A Data Flow Fault Coverage Metric For Validation of Behavioral HDL Descriptions

Qiushuang Zhang and Ian G. Harris
Department of Electrical and Computer Engineering
University of Massachusetts
Amherst, MA 01003
qzhang@ecs.umass.edu, harris@ecs.umass.edu

Abstract—

Behavioral HDL descriptions are commonly used to capture the high-level functionality of a hardware circuit for simulation and synthesis. The manual process of creating a behavioral description is error prone, so significant effort must be made to verify the correctness of behavioral descriptions. Simulation-based validation and formal verification are both techniques used to verify correctness. We investigate validation because formal verification techniques are frequently intractable for large designs. The first step toward a behavioral validation technique is the development of a *validation fault coverage metric* which can be used to evaluate the likelihood of design defect detection with a given test sequence.

We propose a validation fault coverage metric which is based on an analysis of the control data flow description associated with the behavior. The proposed metric identifies a subset of paths through the data flow which must be traversed during testing to detect faults. The proposed metric is a tractable compromise between the statement coverage metric which requires only that each statement be executed, and the path coverage metric which requires that all data flow paths be executed. Data flow paths are identified based on the relative code locations of definitions and uses of variables which may be assigned incorrectly due to a design error. We propose an efficient method to compute all data flow paths which must be traversed, and we generate coverage results for several benchmark VHDL circuits for comparison to other approaches.

I. Introduction

Design validation by simulation-based techniques is the most common approach to verification due to the computational complexity of formal techniques. Validation entails the generation of a test pattern sequence which is applied to the design during simulation to trigger erroneous behavior. A key problem in behavioral validation is how to measure the quality of test vectors. Unlike in the area of manufacturing testing, there does not yet exist a metric which is widely acceptable by the validation community. Some metrics are taken from software testing, such as statement coverage, branch coverage and path coverage. Statement coverage and branch coverage metrics are overly simplistic and cannot reveal sophisticated hardware description language (HDL) faults. Path coverage is a more stringent metric, however the requirement that all control paths be explored makes this metric very pessimistic.

A validation fault model has been developed at the finite state machine level [1] which assumes that each error affects either a single state transition or a single transition output. A behavioral level fault model has been proposed in [2] and [3] which assumes that any single variable assignment in a behavioral description may be incorrect. Mutation analysis has been used for hardware validation previously in [4] by converting a VHDL program into a functionally equivalent Fortran program and then using the Mothra tool for software mutation analysis [5].

Complex hardware systems are commonly described by a behavioral hardware description as a first step in the synthesis process. A design error in a behavioral description must be activated and propagated to a primary output by a test sequence in order to be detected. The activation and propagation of a behavioral design error is associated with a subpath of the control data flow graph which starts with the node where the fault is activated, and ends at a node leading to a primary output. A major difficulty in behavioral validation is the identification of a minimal set of subpaths of the data flow which must be executed to detect all validation faults. Several validation fault coverage metrics have been proposed previously in the area of software testing which identify paths based on their length. For example, statement coverage requires that all statements are executed, effectively executing all paths of length 1. At another extreme, path coverage requires that all paths of maximum length be executed. Path length has been used previously because it limits the complexity of the validation process, but it is only grossly correlated to fault detection. Information embedded in the data flow graph can be used to screen out a large number of unnecessary subpaths, independent of path length.

A large body of research in software testing has studied the use of data flow analysis in this context. The primary goal of data flow analysis is the identification of paths in which a variable is assigned to a value which may be faulty, and that variable is then used to perform an operation. Several test adequacy criteria based on data flow analysis have been developed [6], [7], [8], [9], [10] based on these ideas. The basic criteria include all definitions criterion, all uses criterion and all definition-use-paths criterion. These criteria are concerned mainly with the simplest type of data flow paths that start with a definition of a variable and end with a use of the same variable. We have adapted some of these techniques for the validation fault coverage analysis of behavioral hardware descriptions. The most significant difference between software and hardware descriptions is the need to model the passage of real time in hardware. This is performed in a behavioral HDL through the use of time-varying *signals* and concurrency constructs such as the *process* in VHDL. Data flow analysis for hardware must consider the data flow relationships between multiple processes, and between different instantiations of the same process.

We propose a validation fault coverage metric for behavioral VHDL based on data flow analysis which evaluates the data flow coverage achieved by using a given test pattern sequence. Data flow analysis is an efficient method to reveal design faults. Our approach consists of three steps: First, we generate the flow graph of an HDL description. Second, we select a subset of data flow paths which are required to be executed. Third, the HDL description is simulated

with the candidate test patterns to determine the fraction of required subpaths which are executed.

The remainder of this paper is organized as follows: Section 2 introduces our data flow metric for HDL descriptions and presents our approach to evaluate data flow coverage. Section 3 and 4 presents the results and conclusions respectively.

II. Data flow analysis for HDL descriptions

A. Definition

Data flow analysis for HDL descriptions is concerned with the occurrences of signals and variables in a HDL description. Each signal or variable occurrence in a VHDL description is classified as either a definition occurrence or a use occurrence according the classification in software testing. A *definition occurrence* of a signal or variable describes a statement where a value is bound to the signal or variable. A *use occurrence* of a signal or variable describes a statement which refers to the value of the signal or variable. This occurrence information is added to the flow graph representation as a preprocessing step to facilitate data flow analysis. A number of conventions of flow graph models can be used. In the flow graph model we use, each node represents a single statement in the description and edges represent statement execution order. The flow graph represents not only the computation information, but also relative timing and concurrency information using the *process* statement.

Concurrency adds complexity to data flow analysis of hardware when compared to data flow analysis of sequential software programs. In general, an HDL description has both concurrent statements (e.g. signal assignment) and sequential statements (e.g. variable assignment), timing information (e.g. $X <=' 1'$ *after 1 ns*;), and suspending and resuming control (e.g. "wait" statement, sensitivity list). These features allow complex relationships between operation execution times. For example, a sequential statement executes and finishes instantly, an assignment to a signal X ($X <=' 1'$ after 1 ns;) takes 1 ns to finish, a "wait until" statement takes a variable amount of time to finish. This timing information is unique in an HDL description and must be represented in the flow graph. In a multiple processes description, inter-process communication further complicates the data flow. Since the execution of a process may resume other processes dynamically, a *du* pair can lie in two different processes, and definition occurrence and use occurrence can take place simultaneously, etc.. All this timing information which is unique to an HDL description must be represented in the flow graph.

Example 1: This example shows the flow graph of a VHDL description with simple timing information. A signal assignment statement does not finish instantly, it takes some time to complete the assignment. In this example, since no delay time specified in the statements, the assignment will finish at the end of simulation cycle in which each process executes until next "wait" statement. The VHDL description is as follows:

```
 PROCESS BEGIN
1   wait until clock'event and clock = '1';
2   if(B = '0') then
3      Zout <= P + Q;
4   else
5      Zout <= P - Q;
6   end if;

7   if(A = '0') then
8      P <= C + D;
9   else
10     P <= C - D;
11  end if;
 END PROCESS;
```

Figure 1 shows the data flow representation with timing information of the description. The occurrences of signals are omitted except that of signal *P*. The signals *A*, *B*, *Q* and *clock* are primary inputs and *Zout* is primary output. There are two types of nodes and two types of edges in the graph. The square nodes represents a time point at which statements finish, so we refer square nodes as timing nodes which execute and finish instantly, like sequential statements. The circular nodes are referred as computation nodes, each of which corresponds to a statement in the description. The solid edges in the flow graph represent sequential execution order; the head node of the edge begins to execute after the tail node completely finishes. The dashed edges represent concurrent execution sequence; the head node begins to execute just after the tail node begins to execute and does not finish immediately. In this example, no delay information is given to the assignment statements, so the execution of assignments finish at the end of the current simulation cycle, then go to next "wait" statement. The node "End Simulation Cycle" is a timing node.

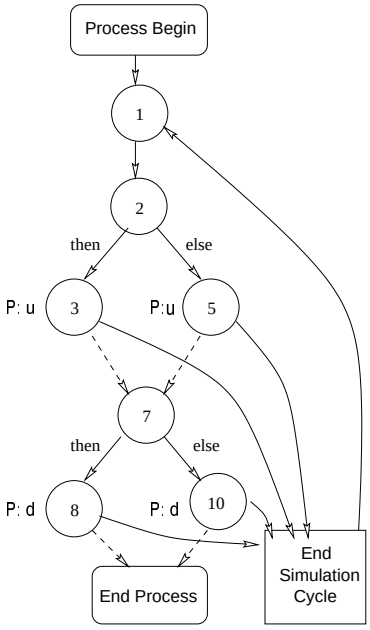

Fig. 1. Flow graph in example 1.

Based on the flow graph model introduced above, a *definition clear path* with respect to signal or variable X is a path in the flow graph without definition occurrence of X. A *definition-use (du) pair* of signal or variable X consists of a definition and a use of variable X which are connected by a definition clear path with respect to X, from the definition to the use. If a *du* pair is exercised in the definition-use sequence by some test patterns, then we say this du pair is covered by the test patterns. *All definition-use (du) pairs metric* [6] requires that all *du* pairs be covered by the test patterns, i.e. every definition to every use of that definition should be exercised. In Figure 1, there are four *du* pairs of variable *P*, $(8 \rightarrow 3)$, $(8 \rightarrow 5)$, $(10 \rightarrow 3)$ and $(10 \rightarrow 5)$, and these *du* pairs are required to be executed by all *du* pairs metric.

Compared to the path coverage metric, all *du* pairs coverage has weaker fault detection ability, but requires much smaller number of test cases because *du* pairs are not redundantly executed. Compared to branch and statement coverage, all *du* pairs coverage has stronger

```
Process(clock)
   variable X, Zout: integer;
begin
1    if A = '0' then
2       X := in1;  -- good version X := 2*in1;
3    else
4       X := in2;
5    end if;

6    if Z = '0' then
7       if X = 0 then
8          Zout := 1;
9       else
10         Zout := 2;
11   end if;
12   else
13      if X > 2 then
14         Zout := 3;
15      else
16         Zout := 4;
17      end if;
18   end if;
end process;
```

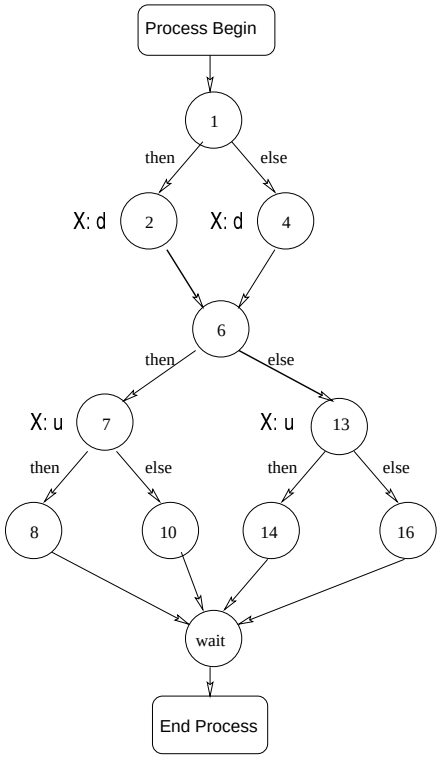

Fig. 2. VHDL description and the flow graph in example 2.

fault detection ability, but requires more test cases than branch and statement coverage because a *du* pair may consists of multiple branches and statements. All *du* pairs coverage represents a tractable tradeoff between test complexity and effectiveness.

B. Validation Fault Assumption

Any validation fault coverage metric must target some set to design faults for detection. The targeted fault set impacts the complexity of the fault metric, as well as the accuracy of the fault metric. Statement coverage assumes the faults occur at a single statement, branch coverage assumes the faults occur at a single branch. In the data flow graph model of a HDL description, a fault can be associated with a set of paths which allow the fault to be detected. All *du* pairs metric assumes a fault effect occurring at a variable definition may only by propagated by a subset of use nodes. In order to ensure propagation of all faults, all *du* pairs must be executed.

Example 2 is shown in Figure 2. The variable X has four *du* pairs, $(2 \rightarrow 7)$, $(2 \rightarrow 13)$, $(4 \rightarrow 7)$ and $(4 \rightarrow 13)$. A fault is injected at node 2, where the correct version of node 2 is "$X := 2 * in1$;". This fault will affect only *du* pair $(2 \rightarrow 13)$, so it can be detected only by the execution of the specific *du* pair $(2 \rightarrow 13)$ via the highlighted path in the flow graph in Figure 2. Execution of the other three *du* pairs can cover all statement and all branches, but cannot detect the fault. This example also illustrates that all *du* pairs metric is stronger than branch coverage and statement coverage.

C. Approach to all du pairs coverage.

Our approach to all *du* pairs coverage consists of three steps:
1. Data flow representation. This step is to generate the data flow representation of a HDL description. Timing information should be represented on the flow graph. At this stage, our implementation can only deal with a subset of VHDL language, such as IF, CASE, LOOP statements, signal and variable assignments without delay and sensitivity lists. Flow graphs of a single process description is similar to the graph in Figure 1 except the 'wait' node is at the end of the process. A concurrent statement will have a solid edge directed to the 'End Simulation Cycle' node and a dashed edge pointing to the next statement without finishing the current statement. 'wait' statements are the points at which processes suspend and resume, and act as inter-process control points. Sensitivity lists of multiple processes control the execution sequence between the processes. If a signal on the sensitivity list of process *B* is assigned in process *A*, the execution of process *A* will cause the execution of process *B*.
2. du pairs identification. After generation of the flow graph of a description, all *du* pairs are identified. There are no *du* pairs involving input and output signals since inputs cannot be assigned a value and outputs can not be used inside descriptions. In a single process, if there is a definition clear path between a *du* pair, then this *du* pair is valid. Note that the outgoing edge of definition node must be a solid edge which means that the definition is complete. In a multiple processes flow graph, a *du* pair may lie between two processes. To identify these inter-process *du* pairs, we first find the synchronization points between processes, including a common signals in sensitivity lists. The search for *du* pairs continues in a similar manner to the single process version by treating the synchronization points as edges between the data flow graphs of the communicating processes.
3. Coverage determination. The behavioral description is simulated with candidate test patterns to determine which *du* pairs are exercised. Timing information adds difficulties to this step because first execution does not necessarily imply first completion. A *du* pair is covered only if the definition is complete before its use.

benchmark	# of statements	# of du pairs	du pairs cov.	statement cov.
ARMS_COUNTER	32	69	0.87	1
BARCODE	44	68	0.69	0.95
TLC	38	47	1	1
BUS_ARBITER	24	45	0.78	1
FIFO	59	92	0.86	1

TABLE I

EXPERIMENTAL RESULTS OF ALL-USES METRIC COMPARED WITH STATEMENT COVERAGE

III. EXPERIMENT RESULTS

We have evaluated our data-flow fault model by computing the all *du* pairs coverage of several behavioral VHDL descriptions and comparing the coverage to statement coverage. The first three examples are ARMS_COUNTER, BARCODE and TLC from the HlSynth92 benchmark suite. FIFO comes from web site of www.vhdl-online.de sponsored by University of Erlangen-Nurnberg. ARMS_COUNTER, BUS_ARBITER and FIFO are multiple processes descriptions. The coverage results are shown in table I. Columns 2 and 3 contain the number of statements which contains signals or variables definitions or uses, and the number of *du* pairs respectively. All *du* pairs metric results are listed in column 4 and statement coverage results are listed in column 5.

To compare statement coverage and all *du* pairs coverage, we select sufficient pseudo-random test patterns to make the statement coverage close to 1. In the ARMS_COUNTER, BARCODE, BUS_ARBITER and FIFO examples, the all *du* pairs coverage is lower than statement coverage which means some test cases are not exercised by test patterns. If faults are in these un-exercised flows, then they cannot be detected by the test patterns. In the TLC example, the coverage of both metrics are 1. This is a special case in which most signals are used only in one statement, so the number of *du* pairs is close to the number of statements. Therefore when all statements are exercised, the all *du* pairs are exercised as well.

The average all *du* pairs coverage over five examples is 0.84 while average statement coverage is 0.99, which means that after statement coverage goal achieved, approximately 16% *du* pairs are left un-executed. Examining these examples, we find that these un-executed *du* pairs are primarily corner cases associated with hard-to-test design faults. For example, in the FIFO benchmark, the case in which the *read* signal is asserted after the *reset* signal is asserted is ignored by the statement coverage metric. Since this case is associated with a *du* pair, the all *du* pair metric reflects that this case is executed.

IV. SUMMARY

We use data flow testing techniques to define a fault coverage metric which enables efficient evaluation of test patterns for behavioral hardware validation. The use of the all *du* pairs metric is based on data flow analysis and shows strong potential in examining data flow faults. However, more investigation is needed to identify infeasible *du* pairs, and to consider observability issues on long data flow paths. We provide all *du* pairs coverage results for several VHDL benchmarks to demonstrate the utility of the approach.

REFERENCES

[1] A Gupta, S. Malik, and P. Ashar, "Toward formalizing a validation methodology using simulation coverage", *in Design Automation Conference*, pp. 740–745, 1997.

[2] F. Fallah, P. Ashar, and S. Devadas, "Simulation vector generation from hdl descriptions for observability enhanced-statement coverage", *in Proceedings of the 36th Design Automation Conference*, pp. 666–671, 1999.

[3] S. Devadas, A. Ghosh, and K. Keutzer, "An observability-based code coverage metric for functional simulation", *in International Conference on Computer-Aided Design*, pp. 418–425, 1996.

[4] G. Al Hayek and C. Robach, "From specification validation to hardware testing: a unified method", *in International Test Conference*, pp. 885–893, 1996.

[5] K. N. King and A. J. Offutt, "A fortran language system for mutation-based software testing", *Software Practice and Engineering*, vol. 21, pp. 685–718, 1991.

[6] S Rapps and E. J. Weyuker, "Selecting software test data using data flow information", *IEEE Trans. on Software Engineering*, vol. SE-11, pp. 367–375, April 1985.

[7] P. G. Frankl and J. E. Weyuker, "An applicable family of data flow testing criteria", *IEEE Trans. on Software Engineering*, vol. SE-14, pp. 1483–1498, Oct. 1988.

[8] S. C. Ntafos, "A comparison of some structural testing strategies", *IEEE Trans. on Software Engineering*, vol. SE-14, pp. 868–874, 1988.

[9] J. Laski and B. Korel, "A data flow oriented program testing strategy", *IEEE Trans. on Software Engineering*, vol. SE-9, pp. 33–43, 1983.

[10] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil, "A formal evaluation of data flow path selection criteria", *IEEE Trans. on Software Engineering*, vol. SE-15, pp. 1318–1332, 1989.