Coverage Discounting: Improved Testbench Qualification by Combining Mutation Analysis with Functional Coverage

Nicole Lesperance, Peter Lisherness, and Kwang-Ting (Tim) Cheng
University of California, Santa Barbara — Email: {nlesperance,timcheng}@ece.ucsb.edu

***Abstract*—Testbench quality is commonly measured using coverage metrics. Code, toggle, and functional coverage all have a significant flaw: they reflect how well the design is activated by the test vectors, but insensitive to the error detection capabilities of the testbench. Mutation testing is an established technique for scrutinizing testbench checkers, but its adoption has been limited by the manual effort required to analyze the results. This paper describes the use of *coverage discounting*, where undetected mutants are explained in terms of functional coverpoints, thereby exposing testbench deficiencies in terms of design functionality rather than synthetic design errors. Two benchmarks are shown to compare this improved flow against regular mutation analysis. A confidence metric and simulation ordering algorithm optimized for coverage discounting are also outlined.**

I. Introduction

The goal of pre-silicon verification is to expose and correct as many design errors as early as possible. Design complexity often prohibits formal verification methods and exhaustive simulation, leading to a choice in how to spend the available resources dedicated to the verification effort.

Functional coverage, which is central to modern validation efforts, has a significant flaw: it is sensitive to how well the test vectors exercise the design, but insensitive to the error detection capabilities of the testbench. This problem has been known for some time, and many solutions have been suggested.

One solution is mutation testing, wherein an error is inserted into the design. If the testbench is incapable of detecting the mutation, it is inferred incapable of detecting real design errors. Mutation testing has garnered interest for hardware validation [1]–[3], but the computational overhead of repeated simulation and the effort needed to analyze the results are a barrier to its adoption. In mutation testing, a *syntactic change* is made to the design under test, potentially resulting in a *functional change*. If the mutation is not detected by the testbench, then any functionality altered by the mutation has not been sufficiently exercised. However, determining *which* functions it changes, if any at all, requires manual analysis by someone familiar with both the verification environment and the design implementation.

Coverage Discounting [4], [5] simplifies this analysis by mapping the mutation results onto functional coverage. Using coverage discounting alongside mutation analysis can reveal *which functions are changed* in the mutant, and in turn what is not being adequately tested if the mutation is undetected. This information, expressed in terms of design coverpoints, is determined by comparing the coverage of *both the original and mutated designs*. Any loss of functional coverage when simulating the mutated design indicates functionality changed by that mutation. The lost coverpoints are no longer considered covered, but rather "discounted".

In this work, we will explain in detail how the discounting process works, and step through a short example. We will then examine two benchmark designs, and show how coverage discounting can expose checker deficiencies, and reduce the number of mutants which require further analysis. We also summarize how our improved analysis can reduce the overhead of mutation analysis by prioritizing the most useful mutant simulations and allowing early termination of the mutation testing process.

II. Related Work

The most closely related works are [4] and [5]. As explained in the introduction, this paper provides clarification and details on the discounting technique omitted from these related works.

A. Mutation Analysis

An enormous amount of previous research has gone into mutation analysis [6]. Typically this has focused on reducing the number of mutants inserted, either by choosing a subset of mutation operators or by crafting mutants based on higher-level descriptions [1], [2]. This serves the same purpose as our work, reducing the runtime and easing the analysis of undetected mutants, but ultimately the validation engineer must analyze synthetic faults rather than functional coverpoints. Moreover, there is nothing to prevent the joint use of discounting with these models.

In [7], the authors attempt to identify equivalent mutants by observing changes in code coverage. This solves one of the problems addressed by using coverage discounting – filtering equivalent mutants – but does not offer the other benefits of discounting in analyzing mutants that are *not* equivalent.

B. Coverage Metrics

This work focuses on functional coverage metrics, which are becoming the dominant coverage methodology in large-scale modern validation environments, superseding code coverage metrics such as statement coverage. To our knowledge, [5] is the only attempt to add propagation or checker sensitivity to

functional coverage. Previous attempts have been made to add propagation sensitivity to statement coverage, most of them related to [8]. The primary difference between these works and ours is their reliance on statement coverage; there is no clear way that their "observability coverage" concept could be adapted to functional coverage. Also, these previous works evaluate only whether potential error effects can propagate to checkers, but not whether the checkers are of sufficient quality to detect errors.

III. Coverage Discounting

A. Discounting Conditions

For a coverpoint C to be discounted by a specific test T, and mutation M, the following conditions must be true:

1) C must be covered by T in the fault-free design, but not covered by T in M
2) M is not detected by the testbench

Coverage discounting relies on the observation that if a testbench cannot even detect the difference between a design which covers certain functionality and a design which does not, the testbench is incapable of detecting potential errors associated with that functionality.

B. Discounting Flow

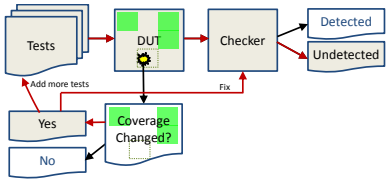


Fig. 1. Discounting Flow

Figure 1 shows the path which leads to discounted coverage for a test and mutant. The discounting algorithm is shown in Figure 2.

Input:
Test Set: T
Fault-Free Design: D
Set of Mutated Designs: M
Coverpoint Set: C

Output: Set of discounted coverpoints for each test: $C_{discount,t}$

```
1   for each t ∈ T:
2       //Initialize discounted point set
3       C_discount,t ← ∅
4       //Simulate and compute fault-free coverage
5       C_t ← computeCoverage(t, D)
6       //Find out which mutants are activated by t
7       for each m ∈ M
8           A_t ← returnActivatedMutants(t, m, D)

9   //Main discounting loop
10  while (t, m) ← selectTestMutantPair(T, A_t)
11      {C_t,m, S} ← computeCoverage(t, m)
12      //Points are discounted if their coverage changes under
13      //the fault and the fault is undetected
14      if S equals pass and C_t − C_t,m is not ∅:
15          C_discount,t ← C_discount,t ⋃ {C_t − C_t,m}
```

Fig. 2. Discounting Algorithm

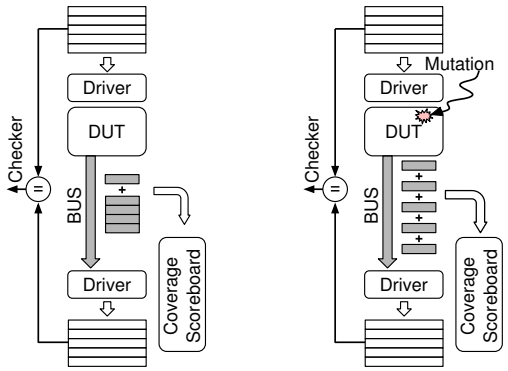
The output of the discounting technique, $C_{discount,t}$ can interpreted in several ways. A stricter definition of a discounted point, p, must satisfy Condition 1:

$$\forall t : p \in C_{discount,t} \tag{1}$$

This definition is used if the designer considers a point adequately covered if at least a single test can meaningfully cover the point. If the designer desires that the point be covered by a larger subset of tests, $T_p \subset T$, Condition 2 must be satisfied:

$$\forall t \in T_p : p \in C_{discount,t} \tag{2}$$

C. Discounting Example



(a) Original Design (b) Mutated Design

Fig. 3. Example – a bus interface controller. A mutant disabling burst mode is not detected, in turn causing the burst mode coverpoint to be discounted.

Suppose we are validating a bus interface controller, as illustrated in Fig. 3a. The portion of the controller which enables burst transactions is described in the following listing:

Listing 1. Original Design

```
1  if (xact_len >= 5)
2    burst:=true, xact_len:=xact_len-4;
3  else
4    burst:=false, xact_len:=xact_len-1;
```

Note that this controller has a bug which is undetected: the number of packets sent in burst mode is assumed to be 5 in the condition and 4 during the transaction length calculation. This causes 4 packets, followed by 1 packet to be sent instead of 5 together.

A testbench sends a transaction through the controller and captures the output on the far side of the bus. The checker[1] ensures that the received data matches the sent data. If so, the test will pass, and as long as a transaction with length ≥ 4 is sent by the testbench, the burst mode will be enabled and considered covered. However, consider the following mutant, where an error has been inserted into the controller's logic:

Listing 2. Mutated Design

```
1  if ( false )
```

[1]A "checker" is a collection of assertions, error checks, and other testbench components that together output a binary pass/fail verdict for each test.

Metric ⇓ \ DUT ⇒		CPU - Original	CPU - Improved	UART
Functional	Total	4816	–	846
	Covered	2091 (44%)	–	842 (99%)
Mutation	Total	9027	–	1588
	Disabled	844 (9%)	–	309 (19%)
	Not activated	1188 (13%)	–	7 (<1%)
	Not propagated	1267 (14%)	–	106 (7%)
	Not detected	2641 (30%)	2595 (29%)	33 (2%)
	Detected (Killed)	3087 (34%)	3133 (35%)	1133 (71%)
Discounting	Total	4816	–	846
	Covered	1439 (30%)	1566 (33%)	839 (99%)
	Discounted	652 (14%)	525 (11%)	3 (<1%)

TABLE I
COVERAGE RESULTS FOR THE BENCHMARKS UNDER FUNCTIONAL, MUTATION, AND DISCOUNTED COVERAGE ANALYSIS.

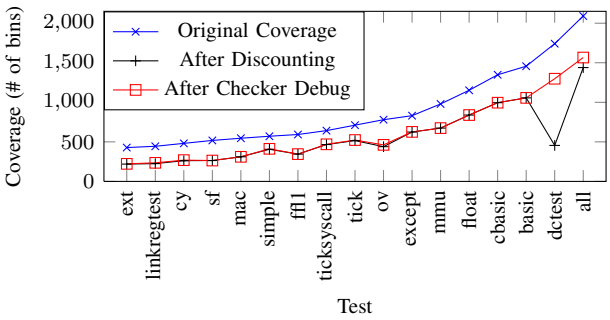


Fig. 4. Discounted coverage of the functional tests with different checkers. The original checker was inadequate, leading to a lower discounted coverage score. By improving the checker, the discounted coverage score was improved.

```
2    burst:=true; len_next:=xact_len-4;
3  else
4    burst:=false; len_next:=xact_len-1;
```

The overall effect of this mutation is illustrated in Fig. 3b. The output seen on the far side of the bus is identical in both the original and mutated design – it merely arrives later due to being broken up into smaller pieces – so this mutation is undetected by the checker. Since the checker cannot distinguish between normal and burst mode operation, clearly the burst functionality has not been meaningfully covered.

Using only mutation analysis, it is easy for a validation engineer to locate mutated source code and analyze its local behavior, but it can be difficult to use this information to determine a mutation's effect on design functionality and uncover test or checker deficiencies. Coverage discounting, in this example, explicitly links the `xact_len >= 4` ⇒ `false` mutation with the bus's burst mode coverpoint because

1) The burst mode coverpoint is no longer covered in the mutated design
2) The mutation is undetected by the testbench

By providing a more accurate coverage score and explaining the meaning of the mutation in terms of design function, it is more likely that the real bug in this controller will be exposed.

IV. COVERAGE VS. MUTATION VS. DISCOUNTING

In this section, we contrast functional coverage, mutation analysis, and discounted coverage for two benchmark designs. The overall metrics are summarized in Table I. Mutation analysis is performed using Certitude [3].

A. OpenRISC CPU

This experiment uses the OpenRISC SoC from OpenCores [9]. There are 16 functional test programs packaged with the CPU. Coverpoints are created from the CPU's top-level signals, and also for each OpenRISC opcode.

1) Functional Coverage: Simulation of all testcases results in 44% coverage. Note that these coverpoints were not carefully scrutinized, and some may be uncoverable. That being said, it is also likely that the tests packaged with this design are inadequate, and the gaps in functional coverage suggest to a validation engineer what additional testing is needed.

2) Mutation Analysis: In total, there are 9027 mutants. Certitude breaks these into 5 categories, as shown in Table I:

Disabled mutants are those considered uninteresting by the mutation tool or statically determined to be untestable.
Not activated mutants never produce a result differing from the unmutated design under the given tests.
Not propagated mutants are those whose error effect did not propagate to the top level of the CPU.
Not detected mutants propagated, but were not detected.
Detected mutants were detected by the testbench. These are commonly referred to as "killed" in mutation literature.

In general, 100% coverage is achieved only if there are no mutants in the **not– activated/propagated/detected** categories. But some of these mutants may be functionally undetectable: for example they may only affect a signal during a "don't care" cycle or cause irrelevant simulation artifacts. There are a total of 5096 such mutants that fall into these categories and require additional analysis.

3) Discounted Coverage: Discounting revealed 652 coverpoints among those originally considered "covered" (2091 in total) that have not been tested thoroughly enough. Many of these are data bus or register value bins, which are not often carefully targeted by validation engineers (additional pseudo-random seeds are used to hit them). Yet others are associated with functions activated by all tests but explicitly tested by none of them (such as reset functions).

Improved Checker: Fig. 4 shows the coverage on a per-test basis, where **Original Coverage** is the functional coverage before discounting, and **After Discounting** is the discounted coverage. "All" refers to the aggregate coverage. Note that most of the tests experienced about the same loss in coverage from discounting, save one: `dctest`.

Analysis of the checker packaged with these tests reveals the culprit: the checker simply searches the debug output for

a specific "passing" value of `8000000d`, and considers the test as passing if it is present. This is sufficient for most of the test programs, which perform numerous internal consistency checks before outputting the passing value.

"`dctest`" outputs large amounts of debug data but does not perform as many internal consistency checks. Presumably the author of this test expected that the output checker would check the entirety of the output against the golden output. Because of this apparent miscommunication, significant portions of the CPU are untested by this test, *despite their showing up as being covered in the original functional coverage score.*

The checker is then modified to compare the entire debug output with the correct response. This leads to the new coverage shown on Fig. 4 as **After Checker Debug**. Improving the checker increases the discounted coverage for many tests, but particularly for the one that relies heavily on the debug output.

Note that the checker improvement is also reflected in the mutation scores, but the difference is negligible. Moreover, the test-by-test comparison that revealed the inadequate checker is not directly possible with the mutation analysis results: the mutations are a completely different set of features than the functional coverpoints, and per-test mutant detection information is incomplete due to fault dropping[2].

B. 16550 UART

This experiment uses a UART (universal asynchronous receiver/transmitter), also available from OpenCores [10]. The testbench is a proprietary OVM-based suite provided by an EDA tool vendor. This testbench includes 75 test sequences of directed pseudo-random test stimuli, a functional checker, and a hand-written set of 846 functional coverpoint bins.

1) Functional Coverage: As shown in Table I, the UART's functional coverage was in excess of 99%. This is expected: the UART has a hand-pruned set of functional coverpoints, so the score is not reduced by irrelevant or invalid bins. It also has a more thorough testbench, 75 tests compared to the CPU's 16, despite being a smaller design.

2) Mutation Analysis: Mutation analysis of this design produces higher scores than the CPU, again due to better testbench quality. Still, 146 mutants are classified **not– activated/propagated/detected**, and manual analysis is required to determine if any of these indicates a testbench deficiency.

3) Discounted Coverage: Discounting identifies 3 of the originally covered functional coverpoints that are not thoroughly checked. These are sufficiently few, and can be analyzed individually. One is related to the timeout interrupt. The testcases activate this function, but do not properly compare the interrupt identification register to see that it contains the correct value. A mutant changing this register's value is undetected, in turn causing this coverpoint to be discounted.

The two other discounted coverpoints are related to the loopback function of the UART. This function is activated by multiple testcases, and even explicitly tested by three of them. Discounting tells us that while the loopback test sequences activate the loopback feature, write some data and read it out, the checkers and test sequences do not ensure that the correct data is read back out. These tests would still pass even if the loopback functionality were not implemented (or incorrect), rendering them effectively useless.

[2]In mutation testing, a "killed" mutant will not be tested any further. This is analogous to "fault dropping" in manufacturing test.

V. Confidence Metric

One open question with discounting is whether or not the faults are sufficient to adequately challenge the quality of the testbench. With a general-purpose fault model such as mutation, it is possible that some coverpoints may never have the opportunity to be discounted. In [4], we propose a metric to determine how thorough the fault insertion has been, and in turn how confident we can be that the discounted coverage reflects the true quality of the testbench. Such a metric can identify coverpoints that are neither discounted nor confidently covered, as well as guide test selection to reduce simulation time.

We show that with our ordering heuristic, the discounted coverage score converges faster than when using Certitude's ordering for both designs. This is clearly advantageous if the designer wishes to stop simulation and start revising the testbench after the first discounted point. Our ordering maximizes the confidence metric as early as possible, with the idea being that a designer can set a confidence threshold, and once that threshold is met, simulation can be stopped.

VI. Conclusion

In this paper we reviewed the coverage discounting technique in detail. We demonstrated how discounting works using mutation analysis of two RTL designs, and outlined a confidence-directed simulation ordering. The two benchmarks are quite different in size, quantity/quality of tests, and type of design. In spite of this, coverage discounting was able to reveal gaps in both verification environments, along with illustrating how the proposed simulation ordering was able to reveal those gaps faster than regular mutation testing.

References

[1] N. Bombieri, F. Fummi, G. Pravadelli, M. Hampton, and F. Letombe, "Functional qualification of TLM verification," in *DATE*, 2009.

[2] A. Sen and M. Abadir, "Coverage metrics for verification of concurrent SystemC designs using mutation testing," in *HLDVT*, 2010.

[3] "Certitude," http://www.springsoft.com/, 2012.

[4] P. Lisherness, N. Lesperance, and K.-T. Cheng, "Mutation analysis with coverage discounting," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '13, 2013, pp. 31–34.

[5] P. Lisherness and K. T. Cheng, "Coverage Discounting: A Generalized Approach for Testbench Qualification," in *HLDVT*, 2011.

[6] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. on Software Engineering*, 2010.

[7] D. Schuler and A. Zeller, "(Un-)Covering Equivalent Mutants," in *Intl. Conference on Software Testing, Verification and Validation*, 2010.

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

[9] "OpenRISC Platform SoC," http://opencores.org/project,orpsoc, 2012.

[10] "UART 16550 Core," http://opencores.org/project,uart16550, 2012.