X) is a test for the fault in the faulty model (M). Let Y be the set of all output bits of the faulty model and Z be the corresponding output bits of the reference model. Further, let $\vec{y}$ represent the subset of faulty model output bits that are *expected* to differ from the corresponding subset of good model output bits ($\vec{z}$). If X covers the fault in M, then z_i xor $\vec{y}_i = 1$ for some $i \ni z_i \in Z, \vec{y}_i \in Y$. The TBG requires the ATPG to indicate which faulty model output signal(s) are to contradict the reference model; that is, the ATPG must specify $\vec{y}$. The bits in $\vec{y}$ are compared with the corresponding $\vec{z}$ bits. All other output bits are ignored. The requirement that $\vec{y}$ be specified prevents ATPG coverage statistics from being inflated by false hits. A false hit occurs when the applied test vector causes an *unexpected* subset of the faulty model output bits to be different from those of the reference model. The test still is said to cover the fault; however, the algorithm which generated the vector is probably not functioning correctly. Confidence in the algorithm is not warranted since similar behavioral faults in other models may or may not be found.

The configuration of Figure 2 is repeated N times within a single test bench in order to apply the N different test vectors to their respective good model/faulty model pairs. The N tests run during a simulation of the test bench have separate signal spaces. Thus, no dependencies exist between the tests.

Shown below is the form of a validation test bench generated by the TBG. The portions of the test bench which vary from simulation to simulation are shown in italics.

```
entity TEST_BENCH is
end TEST_BENCH;
use WORK.all;
architecture entity_name_TEST of TEST_BENCH is
  -- Signal used to begin test process
  signal INIT: BIT;
  -- Comparator output signals
  signal C1,C2,...,CN : BIT;
  -- Good model output signals
  signal Z1,Z2,...,ZN : type_declaration;
  -- Faulty model I/O signals
  signal numbered port declaration signals
  -- Component declarations
  component component_name
    port declaration
  end component;
  component COMP
    port (A, B: IN BIT_VECTOR; C: OUT BIT);
  end component;
  -- Use statements
  for R_i: component_name
   use entity work.entity_name(architecture);
  for F_i: component_name
   use entity work.entity_name_f_i(architecture);
  for X_i: COMP use entity work.COMP(BEHAVIOR);
begin
  -- component instantiation
  INIT < = '1';
  process(INIT)
   begin
     -- test vectors
  end process;
   -- bit comparators
end entity_name_TEST;
```

The elements of the test bench are interpreted as follows:

- The ***entity_name*** identifier provides a device name for the model. For example, an 8-bit register might have an entity_name of REGISTER or REG8. This identifier is supplied by the user during program initialization.
- The **Comparator output signals** convey the results of the individual comparison operations. The signals are all of type bit. The integer following the letter C is the fault number. In general, the fault numbers will not be sequential as implied by the figure.
- The **Good model output signals** transfer the $\vec{z}$ bits of the reference output to the comparator.
- The **Faulty model I/O signals** apply the test vectors to both the faulty and good models, and transfer the $\vec{y}$ bits of the faulty model to the comparator. The faulty model I/O signal declarations are created by appending "*_faultnumber*" to each of the signals in the port declaration for each of the faults in the fault list. This operation creates the N required sets of distinct interconnect signals.
- The **Component declarations** section declares the interface to the source model and, optionally, the interface to the bit-vector comparator. The *component_name* identifier in the source model interface is set equal to the *entity_name* identifier previously described. The *port declaration* part of the source model interface is a copy of the interface specification provided in the entity declaration of the VHDL source model. The comparator module, COMP, is declared for use whenever one or more $\vec{z}$ and $\vec{y}$ subsets are larger than a single bit. The COMP component declaration is absent otherwise.
- The **Use statements** associate binding information with the component labels representing specific instances of a given component. The figure shows the *i*th set of use statements. A simulation involving N models will require N sets. The *component_name* and *entity_name* identifiers have been previously defined. The *architecture* identifier specifies the architectural body name of the source model and is provided by the user during program initialization. The symbol f_i represents the *i*th fault number. The comparator use statement (X_i) will only be present when the $\vec{z}$ and $\vec{y}$ of the *i*th model are larger than a single bit; that is, when a bit-vector comparison is required. Note that a separate reference model (R_i) is instantiated for each instantiated faulty model (F_i). This is necessary since, in the general case, different test vectors are required to detect different faults and for every unique test vector a unique reference response is possible.
- The ***component instantiation*** block contains the **port** map and **generic** map statements required to interconnect the N models, comparators, and sets of test vectors. This section constitutes the bulk of the wiring of the software breadboard. Output signals not being compared are left open using the VHDL **open** keyword.
- The ***test vectors*** section contains the signal assignment statements which transfer the values of the test vectors to the good and faulty model input signals.
- The ***bit comparators*** are single statements of the form: C_i < = z_i xor y_i; . The TBG creates a bit comparison statement for the *i*th model if and only if $\vec{y}$, and, by implication $\vec{z}$, are single bits. A given simulation may contain both bit-vector comparisons and single-bit comparisons.

The Comparator Model

The TBG outputs a VHDL behavioral-level description of an N-bit comparator for use in comparing signals of type bit-vector. The comparator model is analyzed and model generated along with the good and faulty models whenever one or more of the $\vec{y}$ faulty model outputs is a bit-vector. The use of the comparator is transparent to the user. The VHDL for the comparator is as follows:

```
entity COMP is
  port (A, B : IN BIT_VECTOR;  C : OUT BIT);
end COMP;

architecture BEHAVIOR of COMP is
begin
  process(A,B)
    variable TEMP: BIT;
  begin
    TEMP := '0';
    for I in A'Range loop
      TEMP := TEMP or (A(I) xor B(I));
    end loop;
    C < = TEMP;
  end process;
end BEHAVIOR;
```

The Fault Simulation Test Bench

At the users' request, the TBG will produce a fault simulation test bench. The purpose of the fault simulation test bench is to display the response of a VHDL model to a set of arbitrary input vectors. The conceptual difference between the validation test bench and the fault simulation test bench is that the test vectors of Figure 2 are replaced by *input vectors* since no testing_function is implied. Further, there is no requirement that $\bar{y}$ be specified. The response of the faulty model is displayed in a simulation report. The simulation report provides a trace of the faulty model output signals (Y) and, for comparison, a trace of the reference model output (X).

The fault simulation test bench is similar to the ATPG validation test bench with the output comparison functions deleted. The specific differences between the two test benches are as follows:

1. The N-bit comparator is not declared or instantiated in the fault simulation test bench;
2. The bit comparators at the end of the validation test bench are omitted in the fault simulation test bench;
3. The component instantiations in the fault simulation test bench differ slightly from those in the validation test bench since no outputs in the fault simulation test bench are left open.

Results

The Behavioral Fault Mapper and Test Bench Generator have been used in performing behavioral fault simulation for a number of SSI and MSI device models. The models have included 8-bit registers and I/O ports (e.g. Intel 8212), controllable counters, decoders, and experimental models incorporating reconvergent fan-out.

The following analysis derived from our experience thus far suggests the utility of the BFM and TBG in performing behavioral fault simulation. If S is the size of the VHDL source model and N is the number of faults to be injected, an approximate lower bound on the disk space required to store the results of this mapping operation is $[(N+1)S]$ bytes. If we assume a modest VHDL source file size of $S=2K$ bytes and $N=150$ faults, the resulting disk space required for this mapping operation is approximately 302 Kbytes. The reader should be aware that 2K is roughly the size of an MSI behavioral-level part description. Thus, for LSI/VLSI parts, S and N will be much larger and significantly more disk space will be required for a given N.

The size of the Test Bench produced for the same S and N is much more difficult to predict. The Test Bench size is primarily a function of the number of primary inputs, I, the length of the fault list, N, and the average number of time steps, T, required to test a given fault. For MSI parts, the size of the resulting Test Bench may be estimated as 1.5SN. (The Simulation Support Files do not contribute significantly to the size of the file suite and are thus ignored.) In total, $N+1$ models, a single, large Test Bench file and various simulation support files occupying approximately $[(N+1)S + 1.5SN] = (2.5N+1)S$ bytes are produced. It is evident that manually generating the files required for behavioral fault simulation would be prohibitively time consuming and error prone. Thus, the BFM and TBG make the behavioral fault simulation of realistic models and large N practical.

Conclusion

Two tools for VHDL fault simulation have been presented: the Behavioral Fault Mapper and the Test Bench Generator. The Behavioral Fault Mapper is an effective tool for the automatic creation of faulty VHDL models from a source model and a fault list. Eight behavioral fault classes were presented and the effects of their mappings discussed. The extensive size of the typical fault simulation testbench gave rise to the Test Bench Generator. This algorithm provides an effective, easy-to-use means of producing the fault simulation test bench and related simulation support files. Together, the Behavioral Fault Mapper and the Test Bench Generator reduce the time and sophistication required to perform behavioral fault simulation in VHDL.

References

1. Fugiwara, H. and S. Toida, "The Complexity of Fault Detection Problems for Combinational Logic Circuits," IEEE Trans. on Computers, Vol. C-31, No. 6, June 1982, pp. 555-560.
2. *IEEE Standard VHDL Language Reference Manual*, IEEE, Inc., NY, March 1988.
3. Armstrong, J.R., *Chip Level Modeling with VHDL*, Prentice-Hall, Englewood Cliffs, New Jersey, 1989.
4. O'Neill, M.D., D.D. Jani, C.H. Cho and J.R. Armstrong, "BTG: A Behavioral Test Generator", CHDL 1989, Elsevier Science Publishing Co., New York, New York, June 1989, pp. 347-361.
5. Barclay, D.S., and J.R. Armstrong, "A Heuristic Chip-level Test Generation Algorithm", 23rd Design Automation Conf., June 1986, pp. 257-262.
6. Goel, P. and P.R. Moorby, "Fault-Simulation Techniques for VLSI Circuits," VLSI Design, July 1984, pp. 22-26.
7. Armstrong, D.B., "A Deductive Method for Simulating Faults in Logic Circuits," IEEE Trans. on Computers, Vol. C-21, No. 5, May 1972, pp. 464-471.
8. Ulrich, E.G. and T. Baker, "The Concurrent Simulation of Nearly Identical Digital Networks," Proc. 10th Design Automation Workshop, IEEE and ACM, New York, June 1973, pp. 145-150.
9. Renous, R., G. M. Silberman, and I. Spillinger, "Whistle: A Workbench for Test Developement of Library-Based Designs," Computer, Vol. 22, No. 4, 1989, pp. 27-41.
10. *User's Manual for the Standard VHDL 1076 Support Environment (DRAFT)*, USAF Document No. IR-MD-103-3, Intermetrics, Inc., Bethesda, Maryland, August 1988.
11. C. H. Cho, "A Chip Level Fault Coverage Experiment," Research Report, Chip Level Modeling Group, E.E. Dept., Virginia Polytechnic Institute and State University, August 1986.

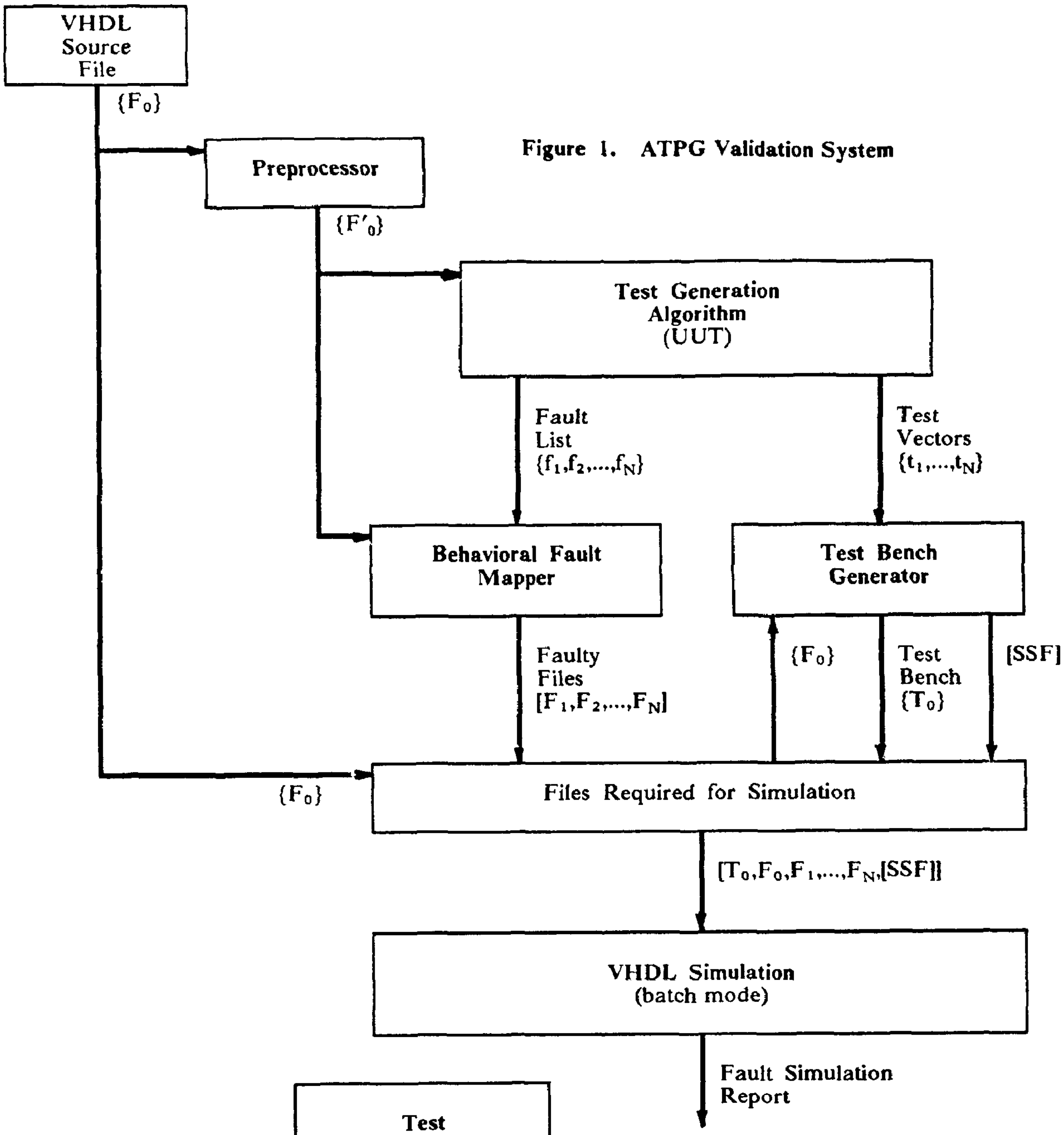
VHDL Source File
{F₀}
Preprocessor
{F'₀}
Test Generation Algorithm (UUT)
Fault List {f₁,f₂,...,f_N}
Test Vectors {t₁,...,t_N}
Behavioral Fault Mapper
Test Bench Generator
Faulty Files [F₁,F₂,...,F_N]
{F₀}
Test Bench {T₀}
[SSF]
{F₀}
Files Required for Simulation
[T₀,F₀,F₁,...,F_N,[SSF]]
VHDL Simulation (batch mode)
Fault Simulation Report

Figure 1. ATPG Validation System

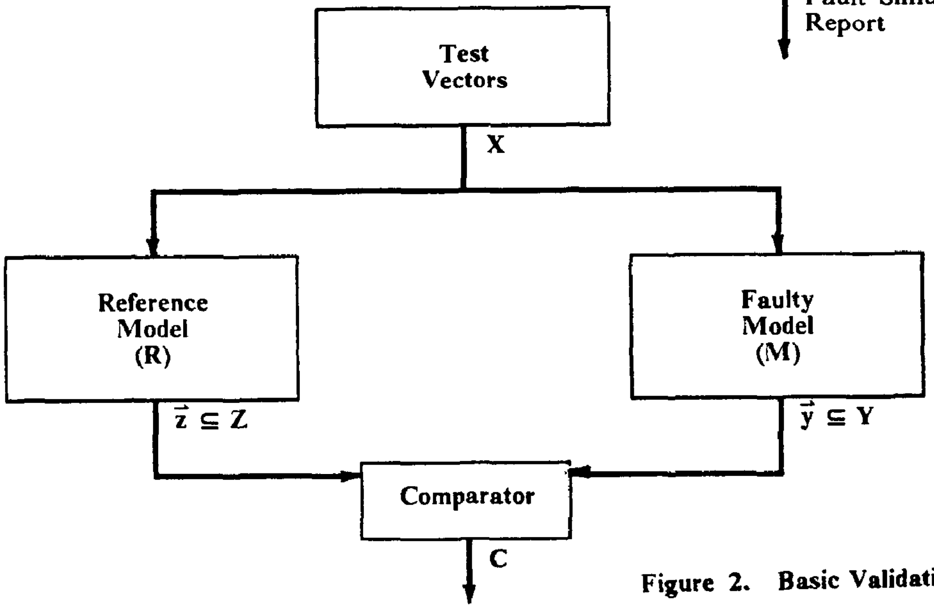

Figure 2. Basic Validation Scheme