Mutation Analysis with Coverage Discounting

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

***Abstract*—Mutation testing is an established technique for evaluating validation thoroughness, but its adoption has been limited by the manual effort required to analyze the results. This paper describes the use of *coverage discounting* for mutation analysis, where undetected mutants are explained in terms of functional coverpoints, simplifying their analysis and saving effort. Two benchmarks are shown to compare this improved flow against regular mutation analysis. We also propose a confidence metric and simulation ordering algorithm optimized for coverage discounting, potentially reducing overall simulation time.**

I. Introduction

Functional coverage has a significant flaw: it is sensitive to how well the test vectors exercise the design, but insensitive to the error detection capabilites 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 resulting metric can be difficult to interpret: 100% mutation coverage is generally not an achievable goal, and determining which of the undetected mutations warrant attention and how to improve their coverage can be labor-intensive.

The primary goal of our work is to simplify this analysis by mapping the mutation results onto functional coverage. We will also show how our improved analysis can reduce the overhead by prioritizing the most useful mutant simulations and allowing early termination of the mutation testing process.

This work builds on "coverage discounting", which is proposed in [4] and demonstrated with a high-level functional simulator and a manually inserted ad-hoc fault model. We apply that basic concept to a scenario that more closely resembles what is available in today's chip validation environments and show how the process can be optimized for this context.

In this paper, we demonstrate for the first time:

1) The use of coverage discounting to analyze mutants
2) Automation of coverage discounting with a commercial mutation tool
3) A metric to estimate the thoroughness of discounting
4) A coverpoint-oriented test/mutant ordering for discovering discounted coverpoints faster

A. Mutation Testing and Coverage Discounting

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 requires manual analysis by someone familiar with both the verification environment and the design implementation.

978-3-9815370-0-0/DATE13/©2013 EDAA

Using coverage discounting for 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 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".

II. Related Work

The most closely related work is [4]. As explained in the introduction, this paper extends the basic concept from [4]. In particular, we automate discounting using a mutated RTL design and propose a confidence metric and optimized simulation ordering.

A. Mutation Analysis

An enormous amount of previous research has gone into mutation analysis [5]. 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 [6], 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, [4] 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 [7]. 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

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.

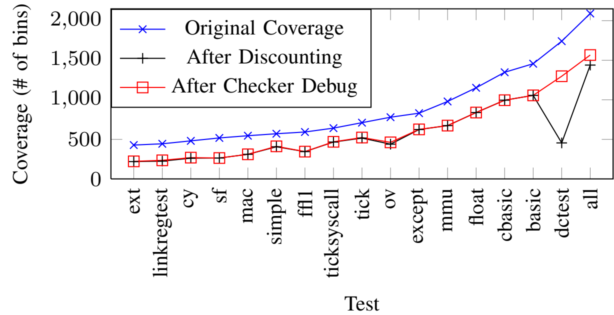


Fig. 1. Discounted coverage of the functional tests with different checkers. The original checker was inadequate, leading to a lower discounted coverage score. By improving the checker, the discounted coverage score was improved.

evaluate only whether potential error effects can propagate to checkers, but not whether the checkers are of sufficient quality to detect errors.

III. 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 [8]. 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. 1 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. 1 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 informa-

tion is incomplete due to fault dropping[1].

B. 16550 UART

This experiment uses a UART (universal asynchronous receiver/transmitter), also available from OpenCores [9]. 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.

IV. Confidence Metric

One open question with discounting is whether or not the faults are sufficient to adequately challenge the quality of the testbench. This issue is avoided in [4], where the authors use an ad-hoc fault model that was guaranteed capable of discounting all coverpoints. With a general-purpose fault model such as mutation, it is possible that some coverpoints may never have the opportunity to be discounted. In this section 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, as described in Sec. IV-B.

Coverage discounting requires that two things happen. First, a coverpoint that was covered in the original DUT must be "suppressed", or prevented from being covered, in a mutated design. Second, that mutation must be undetected.

The first requirement, coverpoint suppression, is strictly related to the interactions of the mutations and the coverpoints. The second requirement, that the mutation be undetected, is a function of the checker and the mutation. The goal of this confidence metric is to measure the thoroughness of meeting the first requirement; the quality of the checker is independent of the coverpoints and is not considered in this metric.

To understand our emphasis on suppression of coverpoints, consider the following pathological scenario: Suppose we have a DUT with many testcases and coverpoints, but absolutely no checking. Regardless of what fault is inserted or which testcase is simulated, the fault is undetected. By extension, any actual design errors that are activated by the testcases are not detected either. Intuitively, the lack of checking means that none of the coverpoints have been meaningfully covered, so all of them should be discounted *if sufficient faults are inserted*.[2]

But what if sufficient faults (either in quality or quantity) are not inserted? Suppose instead that we inadvertently insert only faults that do not change the functional coverage. That is, they suppress no coverpoints. Even if all of these faults are undetected, no coverpoints will be discounted.

A. DECO Score

Based on these observations, we propose a DEtection COnfidence metric, hereafter abbreviated as DECO:

Given:

- T, a set of testcases
- P, the set of functional coverpoints
- $P(t);\, t \in T$, the set of coverpoints covered by test t
- F_s, the set of faults that have been simulated
- $P_f(t);\, f \in F_s, t \in T$, the set of coverpoints covered by testcase t in the presence of fault f

Definitions:

- $S_f(t) := P(t) - P_f(t)$ is the set of coverpoints suppressed by fault f in testcase t.
- $S_{f,p} := \exists t \in T : p \in S_f(t)$ is a suppression test, which is true if fault f has suppressed coverpoint p in any testcase.
- $F_p := \{f \in F_s : S_{f,p}\}$ is the set of faults that have suppressed coverpoint p in any testcase.
- $|F_p|$, or "point confidence", is the number of faults that have suppressed coverpoint p.
- $D_n := \{p \in P : |F_p| \geq n\}$ is the set of coverpoints with point confidence of at least n. The size of this set as a portion of all coverpoints is the "DECO score".

Setting a higher threshold n will require that more mutants be inserted to achieve a given DECO score. This is very similar to the n-detection concept used in manufacturing test; as with n-detection, the suitable threshold n is a heuristic and should be determined empirically. Such a determination requires additional case studies and is beyond the scope of this paper.

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

[2] Note that this pathological case was demonstrated in [4], and produced the expected discounted coverage score of zero.

B. Test and Mutant Selection

While discounting incurs a manageable amount of overhead with respect to mutation analysis, mutation analysis itself is still relatively costly. The most direct way to reduce this cost is to limit the number of mutants simulated or the number of tests simulated per mutant. Some existing techniques that limit the number of simulations are discussed in Sec. II-A.

All of these prior works focus on accurately approximating the mutation metrics. When coverage discounting is used to analyze the mutation results, the goal shifts from selecting tests/mutants that approximate the final mutation coverage metric to selecting tests that help the discounted coverage converge faster. In this section we propose a method for selecting tests and mutants for simulation when discounted coverage is the desired result.

Each time another mutant simulation begins, the test and mutant are chosen (from the set of those not yet simulated) based on the following two criteria:

1) Select the test covering the greatest number of coverpoints with low point confidence $|F_p|$.
2) Of the mutants activated by this test, select the mutant activated by the fewest tests.

This test selection criterion is intended to maximize the DECO score: a test must normally cover a coverpoint for it to be suppressed by a mutation. Therefore, the test covering the most low confidence coverpoints has the best opportunity to suppress them and in turn increase the overall confidence.

The chosen mutant is contingent on the test selected, because inserting a mutation that is not activated by that test is a waste of simulation time. Of these activated mutants, the one activated by the fewest tests is selected due to the following heuristic: when a fault is activated by fewer tests, this indicates that it is modifying less commonly exercised functionality. Such a mutant may be useful for discounting the coverpoints also associated with that functionality.

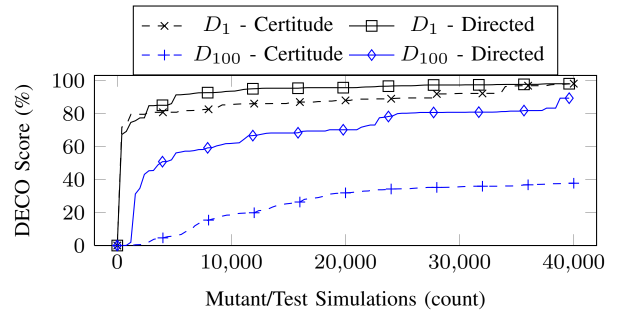
C. Experimental Results

Fig. 2a shows the effect of the proposed ordering method on the DECO scores in the CPU benchmark. As intended, the DECO scores of the directed ordering quickly surpass those of the Certitude ordering. More importantly, this increased confidence is matched by accelerated discounting of coverpoints, as shown in Fig. 2b.

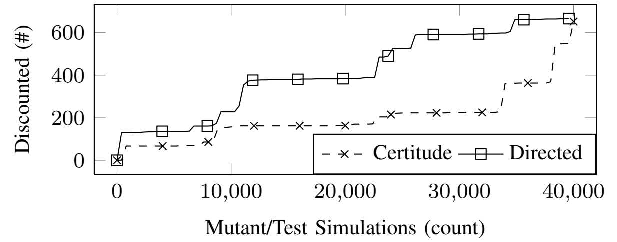
Incidentally, the directed test/mutant ordering also arrives at the final mutation testing score with slightly fewer simulations than the original Certitude ordering. Recall that fault dropping is used, so the ordering will affect how many tests are simulated with a given mutant before it is detected (if at all).

This ordering also allows for early termination of mutation testing. The choice to terminate could be based on a DECO threshold being reached or a sufficient number of discounted coverpoints being identified for further analysis: both of these are achieved faster with the proposed ordering.

Note also that at the end of simulation, a greater number of discounted coverpoints are identified using the new ordering.



(a) CPU DECO scores based on test/mutant ordering



(b) CPU discounted coverpoints based on test/mutant ordering

Fig. 2. DECO scores and discounting progress with Certitude's original test/mutant ordering and the proposed confidence–directed ordering.

This discrepancy is also due to fault dropping: a coverpoint can be discounted in one test by a mutation that is detected by a different test. If the order of these tests is reversed, the mutant will be dropped before it has a chance to discount the coverpoint. Therefore, when discovering discounted coverpoints is more of a concern than simulation time, it may be best not to perform fault dropping. Instead, every mutant should be simulated with every test that activates it.

V. Conclusion

In this paper we automated coverage discounting by using it for mutation analysis of an RTL design. We also proposed a confidence metric and confidence-directed simulation ordering. The two benchmarks we demonstrated 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, and the proposed test 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 and K. T. Cheng, "Coverage Discounting: A Generalized Approach for Testbench Qualification," in *HLDVT*, 2011.

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

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

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

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

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