RUBASTEM: A Method for Testing VHDL Behavioral Models

Anneliese Andrews, Andrew O'Fallon
School of Electrical Engineering
and Computer Science
Washington State University
Pullman, WA 99164
{aandrews, aofallon}@eecs.wsu.edu

Tom Chen
Department of Electrical
and Computer Engineering
Colorado State University
Fort Collins, CO 80523
chen@engr.colostate.edu

Abstract

Verification of high-assurance system designs before they are fabricated on chips reduces time and money costs related to chip testing. Test benches, which are used in simulating and verifying VHDL designs, need to perform efficiently and effectively. Test automation through the automated generation of test patterns increases the efficiency and effectiveness of behavioral verification. We apply heuristic rules to generate test cases for VHDL behavioral designs. Their aim is to increase code coverage. They are based on control-flow and data-flow analysis. We applied the RUle-BAsed Software TEsting Method (RUBASTEM) to small, medium, and large examples. The results of RUBASTEM showed that this method achieves higher branch coverage.

1. Introduction

Verification of design models is a critical component in the hardware design life cycle of high-assurance systems. Verification of embedded systems' chips, for applications such as micro-electro-mechanical systems (MEMS) and micro-robots is an important concern for many system designers. Methods for verifying these hardware designs need to be efficient and effective. High-assurance systems require that the system is designed with confidence.

There are three major phases in the high-assurance hardware design life cycle, as shown in Fig. 1(a). First, the desired function and behavior of the system is modeled in a high level of abstraction using a special purpose language. Next, this behavior is mapped into lower levels of abstraction such as the gate and transistor levels (Fig. 1(b)). Finally, the mapped layout is sent to the factory for manufacturing.

The first phase describes hardware in terms of a program in a hardware design language, such as VHDL. Hardware designers then perform extensive simulations for what they call "behavioral verification", an activity a software engineer might term "validation", or "software testing". Because VHDL is similar to a high-level programming language, we can apply software assurance techniques to a hardware design in order to identify and remove faults. These faults, if carried to the manufacturing phase would be quite expensive. Further, these faults need to be detected through the use of test benches. Test bench automation through the generation of test patterns and test cases increases the efficiency and effectiveness of behavioral verification.

This paper applies a rule-based method for generating test patterns for VHDL designs as described in [3]. Many methods have been defined which address the issue of generating test patterns [2, 4, 7, 8, 11, 12, 13, 14]. Several of these methods require large amounts of test patterns, others are fairly complex.

RUBASTEM is a heuristic, rule-based method for generating test patterns. The method uses the control flow of the VHDL source code to generate inter/intra procedural control flow graphs. The control flow graphs may be analyzed to determine branch locations within the code. The rule-based method also uses data flow analysis to determine definition use pairs and reaching definitions. Data flow analysis enables RUBASTEM to determine which variables and signals affect decisions at branches. In general, rule-based methods try to satisfy testing criteria. Thus, testing criteria are used to determine when adequate verification has been completed. Statement, branch, condition, bit toggle, and statistical criteria are common in behavioral verification. RUBASTEM is primarily interested in meeting branch coverage criteria.

As [3] defines, the rule-based software testing method consists of three phases. These phases include: control flow graph generation, data flow analysis, and rule-based test case generation. The goal of the rule-based method is to generate the least number of test cases which will guarantee that a certain percentage of a coverage criterion will be satisfied. Software metrics are collected which may be used in generating test patterns for future designs. Section 2 describes background information and existing work. Section 3 describes the techniques used in the rule-based approach such as control flow graph generation, data flow analysis, and other tools. Section 4 describes the process for generating the rules. Section 5 describes the results of RUBASTEM on VHDL designs. Section 6 draws conclusions and proposes further work.

2. Background

When discussing behavioral verification of hardware design, we focus on fault coverage and code coverage. Fault coverage is used in detecting manufacturing faults in the structural level designs. Code coverage helps validate the functionality of the behavioral design. Some techniques to increase coverage are described next.

2.1. Hardware testing techniques

1. Random: In random testing, the values of the inputs in each test are selected randomly, sometimes based on user-defined frequencies. Random testing avoids the problem of deterministic test generation that requires structural information about the program to be tested. It is a reasonable choice for obtaining certain levels of coverage. However random testing becomes very inefficient when coverage saturates. [11] proposes a systematic approach to increase the fault coverage obtained by random pattern testing. The paper applies a random vector at the preliminary inputs of the sequential circuit and holds it constant for more than one clock cycle, instead of applying a new random vector every clock cycle. Then a number of flip-flops is chosen to be added to the scan chain, using any of the known flip-flop selection methods. The hold method is highly effective in improving fault coverage. A partial scan can be used to further improve the fault coverage. However this method can be costly. The hold method is only targeted to increase the coverage of circuits which are amenable to random coverage. [8] presents an improved weighted random pattern testing method. A new algorithm is proposed in the paper that throws away weight sets that prove ineffective with simulation. The paper also describes some heuristic rules to enhance the algorithm. The heuristic rules affect the balance between the number of weighted patterns and number of weight sets.

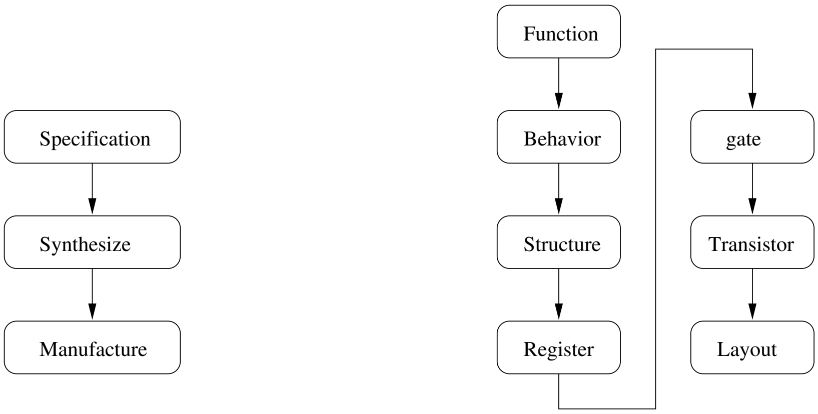
2. Anti-random: The basic premise of anti-random testing is to chose new test vectors that are as far away from existing test inputs as possible. Distance is commonly measured by terms of Hamming or Cartesian Distance. In previous analysis [9], anti-random testing has improved code coverage for boundary conditions, and has proved more efficient than random testing. However the basic anti-random method requires enumeration of the input space and computation of distance for each potential input vector. This prevents scale-up to large test sets and/or long input vectors.

   To deal with this limitation, [14] uses a Fast Anti-Random (FAR) approach to generate new test sequences by centralizing all existing input vectors into one average test vector. FAR finds all potential input test vectors that are orthogonal to the centralized vector and then finds an anti-random vector with maximum distance from the centroid vector. FAR very efficiently generates anti-random test patterns even for large vectors and long sequences of tests.

3. Methods based on Artificial Intelligence: When the number of tests to be run is very large, test case reduction becomes desirable. Neural network methods can help to achieve this. [2] uses a neural net to filter out generated test cases that are not likely to increase coverage. The neural net also reduces the already existing set by throwing out ”coverage duplicates”, trying to keep test cases to a minimum while still achieving the coverage goal. [12] proposes a genetic algorithm for generating test vectors for both combinational and sequential circuits. The approach is based on continuous mutation of a given input sequence and on analyzing the mutated vectors for selecting the test set. Using the approach, test cases are generated with very high coverage.

4. Static Analysis: [13] uses constraint programming techniques to generate test cases for design verification of VHDL programs. The behavioral specification in VHDL is used to generate mathematical constraints. Additionally it is specified which path of the control flow graph of the design should be covered. Constraint solving techniques are then applied that generate a test input, which causes the execution of the desired path. The advantage of this approach is that the process is guaranteed to find a matching test. If it fails, the path is not feasible. However the method is time consuming and not practically applicable to large designs that require the examinations of multiple clock cycles.

5. Fault Injection: Fault injection is an important technique for the evaluation of fault coverage. Hardware-based fault injection includes bombarding the hardware with heavy-ion radiation, injecting voltages on



(a) Phases of Hardware Design

(b) Levels of Hardware Design

Figure 1. Hardware design levels and phases.

the hardware's power rails, and corrupting logic values at the pin level. Fault injection-based automated testing (FIAT) is an example of software-based fault injection. Simulation-based fault injection is another kind of fault inject methods. [7] proposes a method to perform a fault-tolerance analysis on the behavioral level using fault injection techniques on the instruction set architecture (ISA). This allows early generation of faults and therefore the simulation of the design with regards to fault tolerance.

6. Rule-based software test method: [6] proposes a rule-based software test data generator to test Ada programs. In the approach, some heuristic rules and symbol information specific to the subject module under test are used by a rule interpreter to generate test cases. The described test data generator is used in conjunction with a software testing environment for a high level programming language. Heuristic rules are specific to the types of statements in the language. To adopt this approach for VHDL requires reformulating the heuristic rules. [3] applies the rule-based method to VHDL programs. Data flow analysis and control flow analysis are used to generate more test patterns to satisfy branch coverage criterion.

2.2. Testing criteria

Since VHDL is similar to some of the high level programming languages, it is possible to use software testing techniques on hardware designs modeled in VHDL. Additionally, VHDL has its own characteristics such as hardware delays and triggering mechanisms. Therefore, verifying VHDL models requires the development and justification of different test criteria in addition to those used for conventional software.

[1] and [5] use code-coverage metrics similar to those used in software engineering to detect faults in the behavioral code and to reduce the redundant portions of the design. Examples of software code coverage include statement coverage, branch coverage, condition coverage, and path coverage [1]. Examples of hardware code coverage include process-sensitivity-list coverage and signal toggle coverage. RUBASTEM is primarily interested in branch coverage. Branch coverage measures how many times a simulation executes each branch. Incomplete branch coverage can reveal which conditions prevent access to the statements contained at the most nested branch. Branch coverage subsumes statement coverage.

3. Tools

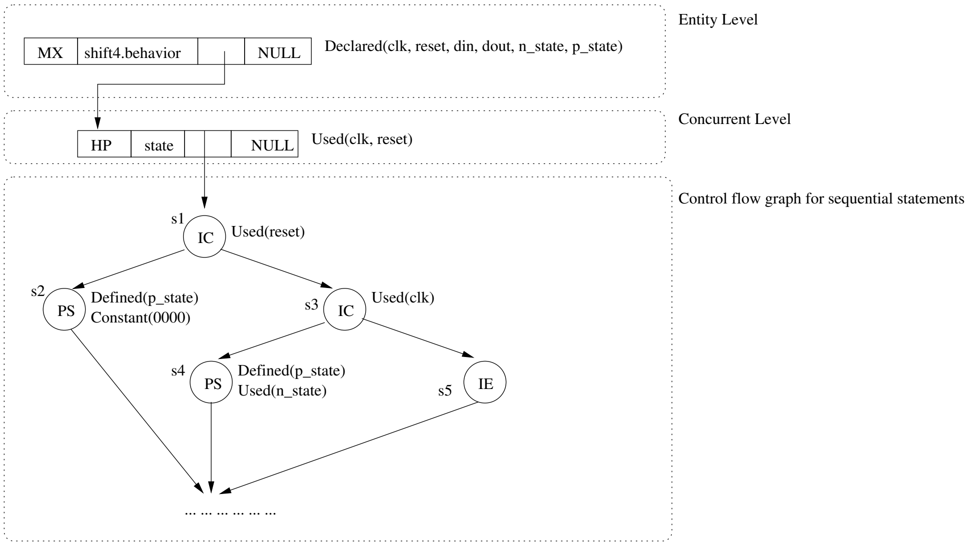
RUBASTEM is based on two tools: a control flow graph generator and a data flow analyzer. "Control Flow analysis determines the control structure of a program, that is the set of all possible execution paths." [15] Calling relationships between procedures are also constructed. "Data Flow Analysis (DFA) examines the flow of scalar values through a program. It solves data flow problems by propagating data flow information along the paths of a control flow graph." [15] In this paper, DFA is used as the static analysis of the possible dynamic modification and usage of variable data values in a program. It determines which variables affect the value of others. This is useful for test generation because, knowing what branches need to be covered, we can determine which input variables affect the branch condition at those branches and create test cases for a particular branch condition. DFA requires a complete anno-

```
...... ......
entity shift4 is
     port (signal clk, reset: in bit;
            signal din: in bit;
            signal dout: out bit_vector (3 downto 0)
     );
end shift4;

architecture behavior of shift4 is
signal p_state, n_state: bit_vector (3 downto 0);
begin
     ......
     state :process (clk, reset)
     begin
          if (reset='1') then
               p_state <= "0000";
          elseif (clk'event and clk='1') then
               p_state <= n_state;
          end if;
     end process state;
     ......
end behavior
```

(a) shift4.vhdl

```
MX shift4.behavior @8
MP @1 {clk IN; reset IN; din IN; dout[3:0] OUT; }
HS{}
MH{

HS { n_state[3:0]; p_state[3:0]; }
 ......
HP state @15 {
PN { clk, reset; }
PD {}
PI @17 {
IC @17 { ((reset EQ "1"[0:0])) }
PS @18 {p_state[3:0] <= (("0000"[3:0]))}
IC @19 {(((EVENT(clk)) AND ((clk EQ "1"[0:0]))))}
PS @20 {p_state[3:0] <= ((n_state[3:0]))}
IE @21 {} @21 } @22 }} @32 }
```

(b) shift4.ivf



(c) Control Flow Graph for Shift4

Figure 2. Example of control flow graph and branch counters.

tated Control Flow Graph to store the necessary information. While building a control flow graph generator, certain software metrics can also be collected. These metrics include the cyclomatic complexity of a process, the number of concurrent constructs, and the number of entities in the program.

3.1. Control flow graph generator

The purpose of control flow analysis is to encode the flow of control of a given VHDL program for use in data flow analysis and test case generation. A CFG comprises the basic component in VHDL programs, i.e. entity, concurrent (process and block) and statements. Thus there are three levels in a CFG.

- Entity level: Entities are represented as a singly linked list. Each node corresponds to an entity. Information stored in each node includes entity type and declared signals/variables. The names of called entities and port maps are also stored. This makes it possible to determine calling relationships between entities.

- Concurrent level: Each entity node in the entity structure may contain a concurrent structure (also represented as a singly linked list). Each node in this linked list corresponds to a concurrent block within the same entity. Stored information includes concurrent type, activation list and declared signals/variables. A signal/variable is defined when it is assigned a value. It is used when it is referenced.

- Control flow graph for sequential statements: Each concurrent node in the concurrent structure may contain a triple $G = (N, E, s)$ [15], where (N, E) is a directed graph, $s \in N$ is the initial node, and there is a path from s to every node in G. Each node $n \in N$ corresponds to a statement and edges represent potential transfers of control between statements. There is an arc $(x, y) \in E$ from node x to node y if control can potentially transfer from x to y at run time. Each node is annotated with the information about the corresponding statement such as statement type, defined signals/variables and used signals/variables/constants.

An example of a CFG is shown in Fig. 2. The graph is built based on an intermediate format of VHDL. We have chosen to extract information from the .ivf file for three reasons: 1) It contains all information present in the VHDL source code; 2) Being in an intermediate format it is easy to parse; 3) Since it is processed by the second phase of compilation the lex/yacc programs for vsim are already present and can be reused with modifications. Therefore the lex/yacc files used by vsim are modified for the CFG generation. Fig. 2(c) shows the resulting CFG.

At the entity level of the example in Fig. 2, a node is built for the entity $shift4$. Within this entity, there is a process block $state$. Thus a node at the concurrent level is created. Each node at the third level corresponds to a sequential statement, for example, node s_1 is built for the IF statement with line number @17, Node s_2, s_3, s_4 and s_5 correspond to statements with line number @18, @19, @20 and @21 respectively. Every node in this CFG contains information which is used in DFA, such as the declared, defined and used signals/variables. When the condition in statement @17 is true, statement @18 is executed. Otherwise statement @20 is executed. Thus, in the CFG, there are two arcs (s_1, s_2) and (s_1, s_3). We define a node s as a branch point if there is more than one outgoing arc.

3.2. Data flow analysis

Data Flow Analysis (DFA) [15] is a static analysis method that determines which variables/signals affect the value of other variables/signals. This is useful for test generation because when we know what branches need to be covered, we can determine which input variables affect the branch condition at those branches. As a result we can create test cases for a particular branch condition, taking into consideration the signals or variables that influence branching behavior at any given branch. Here, DFA is employed to extract signal and variable use-definition information. We use heuristic rules to generate test cases that potentially improve branch coverage. For a VHDL program, the DFA is implemented based on its control flow graph $G = (N, E, s)$.

3.2.1. Use-definition [15] Let $n \in N$, VAR is the set of VHDL variables/signals. We define

- $DEF(n) = \{v \in VAR : n\ contains\ a\ definition\ of\ v\}$
- $USE(n) = \{v \in VAR : n\ uses\ v\ in\ n\}$
- $RD(n, v) = \{n' \in N, n' \neq n : n'\ reaches\ n\}$
  RD is called the set of reaching of definitions for n. A path in G is definition free for a variable/signal v iff v is not defined in any of the path's nodes. We say n' reaches n iff there exists a path from n' to n (except for the endpoints) which is definition free for v.
- $UD(n, v) = \{n' : n' \in RD(n, v)\ and\ v \in DEF(n')\},\ where\ v \in USE(n)$. $UD(n, v)$ is called the use-definition set for n and v. Thus $UD(n, v)$ gives all nodes n' in which v is defined and where definitions reach the use of v in n.

For example for the following code segment,

$$s_1 : A = B + C$$
$$s_2 : IF\ (A > D)\ THEN$$

$DEF(s_1) = \{A\}, USE(s_1) = \{B, C\}, DEF(s_2) = \phi, USE(s_1) = \{A, D\}, RD(s_2, A) = \{s_1\}, UD(s_2, A) = \{s_1\}$.

3.2.2. Dominator [15] Let $G = (N, E, s)$ be a control flow graph, and $n, n' \in N$. We define:

- n dominates n', written $n \leq n'$: each path from s to n' contains n.
- n properly dominates n', written $n < n'$: $n \leq n'$ and $n \neq n'$.
- n direct dominates n', written $n <_d n'$: $n < n'$ and there is no $n'' \in N$ such that $n < n'' < n'$.
- $DOM(n) = \{n' : n' \leq n\}$ is the set of dominators of n.

In Fig. 2, there exists $s_1 <_d s_3 <_d s_5$. $DOM(s_5) = \{s_1, s_3, s_5\}$.

3.2.3. Data flow analyzer and reaching inputs DFA generates use-definition information based on the annotated CFG. There are two types of DFA: intra-module and inter-module [15]. Intra-module is performed at the level of a single procedure (such as Process or Block in VHDL), while inter-module takes the whole program into account. Intra-module DFA computes dominators, use-definitions and local reaching inputs for each node $n \in N$. Inter-procedural DFA determines the calling relationships between procedures of the program. It

Model	Cyclomatic Complexity	Number of Concurrent Statements	Number of Entities
Shift4	3	1	1
Dec2to4	8	1	1
Dec5to32	36	1	1
Gcd	17	1	1
Priority	10	1	1
Tlc	15	1	1
MINTEL	237	7	2

Table 1. Software metrics for VHDL models.

Rule	Statement Types	Precondition	Generation Effect
Rule 0	Any Statement	$\lvert RI(b)\rvert < max$	Generate all combinations of values for $RI(b)$
Rule 1	IF/ELSIF/WHILE	Branch contains constants, c	Set all $v \in RI(b)$ to c, $< c$, $> c$
Rule 2	IF/ELSIF/WHILE	May or may not contain constants	For each $v \in RI(b)$, set v to NOT of v in t_m
Rule 3	ELSE	None	Use Rules 1 and 2
Rule 4	WHEN	Branch contains constants, c	Set all $v \in RI(b)$ to c
Rule 5	WHEN OTHERS	None	Find constants that do NOT occur in $b\prime \in DOM(b)$, for every c, perform Rule 4
Rule 6	Any Statement	None	Generate two test patterns, one with all '1's and one with all '0's
Rule 7	Any Statement	None	For every new test pattern, generate another pattern that is the NOT of this pattern

Table 2. Test generation rules for VHDL designs.

uses the top two levels of the CFG (entity level and concurrent level). We can also get the port maps between signals/variables in one procedure and others in another procedure. Thus local reaching inputs $RI(n)$, which is defined below, can be propagated to other procedure signals/variables, and finally to the signals/variables declared in the main procedure. To implement this, bit vector matrices are used. Every cell in the bit vector matrix records the use-definition relationship between one bit of a signal/variable and another. It is 1 if a use-definition relationship exists, 0 otherwise.

The reaching inputs $RI(n)$ is generated as follows: Let $n \in N$, VAR is the set of program variables/signals. Based on ALGORITHM 3.4 in [15] to compute reaching definitions, we define the following algorithm to compute reaching input $RI(n)$:

1. Initialize $RI(n) = USE(n)$. For each $v \in VAR$ and $v \in USE(n)$, compute $RN = UD(n, v)$
2. Compute local $RI(n)$: While $RN \neq \phi$, for each $n' \in RN$ do
   (a) Update $RI(n) = RI(n) \cup USE(n')$
   (b) For each $v \in VAR$ and $v \in USE(n')$, $RN = RN \cup UD(n', v)$, $RN = RN - \{n'\}$
3. Propagate local $RI(n)$ to $RI(n)$ of the main procedure using inter-module DFA.
4. For each $v \in RI(n)$, if the type of v is not IN or $INOUT$ (signal types defined in .ivf file), $RI(n) = RI(n) - \{v\}$.

We use this algorithm to compute which input signals have an effect on the condition in a node. For a node n, the reaching input $RI(n)$ is initialized to the set of used signals/variables in n. For each v of these signals/variables, RN is set to the use-definition for n and v that is used to keep track of the use-definition chains. While it is not empty, for each node n' in RN, we add the used signals/variables in n' into $RI(n)$. For every v of these signals/variables, we add use-definitions for n' and v into RN and delete n' from it. We continue to do this until RN is empty. This means there are no more nodes that have an effect on n. After we get the local reaching inputs, we propagate them to those declared in the main procedure. The last step of the algorithm is to remove unnecessary signals/variables from $RI(n)$ because we only need to keep those which have type IN or INOUT to be used in later test case generation.

We also define the length of $RI(n)$: $|RI(n)| = \sum |v|$, $v \in RI(n)$. Thus, $|RI(n)|$ is the sum of the lengths of each $v \in RI(n)$.

In SHIFT4 in Fig.2, for example, $RI(s_1) = \{reset\}$, $|RI(s_1)| = 1$.

3.3. Test coverage criteria

We use all branches criterion in RUBASTEM. A branch is defined to be covered if both the true and false parts of the branch are executed at least once. The following branches are required to be covered:

1. The beginning of process bodies. The criterion requires each process to be invoked at least once, in addition to any initialization runs a specific VHDL simulator may perform.
2. The IF statement has branches related to the THEN and ELSE options. Additionally, nested IF statements (ELSIF) must all be covered.
3. FOR LOOP and WHILE LOOP. Branch coverage requires executing the body at least once. In the case of FOR LOOPs branch coverage also requires exactly one execution (the body is executed once, the condition is false, and execution control does not branch back to the beginning of the FOR LOOP), as well as executing the body more than once (the looping condition must be true at least once). Similarly, branch coverage for the WHILE loop requires that the condition for looping must be false at least once (the body of the loop is not executed, control skips to the statement after the loop), as well as another run where the looping condition is true and the body of the loop is executed at least once.
4. The true and false branches of EXIT WHEN and NEXT WHEN statements.
5. Each alternative of a CASE statement, except the infeasible alternatives, if any.
6. The branch following a WAIT ON or WAIT UNTIL.
7. The branches of WHEN-ELSE in the signal assignment statements.
8. The true and false conditions of ASSERT statements.

3.4. Software metrics

Since the VHDL programs are parsed for control flow and data flow analysis, software metrics can be collected. These metrics include cyclomatic complexity [10], the number of concurrent statements, and the number of entities. Table 1 shows these metrics for the executable in 2(a) and six other models. The cyclomatic metric is a cumulative measure of all cyclomatic complexities of concurrent constructs within the design. In future applications of RUBASTEM, these metrics may be applied to approximate the number of test patterns needed to satisfy a coverage criterion. The number of test patterns generated should be proportional to a combination of the cyclomatic complexity, the number of concurrent constructs, and the number of entities in the design.

4. Rule-based approach

Our approach is based on static analysis of the VHDL code and a series of heuristic rules. The static analysis provides a control flow graph and data flow information for the VHDL code. The proposed approach not only draws information from the VHDL source programs, but also makes use of the results of prior tests to guide the generation of additional test cases. Specifically, the simulator has been instrumented to provide coverage information as the VHDL code processes input patterns. Thus, the test analysis system knows which branches are executed with which inputs. In addition, the data flow analysis provides information about the inputs that affect the decision at a branching point. This information is used in rules to generate new inputs that are more likely to execute a previously uncovered branch. In this section, a brief overview of the rule-based approach is presented. Table 2 presents the rules and [3] describes the rules in detail.

4.1. Test generation process

We define more formally the terms test pattern, test case, and branch point. We also define what it means for a test case to reach a branch point.

- Test pattern: An input pattern $t = (i_1, i_2, \cdots, i_k)$, each i_j $(1 \leq j \leq k)$ is an input bit $i_j \in \{0, 1\}$. For example, $t_1 = (0, 1)$ is an input pattern for SHIFT4 in Fig. 2.
- Test case: A test case is a sequence $T = \{t_1, t_2, \cdots, t_n\}$, each t_i $(1 \leq i \leq n)$ corresponds to a VHDL input pattern. For example, $T = \{t_1, t_2\} = \{(0, 1), (0, 0)\}$ is a test case.
- Branch Point: As we mentioned in 3.1, for $n \in N$, n is a branch point if there is more than one outgoing arc. In a given VHDL CFG, n is a branch point if it is one of the types described in section 3.3.

- T reaches b: Given a branch point b and a test case $T = \{t_1, t_2, \cdots, t_m\}$, T reaches b if b is not covered by $t_1 \cdots t_{m-1}$, but covered by t_m.

The test generation process consists of the following steps:

1. Generate an initial test case $T_0 = \{t_{01}, t_{02}, \cdots, t_{0n}\}$ $(n \geq 1)$ by some technique (e.g. random, functional, etc.). Set $i = 0$.
2. Simulate the VHDL model with T_0 and collect branch coverage statistics, including branch coverage for each t_{0j} $(1 \leq j \leq n)$ and summary branch coverage.
3. While ($i <$ a given number max and $coverage < 100\%$), for select branch point b_k which is not covered yet,
   (a) Determine subsequence $S_m\{t_{i1}, t_{i2}, \cdots, t_{im}\}$ of $T_i = \{t_{i1}, t_{i2}, \cdots, t_{in}\}$ $(m < n)$ such that S_m reaches b which is a branch point we want to cover. Let
   $$S_{m-1} = \{t_{i1}, t_{i2}, \cdots, t_{im-1}\}$$
   (b) Using test generation rules, generate additional input patterns
   $$S_{m'} = \{t'_{im}, \cdots, t_{im'}\}, (m' > m)$$
   (c) The new test sequence concatenates S_{m-1} with patterns in $S_{m'}$
   $$T_{i+1} = S_{m-1} \otimes S_{m'}$$
   (d) Simulate T_{i+1}
   (e) Increment i

We assume that there is an initial test sequence of input patterns (step 1) that has been simulated and for which we have step-by-step coverage information (step 2). As long as the coverage is not complete, and as long as a user-defined number of generation steps has not been exceeded, we do the following: first we determine a branch point we want to cover. Then we determine the subsequence that reaches the branch point (step 3a). We replace the test input pattern of this sequence with a sequence of inputs that according to the heuristic rules, is more likely to execute the uncovered branch. The simulator executes this new test sequence and records coverage information.

5. Experimental evaluation

This section describes some of the results that have been obtained by automating the RUBASTEM generator. Like other heuristics, RUBASTEM cannot be proven to always achieve a certain level of coverage. This depends on the particulars of a specific design. Therefore, empirical evaluation of a heuristic approach is important. The questions about the success of the RUBASTEM method include: Is the method effective at reaching a higher percentage of branch coverage, than other methods? Is the method efficient at generating fewer test patterns for satisfying branch coverage, than other methods? [3] reports the results of the ruled-based method on three types of designs. The test generator was applied to small, medium, and large VHDL examples. The results of these experiments indicate that RUBASTEM is a very effective and efficient method for generating test cases to satisfy the branch criterion.

5.1. Small and medium sized examples

Table 3 describes the information for the small and medium sized VHDL examples and the testing results using random test pattern generation. All examples except Tlc can reach 100% branch coverage using the number of test cases shown in the table. Tlc shows a coverage of 92.86% even after applying 8000 random test patterns. Using random generation of test patterns alone is inefficient and ineffective for Tlc.

Table 4 shows the test results using RUBASTEM. Fewer test cases are used to achieve the same branch coverage as with Random pattern generation. RUBASTEM starts with randomly selected input patterns, then applies the rule-based heuristics to generate more patterns. For Tlc, only 54 test cases are used to reach 100% branch coverage, this is less than 1 percent of the 8000 test cases generated by the Random method which do not achieve 100% coverage.

Next, we investigate what types of rules were used by the test generator. Notice that for Dec5to32, Gcd, Priority, and Tlc that Rule 1 was applied. Rule 1 targets IF/ELSIF/WHILE statements. This rule seems to be effective at executing uncovered branches. Rule 7 is also applied to these designs. This rule generates new test patterns that are the NOT of the previous patterns. This rule tries to increase the number of test patterns that are opposite of patterns that have been already generated in order to execute the more difficult branches. According to Tables 1 and 4, the trend seems to be that more rules are applied as the designs contain a higher cyclomatic complexity, and the trend seems to always include Rules 1 and 7.

5.2. Large example-MINTEL

Another large VHDL model we use is MINTEL. The experimental results are shown in Table 5. From the results we observe that we get the highest coverage if we apply DFA together with the Random method.

The results reported in [3] and the results of the examples reported in this paper indicate that RUBASTEM is very effi-

Model	#Inputs	#Branches	Statement Type	#Patterns	Brcov
Dec2to4	3	7	IF/ELSIF, ELSE, WHEN	16	100%
Dec5to32	6	35	IF/ELSE, ELSE, WHEN	250	100%
Gcd	17	16	IF/ELSE, ELSE, WHILE	129	100%
Priority	7	9	IF/ELSE, ELSE	63	100%
Tlc	3	14	IF/ELSE, ELSE, WHEN	8000	92.86%

Table 3. Results of small and medium sized examples using random method only.

Model	Method	#Patterns (Actual)	Brcov	% of Random	Rules Applied
Dec2to4	rand	1	28.57%		
	dfa1	22 (6)	100%		5
	total	23 (7)	100%	7/16=44%	
Dec5to32	rand	1	8.57%		
	dfa1	37 (36)	100%		1,3,7
	total	38 (37)	100%	37/250=15%	
Gcd	rand 87	1	25%		
	dfa1 87	47 (23)	100%		1,2,7
	total	48 (24)	100%	24/129=19%	
Priority	rand seed=1	3	44.44%		
	dfa1 seed=1	35 (27)	100%		1,7
	total	38 (30)	100%	30/63=48%	
Tlc	rand	1	21.43%		
	dfa1	33	92.86%		0,1,2,3,4,7
	rand	20	100%		
	total	54	100%	54/8000=$<$1%	

Table 4. Test coverage results for small and medium sized examples using random and rule-based methods.

Random + DFA				Random Only			
Phase	#P	CC	Brcov	Phase	#P	CC	Brcov
Rand	300	cc1	27.97%	Rand	300	cc1	27.97%
DFA1	895	cc1	30.51%	Rand	895	cc1	28.39%
DFA1	895	cc2	31.78%	Rand	895	cc2	31.36%
Rand	10000	cc1	43.64%	Rand	10000	cc1	43.64%
Rand	10000	cc2	45.34%	Rand	10000	cc2	44.92%

Table 5. Testing results for MINTEL model.

cient at generating new test patterns for uncovered branches. The method is able to target uncovered branches and generate test patterns based on rules described in [3].

6. Conclusion

This paper applies the heuristic rules defined in [3] to more designs. The results of the experiments indicate that applying randomly generated test patterns and RUBASTEM patterns increase the branch coverage of a behavioral VHDL design. The test generator is able to produce new test patterns based on reaching definitions and the coverage results of previously generated patterns. The interesting cases for generating test patterns occur when the coverage begins to saturate. However, RUBASTEM is able to intelligently generate new test patterns that increase the chances of covering more branches. The automated test generation of patterns has clear advantages over random generation of patterns. The test generator producers fewer test patterns than random generation alone.

We plan to investigate whether and how VHDL code metrics can be used to generate effective new test patterns. Candidate metrics include cyclomatic complexity, the number of concurrent constructs, and the number of entities. New heuristic rules may be able to be derived based on the collected metrics and the control graph of the VHDL model. This paper does not infer any techniques for comparing the actual output of the circuit described by the design to the desired output of the circuit. Ultimately, defining new software metrics, testing methods, and combinations of testing methods may lead to higher branch coverage with fewer test patterns. Further research is required to derive new verification methods and heuristic rules, which generate more code coverage with less test cases than existing rules and techniques.

References

[1] M. Abrahams, S. Riches. Optimize ASIC Test Suites Using Code-Coverage Analysis. EDN, May 21, 1998.

[2] C. Anderson, A. von Mayrhauser, R. Mraz. Assessing Neural Networks as Guides for Testing Activities. *Proc. of the 3rd International Symposium on Software Metrics: From Measurement to Empirical Results*, March 1996.

[3] A. Andrews, A. O'Fallon, T. Chen. A Rule-Based Software Testing Method for VHDL Models. *Proc. IFIP International Conf. of VLSI SoC*, Darmstadt, Germany, December 1–3, 2003.

[4] D. P. Appenzeller, A. Kuehlmann. Formal Verification of a PowerPC Microprocessor. *Tech. Report RC (19971)*, IBM T.J. Watson Research Center, Yorktown Heights, NY, March 1995.

[5] B. Barrera. Code Coverage Analysis - Essential To A Safe Design. *Electronic Engineering*, November 1998.

[6] W. H. Deason, D. B. Brown, K. Chang, J. H. Cross II. A Rule-Based Software Test Data Generator. *IEEE Transactions on Knowledge and Data Engineering*, 3(1), March 1991.

[7] T. A. DeLong, B. W. Johnson, J. A. Profeta III. A Fault Injection Technique for VHDL Behavioral-Level Models. *IEEE Design & Test of Computers*, 13(4):24–33, 1996.

[8] R. Kapur, S. Patil, T. J. Snethen, T. W. Williams. A Weighted Random Pattern Test Generation System. *IEEE Transaction On Computer-Aided Design Of Integrated Circuits And Systems*, 15(8), August 1996.

[9] Y. K. Malaiya. Antirandom Testing: Getting the Most out of Black-Box Testing. *Procs. ISSRE '95*, Toulouse, October 1995.

[10] T. J. McCabe, C. W. Butler. Design Complexity Measurement and Testing. *Communications of the ACM*, 32(12):1412–1425, December 1989.

[11] L. Nachman, K. Saluja, A. Upadhyaya, R. Reuse. A Novel Approach to Random Pattern Testing of Sequential Circuits. *IEEE Transactions on Computers*, 47(1), January 1998.

[12] D. G. Saab, Y. G. Saab, J. A. Abraham. Automatic Test Vector Cultivation for Sequential VLSI Circuits Using Genetic Algorithms. *IEEE Transaction On Computer-Aided Design of Integrated Circuits and Systems*, 15(10), October 1996.

[13] R. Vemuri, R. Kalyanaraman. Generation of Design Verification Tests from Behavioral VHDL Programs Using Path Enumeration and Constraint Programming. *IEEE Trans. VLSI Systems*, 3(2):201–214, June 1995.

[14] A. von Mayrhauser, A. Bai. Fast Antirandom (FAR) Test Generation. *3rd IEEE high Assurance Systems Eng. Symp.*, IEEE Computer Society Press, Los Alamitos, Calif., 1998.

[15] H. Zima, B. Chapman, *Supercompilers for Parallel and Vector Computers*. ACM Press, New York, NY, 1991.