

An Efficient Control-Oriented Coverage Metric

Shireesh Verma
shireesh@cecs.uci.edu

Kiran Ramineni
kiran@cecs.uci.edu

Ian G. Harris
harris@ics.uci.edu

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697, USA

ABSTRACT

Coverage metrics, which evaluate the ability of a test sequence to detect design faults, are essential to the validation process. A key source of difficulty in determining fault detection is that the control flow path traversed in the presence of a fault cannot be determined. Fault detection can only be accurately determined by exploring the set of all control flow paths, which may be traversed as a result of a fault. We present a coverage metric that determines the propagation of fault effects along all possible faulty control flow paths. The complexity of exploring multiple control flow paths is greatly alleviated by heuristically pruning infeasible control flow paths using the algorithm that we present. The proposed coverage metric provides high accuracy in designs that contain complex control flow. The results obtained are promising.

1. INTRODUCTION

Design validation is the task of verifying design correctness by simulating (or emulating) it with a test sequence and comparing system responses to known correct responses. Verification of design correctness is known to be a costly task, and validation is a technique for verification, which is well accepted due to the tractability and usability of the simulation process. Validation involves several steps, such as test generation and response evaluation, but coverage metrics are central to all steps in the validation process. A coverage metric defines a set of criteria that are used to determine which faults are detected by a test sequence. A coverage metric provides an empirical measure of the completeness of a test sequence and the fault detection criteria can be used to direct the test generation process.

Most existing behavioral coverage metrics focus on defining the criteria for *fault activation*, placing little emphasis on *fault propagation*. For example the statement coverage metric requires that each statement is executed in order to activate each potentially faulty statement, but propagation of the fault effect to an observable point is not considered. Determining fault propagation exactly would require specific knowledge of the design error in order to compare the faulty machine response to the correct machine response. Exact evaluation of fault propagation is feasible at the gate-level because each signal value is binary; the value of a signal with a fault effect is known to be the inverse of the value of the same signal in the correct circuit. In a behavioral description more complex data types are used which may have very large domains of possible values. As a result, exact

fault propagation is intractable at the behavioral level and some technique must be used to estimate fault propagation using partial fault effect information.

A significant difficulty in predicting fault propagation is the ambiguity in the control flow path executed in the presence of a fault. For example, consider the evaluation of the conditional statement “if ($x > 5$) then ...” in the presence of an unknown fault effect on variable x . If the correct value of x is 0 then the correct value of the conditional predicate is FALSE, but the incorrect predicate value is unknown. The control flow choices at branch points have a major impact on the sequence of instructions executed and the propagation of fault effects. In the simple conditional predicate example above, if the predicate evaluates to FALSE in the faulty design, the fault effect may not propagate, and the fault may remain undetected as a result. The problem of determining fault propagation in the presence of ambiguity in control flow quickly becomes severe as control flow becomes more complex.

Our approach explores all control flow paths which could be executed as a result of a fault. Fault detection is estimated along each control flow path and the detection probabilities along each path are combined to compute total fault detection probability. To alleviate the computational complexity of exploring multiple control flow paths, we use knowledge of the sign of a fault effect to prune infeasible paths, which could never be executed as a result of a fault. Our approach provides an accurate estimate of fault detection probabilities by considering the full range of faulty behaviors.

The remainder of the paper is organized as follows. Related work is summarized in Section 2. The Overview of our approach is summarized in Section 3. The method used to evaluate fault propagation is discussed in Section 4. The technique used to prune infeasible control flow paths is presented in Section 5. We define Fault Detection Probability in Section 6. Section 7 discusses tools used for our experiments. The results are presented in Section 8 and conclusions are described in Section 9.

2. PREVIOUS WORK

Mutation analysis is a coverage metric which was originally developed in the field of software test [6] and has also been applied to hardware validation [5]. A *mutant* is a version of a behavioral description which differs from the original by a single potential design error. A *mutation operator* is a function which is applied to the original program to generate a mutant. A typical mutation operation is *Arithmetic*

Operator Replacement (AOR), which replaces each arithmetic operator with another operator. The local nature of the mutation operations may limit their ability to describe a large set of design errors.

Several coverage metrics have been developed to evaluate behavioral designs, many of which are based on software test coverage metrics. Metrics have been developed to target faults in finite state machines (FSMs). The commonly used FSM coverage metrics [2, 10, 9] are the *state coverage* model, which requires that all states be reached, and *transition coverage*, which requires that all transitions be traversed.

A number of validation coverage metrics are based on the traversal of paths through the control dataflow graph (CDFG) representing the system behavior. Applying these metrics to the CDFG representing a single process is a well understood task. The earliest control-dataflow fault models include the statement coverage, branch coverage and path coverage [1] metrics used in software testing.

The branch coverage metric associates potential faults with each direction of each conditional in the CDFG. Branch coverage requires that the set of all CDFG paths covered during validation include both directions of all binary-valued conditionals. The branch coverage metric has been used for behavioral validation by several researchers for coverage evaluation and test generation [12, 3].

The path coverage metric is a more demanding metric than the branch coverage metric because path coverage reflects the number of control-flow paths taken. The assumption is that an error is associated with some path through the control flow graph and all control paths must be executed to guarantee fault detection. The number of control paths can be infinite when the CDFG contains a loop so the path coverage metric may be used with a limit on path length [11]. Previous work in software test [7] has investigated *dataflow testing* criteria for path selection. In dataflow testing, each variable occurrence is classified as either a definition occurrence or a use occurrence. Paths that connect a definition occurrence to a use occurrence of the same variable are selected. The dataflow testing criteria have also been applied to behavioral hardware descriptions [13].

Many control-dataflow coverage metrics consider the requirements for fault activation without explicitly considering fault effect observability. Researchers have developed observability-based behavioral fault models [4, 8] to alleviate this weakness. The OCCOM approach [4] inserts faults, called *tags*, at each variable assignment to represent a positive or negative offset from the correct signal value. The propagation of these tags to observable variables is determined using a set of propagation rules for behavioral operations. In [8], the software dataflow testing technique is enhanced to identify chains of variable definitions and uses which extend to observable variables. Both of these observability-oriented metrics have difficulty considering the multitude of possible control flow paths, which may be executed in the presence of a fault. Either complex control flow has not been considered or only a small subset of possible control flow paths are considered.

3. OVERVIEW

The inputs to our methodology are a behavioral HDL description of the design and a set of test patterns and the output is the set of faults that propagate to an observable

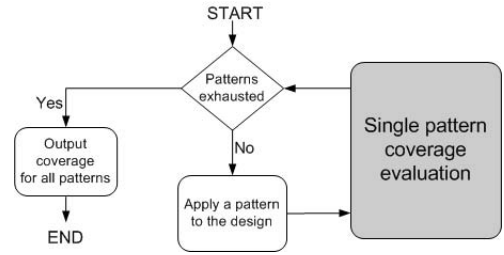


Figure 1: System structure

point for every input pattern. An observable point in a simulation is a variable that may get an incorrect value because of a design fault.

Figure 1 depicts the way we compute the fault coverage value for test patterns. We first automatically generate a list of potential faults in the design. We then simulate the HDL description with each test pattern one at a time until all the generated patterns are exhausted. The fault coverage value for each pattern is computed at the end of its simulation.

Figure 2 illustrates a sequence of steps we follow to compute the fault coverage for each test pattern. We select an arbitrary fault from the generated list and then simulate the HDL description to determine the propagation of the fault. We evaluate its detection probability at the end of the simulation. This sequence of steps is repeated until all the faults in the list are exhausted. The fault coverage value for the given test pattern is computed on the basis of the detection probabilities of the individual faults.

3.1 Fault List Generation

We use the tag model presented in [4] for injecting faults and propagating them. A tag on a variable represents the notion that the variable has the incorrect value in presence of the fault.

We consider every assignment statement and every input variable in the HDL description as potential fault injection points. The tags are injected on to the left hand side (LHS) variable of each assignment statement and on each of the input variables. We parse the HDL description to obtain a list of all the assignment statements and input variables. There are following three types of tags defined:

1. Positive tags: These cause the LHS variable to have a greater value than the correct one.
2. Negative tags: These cause the LHS variable to have a lesser value than the correct one.

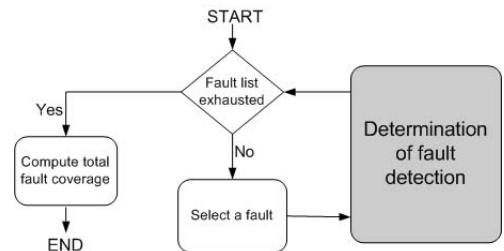


Figure 2: Single pattern coverage evaluation

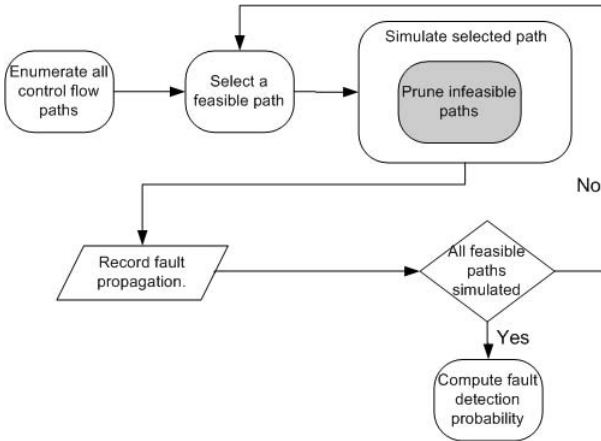


Figure 3: Determination of fault detection

- Unknown tags: These cause the LHS variable to have a greater or lesser value than the correct one.

Since there are three tags which could be injected to the LHS variable and input variable, we have three potential design faults for every assignment statement and input variable encountered. Our fault list essentially consists of a number of faults that is three times the number of assignment statements and input variables in the HDL description.

4. EVALUATION OF FAULT PROPAGATION

The simulation and evaluation of a single fault is depicted in the Figure 3. We compute all the possible control flow paths and maintain a list for these pre-determined paths. We select an arbitrary path from the list, and then the simulation is guided through that path. During each simulation, a fault is injected by inducing a tag using the simulator's directives. During the simulation, we monitor the observable points where an injected tag could manifest itself. This evaluation is repeated at every time step of the simulation in order to propagate a fault.

At the simulation of each statement, the fault propagation is updated. Each injected fault propagates in one of two ways through the behavior.

```

1 begin
2 reg x;
3 integer a;
4 x = 5;
5 a = 2;
6 x = x + a;
7 end
  
```

Figure 4: A simple data flow example

Data flow Propagation A fault can propagate from one variable to another via a direct data dependency between the two variables created by a variable assignment. For example, in the code shown in Figure 4 a tag on variable *a* would propagate to variable *x* as a result of the assignment on line number 6.

<i>a/b</i>	<i>b</i>	<i>b-</i>	<i>b+</i>	<i>b?</i>
<i>a</i>	0	-	+	?
<i>a-</i>	-	-	?	?
<i>a+</i>	+	?	+	?
<i>a?</i>	?	?	?	?

Table 1: Tag calculus for addition

```

1 case(cond)
2   1: x = 5;
3   2: x = 1;
4   3: x = 10;
5 endcase
  
```

Figure 5: A simple control flow example

Previous researches have defined a calculus for tag propagation through each type of behavioral operation[4]. The propagation table used for an addition operation is shown in Table 1. The top most row in the table represents the tags on *b*, while the left most column represents the tags on *a*. +, - and ? signify a positive, a negative and an unknown tag respectively and a 0 signifies absence of a tag. Variables *a* or *b* followed by 0, +, - or ? signify the presence of tags on these variables.

We use the tag propagation calculus presented in [4] to perform dataflow propagation.

Control flow Propagation It is possible for a fault to change the executed control flow path by changing the result of a conditional. Changing the control flow path can have drastic effects on fault propagation by introducing indirect variable dependencies and by changing the sequence of dataflow operations.

An indirect data dependency can be seen in Figure 5 between variables *cond* and *x*. Although there is no dataflow dependency between the two variables, it is clear that the value of *cond* determines the value of *x*. If the correct value of the *cond* variable is 1 and it has a positive tag, then either branch 2 or 3 could be taken. The sign of tag on *x* depends on the magnitude of the error on *cond*. *cond* = 2 creates a negative tag on *x* and *cond* = 3 creates a positive tag on *x*.

When a tag changes the result of a conditional predicate, it alters the sequence of dataflow operations executed. This consequently changes all subsequent dataflow propagation. The example shown in Figure 5 is simple, but each branch may contain a subprogram of arbitrary complexity. Fault propagation depends on the sequence of dataflow operations, so any change to that sequence will also alter the dataflow fault propagation.

In order to determine the impact of tags which affect control flow, it is necessary to simulate each feasible path. The number of control flow paths is exponential in the number of conditionals.

At the end of simulation of each feasible control flow path the tag propagation data is recorded and the tags on the variables are stored. The propagation data is the set of variables that contain tags at the end of simulation. The stored tag values at the end of each simulated feasible control flow path are used to determine the tags on the variables at the start of simulation of next feasible control flow path. This process is repeated until there are no feasible paths left to simulate in the control flow path list. Once all the paths in the list are simulated, the fault detection probability for the injected fault is computed using the fault propagation data recorded for individual paths.

5. PRUNING INFEASIBLE PATHS

If a tag is present on a variable involved in a conditional predicate, then we determine whether or not the outcome of the predicate depends on the magnitude of the tag. The alternate control flow paths can then be pruned from consideration, based on our determination. This pruning can greatly reduce the time complexity of this approach. In this section, we demonstrate how a decision tree is built from an HDL description and how pruning is performed on the resulting tree.

5.1 Generating Decision Tree

We generate a decision tree from the HDL description at the parsing step. A path through the decision tree from root to leaf corresponds to a control flow path in the HDL description. The tree consists of following two types of nodes.

Condition node Each condition node corresponds to a control flow decision that is made during simulation. A single conditional predicate in the HDL may map to many decision nodes since each predicate may be evaluated on many control flow paths. Each condition node has a number of children corresponding to the number of outcomes of the conditional predicate. A condition node corresponding to an IF.Else statement will have two children, TRUE and FALSE. A condition node corresponding to a case statement will have as many children as the case has branches. These nodes are represented as white nodes.

Leaf node Leaf nodes terminate the paths in the decision tree. The path from the root to each of these nodes represents one control flow path through the HDL description. These nodes are represented as filled black nodes.

The decision tree for the HDL description in Figure 6 is shown in Figure 7.

A sequence of nodes starting with the top most condition node and ending at a leaf node of the tree constitutes a control flow path. This signifies that there is a one-to-one mapping between the leaf nodes and the control flow paths. In this case, we have six control flow paths each corresponding to a leaf node. These control flow paths are used to guide the simulation.

5.2 Pruning Algorithm

It is possible to reduce the number of control flow possibilities to be considered by using dynamic tag information computed during simulation. A child c of a conditional node n can be pruned if the predicate $P(n)$ associated with

```

1 begin
2 a = 5;
3 b = a+5;
4 if (b > 10)
5 begin
6 if (a < 5)
7 out = 1 + a;
8 else
9 out = 1;
10 end
11 else
12 out = 1 - a;
13 if (out < a)
14 out = 0;
15 else
16 out = 1;
17 end

```

Figure 6: HDL example for illustrating pruning

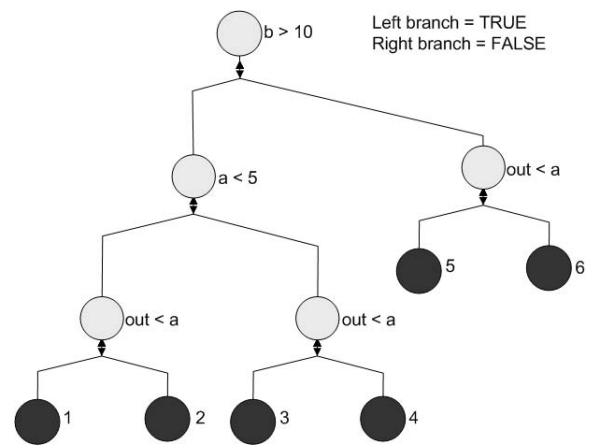


Figure 7: Decision tree for HDL example

node n can never evaluate to the value required to lead to child c in the presence of the fault. For example, the root node in Figure 7 has two children, one associated with a TRUE predicate value and one associated with a FALSE predicate value. If during simulation the variable b has the value 11 and b has a positive tag, then it is not possible for the predicate to evaluate to FALSE in the presence of the fault. In this case, the child in the FALSE direction can be pruned from consideration without reducing the accuracy of the approach.

When simulation encounters a conditional predicate, the following steps are executed to determine if pruning can be performed.

1. **Determine the set of possible faulty predicate values, V_f** - This step is performed by limiting the range of the variables in accordance with the signs of their tags and determining if the predicate is satisfiable or not. This problem in the worst case is a version of the SATISFIABILITY problem, but in practice it is trivial given the monotonic nature of the vast majority of predicates in real examples.
2. **Prune children corresponding to impossible predicate values** - Once the possible predicate values are

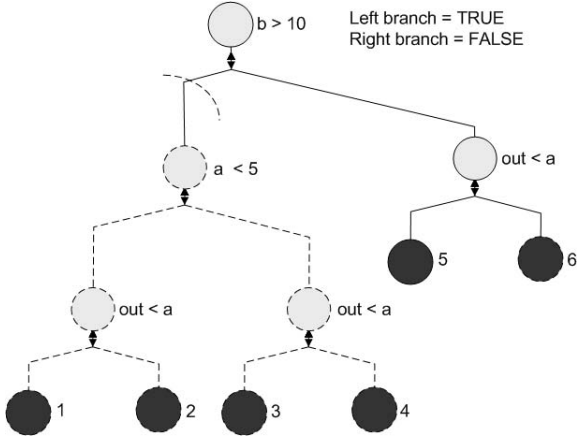


Figure 8: Pruning case with negative tag on a

known, all children of the decision node that can be reached only by impossible predicate values can be safely pruned.

We will use the example used in Figure 6 in order to illustrate the pruning of the decision tree in Figure 7. Let us consider the following cases of tag injection on variable a .

Negative tag at line number 2 This results in a negative tag for b at the assignment statement $b = a + 5$ at line number 3. The condition node $b > 10$ evaluates to FALSE as there is a negative tag on b and its value is 10. This allows us to prune TRUE branch, which would never be taken in presence of the fault. In the FALSE branch, assignment statement $out = 1 - a$ at line number 12 is evaluated. out gets a value -4 and a positive tag since a which has negative tag, is being subtracted. The value of the next condition $out < a$ can not be uniquely determined because it depends on the relative magnitudes of tags out and a .

For example, $out < a$ would evaluate to TRUE when magnitudes of tags on out and a are 1 and 1 respectively in which case the resultant expression becomes $((-4 + 1) < (5 - 1))$, which evaluates to TRUE. However, when the magnitudes of the tags are 5 and 6 respectively for out and a , the resultant expression $((-4 + 5) < (5 - 6))$ evaluates to FALSE. Pruning can not occur at the decision node $out < a$ on account of ambiguity resulting from the magnitudes of the tags.

The pruned tree is depicted in Figure 8. The dashed lines represent the pruned part of the tree.

Positive tag at line number 2 This results in a positive tag for b at the next assignment statement $b = a + 5$ at line number 3. The condition node $b > 10$ evaluates to TRUE as there is a positive tag on b and its value is 10. This allows to prune FALSE branch which would never be taken. The next condition node $a < 5$ evaluates to FALSE as a has positive tag and its value is 5. The TRUE branch is pruned at the decision node $a < 5$. The assignment statement $out = 1$ at line number 9 is evaluated. The next decision node $out < a$ evaluates to TRUE as there is positive tag on a and its value

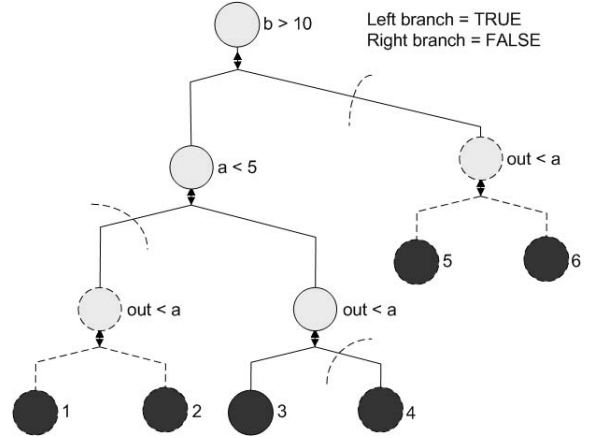


Figure 9: Pruning case with positive tag on a

is greater than that of out . So, the FALSE branch is pruned. The final value of out is 0. The pruned tree is depicted in Figure 9. The dashed lines represent the pruned part of the tree.

6. EVALUATING FAULT DETECTION PROBABILITY

A fault is considered detected if any of observable points obtains a tag at the end of the simulation. Fault Detection Probability of a variable v is the probability that v gets a tag at the end of simulation. The Number of simulations required to be run for a fault f is represented by $S(f)$. $D(V, f)$ represents number of simulations where the fault propagated to an observable point and V is the set of observable points where the fault is detected. $FDP(V, f)$ is the probability of a single fault f being detected at any of the observable points in V . $FC(T)$ is the fault coverage value for a single test pattern T .

$$FDP(V, f) = \frac{D(V, f)}{S(f)}$$

$$FC(T) = 1 - \prod_{i=1}^f (1 - FDP(V, i))$$

7. VERILOG PROCEDURAL INTERFACE (VPI)

We use Cadence Verilog-XL simulator to run our experiments. It supports the VPI library of Verilog Programming Language Interface (PLI 2.0). We developed a C application that interacts with the simulator while running a simulation. First, we parse the verilog input and construct the decision tree. Then we assign simulation callbacks for each of the assignment statements so that we can interact with the internal data structures of the Verilog-XL simulator. A callback associated with a statement forces the simulator to acquiesce control to VPI when the statement is reached while simulating. At that point, we apply our tag calculus for each of the assignments and store

subsequent changes in the tag values for each of the variables affected.

8. RESULTS

In order to evaluate our methodology, we used a set of the ITC'99 benchmarks. We took the RT-level VHDL descriptions and converted them into Verilog descriptions since our methodology makes use of Verilog Procedural Interface (VPI). We generated test sequences in a random fashion.

Bench mark	Faults (Tags)	% Cvg.	Avg. no. per tag		% Prune
			Sims	Prune	
b01	195	100	11	6.2	36.6
b02	66	100	9	8	47.1
b03	156	100	3.4	13.6	79.7
b06	168	100	7.66	6.33	37.3
b07	90	100	1.66	11.3	86.9
b08	60	100	5	4	44.8
b09	111	67.3	3.1	5.9	65.4
b10	279	50.1	10	15	60.0

Table 2: Results

Table 2 shows results for our methodology when applied to eight of the ITC'99 benchmarks. Each row in the table shows the results for the benchmark whose name is listed in the first column. The second column lists the total number of faults injected for every design. The third column shows the percentage fault coverage values computed as described in Section 6. The next two columns list the average number of simulations run and pruned respectively per tag. The last column shows the percentage of simulations pruned. We do not present direct performance results, however the performance overhead of applying our methodology can be approximated as the average number of simulations per tag as shown in Table 2. There is an additional overhead associated with pruning but it is small compared to the cost of performing multiple simulations required by our methodology.

9. CONCLUSIONS

We have presented a coverage metric, which manages the ambiguity in control flow that arises in the presence of a fault. Our approach identifies a subset of possible control flow sequences, which may be executed due to a fault and determines fault propagation in each case. Accuracy is gained in fault propagation by considering all possible control flow paths rather than just a single path. Time complexity is controlled by using fault propagation information to prune infeasible control flow possibilities. Our results are promising, and we will extend this technique to deal with the subtleties present in real designs.

10. REFERENCES

- [1] B. Beizer. *Software Testing Techniques, Second Edition*. Van Nostrand Reinhold, 1990.
- [2] K.-T. Cheng and J.-Y. Jou. A functional fault model for sequential machines. *IEEE Transactions on*

Computer-Aided Design, 11(9):1065–1073, September 1992.

- [3] F. Corno, M. S. Reorda, G. Squillero, A. Manzone, and A. Pincetti. Automatic test bench generation for validation of RT-level descriptions: an industrial experience. In *Design Automation and Test in Europe*, pages 385–389, 2000.
- [4] F. Fallah, S. Devadas, and K. Keutzer. Occom: Efficient computation of observability-based code coverage metrics for functional verification. In *Design Automation Conference*, pages 152–157, June 1998.
- [5] G. A. Hayek and C. Robach. From specification validation to hardware testing: A unified method. In *International Test Conference*, pages 885–893, October 1996.
- [6] K. N. King and A. J. Offutt. A fortran language system for mutation-based software testing. *Software Practice and Engineering*, 21(7):685–718, 1991.
- [7] J. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Trans. on Software Engineering*, SE-9:33–43, 1983.
- [8] T. Lv, J. Fan, and X. Li. An efficient observability evaluation algorithm based on factored use-def chains. In *Asian Test Symposium*, pages 161–166, 2003.
- [9] N. Malik, S. Roberts, A. Pita, and R. Dobson. Automaton: an autonomous coverage-based multiprocessor system verification environment. In *IEEE International Workshop on Rapid System Prototyping*, pages 168–172, June 1997.
- [10] D. Moundanos, J. A. Abraham, and Y. V. Hoskote. Abstraction techniques for validation coverage analysis and test generation. *IEEE Transactions on Computers*, 47(1):2–14, January 1998.
- [11] R. Vemuri and R. Kalyanaraman. Generation of design verification tests from behavioral vhdl programs using path enumeration and constraint programming. *IEEE Transactions on Very Large Scale Intergration Systems*, 3(2):201–214, 1995.
- [12] A. von Mayrhauser, T. Chen, J. Kok, C. Anderson, A. Read, and A. Hajjar. On choosing test criteria for behavioral level hardware design verification. In *High Level Design Validation and Test Workshop*, pages 124–130, 2000.
- [13] Q. Zhang and I. G. Harris. A data flow fault coverage metric for validation of behavioral hdl descriptions. In *International Conference on Computer-Aided Design*, November 2000.