Event-Driven Observability Enhanced Coverage Analysis of C Programs for Functional Validation

Farzan Fallah† Indradeep Ghosh† Masahiro Fujita‡

† Fujitsu Laboratories of America
Sunnyvale, CA 94086
USA
<farzan,ighosh>@fla.fujitsu.com

‡ Dept. of Electronic Engineering
University of Tokyo
Tokyo 113-8654, Japan
fujita@ee.t.u-tokyo.ac.jp

Abstract

Software programs written in some programming languages like C, C++, Java, *etc*, are mostly verified by functional simulation. Since exhaustive functional simulation is impossible for even a small C program, it is important to quantitatively measure the extent of design verification during simulation by a set of test vectors. Various coverage metrics have been proposed for measuring the degree of design verification. Most of them compute the extent of design excitation (controllability) but are unable to say whether the excitation responses have propagated to observable points in the program (observability). In this paper we propose a metric for code coverage analysis of C programs that addresses not only controllability but tackles observability as well. Thus, this metric is able to tell what percentage of the simulation responses have been propagated to observable points in the program like primary outputs or printed variables. We improve upon a recently proposed observability enhanced software coverage metric by increasing the accuracy of the analysis as well as decreasing the simulation runtime overhead by using an event-driven coverage analysis method. We report some experimental results of using our coverage analysis tool for several C programs.

1 Introduction

With the advent of better compilers and automated synthesis tools, C programs are being widely used for describing hardware, writing software, or specifying mixed hardware/software systems. C programs may be used to create software used in embedded or portable systems, to write initial descriptions or specifications of hardware or even to describe complete systems in case of hardware/software codesign. Traditionally C has been used mostly for writing large software applications but as mentioned above many design methodologies now use C for both describing hardware and writing software. The fact that hardware and software design methods are trying to move towards a unified C based description language can be observed in the recent activities with languages like *SystemC* and *SpecC* [1], [2].

In any software or hardware design flow one of the fundamental problems is verification. Since software programs are error prone, it is necessary to check or validate the code for correctness and detect errors. Also, the hardware synthesized from C or assembly code generated from C programs can be quite sensitive to the quality of the initial C description. Therefore, there are bound to be changes made to the initial C program. At each step of these design iterations, it is absolutely necessary to verify or validate the correctness of the C description.

Simulation is still the most popular technique used to verify the correctness of hardware and software designs. In simulation based verification or validation the unit under test is exercised with a set of input stimuli and the output responses are examined for correctness. One of the major problems in this technique is to determine how good the test vectors are in terms of exercising the complete behavior of the design. If large portions of the C program remain untouched by the simulation vectors, the chances of the presence of bugs in those portions of the C description increases drastically.

In order to alleviate the above problem a large number of software coverage metrics have been proposed so far that give a percentage of the amount of the description that has been exercised by the simulation test vectors. Most of these metrics consider excitation but fail to address observation. That is they tell us how much of the description has been exercised by the test vectors (controllability) but fail to say whether the effects of the excitation has been propagated to observable points like outputs or printed variables (observability). Observability is as important as controllability since if the error response of a bug is unobservable during simulation of the test vectors, then the bug remains undetected.

In this paper, we address the above issue by proposing an observability enhanced coverage metric that not only gives a measurement of controllability of the test set but also tackles observability. Our work is motivated by previous work on coverage of hardware description language (HDL) designs [3]. In this paper we have enhanced and modified the above coverage metric to take care of software constructs like pointers, recursions, and floating-point operations that are not present in hardware languages. We have increased the accuracy of the coverage metric over previous work presented in [9] by detailed analysis of floating-point operations, expressions, types and type casts. We also reduce the coverage analysis overhead by using an event-driven simulation method. We present some experimental results that show the overhead of simulating a design with this coverage metric is tolerable.

2 Previous Work

Previously a lot of work has been done on software testing techniques [5],[6]. Since it is impossible to ensure that a software program is completely free from bugs, most of these techniques have concentrated on coverage metrics. The most important ones are statement, branch and path coverage. There is also multi-condition coverage and loop coverage. All the above metrics just target excitation or controllability. They do not target observability which is equally important for catching bugs. The path coverage metric will satisfy observability requirements if all paths from program inputs to program outputs are exercised and the values of variables are such that the erroneous values are not masked. However, this metric also does not explicitly evaluate whether the effect of an error is observable at a primary output. Moreover, due to the exponential number of paths in a program 100% path coverage is impractical. Further, the presence of false paths makes full path coverage very tricky to achieve.

Impact and sensitivity analyses proposed for software testing take into account observability requirements. Voas [7] introduced sensitivity analysis that estimates probabilities that an error on a location will result in an output error under a specific input distribution. The program is executed for a large number of random inputs consistent with the input distribution to estimate the error propagation probability for a single location. Impact analysis [8] estimates impact strengths of all entity instances in an execution in a time proportional to the execution time. The impact strength of a statement or variable y serves as a quantitative measure of the error-sensitivity of the paths from y to the output. The above two metrics both deal with injection of some error at some part of the program and compute the probability of that error effect reaching an output. They do not compute a complete coverage of the program based on observability requirements. Though weak mutation analysis [6] comes close to the error model used in this work, the errors modeled here are a strict superset of all errors that can be modeled by weak mutation analysis and again unlike this work observability is not explicitly tackled in weak mutation analysis.

In [9], an observability enhanced coverage metric has been proposed. In that work the authors modify the software program by adding a function call for each assign statement in the program. In case of an assignment, a control function is added after the assignment. When there is a call to an output function an observe function is added. A list of dependencies is maintained for each statement and augmented with the program flow. When an observe statement is reached, all the statements in its dependency list are marked as observable. Since the method does not take into account the type of errors in a particular statement, the method can be inaccurate. Furthermore, error masking due to cancelation of errors is ignored. Also, for statements like *if (a<15) else ...*, a more accurate error representation needs to be made to determine whether the **if** block or the **else** block has been taken due to the error. From the description of the paper it seems that the dependency list is built up through a forward traversal of the program. This results in an expensive algorithm of $O(n^2)$ where n is the number of lines in the program. The high simulation run time overheads reported in the paper corroborates this. It is possible to have a linear time algorithm by building the dependency list through a backward traversal. However, this was not done in the paper. In this paper we have improved the accuracy of the coverage by implementing a more accurate error model. Also, our simulation run time overheads are much better than the ones reported in [9].

Some work has already been done on observability enhanced coverage analysis of RTL HDL descriptions. A rather inefficient and restrictive observability enhanced HDL coverage metric was proposed in [4]. The above work in [9] as well as the work presented in this paper is inspired by previous work on HDL coverage presented in [3].

3 Proposed Method

In this section, we describe our method for performing coverage analysis for a C program. First we explain what tags are. After that we will explain how we handle software specific constructs like pointers, arrays, recursive functions, and data structures. We will also present some enhancements to the method used in [3] to better handle C programs. After explaining our event-driven coverage analysis method, we will conclude the section by briefly describing the complexity of our coverage analysis algorithm.

3.1 Tags

A **tag** at a location represents the possibility that an incorrect value was computed at that location. The tags on variables are not tied to particular errors; they serve as a mechanism for extending standard coverage metrics to include observability requirements [3]. Each location corresponds to an assigned variable in some statement in the C program. Our goal, given a set of functional vectors and a C program, is to determine if a tag **injected** in any particular location is **propagated** to the program output. That is, we want to see if incorrectly computed values are propagated to program outputs, or not. This is dependent on the data values at other variables of the program. Other data values may **block** the tag from reaching any program output. The quality of a vector set is determined by how many injected tags are propagated to the output. The percentage of propagated tags is what we call **tag coverage** under the metric.

Note that while there is full observability of internal locations in the C program during simulation, the particular data values at internal locations may be incomprehensible to the designer. For example, the designer may be able to verify that the output of a Wallace tree multiplier is an incorrect 6, for inputs of 4 and 4, but will not be able to determine if an arbitrary internal wire is at a faulty 1 or 0.

We will use the **single tag** model, where the effects of exactly one injected tag are computed, for many different injected tags. Each injected tag can be thought of creating a distinct "faulty" C program. Two tags are injected for every variable and expression.

A tag is represented by the symbol Δ which signifies a possible change in the value of the variable due to an error. Both positive and negative tags are considered, $+\Delta$ written simply as Δ, and $-\Delta$. If the presence of the tag is not known, an unknown tag is used. An unknown tag is shown by "?". In case that there is an error, but we don't know its sign we use Δ'. Approach presented in [3] used "?" to model this type of errors. As we will see in the next subsection, introducing Δ' helps us to better handle pointers and get some meaningful coverage results.

As an example consider the following piece of C program:

```
a = 1;
b = 2 - a;
fprintf(outFile, ``%d'', b);
```

If the intended assignment in the first line was $a = 0$, the value of variable a will be greater than its intended value. We can represent the possibility of this by putting a Δ on variable a. The tag on variable a is propagated to variable b when the second line is executed. As a result, variable b will have $-\Delta$. Finally, when the **fprintf** instruction is executed, the value which is printed in **outFile** will be erroneous. This will signal the presence of an error in the program.

3.2 Pointers

One major difference between C language and Verilog HDL is the use of pointers in C. In this subsection we explain how we handle pointers. A pointer is a variable that holds the address of another variable in a memory. If there is an error in the value of the pointer, the pointer points to a location different than the desired one. Hence, any memory operation performed using the pointer may result in an error. If the pointer is used to read the memory location, a wrong location may be read resulting in an erroneous value. Consider the following example:

```
a = *p;
```

If there is a tag on variable p, we read a location in the memory which is different than the one intended. Hence, there will be an error on variable a. Due to the fact that the exact magnitude of the tag on p is unknown, we don't know the sign of the tag that may appear on variable a. We use Δ' to represent the presence of an error on variable a.[1]

Now let's assume that the pointer is used to write a memory location. In this case a wrong location may be written which might result in errors in both the erroneous and the intended locations. This means that we need to inject two tags in two different locations to model the error. In the following assignment:

```
*p = a;
```

if p is tag free, the tag present on a is propagated to $*p$. If p is not tag free, there will be two effects,

1. An erroneous memory location will be modified.
2. The intended memory location will not be modified.

As a result the tag on p may propagate to two different memory locations. To capture the first effect, it is necessary to put a tag on the memory location pointed by p in presence of the tag. The second effect can be modeled by putting a tag on the memory location pointed by p in absence of the tag. Because the magnitude of the tag is not known, it is not clear which memory location should have been altered in the correct case. It is possible to assume a tag on every location whose address is more (less) than p when there is a negative (positive) tag but this approach leads to too many tags on memory locations. This problem may be solved by ignoring the propagation of the tag by this effect and concentrating only on the tag generated by the first effect. As an example, consider the statement $*p = 20$ with the memory contents given in Figure 1 (a). Figure 1 (b) shows how the memory contents will change if there is no tag on p (i.e., the correct behavior of the program). Note that only the content of one location has changed. Figure 1 (c) shows the memory contents after executing the statement in presence of a tag with magnitude one on p. As one can see, the statement changes the content of a wrong location which results in a negative tag on that location. Also, there will be a positive tag on the intended location because it has not been changed to 20. Figure 1 (d) shows how the memory contents are changed in presence of a tag with magnitude two on p. In this case, the error appears on the intended location and a new location. Note that the tag on the erroneous location appears independent of the magnitude of the tag on p. In practice the magnitude of the tag is not known. Hence, only the negative tag on the erroneous location will be considered and the tag which may appear in the correct location will be ignored. This simplification makes our coverage analysis pessimistic.

In some cases the above approximation may be improved by a more accurate analysis of the program. As an example consider the following code segment.

```
 p = p + m;
*p = a;
```

where m is a binary variable. If the tag on p is due to the tag on m, it is possible to find the exact magnitude of the tag on p. This makes it possible to find the memory location that will be wrongly modified because of the tag.

Tag propagation in the presence of arrays can be done in a similar fashion explained above.

3.3 Recursions

The method explained in [3] for handling functions can be used directly to handle recursions. We give a brief description of the method here. Upon calling a function, the tags on its actual

[1] Δ' is the only type of tag used in [9]. While this may seem adequate for analyzing errors on pointers, it may not be powerful enough for measuring the coverage of every part of a C program.

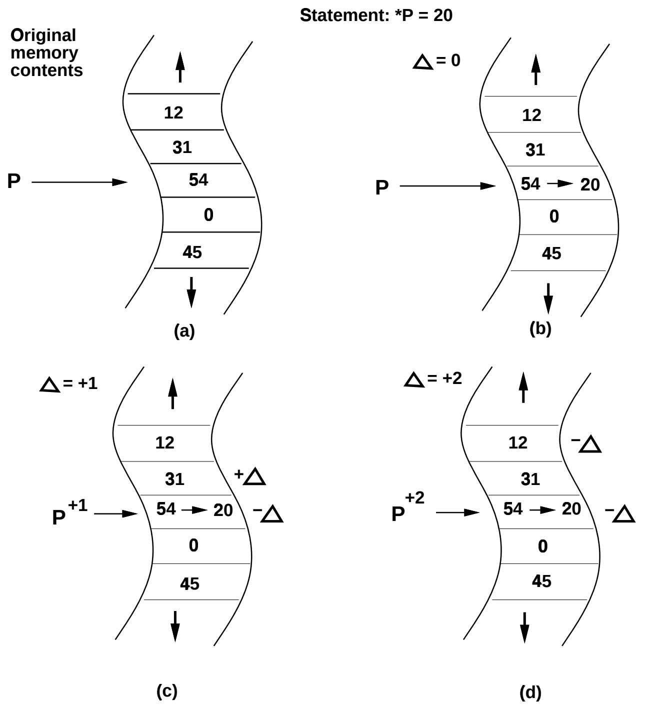

Figure 1: Effects of tags on pointers

variables are copied to their corresponding formal variables and the tag propagation is performed as usual. If the line of the program on which we want to inject a tag is inside the function, a tag will be injected when executing that line. Once returning from the function, the tags on the formal variables are copied to their corresponding actual variables.

To handle recursive functions, tags have to be copied from actual (formal) variables to formal (actual) variables on each recursive call (return). To prevent the interaction of the tag on a variable in one recursion depth with the ones in a different depth, the tags on variables are saved and restored after each recursive call and return, respectively. Also, if the line of the program on which we want to inject a tag is inside the recursive function, a new tag is injected on that line in every recursion.

The need to save and restore the status of all variables for every recursive call will make the above approach inefficient for handling very deep recursive functions. This problem can be easily solved by performing the save and the restore only for the tagged variables which are typically a small fraction of the total variables.

3.4 Structures

Structures are handled in the same way that simple variables are handled. The only difference is that, the tag on every field of a structure is considered separately.

3.5 Expressions

In [3], the tags are injected on assignments only. To better handle some cases like having an error in the control clause of an **if** statement, we inject tags on expressions as well. This also assists in detecting errors corresponding to calling functions with wrong arguments. For example, if there is an error in the following code segment:

```
if(C)
  fprintf(outFile, ``%d'', a*b);
```

we can detect it only if we inject a tag on the expression $a * b$. Otherwise, there will be no tag and no activation requirement for the **fprintf** instruction.

3.6 Floating-Point Arithmetic

While working with floating-point variables, the tag may be blocked depending on the relative values of the variables. Our coverage analysis method can detect this tag-blockage and achieve more accurate results for floating-point numbers. As an example consider the following statement,

```
A = B + C;
```

If the variables are integer, the value of A will depend on the values of both variables B and C. As a result, if there is a tag on one of the variables in the right hand side and the other variable is tag free, the tag will propagate to the left hand side. Now, assume the variables are floating-point. In this case, it is possible that the value of variable A will depend on only one of the variables in the right hand side. More precisely, if the value of one of the variables is much larger than the other, the result of the operation will depend only on the former one. For example, if $B = 10^{20}$ and $C = 10^{-20}$, the value of variable A will be 10^{20}. As a result if there is a tag on variable C and the tag's magnitude is in the same order as the value of the variable, it will not propagate to the left hand side. Because it is very time consuming to keep track of the magnitude of tags, we make the conservative assumption that the magnitude of a tag on a variable is always in the same order as the variable's value. The approach presented in [9] does not consider this tag-blockage on floating-point numbers.

3.7 Types and Type Casts

Many complex types are used in software programs to define variables. This may result in choosing an improper type for a variable or loosing some information while casting one type to another.[2] To target bugs related to casting errors, we can inject a new tag once an overflow happens during performing an operation and perform the tag propagation to see if the effect of that error can be detected in one of the outputs.

3.8 Event-Driven Coverage Analysis

We use event-driven coverage analysis in order to increase the speed of our method [11]. In this method, we process a statement only if there is at least one tagged-variable in its right

[2]Ariane 5, a rocket launched by the European Space Agency, exploded 40 seconds after its lift-off due to an overflow error which happened because of converting a 64-bit floating-point number to a 16-bit integer.

```
Coverage-Analysis(Graph G, List VARS) {
  WHILE (VARS is not empty) {
    Topologically sort VARS;
    V = Remove the first element of VARS;
    Perform the tag simulation for the
    statement whose left-hand side variable
    is V.

    If (tag is propagated to V)
      Insert all variables (i.e., nodes in G)
      in the fanout of V into VARS;
  }
}
```

Figure 2: The event-driven coverage analysis algorithm

hand side. This helps to reduce the CPU time wasted while there is no tag on the right hand side of a statement. Figure 3.8 shows the even-driven algorithm in pseudo code.

The input of the algorithm is graph G and list VARS. G is a graph constructed from the C program. Every node in G represents a variable in the C program. There is an edge in G corresponding to every statement in the C program. VARS is a list of variables which may have tag during coverage analysis. Before calling the coverage analysis routine, the values of variables are computed by performing plain simulation. These values are used later during tag simulation. Tag simulation starts with selecting a variable and injecting a tag on it. The variables on the fanout of the selected variable is inserted in the VARS list and the function is called. The tag simulation continues with topologically sorting variables and removing the first variable from the list. In the next step, tag simulation is performed for the corresponding statement with the help of values computed through simulation. If a tag appears on variable V, the variables in its fanout are inserted into VARS. This process continues until there are no more variables in VARS. Since in practice only a small portion of variables are tagged during tag simulation, this method performs much better than straight forward tag simulation.

3.9 Complexity

In our coverage analysis algorithm, we need to inject two tags for every assignment or expression. After that, it is necessary to perform the tag simulation for every tag. Assuming one assignment or expression per line, the number of tags will be $O(l)$ where l is the number of lines of the C program. For every tag, it is necessary to perform the tag simulation once. This suggests the CPU time for the tag simulation will be $O(l \times n)$ where n is the number of lines that are executed.[3]

If there is a recursive function, it will be necessary to save and restore the status of variables in every recursion. The amount of work will be on the same order of the work done for creating the stack while executing the recursive function. Taking into account this effect, the CPU time will be $O(l \times t)$ where t is the execution time of the program. In practice, the time it takes to perform the tag simulation is less than $O(l \times t)$ because some tags may not be excited when simulating the program. Also, it may not be necessary to process the entire program for every injected tag due to early detection or blockage. The tag simulation speed can be further improved by implementing the speed-up techniques used in concurrent fault simulation [10].

[3]Note that n can be greater than l if some lines are executed more than once.

4 Experimental Results

We report experimental results on five example C programs. The programs are first compiled into a graph-based ASCII intermediate format. The tag injection and propagation algorithm works better on this simple format than on a complex direct parse tree representation of the C program. Table 1 reports the experimental results on the examples. In the Table, *Fib* is a program to calculate Fibonacci numbers. *String Match* is a program that reads a stream of characters and detects the occurrence of a specific string. *Fourier* is the Fast Fourier Transform program. *PROG1* and *PROG2* are two large inhouse programs with many *if* statements. In Column 2 of the table the number of lines in each program is shown. In Column 3 the test set size is shown. We use three different test sets for the first three programs. For *Fib*, the first test set consists of the vectors (0, 1, and 3). The second set consists of the vectors (0, 1, 2, 3, and 4). The third set consists of 1000 random vectors. For *String Match*, the first set consists of 15 vectors with no match. The second set consists of 15 vectors with a match. The third set consists of 1000 random vectors. For *Fourier*, we used dirac, constant, and sinusoidal inputs. For *PROG1* and *PROG2*, we used long randomly generated vectors. Column 4 shows the traditional statement coverage numbers while Column 6 shows our tag coverage numbers. Column 5 shows the total number of tags injected for each program. As it can be seen from the results, the tag coverage numbers reflect the poor coverage of a test set even when statement coverage is 100%. This provides a more accurate measure of the effectiveness of the test set in testing the program. Note that the second test set for *Fib* has one vector more than the test set given in [9]. This is because our tag coverage scheme handles the error responses in a more detailed manner than the scheme in [9]. In case of *String Match* the coverage numbers are also different from [9] because of more accurate modeling of errors. The tag coverage numbers in this example are low because there is only one observable statement in the program. Similarly, the tag coverage for PROG2 is very low due to small number of observable statements.

In Columns 7 and 8 the CPU times of running the programs and simulating them using our tool without performing the coverage analysis are reported. Columns 9 and 10 show the coverage analysis times using the straight forward and the event-driven tag simulation methods. The simulation times are for a Sun Ultra 60 with 360MHz CPU and 786MB memory. In the small examples, the CPU times are mostly dominated by parsing and initializing times. Therefore, there is not much difference between the CPU time of the straight forward and the event-driven tag simulations. For large examples (i.e., *PROG1* and *PROG2*), simulating the programs using our tool

Table 1: Experimental Results

Program	Size # C lines	# Vectors	Stmt. Cov. (%)	# Tags	Tag Cov. (%)	Orig. CPU (sec)		Tag. Cov. CPU (sec)	
						C	Sim.	Reg.	Event
Fib	24	3	100	24	79.2	0.01	0.29	0.31	0.33
		5	100		100	0.01	0.29	0.33	0.34
		1000	100		100	0.02	0.63	1.13	1.02
String Match	45	15	87.5	36	2.8	0.01	0.06	0.06	0.06
		15	100		41.7	0.01	0.06	0.06	0.06
		1000	100		63.9	0.02	0.29	1.27	1.06
Fourier	129	8	100	118	51.6	0.01	2.18	5.64	2.91
		8	100		100	0.01	2.19	5.66	2.90
		8	100		100	0.01	2.18	5.66	2.91
PROG1	2288	1000	100	4000	69	5.83	13.5	8180.84	27.70
PROG2	4897	10000	100	9056	31	68.8	337.62	39295.12	766.57

is between 2x and 5x slower than running the actual C codes.[4] The straight forward tag simulation, on the other hand, is several orders of magnitude slower than the plain simulation. For these examples, the event-driven tag simulation is only 5x and 11x slower than running the actual C codes. This enables us to use our tag simulator on large programs. The good performance of the event-driven tag simulation is due to the fact that during the tag simulation many variables are not tagged. As a result it is not necessary to perform coverage analysis for many statements of the C program and they can be ignored. In addition, the fact that many tags are observed after simulating a small number of vectors helps to decrease the CPU time. Overall the overhead of the coverage analysis using our method is at most 3x when compared with time it takes to simulate the program using our tool.

Note that it is possible to implement the tag simulation scheme by direct instrumentation of the C program. However, in case of a hardware/software co-simulation environment which this method can target the C program needs to be simulated in an intermediate format like ours. In this scenario, the CPU time overheads are about 2 times the original simulation times. Thus, the tag simulation is not very expensive in terms of the computational resources.[5]

5 Conclusions

In this paper we have presented a coverage metric that can be used for observability enhanced functional simulation of C programs. This metric tells us not only which statements are executed by a test set but also if the statements have any effect on the output responses. This metric is more accurate than the previously proposed observability based metric as it can model various types of errors and can handle cancelation of errors. Our method is capable of handling floating-point arithmetic and can perform an accurate analysis for them. It can also perform coverage analysis targeting type casting errors. The CPU time overhead while simulating with this metric in event-driven mode is superior than the previous work in [9]. This metric can be used in embedded software testing or during hardware/software co-simulation. We are currently working on new heuristics to make the coverage analysis more efficient for large examples. Some possibilities are using fault simulation methods like fault collapsing and dominating faults to skip some parts of computation.

[4] The numbers in column 8 do not include the time it takes to output data. They only present the time it takes to compute variables' values, as this is what we need to perform the tag simulation.

[5] The simulation overhead in case of [9] was 300 to 900 times.

References

[1] *http://www.systemc.org*

[2] D. Gajski, J. Zhu, *et al.*, *SpecC: Specification language and design methodology*, Kluwer Academic Publishers, New York, 2000.

[3] F. Fallah, S. Devadas, and K. Keutzer, "OCCOM: Efficient computation of observability-based code coverage metrics for functional simulation," in *Proc. Design Automation Conf.*, pp. 152-157, June 1998.

[4] S. Devadas, A. Ghosh, and K. Keutzer, "An observability based code coverage metric for functional simulation," in *Proc. Int. Conf. Computer-Aided Design*, pp. 418-425, Nov. 1996.

[5] B. Beizer, *Software Testing Techniques*, Van Nostrand Rheinhold, New York, second edition, 1990.

[6] B. Marick, *The Craft of Software Testing*, Prentice-Hall, Englewood Cliffs, New Jersey, 1995.

[7] J.M. Voas, "PIE: A dynamic failure-based technique," *IEEE Trans. on Software Engineering*, Vol. 18-8, pp. 717-727, August 1992.

[8] T. Goradia, "Dynamic impact analysis: A cost effective technique to enforce error propagation," in *Proc. Int. Symp. on Software Testing and Application*, Mar. 1993.

[9] J.C. Costa, S. Devadas, and J.C. Monteiro, "Observability analysis of software for coverage-directed validation," in *Proc. Int. Conf. Computer-Aided Design*, pp. 27-32, Nov. 2000.

[10] M. Abramovici, M.A. Breuer, and A.D. Friedman, *Digital Systems Testing and Testable Design*, IEEE Press, New York, 1990.

[11] D. M. Lewis, *Hierarchical Compiled Event-Driven Logic Simulation*, in *Proc. Int. Conf. Computer-Aided Design*, pp. 498-500, 1989.