A Domain Coverage Metric for the Validation of Behavioral VHDL 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

Topics: Design Validation

Presenter: Qiushuang Zhang

Abstract

During functional validation, corner cases and boundary conditions are known to be difficult to verify. We propose a functional fault coverage metric for behavioral VHDL descriptions which evaluates the coverage of faults which cause the behavior to execute an incorrect functional domain. Although domain faults are known to be a source of design errors, they are not targeted explicitly by previous work in functional validation. We propose an efficient method to compute domain coverage in VHDL descriptions and we generate domain coverage results for several benchmark VHDL circuits for comparison to other approaches.

1 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 inherited from manufacturing testing, such as state coverage and transition coverage, are impractical for systems with a large number of states and transitions. 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.

Behavioral HDL descriptions have many similarities with procedural programming languages, so it is expected that faults in behavioral HDL descriptions should be similar to software faults. Software testing research is more advanced than work in behavioral hardware validation because software faults have always been described at the behavioral level, while behavioral descriptions are only recently being accepted in the hardware community. Howden [10] has presented a classification of software faults, which is well accepted in the software test community [19, 16, 18] and divides faults into two classes. **Computation faults** are found in assignment statements and cause them incorrect data to be written to a variable. **Domain faults** exist in the control of the flow and cause an incorrect control flow path to be followed during execution. A **domain** is a subset of the input space of a program in which every element causes the program to follow a common control path. A **domain fault** causes program execution to switch to an incorrect domain. Domain faults may be stimulated by test points anywhere in the input space, but they are most likely to be stimulated by inputs which cause the program to be in a state which is "near" a domain boundary. We use this observation to determine whether or not a fault is detected by examining the distance of the test points from the appropriate boundary.

The similarities between HDLs and procedural programming languages lead us to expect that domain faults are a significant source of errors in HDLs. We propose a validation coverage metric for VHDL based on domain analysis which evaluates the domain coverage achieved by using a given test pattern sequence. A heuristic approach to domain coverage analysis to ameliorate the high complexity of enumerating all domains in a large VHDL description which contains non-linear predicates. Using some simplifying assumptions, we can evaluate the domain fault coverage via simulation. Our approach consists of two steps: First, all test points which might potentially detect domain faults are identified through simulation of the good machine. Second, faulty machines are simulated with the candidate test patterns to determine which activated faults are propagated to an output. Our approach evaluates fault coverage over domain faults which are often difficult to detect. Domain faults also dominate some simpler fault classes, so the coverage accuracy is extended to these classes of faults as well. Previous work in functional validation does not target domain faults explicitly, but the significance of this fault class motivates specfic attention to it.

The remainder of this paper is organized as follows: Section 2 summarizes previous work. Section 3 outlines our approach and introduces background knowledge of domain analysis. Section 4 presents the features of domains in VHDL descriptions. Our algorithm to evaluate domain coverage is presented in section 5. Sections 6 and 7 present the results and conclusions respectively.

2 Previous Work

Fault models have been developed at different levels of abstraction, each model defining a set of expected defects. Logic level models [13, 1] assume defects such as the use of an incorrect gate, insertion of an extra line, deletion of a line, and deletion of a gate. In [1] several theorems are presented which prove that an automatic test generation tool for single stuck-at faults is sufficient, in many cases, to guarantee validation defect coverage as well. In [13], the defect model is used to direct an automatic test generation tool which is presented. A more broad logic level defect model is presented in [12] which considers any defect which can be repaired by re-synthesizing a single signal in the circuit. Together with the fault model, [12] also presents a fault simulation method which determines the detection of faults in their model. In [8] a fault model is presented at the finite state machine level which assumes that each error affects either a single state transition or a single transition output.

Fault models have been developed directly at the behavioral level in [5] and [4] where a fault model assumes that any single variable assignment in a behavioral description may be incorrect. This is represented by associating each variable assignment with both a positive and negative tag to represent both assignment incorrect possibilities. The tags are propagated through the control-flow graph using a set of tag propagation rules which consider masking effects. In [6], the authors use the fault model presented in [4] to build a test generation tool based on the 3-Satisfiability problem. Mutation analysis has been used for hardware validation previously in [9] by converting a VHDL program into a functionally equivalent Fortran program and then using the Mothra tool for software mutation analysis [14].

The behavioral level fault models discussed to this point automatically define the validation goals, but the following two techniques require the user to define the validation goals. Researchers have applied software path testing to VHDL by allowing the user to select control-flow paths to stimulate, and using constraint programming to identify tests to stimulate the chosen paths [17]. The tool presented in [7] act as a simulator and data collector, allowing the user to specify the nature of the fault coverage to be computed.

Software researchers have been studying the problem of validating behavioral descriptions and have developed several techniques which can be applied in hardware validation. The earliest software fault coverage metrics include statement coverage, branch coverage, and path coverage [2]. Statement coverage assumes that the execution of a faulty statement will guarantee the detection of the fault. Statement coverage does not consider control-flow branches explicitly and will ignore the presence of a branch if it does not contain executable code. The branch coverage metric complements statement coverage by reflecting the number of

branches which are taken at some point during testing. The path coverage metric is a more demanding metric than either the statement or branch coverage metrics because path coverage reflects the number of control-flow paths taken. Since the total number of control-flow paths grows exponentially with the number of conditional statements, achieving high path coverage is a highly complex task.

Mutation testing [14, 15] is a flexible alternative between the weaker statement and branch coverage metrics, and the stronger path coverage metric. In principle, mutation analysis is similar to fault simulation using a set of *mutation operations* which describe the expected defects. The number of mutants can be high, making this approach time consuming, but research has been performed to limit the number of mutants [15], and to weaken the mutation detection requirements [11].

3 Proposed Functional Fault Model

A *functional fault model* is needed to describe the behavior of validation defects. Such a functional fault model will be used during fault simulation and test generation. In order for a functional fault model to be suitable, the fault model must be both accurate and efficient. Accuracy necessitates that the detection of all faults ensures the detection of all defects, in this case, all domain defects. Domain faults may be stimulated by test points anywhere in the input space, but they are most likely to be stimulated by inputs which cause the program to be in a state which is "near" a domain boundary. We use this observation to determine whether or not a fault is likely to be detected by examining the distance of the test points from the appropriate boundary. The proximity of the test points to various boundaries is used to determine which faults are activated. It is also necessary to determine which activated fault effects are propagated to an output. To determine fault effect propagation, we create a small number of mutants of the description being tested, and we simulate each mutant with the test set to determine with faults are detected. To ensure that all of the domain defects are covered, we propose a method to create a set of mutants whose differentiation ensures that all domain defects are detected. The set of mutants is created by analyzing the test set itself to determine its potential to detect domain faults.

3.1 Domain Analysis

Domain analysis was first introduced in the area of software testing [19, 2]. The input space of a program is partitioned into domains. Domains are defined by conditions at branch points referred to as predicates. The majority of domains found in behavioral descriptions have linear boundaries. Initially, we will restrict our analysis to linear domain boundaries in order to simplify the problem formulation. If a domain includes points on a boundary, this boundary is denoted as a **closed boundary** with respect to this domain. Otherwise it is an **open boundary**. An example in Figure 2 shows four domains in a two-dimensional input space. A domain boundary with shading on one side represents that the boundary is closed with respect to the shaded domain. The VHDL description corresponding to Figure 2 is shown in Figure 1.

Each predicate may form one or more domain **boundary segments**. A boundary segment is a part a boundary which lies between only two domains. For example, in Figure 2, the predicate $x \leq y + 2$ forms two boundary segments, one between domains I and IV, and another between domains II and III. In our approach, we assume that each predicate in the VHDL description forms only one domain boundary segment. Under this assumption, several segments on a boundary are considered as one segment only, so the number of boundary segments is greatly reduced and computational effort to evaluate the segments is reduced.

Each predicate defines a set of *local domains* which partition the space defined by the *local variables* used in the predicate. If the local variables of a local domain are all input signals, then the local domain is also a *global domain*. In general, the problem of mapping local domains to global domains is NP-complete, so we will first apply domain analysis to local domains. We will use the term **simulation point** to refer to

```
IF x >= y + 2 THEN
  IF x <= 10 - y THEN
    -- Computation of domain 3.
  ELSE
    -- Computation of domain 4.
  END IF;
ELSE
  IF x <= 10 - y THEN
    -- Computation of domain 2.
  ELSE
    -- Computation of domain 1.
  END IF;
END IF;
```

Figure 1: Behavioral VHDL Description with 4 Domains

the values of all of the local variables of a predicate when the predicate is executed. When a behavioral description is simulated with a test sequence, each predicate may be executed a number of times, and will be associated with a simulation point for each execution. Each simulation point is classified by its proximity to the domain boundaries. A **boundary point** is a simulation point on a domain boundary, which satisfies the equality condition of the predicate. Points A and B in Figure 2 are boundary points. An **extreme point** lies at the intersection of two or more boundaries. Point E in Figure 2 is an extreme point. Extreme points are of interest because they are typically useful in detecting domain faults across multiple domains. An **on point** is a point which lies on a boundary. An **off point** is the point near an open boundary. In order for an off point to be sensitive to a domain fault, it must lie near the domain boundary. Points A, B and E in Figure 2 are on points, D is an off point while C is not an off point because the nearby boundary is closed with respect to the domain containing C.

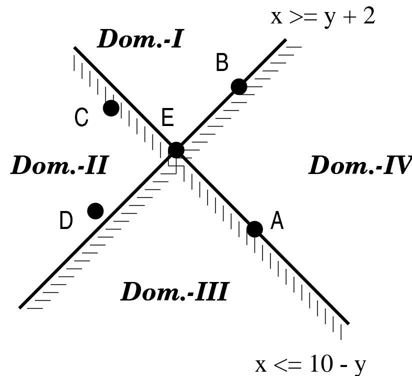

Figure 2: Four domains formed by $x >= y + 2$ and $x <= 10 - y$

A domain defect may alter the domain boundaries in a number of different ways, and it is important that our domain fault model covers all domain defects, regardless of how they alter the domain boundaries. Domain defects involving a linear boundary can be classified into the following sets [2]:

- A **closure fault** occurs when a closed boundary becomes open or vice versa.
- A **shifted boundary** describes moving a correct boundary to a position which is parallel to its original

position.

- A **tilted boundary** describes the creation of a new boundary which intersects the original boundary.
- An **extra boundary** describes the creation of an arbitrary incorrect boundary.
- A **missing boundary** is the deletion of a correct boundary.

Figure 3a depicts a local domain containing three simulation points, and Figures 3b - 3f shows each of the five classes of domain defects.

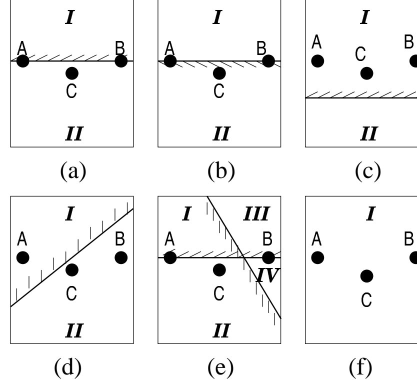

Figure 3: 2 on points and 1 off point detect domain faults on a 2-dimensional domain boundary. (a) correct boundary, (b) closure fault, (c) shifted down boundary, (d) tilted boundary, (e) extra boundary, (f) missing boundary.

3.1.1 Detection of Domain Defects

A simulation point can only detect a domain defect if that simulation point is moved to the other side of a domain boundary as a result of the defect. Previous work in domain analysis [2] has identified minimal requirements for the detection of domain defects which we will use to evaluate the potential of an arbitrary test sequence to detect domain defects. For an N-dimensional linear boundary, all domain defects can be collapsed to N+1 faults, where N points are on points and 1 point is an off point. If the N+1 faults are detected, then all domain defects related to the boundary are detected. This is depicted in Figures 3a - 3f. Points A, B, and C are the 2 on points and the off point which are needed to detect all domain defects associated with the domain boundary in Figure 3a. Each of the Figures 3b - 3f show a domain defect class, and show that at least one of the three simulation points is moved across the domain boundary in each case. To guarantee detection of all domain defects associated with a boundary segment, the test data must include N on points and 1 off point to activate the faults, and the fault effects must also propagate to the output. These detection criteria are associated with the $N \times 1$ software domain testing method and these criteria can be used to evaluate a test sequence.

4 Domains in VHDL Descriptions

The input space can be partitioned into domains in which all points execute the same computation (i.e. follow the same control-flow path in the VHDL description) to generate outputs and next state. Domain analysis can be applied to determine the coverage of domain faults across these domain boundaries.

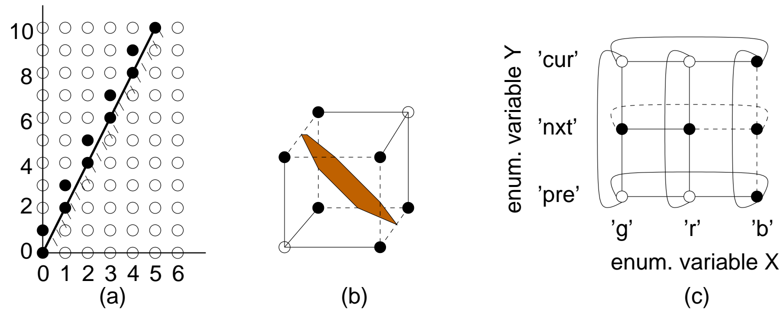

Figure 4: Three domain types

The most common data types in the VHDL benchmarks we examined are integer, boolean and enumerated types. The analysis of domains formed by these types can be easily extended to domains formed by other data types. Domains defined by integer variables and signals can be approximated by the traditional linear domains. Extreme points, boundary points, and on/off points are well defined. Fault detection can be determined by evaluating the $N \times 1$ detection requirements at each domain boundary. An advantage of an integer domain is that the smallest distance from off point to boundary is 1 in most cases and no other points lie between the off point and boundary. This advantage will eliminate the error which can be introduced by significant distance from the boundary. Figure 4a shows a 2-dimensional integer input space, and the black circles in the figure represent on/off points.

Domains defined by boolean variables and signals are different since each dimension has only two values. The domain boundary is not linear. The input space can be mapped to an n-dimensional hypercube for which each vertex is a unique combination of the values of variables. Two adjacent vertices differ only in the value of one variable. If two adjacent vertices belong to different domains, these two points are on point and off points with respect to the boundary defined by the variable on which differ. Figure 4b depicts a three dimensional boolean space. Two domains are divided by the plane inside the cube. Six dashed edges go through the boundary. On/off points represented by black circles are the ending vertices of dashed edges. To detect domain faults of this boundary, all on and off points are required.

Domains defined by enumerated variables and signals can be treated as an extension of the domains defined by boolean variables. Each dimension has k($\geq$ 2) values and these k points are adjacent to each other. Figure 4c shows a two dimensional enumerated input space. Each point is adjacent to four other points in the space. If on points ('b','nxt') form a domain, then all other points compose a different domain, and the on point is adjacent to the four off points connected by dashed edges.

In general, domains in VHDL are often defined by variables and signals of more than one type which make the analysis of domains more difficult. Domain analysis becomes intractable when the number of inputs and states is large. In the next section we propose a heuristic approach to improve the efficiency of domain fault analysis.

5 Domain Fault Coverage Metric

Based on previous work in domain analysis [2], we know that all domain defects are activated if the test sequence meets the *N+1* criteria, to contain N on points and 1 off point for each domain boundary segment in an N dimensional input space. We evaluate the domain coverage of a test sequence by determining how closely the test sequence matches the N+1 criteria. This is performed in 3 steps:

1. **Simulation Point Condition Generation** - For each predicate, we generate a set of linear equations which define the set of N+1 simulation points required to activate all domain defects.
2. **Test Sequence Simulation** - The original VHDL description is simulated with the test sequence to identify all simulation points. The simulation points are checked for satisfaction of the simulation point conditions.
3. **Mutation Analysis** - A mutant is created corresponding to each simulation point which is found to satisfy a simulation point condition. Simulation of each mutant is performed to determine if the domain fault effects propagate to a primary output.

5.1 Step 1: Simulation Point Condition Generation

In this step, a VHDL description is analyzed to derive a list of *test conditions* for each predicate in the description. Each test condition is a linear predicate which describes the condition required to activate domain faults along a particular boundary. Each test condition corresponds to one of the N+1 test points required to activate domain faults associated with a boundary. We define some rules used to derive the test conditions according the type of VHDL predicate. Predicates are classified into two types: a **simple predicate** is an expression with only a relation operator($\leq$, $\geq$ ...), and a **compound predicate** consists of several simple predicates connected by logical operators (and , or ...). Rules 1 and 2 are used for simple predicates while rule 3 is for compound predicates.

Rule 1. *Simple predicate with integer variables and signals:* The test points required are N on points and one off point. The N on points must be on an N dimensional plane. The off point must be between some of the N on points.

Rule 2. *Simple predicate with boolean or enumerated variables and signals:* The on/off test points for a boolean predicate are all the points on the hypercube which represents the input space. For enumerated predicates, each on test point is accompanied by one off test point in each dimension.

Rule 3. *Compound predicate:*

If we consider each simple predicate to be boolean variable, then a compound predicate consists of a combination of boolean variables, and the domain defined by boolean variables is a set of points on a hypercube representing the input space. A domain boundary exists between any two adjacent points on the hypercube which belong to different domains. Since a boolean variable defines a non-linear boundary, we have to test every point near a boundary, i.e. every point on the hypercube which has at least one adjacent point in a different domain. A boolean variable at a test point is **sensitive** if a change of the variable value will change the output of the predicate. The sensitive boolean variables of a boundary point are expanded using the simple predicate rules 1 and 2.

Consider the following example VHDL description, where x and y are integer variables and st is a boolean variable:

```
IF (x > y) and (st = '1') THEN
    output <= '1';
ELSE
    output <= '0';
END IF;
```

This predicate is a compound predicate consisting of the conjunction of two simple predicates, P1: $x > y$, P2: st. The required test conditions are listed in table 1. Consider P1 and P2 as boolean variables, the

space of (P1, P2) includes four points (00, 01, 10, 11), where point (11) belongs to 'true' domain of the predicate and the points belongs to the 'false' domain. Three points (01, 10, 11) have sensitive variables. Since point (01) is sensitive to P1, we expand P1 to the linear predicate $x > y$. Since on points of $x > y$ are in 'false' domain of P1, we test two on points for P1, while P2 should be satisfied. T1 and T2 describe the two points (P1, P2) = (01). Similarly, (10) point is sensitive to P2, and is described by test point T4. Two test points are required for point (11) which is sensitive to both P1 and P2. Since the test point sensitive to P2 is dominated by the other point sensitive to P1, we just require T3 sensitive to P1, an off point of P1.

Test Point	**Condition**
T1	$x = y$ and $x = x1$ and $st =' 1'$
T2	$x = y$ and $x = x2$ and $st =' 1'$
T3	$x = y + 1$ and $x2 < x < x1$ and $st =' 1'$
T4	$x > y$ and $st =' 0'$

Table 1: Test points required for test predicate in example.

In Table 1, $x1$ is the maximum number which satisfies T1's conditions, while $x2$ is the minimum number. The values of $x1$ and $x2$ are determined in step 2 of the algorithm when simulation is performed. T3 is an off point between T1 and T2.

5.2 Step 2: Test Sequence Simulation

In this step, we simulate the behavioral description with the test set to identify all simulation points. Each simulation point is then compared to the set of simulation point conditions to determine if the simulation point will activate ant domain defects. Domain defects corresponding to boundaries whose simulation point conditions have not been satisfied are considered to be undetected. Simulation points which satisfy simulation point conditions are further analyzed using mutation analysis.

5.3 Step 3: Mutation Analysis

In this step, the propagation of fault effects to outputs is determined by simulating mutants corresponding to each activated test point. A mutant is created for each simulation point by modifying the corresponding predicate so that the simulation point is moved to the other side of the nearby domain boundary. For example, suppose that in testing the predicate $x \leq y$, the simulation point $(x, y) = (3, 3)$ is identified as an on point. A mutant would be created which substitutes the original predicate with $(x \leq y) \cap \overline{(x = 3) \cap (y = 3)}$. In the mutant, the simulation point no longer satisfies the equality condition of the predicate, and is moved to a new domain to activate potential defects.

Domain fault coverage is defined as:

$$domain\ coverage = \frac{\#\ of\ detected\ faults\ on\ test\ points}{\#\ of\ total\ test\ point\ required}$$

6 Domain Fault Coverage Results

We have evaluated our domain fault model by computing the domain fault coverage of several behavioral VHDL descriptions and comparing the coverage to branch coverage. The first example is a 4 bit ARBITER containing 8 predicates. BARCODE and TLC are from the HLSynth'92 benchmark suite. FIFO is an

Benchmark	# of lines	# of preds	# of patts	Domain metric			Branch metric		
				sim. pts.	det. flts.	cov.	total branches	branches executed	cov.
ARBITER	102	8	16	27	24	0.89	22	19	0.86
BARCODE	77	5	540	14	8	0.57	10	10	1
FIFO	114	13	36	27	21	0.78	27	27	1
TLC	58	5	100	15	14	0.93	12	11	0.92
FP_ADDER	428	42	417	135	124	0.92	102	101	0.99

Table 2: Domain coverage results compared with branch coverage.

example modified from an existing Verilog model. The FIFO's depth is 16 and it has two write ports and one read port. FP_ADDER is a benchmark from HLSynth'95.

The coverage results are shown in Table 2. Columns 2, 3 and 4 contain the number of lines of code, the numbers of predicates and the number of test patterns applied respectively. Domain metric results are listed in columns 5, 6 and 7. Branch metric results are shown in columns 8, 9 and 10.

From the results, we see that the number of test points required by domain metric is greater than or equal to the number of total branches. The number of undetected domain faults is greater than or equal to the number of uncovered branches. In the ARBITER and TLC examples, all predicates are boolean and numerical predicates. As we have shown, for simple boolean or enumerated predicates, the two metrics are equivalent to each other. In ARBITER, 3 domain test point conditions are not satisfied and exactly 3 branches are not covered because those 3 predicates are simple boolean predicates. A similar situation occurs in TLC example.

In the BARCODE and FIFO examples, there exist linear predicates. Only those points near the boundary may be test points in domain metric while any point in a domain can cover a branch. So branches branch coverage should be much easier achieve than domain coverage. Our results show that branch coverage of both examples are 1, while domain coverage are 0.57 and 0.78 respectively.

FP_ADDER is a larger design. The 417 test patterns were designed to check the equivalence between behavioral description and low level implementation. Although the test patterns cover 99% branches of the design, our domain metric still identifies 11 domain test points not satisfied and yields 92% domain coverage. The 42 predicates include 30 simple, 12 compound predicates; or include 26 boolean, 3 enumerated, 10 linear and 3 mixed predicates. 14 out of 42 predicates are simple boolean one-dimensional predicates for which the domain metric is equivalent to the branch metric.

7 Summary

We use domain testing techniques to define a functional fault model which enables efficient evaluation of test patterns for validation. The use of a behavioral validation metric based on domain testing shows strong potential in examining faults which are overlooked by other validation metrics. However, more investigation is needed to refine the domain metric. We provide domain coverage results for several VHDL benchmarks to show the utility of the approach.

References

[1] M. S. Abadir, J. Ferguson, and T. E. Kirkland. Logic Verification via Test Generation. *IEEE Transactions on Computer-Aided Design*, 7(1):138–148, January 1988.

[2] B. Beizer. *Software Testing Techniques, Second Edition.* Van Nostrand Reinhold, 1990.

[3] T. A. Budd and D. Angluin. Two Notions of Correctness and Their Relation to Testing. *Acta Informatica*, 18:31–45, 1982.

[4] S. Devadas, A. Ghosh, and K. Keutzer. An Observability-Based Code Coverage Metric for Functional Simulation. In *International Conference on Computer-Aided Design*, pages 418–425, November 1996.

[5] 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*, pages 666–671, June 1999.

[6] F. Fallah, S. Devadas, and K. Keutzer. Functional Vector Generation for HDL models Using Linear Programming and 3-Satisfiability. In *Design Automation Conference*, pages 528–533, June 1998.

[7] R. Grinwald, E. Harel, M. Orgad, S. Ur, and A. Ziv. User Defined Coverage - A Tool Supported Methodology for Design Verification. In *Design Automation Conference*, pages 158–163, June 1998.

[8] A. Gupta, S. Malik, and P. Ashar. Toward Formalizing a Validation Methodology Using Simulation Coverage. In *Design Automation Conference*, pages 740–745, June 1997.

[9] G. Al Hayek and C. Robach. From Specification Validation to Hardware Testing: A Unified Method. In *International Test Conference*, pages 885–893, October 1996.

[10] W. E. Howden. Reliability of the Path Analysis Testing Strategy. *IEEE Transactions on Software Engineering*, SE-2(3):208–215, 1976.

[11] W. E. Howden. Weak Mutation Testing and Completeness of Test Sets. *IEEE Transactions on Software Engineering*, SE-8(4):371–379, July 1982.

[12] S.-Y. Huang, K.-T. Cheng, K.-C. Chen, and J.-Y. J. Lu. Fault-Simulation Based Design Error Diagnosis for Sequential Circuits. In *Design Automation Conference*, June 1998.

[13] S. Kang and S. A. Szygenda. Design Validation: Comparing Theoretical and Empirical Results of Design Error Modeling. *IEEE Design & Test of Computers*, 11(1):18–26, Spring 1994.

[14] 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.

[15] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An Experimental Determination of Sufficient Mutant Operators. *ACM Transactions on Software Engineering Methodology*, 5(2):99–118, April 1996.

[16] D. Richardson and L. Clarke. Partition Analysis: A Method Combining Testing and Verification. *IEEE Transactions on Software Engineering*, SE-11(12):1477–1490, 1985.

[17] 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.

[18] E. Weyuker and T. Ostrand. Theories of Program Testing and the Application of Revealing Subdomains. *IEEE Transactions on Software Engineering*, SE-6(3):236–246, 1980.

[19] L. White and E. Cohen. A Domain Strategy for Computer Program Testing. *IEEE Transactions on Software Engineering*, SE-6(3):247–247, 1980.