Improving Validation Coverage Metrics to Account for Limited Observability

Peter Lisherness and Kwang-Ting (Tim) Cheng
Electrical and Computer Engineering Department
University of California, Santa Barbara
Email: {peter,timcheng}@ece.ucsb.edu

***Abstract*—In both pre-silicon and post-silicon validation, the detection of design errors requires both stimulus capable of activating the errors and checkers capable of detecting the behavior as erroneous. Most functional and code coverage metrics evaluate only the activation component of the testbench and ignore propagation and detection. In this paper, we summarize our recent work in developing improved metrics that account for propagation and/or detection of design errors. These works include tools for observability-enhanced code coverage and mutation analysis of high-level designs as well as an analytical method, Coverage Discounting, which adds checker sensitivity to arbitrary functional coverage metrics.**

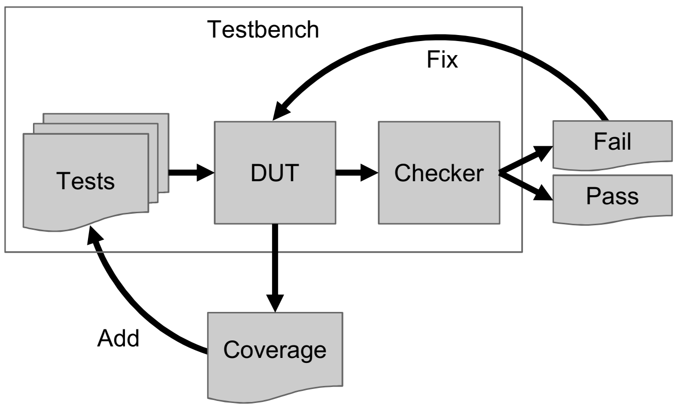
I. Introduction

In order to detect (and subsequently fix) a design error, it is necessary to 1) activate the faulty functionality, 2) propagate the erroneous behavior to a checker, and 3) detect that behavior as erroneous. Functional coverage, as it exists today, typically evaluates only the first of these [1].

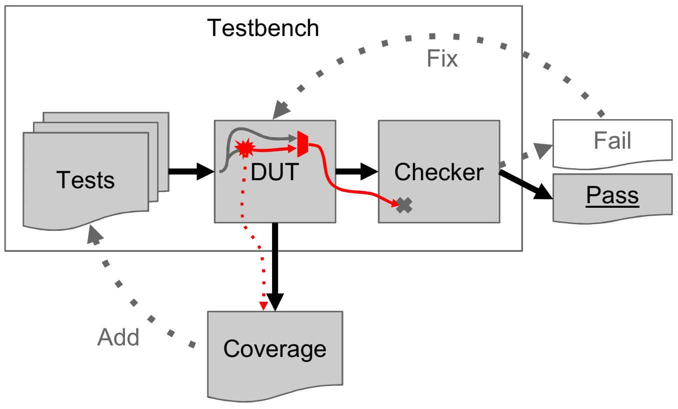
To understand the danger of ignoring propagation and detection, consider the simplified validation flow in Fig. 1a. In the testbench, a set of functional tests are applied to the design under test. The outputs of that design (as well as some internal signals) are checked for correctness by a checker, which may be specific to the test or encode some properties of the design's functionality. Debugging begins when the checker fails a test – the source of this failure is diagnosed and fixed (shown as the "Fix" back edge). At the same time, coverage monitors record the functional coverage of the design. If the coverage is not high enough, additional testcases are written to cover the coverage holes. These two cycles, debug and coverage, are the essence of validation.

Activation-only coverage metrics compromise the integrity of these cycles. Observe Fig. 1b: a fault in the DUT has been activated by a test. If it is not propagated to a checker or the checker that it *does* propagate to is unable to detect it as erroneous, then the test will pass. This provides no opportunity to fix the bug. More troublingly, the functionality associated with that bug will still be considered covered – it *was* activated. This breaks the coverage feedback cycle, as no additional testcases will be written targeting that functionality.

Our solution to this problem is to develop coverage metrics that will only be covered when there is some degree of confidence that any errors were at least propagated to a checker, and ideally detected. The first of these is an observability-enhanced code coverage metric capable of tracking data flows through



(a)



(b)

Fig. 1. (a) Validation flow with debug and test generation cycles. (b) Activated but undetected bugs can break both cycles.

high-level designs, testbench components, and checkers [2]. The second is a mutation coverage tool for C/C++/SystemC designs that provides easy integration with existing flows by hooking directly into the GCC toolchain [3]. Finally, we developed an analytical technique, Coverage Discounting, that uses fault insertion to add checker sensitivity to existing functional coverpoints [4].

II. Observability Coverage

Observability-enhanced metrics, originally proposed in [5], are a class of code coverage metrics that attempt to evaluate the likelihood that bugs are propagated to an output. They do so by extending the notion of coverage from control flow to data flow, where coverage of def-use (DU) or use-def (UD) chains (a data flow analogue to CFG paths) is the additional coverage requirement. Dynamic analysis, like the techniques proposed in [6]–[8], can determine how thoroughly the test stimuli cover these chains and subsequently propagate potential design errors to the outputs.

A. Compatibility with High-Level Models

These metrics are generally only compatible with RTL because its structural requirements allow static analysis to determine all possible data flows. Simulating system-level designs in RTL has become impractical, so large-scale validation typically employs high-level models (ex. SystemVerilog classes) for at least some of the design's components. Since errors may propagate to and back from these high-level descriptions before propagating to a system output, the data-tracking capabilities need to be extended beyond what can be done with static analysis alone.

We recently developed a tool [2] to address this challenge. It uses a number of techniques to expose hidden data flows that cannot be resolved with static analysis, such as the dynamic behavior of pointers, function calls, and object orientation. Moreover, it is capable of tracking control-flow based data dependencies, such as the dependency of `b` on `a` in "$if(a)b = 0$", an extremely common situation which is not adequately addressed by previous observability metrics.

An experimental evaluation of this tool on an instruction set simulator showed coverage results similar to previous RTL-only tools when compared to regular statement coverage. Moreover, the instrumentation overhead was so negligible as to be unmeasurable; although our tool required multiple iterations (32 for this design) to track all dataflow sources, the per-iteration runtime was *less* than that of a regular statement coverage run.

B. Limitations of Observability Coverage

While our results (and those of previous studies) show that observability coverage is an improvement over regular code coverage, it still carries several limitations. The dataflow tracking is approximate, so it can provide incorrect results in the presence of masking or reconvergent fanouts. Also, all existing implementations have greater-than-constant overhead (w.r.t. either the design size or the number of coverpoints) and therefore limited scalability.

Even if these limitations were overcome, observability coverage would still be unable to address two key challenges: First, it is an augmentation of code coverage and there is no clear way to extend it to functional coverage. Second, and more importantly, it assumes the checkers are of sufficient quality. Building a high quality functional checker is challenging – it is effectively impossible to encode the entirety of the correct functional behavior. And unlike manufacturing test, simply comparing the output to known good responses is not generally an option: most modern systems exhibit a degree of nondeterminism, and the correct output is not always known.

TABLE I
ERROR/MUTANT TYPES IN SCEMIT

Name	Full Name	Example
OPR	Operator Replacement	a=b+1 ⇒ a=b-1
VCR	Var⇒Constant Repl.	if (a) ⇒ if (true)
CCR	Constant Replacement	a=b+1 ⇒ a=b+0
ROR	Relational Op. Replacement	a¿=b ⇒ a==b

III. MUTATION / FAULT INSERTION

The importance of propagation and detection as well as the relatively simple activation criteria of code coverage metrics led to the development of mutation analysis, originally for the purpose of software testing [9]. In mutation analysis, the original source code is *mutated* by one of a number of *mutation operators*, which inject errors such as changing an addition operator to a subtraction operator or replacing a variable in the LHS of an assignment with a constant. This *mutation* produces a *mutant* variant of the original source code, which is then tested with the verification test suite. If the test suite cannot distinguish the *mutant* as erroneous, there are two possible conclusions: 1) the test set is inadequate, as it cannot distinguish an intentionally buggy version of the source code from the original, or 2) the injected error produced a functionally equivalent piece of software. In the first case, the verification engineer is expected to improve the test set, adding a test to distinguish this *mutant* from the original. This is analogous to the gate-level stuck-at fault model, where an undetected fault either guides additional test generation or produces equivalent logic (such that no test can be generated).

A. SCEMIT

To facilitate mutation analysis research in high-level flows using SystemC models, we developed "SCEMIT": A SystemC Mutation and Error Insertion Tool [3]. SCEMIT is implemented as a plugin for GCC, and thus supports error injection in C, C++, and SystemC (using the OSCI [10] libraries) models.

1) Error Models: The mutation operators/error models supported by SCEMIT are summarized in Table I, and described in greater detail here:

- The OPR error type will replace the binary (i.e. taking two operands, as opposed to *unary*) operator in an expression with every other valid operator. The set of operators available in the GCC intermediate representation includes basic arithmetic, bitwise logic, and the min&max functions.
- The VCR and CCR error types replace an operand or value in the RHS of an assignment or the condition of a branch with a constant value. The constant currently takes *true* or *false* for branch conditions and values 0, 1, and -1 (or MAXINT for unsigned) for integer data types, but additional values can be forced if needed. CCR additionally perturbs the existing constant value by +1 and -1 (again, additional values can be forced), and omits any replacements to the same constant value (i.e. it will not attempt to replace 0 with 0).

- The ROR error type replaces a relational or equality operator (==, !=, >=, <=, <, >) with every other equality operator, typically within the condition of a branch.

2) Mutation Coverage Flow Example: Figure 2 shows an exemplary flow for using SCEMIT to compute selective mutation coverage, limited of course to the subset of mutant operators supported by SCEMIT. Given an existing design built with the 'make' command and the test set executed with 'make check' that generates a golden output, the following steps would be needed:

1) The program is built and run *without* error injection enabled but *with* the SCEMIT plugin loaded using the appropriate compiler flags (typically with the CFLAGS or CXXFLAGS environment variables). The list of error sites is gathered from the build (output by default on STDERR) and a golden reference output is saved from the tests.
2) A shell script repeatedly invokes the build process and tests again, passing the indexes of each of the desired mutants to SCEMIT (again through compiler flags) and storing the resulting output. An execution timeout is generally needed in this loop, as some injected errors will prevent the simulation from halting.
3) Once all errors have been injected and simulated, the 'diff' command is used to determine which of the errors caused an observable difference in the output.

While this is only a simple example, relatively few changes are needed to adapt it for different build/simulation flows or other types of analysis.

B. Limitations of Mutation

While mutation analysis is a well studied technique, it has several shortcomings limiting its adoption in industrial validation environments. First, the runtime can be prohibitive, particularly when dealing with large RTL designs. Second, the analysis of undetected mutants can be very challenging. The validation engineer is tasked with analyzing the mutant and devising a test that activates it, propagates the faulty value, and is capable of detecting it – all of which require significant understanding of the implementation details. Finally, and most problematically, mutants are not guaranteed testable at all. The untestable, or "equivalent", mutants can require an enormous amount of manual effort to determine that they are indeed untestable. Such efforts are essentially wasted: no new testcases are written, no new checkers are developed, and no additional insight into the real errors present in the design are gained.

IV. Coverage Discounting

Coverage discounting [4] is an analytical method that, for any given implicit or functional coverage metric and any given fault model or injection mechanism, extracts meaningful coverage holes from the functionally meaningless or misleading fault insertion results. The basic premise is as follows:

1) An inserted fault may change the design's functionality.

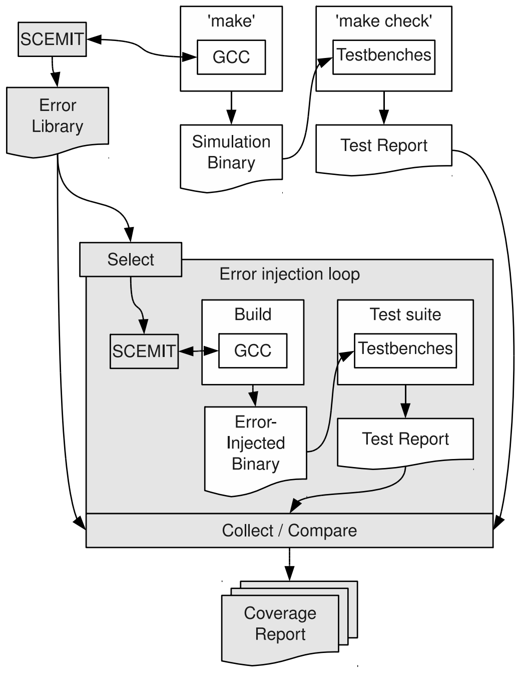


Fig. 2. Flow for computing mutation coverage with SCEMIT. Items shown in gray are not part of the original simulation flow.

2) This change may result in a different functional coverage score.
3) If the fault is not detected, then those covered points which are no longer covered in the presence of the fault should not be considered covered in the fault-free design either.

Thus, the detection information from fault insertion is mapped back onto the coverage metric, and any undetected coverpoints are removed from the "covered" set. After multiple fault insertions, the revised coverage score will better reflect the portion of the coverpoints that were meaningfully tested. The removed coverpoints still represent test holes that need to be more carefully targeted.

Our technique can be applied to any activation-based coverage metric, including implicit and functional metrics, and the fault detection can use any fault insertion methods. This methodology is adaptable to any tool, design language, or simulation environment that is capable of producing coverage and fault detection data; all that is needed is a simple tool to parse and manipulate that data.

A. Discounting Example

In this section, we will walk through a simplified example of how coverage discounting works for a single coverpoint. For ease of explanation we use statement coverage as the coverage metric and mutation analysis as the fault model. Note that functional coverage is used in the case study of Sec. IV-B: although discounting also works with implicit metrics such as statement coverage, functional coverage is more frequently relied upon in practice.

The following listing shows a block of code (the DUT or design under test) with a built-in assertion checker:

Listing 1. Example Design

```
1 //Test Vectors: $a_1$=2, $a_2$=1, $a_3$=0
2 input a; output b;
3 if (a>=0)
4 ⇒ b=1+a;
5 else
6    b=1-a;
7 //Test Outputs: $b_1$=3, $b_2$=2, $b_3$=1
8 assert(b!=0); //The checker
```

After simulation of vector a_1=2, the code coverage report shows that line 4 is covered (according to the statement coverage metric).

We will next run a fault insertion campaign using mutation analysis. One of the inserted faults changes the code to the following:

Listing 2. Example Design with Mutant

```
1 //Test Vectors: $a_1$=2, $a_2$=1, $a_3$=0
2 input a; output b;
3 if (a<=0) //Fault, orig. (a>=0)
4 ⇏ b=1+a;
5 else
6    b=1-a;
7 //Test Outputs: $b_1$=-1, $b_2$=0, $b_3$=1
8 assert(b!=0); //The checker
```

Here, the inequality on line 3 has been switched from ">=" to "<=". Simulation of this fault with the test vector a_1=2 and checker `assert(b!=0)` will pass, seeing as the checker is insufficient to detect the result b_1=-1 as erroneous. Simulating only this test vector, the fault is undetected.

Also, the coverage report has changed: line 4 was no longer covered in the presence of the fault. From this we conclude that the stimulus and checker were not sufficient to thoroughly cover line 4, so it is removed from vector a_1's coverage score. This is the essence of coverage discounting: we reduce coverage scores (in this case removing line 4 from a_1's covered set) to reflect the validation holes revealed by undetected faults.

1) Fixing the Hole: Although we determined that a_1 did not meaningfully cover that statement, some other stimulus in the test set may. For instance, the test vector a_2=1 would cover that statement *and* allow the checker to detect the fault. Alternatively, a more complete checker, such as `assert(b==abs(a)+1)`, would allow even the original vector a_1 to cover line 4.

2) Redundant Faults: The primary problem with mutation analysis is its tendency to create equivalent mutants, a form of redundant fault that is by definition not testable. Although some can be automatically classified as redundant, the rest can require enormous manual effort to diagnose.

The listing below shows an example of an equivalent mutant:

Listing 3. Example Design with Equivalent Mutant

```
1 //Test Vectors: $a_1$=2, $a_2$=1, $a_3$=0
2 input a; output b;
3 if (a>0) //Fault, orig. (a>=0)
4 ⇒ b=1+a;
5 else
6    b=1-a;
7 //Test Outputs: $b_1$=3, $b_2$=2, $b_3$=1
8 assert(b!=0); //The checker
```

The fault on line 3 does not change the function of the code, because when $a = 0$ (the only input affected), both $b = 1 + a$ and $b = 1 - a$ evaluate to $b = 1$. When performing fault insertion alone, unless identified as a redundant fault, this undetected fault would be considered a fault coverage hole.

Let us examine the behavior of coverage discounting in this scenario. The test vector a_3=0 will cover line 4 normally, but not in the presence of the fault. If the vector a_3=0 is the only vector in the test set, line 4 will not be considered covered. This is a fair assessment: under that vector, there is a redundant code path in the original, unmodified code.

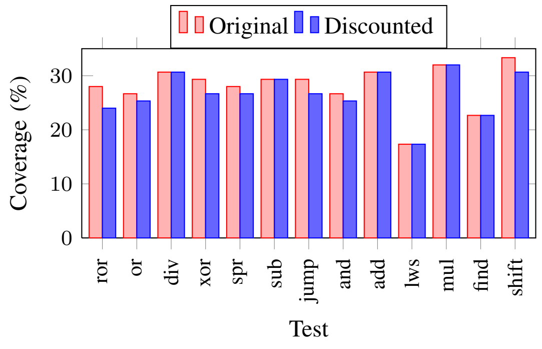
But this statement coverage hole can be filled by other vectors, unlike a redundant fault: a different test vector that exercises line 4, such as the vector a_2=1, can cover this statement. So even though the fault is untestable, any holes it opens when we discount the coverage score can still be covered. This is one of the primary benefits of coverage discounting: redundant faults do not need to be identified, as they are either ignored or expose fundamental redundancies in the underlying implementation.

B. Results from Case Study

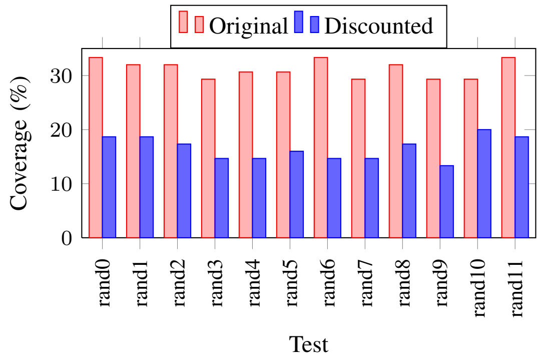
In this case study, we evaluated coverage discounting on the OpenRISC processor. The functional coverage used in this experiment was opcode coverage, and an ad-hoc fault model that suppressed individual opcodes was used.

1) Test Quality: Hand Made vs. Random: As in [5], we can evaluate our coverage methodology based on its ability to distinguish hand-made tests from random tests. This comparison accentuates the problems with activation-only coverage: random testing is very good at activating many functional corners, but hand-written tests tend to do a far better job of propagating the tested behavior to the output, where checkers can detect whether or not it is erroneous.

Fig. 3 shows the original and discounted functional coverage scores for each of the thirteen hand-written functional tests along with the first twelve randomly generated pseudofunctional tests. These results match with what we would anticipate: the functional test scores are not affected much by discounting, since they are written to propagate and thoroughly check the results of every operation. Note that the random tests start with a similar coverage score before discounting but, unlike the functional tests, lose about half of that score after discounting.



(a) Hand-Written Tests



(b) Pseudofunctional Tests

Fig. 3. Coverage of different test sets before and after discounting. Both test sets are approximately comparable before discounting. As expected, the hand-written functional tests (a) do a good job of propagating and checking the results. This can be seen in the relatively small change in coverage score after discounting. The random test programs (b) are not as carefully constructed, and coverage discounting exposes this shortcoming.

TABLE II
COVERAGE DISCOUNTING SUMMARY

Test Set	Functional		Random		Both (F+R)	
Value	Avg.	Total	Avg.	Total	Avg.	Total
Coverage (%)	28	57	31	93	29	100
Disc. Cov. (%)	27	57	17	51	20	69
Change (%)	4	0	47	46	31	31

Table II summarizes both the per-test average scores (coverage before and after discounting) and the cumulative coverage for each test set, as well as the average and cumulative coverage for both test sets combined. It also lists the relative change in coverage score caused by discounting. Note that the total scores are not equal to the sum of the per-test coverage: any coverpoint covered by more than one test will only be counted once.

There are a number of details in Table II worth noting. First, observe that the while some functional tests experienced a small loss of coverage from discounting, the functional test set's total coverage did not change. This indicates that the discounted coverpoints from some tests were covered by others, as expected from a test set where each test is focusing on a specific function. Note also that the aggregate coverage of the random tests is higher than the functional test set before discounting but lower afterwards. While random tests are very good at exploring a wide range of functionality, there is no guarantee that they exercise it meaningfully.

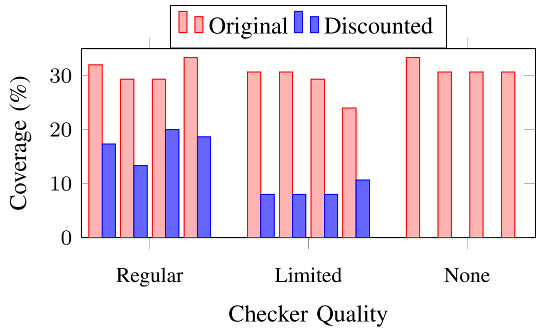


Fig. 4. Coverage for random tests with varying checker quality. The "regular" tests have a thorough checker which compares the entire output to its expected value. The "limited" tests have a checker which only compares the least significant bit of the output, and the "none" tests have no checker at all. Note that all 12 of these tests are different (since the checkers are partially integrated into the tests), leading to the variation in original coverage score.

2) Checker Quality: One of the benefits of coverage discounting over observability coverage metrics is the ability to diagnose checker deficiencies. To demonstrate this capability, additional random tests with less complete checkers but equal observability were generated. Fig. 4 shows the original and discounted coverage scores for 12 random pseudofunctional tests broken into three categories. The first category, "regular", has a checker that compares the entire output to its expected value. These tests are the same as rand8 - rand11 from Fig. 3b, and are repeated here to ease comparison with the other categories.

The second category, "limited" has a checker that compares the least significant bit of the output with its expected value. This leads to a decrease in coverage after discounting because a less thorough checker detects fewer faults, and therefore the tests as a whole have lower coverage. Finally, the "none" category contains tests without any output checker. This is the pathological case: it shows that nothing is meaningfully covered in the absence of checkers, and that coverage discounting correctly identifies this situation.

V. Conclusion

Validation has been continually growing as a portion of total hardware engineering costs. To mitigate this growth, coverage metrics are needed that can provide better confidence that the design has been validated, and more precisely guide the generation of additional test cases or testbench improvements if it hasn't.

In this paper we summarized our recent work in this area, which included tools for measuring observability-aware code coverage and mutation coverage in high-level design descriptions, and coverage discounting, our analytical method for mapping mutation or fault insertion results back onto functional coverage metrics.

These techniques only begin to address the problem. Extensions to them, or new approaches entirely, are needed to further analyze the coverage and determine which portions of the functionality we can be confident were tested thoroughly. In doing so, they will need to consider the testbench holistically (including the checkers) and maintain enough flexibility to be compatible with high-level synthesis and other emerging design methodologies.

Acknowledgment

This research was supported by the Semiconductor Research Corporation GRC Computer Aided Design & Test Sciences under task 1853.001.

References

[1] R. Grinwald, E. Harel, M. Orgad, S. Ur, and A. Ziv, "User defined coverage-a tool supported methodology for design verification," in *DAC*, 1998.

[2] P. Lisherness and K.-T. T. Cheng, "An Instrumented Observability Coverage Method for System Validation," in *High Level Design Validation and Test Workshop (HLDVT)*, 2009.

[3] ——, "SCEMIT: A SystemC Error and Mutation Injection Tool," in *Design Automation Conference (DAC)*, 2010.

[4] ——, "Coverage Discounting: A Generalized Approach for Testbench Qualification," in *High Level Design Validation and Test Workshop (HLDVT)*, 2011.

[5] F. Fallah, S. Devadas, and K. Keutzer, "Occom-efficient computation of observability-based code coverage metrics for functional verification," *IEEE TCAD-ICS*, 2001.

[6] 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 (ICCAD)*, 2000.

[7] T. Lv, L. yi Liu, Y. Zhao, H. wei Li, and X. wei Li, "An observability branch coverage metric based on dynamic factored use-define chains," in *Asian Test Symposium (ATS)*, 2006.

[8] T.-Y. Jiang, C.-N. J. Liu, and J.-Y. Jou, "An observability measure to enhance statement coverage metric for proper evaluation of verification completeness," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2005.

[9] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Theoretical and empirical studies on using program mutation to test the functional correctness of programs," in *Symposium on Principles of Programming Languages (POPL)*, 1980.

[10] "Open systemc initiative (osci)," http://www.systemc.org/, Open SystemC Initiative (OSCI). [Online]. Available: http://www.systemc.org/