The Risks of Coverage-Directed Test Case Generation

Gregory Gay, *Member, IEEE,* Matt Staats, Michael Whalen, *Senior Member, IEEE*, and Mats P.E. Heimdahl, *Senior Member, IEEE*

Abstract—A number of structural coverage criteria have been proposed to measure the adequacy of testing efforts. In the avionics and other critical systems domains, test suites satisfying structural coverage criteria are mandated by standards. With the advent of powerful automated test generation tools, it is tempting to simply generate test inputs to satisfy these structural coverage criteria. However, while techniques to produce coverage-providing tests are well established, the effectiveness of such approaches in terms of fault detection ability has not been adequately studied.

In this work, we evaluate the effectiveness of test suites generated to satisfy four coverage criteria through counterexample-based test generation and a random generation approach—where tests are randomly generated until coverage is achieved—contrasted against purely random test suites of equal size. Our results yield three key conclusions. First, coverage criteria satisfaction alone can be a poor indication of fault finding effectiveness, with inconsistent results between the seven case examples (and random test suites of equal size often providing similar—or even higher—levels of fault finding). Second, the use of structural coverage as a supplement—rather than a target—for test generation can have a positive impact, with random test suites reduced to a coverage-providing subset detecting up to 13.5% more faults than test suites generated specifically to achieve coverage. Finally, Observable MC/DC, a criterion designed to account for program structure and the selection of the test oracle, can—in part—address the failings of traditional structural coverage criteria, allowing for the generation of test suites achieving higher levels of fault detection than random test suites of equal size.

These observations point to risks inherent in the increase in test automation in critical systems, and the need for more research in how coverage criteria, test generation approaches, the test oracle used, and system structure jointly influence test effectiveness.

Index Terms—Software Testing, System Testing

1 Introduction

In software testing, the need to determine the adequacy of test suites has motivated the development of several classes of test coverage criteria [1]. One such class is *structural coverage criteria*, which measure test suite adequacy using the coverage over the structural elements of the system under test, such as statements or control flow branches. In the critical systems domain—particularly in avionics—demonstrating structural coverage is required by standards [2].

In recent years, there has been rapid progress in the creation of automated test generation tools that direct the generation process towards the satisfaction of certain structural coverage criteria [3], [4], [5], [6]. Such tools promise to improve coverage and reduce the cost associated with test creation.

In principle, this represents a success for software engineering—a mandatory, and potentially arduous, engineering task has been automated. Nevertheless, while there is evidence that using structural coverage to guide random test generation provides better tests than purely random tests [7], the effectiveness of test suites automatically generated to satisfy various structural coverage criteria has not been firmly established.

In previous work, we found that test inputs generated specifically to satisfy three structural coverage criteria via counterexample-based test generation were *less effective* than random test inputs [8]. Additionally, we found that reducing larger test suites for a given coverage metric—in our study, MC/DC—while maintaining the same level of coverage reduced their fault finding significantly, hinting that it is not always wise to build test suites solely to satisfy a coverage criterion [9]. These findings were confirmed in a larger study, where we found that test suites generated to provide branch and MC/DC coverage were less effective than random test suites of the same size [7]. The same study found that using structural coverage as a supplement to random testing was a far more effective practice than generating tests specifically for satisfying that criterion.

More recent work suggests that a number of factors that are currently not well understood can strongly impact the effectiveness of the testing process—for example, the oracle used, the structure of the program under test, etc. [10], [11], [12]. The results of these studies indicate that adequacy criteria that do not take such factors into account may be at a disadvantage to those that do.

These results are concerning. Given that common standards in critical systems domains require test suites to satisfy certain structural coverage metrics—and the rise of automated tools that can provide such coverage—the temptation exists to automate the entire testing process. Before we can recommend

G. Gay is with the Department of Computer Science & Engineering, University of South Carolina. E-Mail: greg@greggay.com
M. Whalen and M. Heimdahl are with the Department of Computer Science and Engineering, University of Minnesota. E-Mail: whalen@cs.umn.edu, heimdahl@cs.umn.edu
M. Staats is with Google, Inc. E-Mail: staatsm@gmail.com
This work has been supported by NASA Ames Cooperative Agreement NNA06CB21A, NSF grants CCF-0916583, CNS-0931931, and CNS-1035715, and the Fonds National de la Recherche, Luxembourg (FNR/P10/03). We would additionally like to thank Rockwell Collins for their support.

such action, however, is it essential to establish the efficacy of the generated test suites.

In earlier work, we reported the results of a study measuring the fault finding effectiveness of automatically generated test suites satisfying two structural coverage criteria: decision coverage and Modified Condition/Decision Coverage (MC/DC coverage), as compared to randomly generated test suites of the same size on four production avionics systems and an example system from NASA [7]. In the work presented in this paper, we have expanded the initial pilot study to cover a wider range of structural coverage criteria, with the goal of making new observations and establishing more generally applicable conclusions.

In this study, we generate tests for seven industrial critical systems—four Rockwell Collins systems from previous work, a NASA system, as well as two subsystems of a complex real-time infusion pump—using both a random test generation approach and counterexample-based test generation [13]—directed to satisfy condition, decision, MC/DC and Observable MC/DC coverage (a coverage metric designed to propagate the impact of condition choices to a program point where they can be observed by the test oracle [12]). Both the generated test suites and random test suites were reduced while maintaining coverage and compared to purely random test suites of equal size. We determined effectiveness of the resulting test suites through mutation analysis [14]—and, for two systems, a set of real faults—using two expected value test oracles: an output-only test oracle and a maximally powerful test oracle (an oracle observing all output and internal state variables).

Our results show that for three of the four coverage criteria—in four of our industrial systems—the automatically generated test suites perform *significantly worse* than randomly generated test suites of equal size (up to 40.6% fewer faults found when coupled with an output-only oracle). For the NASA example and the two infusion pump systems, which were selected specifically because their structures were significantly different from the Rockwell Collins systems, and the Observable MC/DC (OMC/DC) criterion—which was selected for its potential to overcome the shortcomings of MC/DC–test suites generated to satisfy structural coverage—performed better, matching or improving on randomly generated test suites of the same size. However, these tests were still not always effective in an absolute sense, generally finding fewer than 50% of the faults with the standard output-only oracle. Similar trends can be observed when examining systems with real faults, with coverage-directed test generation yielding up to 93.4% worse fault-detection performance.

Finally, we found that for most combinations of coverage criteria and case examples, randomly generated test suites reduced while maintaining structural coverage sometimes find more faults than pure randomly generated test suites of equal size (finding up to 13.1% more faults).

We draw three conclusions from these results. First, automatic test generation to satisfy structural coverage does not, for many of the systems investigated, yield effective tests relative to their size for the commonly-used condition, decision, and MC/DC coverage criteria. This, indicates that satisfying even a "rigorous" coverage criterion can be a poor indication of test suite effectiveness. Furthermore, even when coverage-directed tests yield superior performance, the tests may still miss a large number of potentially severe faults.

Second, the use of structural coverage as a supplement—not a target—for test generation (as Chilensky and Miller recommend in their seminal work on MC/DC [15]) can be an effective practice.

Finally, OMC/DC, unlike other coverage criteria, generally provides the same or better fault finding than random test suites of equal size in our study, indicating that the extensions to consider propagation at least partly—though not completely—address issues related to MC/DC.

These observations have serious implications and point to the risks inherent in the increased use of test automation. The goal of this study is not to discourage the use of one particular form of automated test generation—or to recommend another—but to raise awareness of the risks of assuming that code coverage equates to effective testing. To the best of our knowledge, this paper is the largest such study to date. It demonstrates the potential for automatic test generation to reduce the fault finding effectiveness of test suites and coverage criteria relative to random testing, and it is one of the few such studies that use real-world avionics systems. While our focus is on critical systems, test generation techniques and coverage measurements are used in the verifications of systems across many disparate domains; the issues raised are relevant to those domains as well.

Our results highlight the need for more research in how the coverage criterion, the test generation approach, the chosen test oracle, and the structure of the system under test jointly influence the effectiveness of testing. The increasing availability and use of advanced test-generation tools coupled with the increased use of code coverage in certification—and our lack of knowledge of the effectiveness of such tools and metrics—is worrisome and careful attention must be paid to their use and acceptance.

2 Related Work

A number of empirical studies exist comparing structural coverage criteria with random testing, with mixed results. Juristo et al. provide a survey of much of the existing work [16]. With respect to branch coverage, they note that some authors (such as Hutchins et al. [17]) find that branch coverage outperforms random testing, while others (such as Frankl and Weiss [18]) discover the opposite. Namin and Andrews have found coverage levels are positively correlated with fault finding effectiveness [19]. However, recent work from Inozemtseva and Holmes found low-to-moderare correlation when the number of test cases is controlled for and that stronger forms of coverage do not necessarily lead to stronger fault-finding results [20]. Theoretical work comparing the effectiveness of partition testing against random testing yields similarly mixed results. Weyuker and Jeng [21], and Chen and Yu [22], indicate that partition testing is not necessarily more effective than random testing. Hamlet and Taylor additionally found that partition testing—as commonly used—is often ineffective and has little value in gaining confidence in a

system [23]. Later theoretical work by Gutjahr [24], however, provides a stronger case for partition testing. Arcuri et al. [25] recently demonstrated that in many scenarios, random testing is more predictable and cost-effective at reaching high levels of structural coverage than previously thought. The authors have also demonstrated that, when cost is taken into account, random testing is often more effective at detecting faults than a popular alternative—adaptive random testing [26].

Most studies concerning automatic test generation for structural coverage criteria are focused on how to generate tests quickly and/or improve coverage [27], [3]. Comparisons of the fault-finding effectiveness of the resulting test suites against other methods of test generation are few. Gopinath et al. compared a number of manual and automatically-generated test suites for statement, block, brand, and path coverage for their ability to find fault [28]. They concluded that statement coverage led to the highest level of fault-finding effectiveness. Others that exist apart from our own limited previous work and Gopinath's study are, to the best of our knowledge, studies in concolic execution [4], [5]. One concolic approach by Majumdar and Sen [29] has even merged random testing with symbolic execution, though their evaluation only focused on two case examples, and did not explore fault finding effectiveness.

Despite the importance of the MC/DC criterion [15], [2], studies of its effectiveness are few. Yu and Lau study several structural coverage criteria, including MC/DC, and find MC/DC is cost effective relative to other criteria [30]. Kandl and Kirner evaluate MC/DC using an example from the automotive domain, and note less than perfect fault finding [31]. Dupuy and Leveson evaluate the MC/DC as a complement to functional testing, finding that the use of MC/DC improves the quality of tests [32]. None of these studies, however, compare the effectiveness of MC/DC to that of random testing. They therefore do not indicate if test suites satisfying MC/DC are truly effective, or if they are effective merely because MC/DC test suites are generally quite large.

More concerning than the negative results regarding the ability of structural coverage to enhance fault finding is the overall lack of consensus one way or the other. Certain coverage metrics are used as though their use guarantees effective testing when, in practice, there is no universal evidence of their utility.

Our study applies counterexample-based test generation using the JKind model checker [13], [33] to directly generate test inputs for multiple coverage criteria. In this work, we find issues with the effectiveness of tests generated using this approach. Several problems that can arise when using model checkers to generate tests are discussed by Fraser et al. [34]; however, issues regarding fault-finding effectiveness are not among them. Counterexample-based test generation is simply one method of automated test generation—others include symbolic [35] and concolic execution [4], model-based test generation [36], combinatorial testing [37], and search-based testing [38], among others. For a comprehensive survey on automated test generation, see [6].

The work presented in this paper is an extension of a conference publication [7]. Our current work differs primarily in the number of criteria explored (four, rather than two), and—in particular—the use of OMC/DC [12], a coverage criterion whose definition was in part motivated by the issues raised in our earlier work [7]. We also expand on the programs studied, and include real faults—as opposed to seeded mutations—for two of the systems.

3 Experiment

We are interested in two approaches for test generation: random test generation and directed test generation. As the name implies, in random test generation, tests are randomly generated. Suites of these tests can later be reduced with respect to the coverage criterion—this is akin to the practice of using coverage as an adequacy criterion, where one tests until coverage is achieved. Such an approach that is useful as a gauge of the value of a coverage criterion—if tests randomly generated and reduced with respect to a coverage criterion are more effective than pure randomly generated tests, we can safely conclude the use of the criterion led to the improvement. Unfortunately—other than our previous work [7]—evidence demonstrating this is, at best, mixed for coverage metrics such as decision or condition coverage [16], and non-existent for more stringent forms of coverage, such as MC/DC.

In directed test generation, tests are created specifically for the purpose of satisfying a coverage criterion. Examples include heuristic search methods [38] and approaches based on reachability [27], [3], [4]. Such techniques have advanced to the point where they can be effectively applied to production systems. Although these approaches can be slower than random testing, they offer the potential to improve the coverage of the resulting test suites.

It has been suggested that structural coverage criteria should *only* be used as adequacy metrics—to determine if a test suite has failed to cover functionality in the source code [1], [19]. However, an adequacy criterion can always be transformed into a test suite generation target. In mandating that a coverage criterion be used for measurement, it seems inevitable that some testers will opt to perform generation to speed the testing process, and tools have been built for that purpose [39].

Therefore, in our study, we aim to determine if using existing directed generation techniques with these criteria results in test suites that are more effective at fault detection than randomly generated test suites. We expect that a test suite satisfying the coverage criterion to be, at a minimum, at least as effective as randomly generated test suites of equal size. Given the central—and mandated—role the coverage criteria play within certain domains (e.g., DO-178C for airborne software [40]), and the resources required to satisfy them, this area requires additional study. We thus seek answers to the following research questions:

RQ1: *Are random test suites reduced to satisfy various coverage criteria more effective than purely randomly generated test suites of equal size?*

RQ2: *Are test suites directly generated to satisfy various coverage criteria more effective than randomly generated test suites of equal size?*

	# Simulink Subsystems	# Blocks
DWM_1	3,109	11,439
DWM_2	128	429
Vertmax_Batch	396	1,453
Latctl_Batch	120	718

	# States	# Transitions	# Vars
Docking_Approach	64	104	51
Infusion_Mgr	27	50	36
Alarms	78	107	60
Infusion_Mgr (faulty)	30	47	34
Alarms (faulty)	81	101	61

TABLE 1
Case Example Information

3.1 Experimental Setup Overview

In this study, we have used four industrial systems developed by Rockwell Collins Inc., a fifth system created as a case example at NASA, and two subsystems of an infusion pump created for medical device research [41]. The Rockwell Collins systems were modeled using the Simulink notation and the remaining systems using Stateflow [42], [43]; all were translated to the Lustre synchronous programming language [44] to take advantage of existing automation. In practice, Lustre would be automatically translated to C code. This is a syntactic transformation, and if applied to C, the results of this study would be identical.

Two of these systems, *DWM_1* and *DWM_2*, represent portions of a Display Window Manager for a commercial cockpit display system. The other two systems—*Vertmax_Batch* and *Latctl_Batch*—represent the vertical and lateral mode logic for a Flight Guidance System (FGS). The NASA system, *Docking_Approach*, was selected due to its structure, which differs from the Rockwell Collins systems in ways relevant to this study (discussed later). *Docking_Approach* describes the behavior of a space shuttle as it docks with the International Space Station. The remaining two systems, *Infusion_Mgr* and *Alarms*, were chosen because they come with a set of real faults that we can use to assess real-world fault-finding. These systems represent the prescription management and alarm-induced behavior of an infusion pump device [1].

Information related to these systems is provided in Table 1. We list the number of Simulink subsystems, which are analogous to functions, and the number of blocks, analogous to operators. For the examples developed in Stateflow, we list the number of Stateflow states, transitions, and variables. As we have both faulty and corrected versions of Infusion_Mgr and Alarms, we list information for both.

Note that Lustre systems, and the original Simulink systems from which they were translated, operate in a sequence of steps. In each step, input is received, internal computations are done sequentially, and output is produced. Within a step, no iteration or recursion is done—each internal variable is defined, and the value for it computed, exactly once. The system itself operates as an large loop.

For each case example, we performed the following steps:

1) **Generated mutants:** We generated 250 mutants, each containing a single fault, and removed functionally equivalent mutants. (Section 3.2.)
2) **Generated structural tests:** We generated test suites satisfying condition, decision, MC/DC, and Observable MC/DC coverage using counterexample-based test generation. (Section 3.4.)
3) **Generated random tests:** We generated 1,000 random tests of test lengths between 2-10 steps. (Section 3.4.)
4) **Reduced test suites:** We generated reduced test suites satisfying condition, decision, MC/DC, and OMC/DC coverage using the test data generated in the previous two step. (Section 3.5.)
5) **Random test suites:** For each test suite satisfying a coverage criterion, we created a single random test suite of equal size. In addition, we created test suites of sizes evenly distributed from sizes 1 to 1,000. (Section 3.5.)
6) **Computed effectiveness:** We computed the fault finding effectiveness of each test suite using both an output-only oracle and an oracle considering all outputs and internal state variables (a *maximally powerful* oracle) against the set of mutants and—for the infusion pump examples—against the set of real faults. (Section 3.6.)

3.2 Mutant Generation

We have created 250 *mutants* (faulty implementations) for each case example by automatically introducing a single fault into the correct implementation. Each fault was seeded by either inserting a new operator into the system or by replacing an existing operator or variable with a different operator or variable. Constructing the specific mutants involved a randomized process in which a list of possible mutants was enumerated. From this list, 250 mutants were selected for generation, with a roughly even distribution of fault types across the system occurring naturally.

The mutation operators used in this study are fairly typical and are discussed at length in [45]. They are similar to the operators used by Andrews et al. where they conclude that mutation testing can be an adequate proxy for real faults for the purpose of investigating test effectiveness [46].

One risk of mutation testing is *functionally equivalent* mutants—the scenario in which faults exist, but these faults cannot cause a *failure* (an externally visible deviation from correct behavior). This presents a problem when using oracles that consider internal state—we may detect failures that can never propagate to the output. We have used the JKind model checker [13] to detect and remove equivalent mutants for the four Rockwell Collins systems[2]. This is made possible thanks to our use of synchronous reactive systems—each system is finite, and thus determining equivalence is decidable[3].

The cost of determining non-equivalence for the Docking_Approach, Infusion_Mgr, and Alarms system is, unfortunately, prohibitive. However, for every mutant reported as killed in our study for the output-only oracle, there exists

1. These two models are available to download from http://crisys.cs.umn.edu/PublicDatasets.shtml

2. The percentage of mutants removed is very small, 2.8% on average

3. Equivalence checking is fairly routine on the hardware side of the reactive system community; a good introduction can be found in [47].

at least one trace that can lead to a user-visible failure, and all fault finding measurements for that oracle indeed measure faults detected.

3.3 Real Faults

For both of the infusion pump systems—Infusion_Mgr and Alarms—we have two versions of each case example. One is an untested—but feature-complete—version with several faults, the second is a newer version of the system where those faults have been corrected. We can use the faulty version of each system to assist in determining the effectiveness of each test suite. As with the seeded mutants, effective tests should be able to surface and alert the tester to the residing faults.

For the Infusion_Mgr case example, the older version of the system contains seven faults. For the Alarm system, there are three faults. Although there are a relatively small number of faults for both systems, several of these are faults that required code changes in several locations to fix. The real faults used in this experiment are non-trivial faults—these were not mere typos or operand mistakes, require specific conditions to trigger, and extensive verification efforts were required to identify these faults. Faults of this type are ideal, as we do not want the generated test cases to trivially fail on the faulty models.

A brief description of the faults can be seen in Table 2.

3.4 Test Data Generation

In this research, we explore four structural coverage criteria: condition coverage, decision coverage, Modified Condition/Decision Coverage (MC/DC) [16], [15], and Observable Modified Condition/Decision Coverage (OMC/DC) [12].

Condition coverage is a coverage criterion based on exercising complex Boolean conditions (such as the ones present in many avionics systems). For example, given the statement `((a and b) and (not c or d))`, achieving condition coverage requires tests where the individual atomic boolean conditions `a, b, c, and d` evaluate to true and false.

Decision coverage is a criterion concerned with exercising the different outcomes of the Boolean decisions within a program. Given the expression above, `((a and b) and (not c or d))`, tests would need to be produced where the expression evaluates to true and the statement evaluated to false, causing program execution to traverse both outcomes of the decision point. Decision coverage is similar to the commonly-used branch coverage. Branch coverage is only applicable to Boolean decisions that cause program execution to branch, such as that in if or case statements, whereas decision coverage requires coverage of all Boolean decisions, whether or not execution diverges. Improving branch coverage is a common goal in automatic test generation.

Modified Condition/Decision Coverage further strengthens condition coverage by requiring that each decision evaluate to all possible outcomes (such as in the expression used above), each condition take on all possible outcomes (the conditions shown in the description of condition coverage), and that each condition within a decision be shown to independently impact the outcome of the decision. Independent effect is defined in terms of *masking*, which means that the condition has no effect on the value of the decision as a whole; for example, given a decision of the form `x and y`, the truth value of `x` is irrelevant if `y` is false, so we state that `x` is masked out. A condition that is not masked out has *independent effect* for the decision.

Suppose we examine the independent affect of `d` in the example; if `(a and b)` evaluates to false, than the decision will evaluate to false, masking the effect of `d`; Similarly, if `c` evaluates to false, then `(not c or d)` evaluates to true regardless of the value of `d`. Only if we assign `a`, `b`, and `c` true does the value of `d` affect the outcome of the decision.

MC/DC coverage is often mandated when testing critical avionics systems. Accordingly, we view MC/DC as likely to be effective criteria, particularly for the class of systems studied in this report. Several variations of MC/DC exist—for this study, we use Masking MC/DC, as it is a common criterion within the avionics community [48].

Observable MC/DC (OMC/DC) is an enhanced version of MC/DC that requires that tests not only exercise the Boolean conditions and decisions within program expressions, but that tests also offer a path of propagation from that condition to the program output as well. One can view MCDC as determining independent affect of a condition within a decision; OMC/DC requires an analogous effect on some observable quantity for the test (such as a program output variable). While MC/DC ensures that Boolean faults will not be masked at the decision level, it is often the case that the decision will itself be masked before propagating to an output variable. OMC/DC tests have the potential to overcome this weakness by requiring that a path of propagation exists between the condition and an output. For example, consider the following block of pseudocode:

```
x = ((a and b) and (not c or d));
y = x or c;
output = y and (b or d);
```

In MC/DC it is necessary to show that the result of `d` influences the outcome of `x`. In OMC/DC, we must not only demonstrate the independent impact of `d` on `x`, but the independent impact of `x` on `y` and the independent impact of `y` on `output`—thus establishing a propagation path for `(not c or d)` from `x` to the program output.

For our directed test generation approach, we used counterexample-based test generation to generate tests satisfying the four coverage criteria [27], [3]. In this approach, each coverage obligation is encoded as a temporal logic formula and a model checker can be used to detect a counterexample (test case) illustrating how the coverage obligation can be covered. By repeating this process for each coverage obligation for the system, we can use the model checker to automatically derive test sequences that are guaranteed to achieve the maximum possible coverage of the model.

This coverage guarantee is why we have elected to use counterexample-based test generation, as other directed approaches (such as concolic/SAT-based approaches) do not offer such a straightforward guarantee. In the context of avionics systems, the guarantee is highly desirable, as achieving maximum coverage is required [2]. We have used the JKind

	Infusion_Mgr
1	When entering therapy mode for the first time, infusion can begin if there is an empty drug reservoir.
2	The system has no way to handle a concurrent infusion initiation and cancellation request.
3	If the alarm level is $>= 2$, no bolus should occur. However, intermittent bolus mode triggers on alarm $<= 2$.
4	Each time step is assumed to be one second.
5	When patient bolus is in progress and infusion is stopped, the system does not enter the patient lockout. Upon restart, the patient can immediately request an additional dosage.
6	If the time step is not exactly one second, actions that occur at specific intervals might be missed.
7	The system has no way to handle a concurrent infusion initiation and pause request.

	Alarms
1	If an alarm condition occurs during the initialization step, it will not be detected.
2	The Alarms system does not check that the pump is in therapy before issuing therapy-related alarms.
3	Each time step is assumed to be one second.

TABLE 2
Real faults for infusion pump systems

model checker [13], [33] in our experiments because we have found that it is efficient and produces tests that are easy to understand [8].

For the systems with real faults, we generate tests twice. When calculating fault-finding effectiveness on generated mutants, we generate tests using the corrected version of the system (as the Rockwell Collins systems are free of known faults). However, when assessing the ability of the test suites to find the real faults, we generate the tests using the faulty version of the system. This reflects real-world practice, where—if faults have not yet been discovered—tests have been generated to provide coverage over the code as it currently exists.

We have also generated a single set of 1,000 random tests for each case example. The tests in this set are between two and ten execution steps (evenly distributed in the set). For each test step, we randomly selected a valid value for all inputs. As all inputs are scalar, this is trivial. We refer to this as a *random test suite*. After generating coverage-based test suites, we resample from this test suite to create random test suites of equal size for each coverage-based test suite. Note that as all of our case examples are modules of larger systems, the tests generated are effectively *unit tests*.

3.5 Test Suite Reduction

Counterexample-based test generation results in a separate test for each coverage obligation. This leads to a large amount of redundancy in the tests generated, as each test likely covers several obligations. Consequently, the test suite generated for each coverage criterion is generally much larger than is required to provide coverage. Given the correlation between test suite size and fault finding effectiveness [19], this has the potential to yield misleading results—an unnecessarily large test suite may lead us to conclude that a coverage criterion has led us to select effective tests, when in reality it is the size of the test suite that is responsible for its effectiveness. To avoid this, we reduce each naïvely generated test suite while maintaining the coverage achieved. To prevent us from selecting a test suite that happens to be exceptionally good or exceptionally poor relative to the possible reduced test suites, we produce 50 different reduced test suites for each case example using the process described below.

Per *RQ1*, we have also created tests suites satisfying the coverage criteria by reducing the random test suite with respect to the coverage criteria (that is, the suite is reduced while maintaining the coverage level of the unreduced suite). Again, we produce 50 tests suites satisfying each coverage criterion.

For both counterexample-based test generation and random testing reduced with respect to a criterion, reduction is done using a simple greedy algorithm. We determine the coverage obligations satisfied by each test generated, and initialize an empty test set $reduced$. We then randomly select a test from the full set of tests; if it satisfies obligations not satisfied by any test input in $reduced$, we add it to $reduced$. We continue until all tests have been examined in the full set of tests.

For each of our existing reduced test suites, we also produce a purely random test suite of equal size using the set of random test data. Recall that each system operates as a large loop receiving input and producing output. Each generated test is thus a finite number of "steps", with each step corresponding to a set of inputs received by the system. We measure test suite size in terms of the number of total test steps, rather than the number of tests, as random tests are on average longer than tests generated using counterexample-based test generation. These random suites are used as a baseline when evaluating the effectiveness of test suites reduced with respect to coverage criteria. We also generate random test suites of sizes varying from 1 to 1,000 steps. These tests are not part of our analysis, but provide context in our illustrations.

When generating tests suites to satisfy a structural coverage criterion, the suite size can vary from the minimum required to satisfy the coverage criterion (generally unknown) to infinity. Previous work has demonstrated that test suite reduction can have a negative impact on test suite effectiveness [9]. Despite this, we believe the test suite size most likely to be used in practice is one designed to be small—reduced with respect to coverage—rather than large (every test generated in the case of counterexample-based generation or, even more arbitrarily, 1,000 random tests). Note that one could build a counterexample-based test suite generation tool that, upon generating a test, removes from consideration *all* newly covered obligations, and randomly selects a new uncovered obligation to try to satisfy, repeating until finished. Such a tool would produce test suites equivalent to our reduced test suites, and

thus require no reduction; alternatively, we could view such test suites as pre-reduced.

3.6 Computing Effectiveness

In our study, we use what are known as *expected value oracles* as our test oracles [49]. Consider the following testing process for a software system: (1) the tester selects inputs using some criterion—structural coverage, random testing, or engineering judgement; (2) the tester then defines concrete, anticipated values for these inputs for one or more variables (internal variables or output variables) in the program. Past experience with industrial practitioners indicates that such oracles are commonly used in testing critical systems, such as those in the avionics or medical device fields.

We explore the use of two types of oracles: an *output-only oracle* that defines expected values for all outputs, and a *maximum oracle* that defines expected values for all outputs and all internal state variables. The output-only oracle represents the oracle most likely to be used in practice. Both oracles have been used in previous work, and thus we use both to allow for comparison [10], [49]. The fault finding effectiveness of the test suite and oracle pair is computed as the number of mutants detected (or "killed").

For all seven of our example systems, we assess the fault-finding effectiveness of each test suite and oracle combination by calculating the ratio of mutants killed to total number of mutants (with any known non-equivalent mutants removed).

For Infusion_Mgr and Alarms, we also assess the fault-finding effectiveness of each test suite and oracle combination against the version of the model with real faults by measuring the ratio of the number of tests that fail to the total number of tests for each test suite. We use the number of tests rather than number of real faults because all of the real faults are in a single model, and we do not know which specific fault led to a test failure. However, we hypothesize that the test failure ratio is a similar measure of the *sensitivity* of a test suite to the mutant kill ratio.

4 Results and Discussion

In Tables 3, 4, 5, and 6, we present the fault finding results from our experiments when using the mutant kill ratio as the evaluation criteria. These tables list—for each case example, coverage criterion, test generation method, and oracle—the median percentage of found faults for test suites reduced to satisfy a certain criterion, next to the median percentage of found faults for random test suites of equal size[4], the relative change in median fault finding when using the test suites satisfying the coverage criterion versus the random test suite of equal size, and the p-value for the statistical analysis below. To give an example, test suites generated to satisfy decision coverage for the Latctl_Batch system find a median of 89.3% of faults, while purely random test suites of the same size find a median of 88.1% of faults—a 1.4% improvement in fault finding. In Tables 7, 8, 9, and 10, we display the results for the systems where real faults were available. In this case, we display the median percentage of tests to fail in the generated test suites. Note that negative values for *% Change* indicate the test suites satisfying the coverage criterion are less effective on average than random test suites of equal size.

We present the coverage achieved by the coverage-directed test generation in Table 11 and the randomly-generated test suites in Table 12. For decision, condition, and MC/DC coverage, the random suites are able to reach or come close to reaching 100% coverage of the test obligations for the Rockwell systems. OMC/DC is a stronger coverage criterion, and it is more difficult to achieve full coverage. However, the random test suites reduced to satisfy OMC/DC are still able to come within 20 percentage points of full coverage for the Rockwell systems. Random testing is less capable of achieving coverage on the Docking_Approach, Infusion_Mgr, and Alarms systems, covering—at most—around 70% of the decision coverage obligations for Alarms.

4.1 Statistical Analysis

For both *RQ1* and *RQ2*, we are interested in determining if test suites satisfying structural coverage criteria outperform purely random test suites of equal size. We begin by formulating statistical hypotheses H_1 and H_2:

- H_1**:** A test suite generated using random test generation to provide structural coverage will find more faults—or, for real faults, find more failing test cases—than a pure random test suite of similar size.
- H_2**:** A test suite generated using counterexample-based test generation to provide structural coverage will find more faults—or, for real faults, find more failing test cases—than a random test suite of similar size.

We then formulate the appropriate null hypotheses:

- $H0_1$**:** The fault finding results of test suites generated using random test generation to provide structural coverage and pure random test suites of similar size are drawn from the same distribution.
- $H0_2$**:** The fault finding results of test suite generated using counterexample-based test generation to provide structural coverage and random test suites of similar size are drawn from the same distribution.

Our observations are drawn from an unknown distribution; therefore, we cannot fit our data to a theoretical probability distribution. To evaluate $H0_1$ and $H0_2$ without any assumptions on the distribution of our data, we use a one-sided (strictly greater) Mann-Whitney-Wilcoxon rank-sum test [50], a non-parametric hypothesis test for determining if one set of observations is drawn from a different distribution than another set of observations. As we cannot generalize across non-randomly selected case examples, we apply the statistical test for each pairing of case example, coverage criterion, and oracle type with $\alpha = 0.05$[5].

4. "Random of Same Size" refers only to random test suites generated specifically to be the same size in terms of number of test steps as those suites reduced with respective coverage.

5. Note that we do not generalize across case examples, oracles or coverage criteria, as the needed statistical assumption, random selection from the population of case examples, oracles, or coverage criteria, is not met. The statistical tests are used to only demonstrate that observed differences are unlikely to have occurred by chance.

		Counterexample Generation Comparison				Random Generation Comparison			
Case Example	**Oracle**	**Satisfying Condition**	**Random of Same Size**	**% Change**	**p-val**	**Satisfying Condition**	**Random of Same Size**	**% Change**	**p-val**
Latctl_Batch	MX	93.8%	97.9%	-4.2%	1.00	98.8%	97.1%	1.7%	< 0.01
	OO	48.6%	83.5%	-41.9%		81.5%	80.2%	1.5%	0.06
Vertmax_Batch	MX	100.0%	95.6%	4.6%	< 0.01	100.0%	96.0%	4.2%	< 0.01
	OO	45.6%	76.2%	-40.2%	1.00	81.9%	75.8%	7.9%	
DWM_1	MX	92.8%	92.8%	0.0%	0.61	97.5%	96.6%	0.8%	
	OO	18.2%	25.0%	-27.1%	1.00	34.3%	33.1%	3.6%	0.35
DWM_2	MX	94.7%	92.6%	2.3%	< 0.01	99.2%	94.2%	5.2%	< 0.01
	OO	75.7%	77.8%	-2.6%	0.99	88.9%	82.7%	7.5%	
Docking Approach	MX	70.7%	19.4%	264.4%	< 0.01	18.5%	18.5%	0.0%	0.79
	OO	21.7%	2.0%	985.0%		2.0%	2.0%		1.00
Infusion_Mgr	MX	68.6%	25.1%	173.3%		20.7%	20.6%	0.5%	0.11
	OO	27.0%	10.1%	167.3%		7.3%	7.3%	0.0%	0.42
Alarms	MX	74.9%	47.4%	58.0%		47.0%	47.0%	0.0%	0.46
	OO	40.5%	14.6%	177.4%		15.0%	14.2%	5.6%	< 0.01

TABLE 3
Median percentage of faults identified, condition coverage criterion. OO = Output-Only, MX = Maximum Oracle

		Counterexample Generation Comparison				Random Generation Comparison			
Case Example	**Oracle**	**Satisfying Decision**	**Random of Same Size**	**% Change**	**p-val**	**Satisfying Decision**	**Random of Same Size**	**% Change**	**p-val**
Latctl_Batch	MX	89.3%	88.1%	1.4%	< 0.01	97.9%	95.9%	2.1%	< 0.01
	OO	34.2%	57.6%	-40.6%	1.00	80.7%	77.8%	3.7%	0.03
Vertmax_Batch	MX	85.1%	70.6%	20.5%	< 0.01	87.9%	84.3%	4.3%	< 0.01
	OO	31.0%	41.12%	-24.25%	1.00	61.7%	57.3%	7.7%	
DWM_1	MX	82.6%	97.0%	-14.6%		97.5%	96.2%	1.3%	0.03
	OO	13.1%	32.2%	-59.2%		33.5%	32.6%	2.6%	0.34
DWM_2	MX	80.2%	88.1%	-8.9%		94.7%	91.8%	3.1%	< 0.01
	OO	48.1%	70.8%	-31.9%		80.1%	77.4%	4.3%	
Docking Approach	MX	67.5%	19.4%	247.9%	< 0.01	18.5%	18.5%	0.0%	0.90
	OO	20.1%	2.0%	905%		2.0%	2.0%		1.00
Infusion_Mgr	MX	65.2%	23.5%	177.4%		20.7%	19.8%	4.5%	0.24
	OO	25.5%	8.9%	186.5%		6.9%	6.9%	0.0%	0.54
Alarms	MX	74.9%	47.8%	56.7%		47.0%	47.0%	0.0%	1.00
	OO	35.7%	14.6%	144.5%		14.6%	13.8%	5.8%	< 0.01

TABLE 4
Median percentage of faults identified, decision coverage criterion. OO = Output-Only, MX = Maximum Oracle

		Counterexample Generation Comparison				Random Generation Comparison			
Case Example	**Oracle**	**Satisfying MC/DC**	**Random of Same Size**	**% Change**	**p-val**	**Satisfying MC/DC**	**Random of Same Size**	**% Change**	**p-val**
Latctl_Batch	MX	96.7%	99.6%	-2.3%	1.00	99.6%	98.8%	0.8%	< 0.01
	OO	79.8%	93.4%	-14.5%		89.7%	87.7%	2.3%	
Vertmax_Batch	MX	100.0%	96.4%	3.8%	< 0.01	100.0%	95.2%	5.1%	
	OO	59.3%	78.2%	-24.2%	1.00	81.9%	76.6%	6.8%	
DWM_1	MX	88.6%	97.5%	-9.1%		97.9%	97.5%	0.4%	
	OO	18.6%	36.0%	-48.2%		34.7%	34.3%	1.2%	0.45
DWM_2	MX	96.3%	96.3%	0.0%	0.83	99.6%	97.1%	2.5%	< 0.01
	OO	79.8%	86.0%	-7.2%	1.00	90.9%	88.1%	3.3%	
Docking Approach	MX	72.3%	19.4%	272.7%	< 0.01	18.5%	18.5%	0.0%	0.62
	OO	23.3%	2.0%	1065.0%		2.0%	2.0%		1.00
Infusion_Mgr	MX	69.6%	24.7%	44.9%		20.6%	21.9%	-1.3%	0.99
	OO	31.6%	11.3%	20.3%		6.9%	7.3%	-0.4%	0.02
Alarms	MX	78.9%	47.8%	65.1%		47.8%	47.4%	0.8%	0.01
	OO	40.5%	14.6%	177.4%		15.0%	14.6%	2.7%	< 0.01

TABLE 5
Median percentage of faults identified, MC/DC criterion. OO = Output-Only, MX = Maximum Oracle

		Counterexample Generation Comparison				Random Generation Comparison			
Case Example	**Oracle**	**Satisfying OMC/DC**	**Random of Same Size**	**% Change**	**p-val**	**Satisfying OMC/DC**	**Random of Same Size**	**% Change**	**p-val**
Latctl_Batch	MX	99.2%	99.2%	0.0%	1.00	99.6%	99.6%	0.0%	0.28
	OO	95.3%	90.1%	5.8%	< 0.01	97.1%	95.9%	1.3%	< 0.01
Vertmax_Batch	MX	99.6%	96.4%	3.3%		99.2%	97.6%	1.6%	
	OO	97.5%	77.4%	26.0%		90.7%	80.2%	13.1%	
DWM_1	MX	100.0%	100.0%	0.0%	0.16	100.0%	100.0%	0.0%	1.00
	OO	90.5%	83.5%	8.4%	< 0.01	97.5%	92.8%	5.1%	< 0.01
DWM_2	MX	98.7%	97.5%	1.2%		100.0%	100.0%	0.0%	
	OO	95.7%	88.9%	7.6%		98.4%	96.7%	1.8%	
Docking Approach	MX	73.9%	19.4%	280.9%	< 0.01	19.4%	19.4%	0.0%	0.56
	OO	26.9%	2.0%	1245.0%		2.0%	2.0%		1.00
Infusion_Mgr	MX	70.0%	27.5%	154.5%		23.1%	23.1%		0.20
	OO	43.3%	12.1%	257.9%		8.5%	8.5%		0.19
Alarms	MX	81.0%	48.2%	68.0%		48.6%	48.6%	0.0%	0.02
	OO	58.3%	15.4%	278.6%		15.4%	15.4%	0.0%	0.81

TABLE 6
Median percentage of faults identified, OMC/DC criterion. OO = Output-Only, MX = Maximum Oracle

		Counterexample Generation Comparison				Random Generation Comparison			
Case Example	**Oracle**	**Satisfying Condition**	**Random of Same Size**	**% Change**	**p-val**	**Satisfying Condition**	**Random of Same Size**	**% Change**	**p-val**
Infusion_Mgr	MX	55.8%	36.7%	52.0%	< 0.01	42.9%	36.4%	17.9%	0.16
	OO	10.5%	29.0%	-63.8%	1.00	35.7%	30.0%	19.0%	0.07
Alarms	MX	93.0%	93.8%	-0.9%	0.98	92.9%	93.0%	-0.1%	0.70
	OO	91.2%		-2.8%	1.00				

TABLE 7
Median percentage of tests failed after identifying real faults, condition coverage criterion.

		Counterexample Generation Comparison				Random Generation Comparison			
Case Example	**Oracle**	**Satisfying Decision**	**Random of Same Size**	**% Change**	**p-val**	**Satisfying Decision**	**Random of Same Size**	**% Change**	**p-val**
Infusion_Mgr	MX	42.3%	40.0%	5.8%	< 0.01	33.3%	33.3%	0.0%	0.44
	OO	6.3%	31.8%	-80.2%	1.00	28.6%	28.6%	0.0%	0.26
Alarms	MX	92.6%	94.3%	-1.8%	1.00	93.8%	94.1%	-0.3%	0.33
	OO	90.4%		-4.1%					

TABLE 8
Median percentage of tests failed after identifying real faults, decision coverage criterion.

		Counterexample Generation Comparison				Random Generation Comparison			
Case Example	**Oracle**	**Satisfying MC/DC**	**Random of Same Size**	**% Change**	**p-val**	**Satisfying MC/DC**	**Random of Same Size**	**% Change**	**p-val**
Infusion_Mgr	MX	46.0%	34.4%	33.7%	< 0.01	33.3%	30.0%	11.0%	0.02
	OO	1.8%	27.3%	-93.4%	1.00	30.0%	25.0%	20%	0.05
Alarms	MX	94.6%	93.8%	0.9%	0.05	95.2%	94.1%	1.2%	0.02
	OO	94.4%		0.6%	0.36				

TABLE 9
Median percentage of tests failed after identifying real faults, MC/DC criterion.

		Counterexample Generation Comparison				Random Generation Comparison			
Case Example	**Oracle**	**Satisfying OMC/DC**	**Random of Same Size**	**% Change**	**p-val**	**Satisfying OMC/DC**	**Random of Same Size**	**% Change**	**p-val**
Infusion_Mgr	MX	47.9%	38.3%	25.1%	< 0.01	42.1%	35.3%	19.3%	< 0.01
	OO	40.4%	30.3%	33.3%		30.0%	28.6%	4.9%	0.09
Alarms	MX	93.0%	94.4%	-1.5%	1.00	92.4%	94.7%	-2.4%	1.00
	OO	40.4%		-57.2%					

TABLE 10
Median percentage of tests failed after identifying real faults, OMC/DC criterion.

	Decision Coverage	Condition Coverage	MC/DC Coverage	OMC/DC Coverage
DWM_1	100.00%	100.00%	100.00%	99.90%
DWM_2	100.00%	100.00%	95.28%	89.79%
Vertmax_Batch	100.00%	100.00%	100.00%	98.15%
Latctl_Batch	100.00%	100.00%	100.00%	93.42%
Docking_Approach	97.94%	95.85%	91.94%	50.86%
Infusion_Mgr	92.06%	93.56%	92.36%	56.06%
Alarms	91.85%	93.17%	91.32%	76.24%

TABLE 11
Coverage Achieved by Coverage-Directed Test Suites Reduced to Satisfy Coverage Criteria

	Decision Coverage	Condition Coverage	MC/DC Coverage	OMC/DC Coverage
DWM_1	100.00%	100.00%	100.00%	99.85%
DWM_2	100.00%	100.00%	93.15%	89.25%
Vertmax_Batch	100.00%	99.83%	99.40%	84.18%
Latctl_Batch	100.00%	100.00%	100.00%	89.21%
Docking_Approach	58.10%	56.90%	48.59%	2.20%
Infusion_Mgr	62.77%	62.33%	48.86%	11.16%
Alarms	69.33%	68.85%	64.22%	32.72%

TABLE 12
Coverage Achieved by Randomly Generated Test Suites Reduced to Satisfy Coverage Criteria

4.2 Evaluation of RQ1

Based on the p-values less than 0.05 in Tables 3-6, we *reject* $H0_1$ for nearly all of the four Rockwell case examples and the respective coverage criteria when using either oracle. Note that we do not reject $H0_1$ for the *DWM_1* case example when using decision, condition, and MC/DC coverage and the output-only oracle. That is, the use of coverage as a supplement to random testing—using coverage to decide when to stop random testing—leads to improved fault-finding results. However, for the majority of coverage criterion and oracle combinations for the Docking_Approach, Alarms, and Infusion_Mgr systems, we fail to reject $H0_1$. For many of these combinations, there is no evidence of improvement from using coverage as an adequacy metric for random testing. Across all system, for cases with differences that are statistically significant, test suites reduced to satisfy coverage criteria are clearly more effective than purely randomly generated test suites of equal size—for these combinations, we accept H_1.

The difference in results between the four Rockwell Collins systems and the Docking_Approach, Alarms, and Infusion_Mgr systems—when examining mutations—can be somewhat explained through the coverage achieved by random tests on those systems. As can be seen in Table 12, random testing is able to cover almost all of the obligations for the four coverage criteria on the Rockwell systems. In those cases, we can use coverage as a method of guiding the selection of tests. We can cut off random testing once coverage is achieved, filter out superfluous tests, and potentially present a small, powerful test suite. On the other systems, however, random testing is unable—even after generating our full pool of 1000 tests—to achieve full coverage for any of the coverage criteria. In those cases, it is hardly surprising that using coverage to guide the selection of random tests fails to lead to an improvement in fault-finding. Even if there is potential power to be gained from the guidance of coverage, failing to achieve coverage will also imply failing to gain that predictive power. It may well be that *any* suite of random tests of the same size will result in similar coverage and fault-finding, and using coverage to choose those suites may be no more effective than choosing from the pool of tests at random.

When evaluating test suite effectiveness against the set of real faults for the Infusion_Mgr and Alarms systems, we see similar results—see tables 7-10. We reject $H0_1$ for the MC/DC and OMC/DC criteria and the maximum oracle and the MC/DC criteria for the output-only oracle for the Infusion_Mgr system. However, we fail to reject $H0_1$ for the remaining criteria and oracle combinations for these two systems. Note that, in several cases, we do see an improved median fault-finding result, but we do not see a corresponding achievement of statistical significance. In those cases, there is a large amount of variance in the results from the random test suites. While the median case has improved, the the *distribution* of results has not changed. Samples from the distributions of random tests guided by coverage are not significantly better at finding faults than samples from the distribution of purely random tests. In some cases, we see a small improvement in fault-finding effectiveness when we use coverage as an adequacy criteria and, even in the other cases, using coverage as an adequacy metric does not result in *worse* fault-finding results.

From our results, we can weakly confirm that all four of these coverage criteria *can* be effective metrics for test suite adequacy within the domain of critical avionics systems: reducing test suites generated via a non-directed approach to satisfy structural coverage criteria is at least not harmful, and in some instances improves test suite effectiveness relative to their size by up to 13%. Thus, the use of structural coverage as an adequacy criteria for random testing can potentially lead to a positive, albeit slight, improvement in test suite effectiveness. This indicates that the core intuitions behind these coverage metrics—i.e., covering branches, conditions and combinations of conditions—appear valid, and thus, given

a constant generation strategy, covering code yields benefits as long as coverage is actually achieved.

4.3 Evaluation of RQ2

Based on the p-values less than 0.05 in Tables 3- 5, we fail to reject $H0_2$ for the four Rockwell Collins case examples and the decision, condition, and MC/DC coverage criteria when using the output-only oracle. For all but one of these case examples, test suites generated via counterexample-based test generation are *less* effective than pure random test suites by 2.6% to 59.2%; we therefore conclude that our initial hypothesis H_2 is false—at least, with respect to the Rockwell systems—with regard to decision, condition, and MC/DC coverage when using an output-only oracle.

When using the maximum oracle, the test suites generated via counterexample-based test generation to satisfy decision, condition, and MC/DC coverage fare better. In select instances, countereexample-based test suites outperform random test suites of equal size (notably *Vertmax_Batch*), and otherwise close the gap, being less effective than pure random test suites by at most 14.6%. Nevertheless, we note that for most combinations of the Rockwell case examples and those three coverage criteria, random test suites of equal size are still more effective. When these criteria are used as targets for test generation, the test suites produced are generally *less* effective than random testing alone, with decreases of up to 59.2%.

This indicates that decision, condition, and MC/DC coverage are—by themselves—not necessarily good indicators of test suite effectiveness; factors other than coverage impact the effectiveness of the testing process. In contrast, to the more traditional structural coverage criteria—notably MC/DC coverage—results for OMC/DC coverage are more positive in terms of the value of directed test generation. From the p-values in Table 6, we reject $H0_2$ for all case examples except Latctl_Batch with the maximum oracle and DWM_1 with the maximum oracle when examining the mutation-based faults. It appears that test suites generated to satisfy OMC/DC coverage are more effective than purely random tests of the same size, and—as when using coverage as a supplemental criteria for random testing—generating tests in order to satisfy OMC/DC is no worse than just constructing random tests. Thus, we can see that considering how covered obligations propagate to observable points can yield dividends during test generation.

The converse of H_2—that randomly generated test suites are more effective than equally large test suites generated via counterexample-based test generation—is also not universally true, as the Docking_Approach, Alarms, and Infusion_Mgr examples illustrate. For the Docking_Approach example, random testing is effectively useless, finding a mere 2% of the faults on average when using an output-only oracle and 19.4% with the maximum oracle. Similarly, for the Alarms and Infusion_Mgr systems, the use of counterexample-based tests does improve fault-finding effectiveness by up to 278.6% over random test suites of similar size. However, it should be noted that *improved* fault-finding is not always the same as *good* fault-finding. The maximum oracle finds up to 81% of the faults for the Alarms system; however, maximum oracles are often prohibitively expensive to employ, as they require a specification of correctness for all variables. The output-only oracle, a far more common option [49], only manages to find slightly over half of the faults for a single case example and coverage combination—OMC/DC testing on the Alarms system. There is clearly room for improvement.

Contrasting the performance of counterexample-based test generation and random test generation on the systems with real faults yields a number of observations. From the results for Infusion_Mgr in Table 7-10, we can see that the use of coverage-directed test generation yields a higher percentage of failing tests for the maximum oracle. However, for the output-only oracle, the opposite is true—random tests are far more capable at detecting faults than the coverage-directed tests. This difference is likely due to *masking*—some expressions in the systems can easily be prevented from influencing the outputs. When covered expressions do not propagate to an observable output, faults cannot be observed. The coverage-directed tests tend to be short, one or two test steps at most. The real faults embedded within this system require specific combinations of events to trigger, and may take some time before they influence the output. As a result, the coverage-based tests may trigger more of the difficult-to-reach faults, but are not long enough for the effects of those faults to influence the outward behavior of the system. The random tests, on the other hand, tend to be longer (up to ten steps), which may be long enough that many of the faults do propagate to the system output.

On the Alarms system, the random tests are more capable of detecting faults than the coverage-directed tests, with the exception of the MC/DC criterion. For the decision and condition coverage criteria, this difference is relatively small, up to a 4.1% difference. However, for the OMC/DC criterion, random testing is far more capable at detecting the embedded faults. This result can be explained by examining the type of faults that exist in this system, as listed in Table 2. In particular, the first fault—that if an alarm condition occurs during the first initialization step of execution, it will not be detected in the faulty version of the system—helps to explain the particular results that were observed. The coverage criteria employed in this experiment all, to a varying degree, require that certain combinations of Boolean conditions are satisfied. As a result, the generated tests will be biased towards particular input values. In many cases, these input values would be not trigger alarm conditions immediately upon system activation. The random tests, on the other hand, tend towards extreme input values (or, at least, make use of the full range of possible input combinations) and, as a result, trigger this particular fault in almost *every* test case. As a result, random testing results in a higher percentage of failing tests, but the majority of those failures are due to the fault in the initialization step.

Another factor that helps explain the differing results on the version of Alarms with real faults is that coverage-directed test generation is unable to achieve a level of OMC/DC coverage as high as that achieved by MC/DC, condition, and decision coverage—tests generated on the faulty version of Alarms only achieve 72% OMC/DC coverage, whereas 88% MC/DC coverage is achieved. This means that some portion of the state

space *is not being explored* by the OMC/DC-directed tests. If the uncovered state space overlaps with one of the embedded faults, then the existing OMC/DC tests may not execute the tests in such a way that the fault will be triggered.

The results when examining real faults differ from those seen for the Infusion_Mgr and Alarms systems when examining seeded mutations. In the latter case, coverage-directed generation yielded stronger tests. Often, in the former case, the randomly-generated tests yielded stronger results. This shift can likely be explained by examining the types of faults seeded in both scenarios. The seeded mutations are all code-based errors—using the wrong operation, changing a constant, using a stored value for a variable from a previous computation. However, the real faults, listed in Table 2, tend to be more *conceptual* in nature. Largely, the real faults are problems of omission—the developers forgot to implement a feature or the system specification left an outcome ambiguous. Code coverage cannot be expected to account for code that does not exist, and thus, is unlikely to yield tests that account for such faults. This explains the different results for these systems when switching from seeded faults to real faults—coverage-directed tests can help find faults when the faults are the result of mistakes in the code that is being exercised, but are not guaranteed to be effective when faults are due to conceptual mistakes.

4.4 Implications

Given the important role of structural coverage criteria in the verification and validation of safety-critical avionics systems, we find these results quite troublesome. In the remainder of this section, we discuss the immediate practical implications of this as well as the implications for future work. We begin by discussing why traditional structural coverage criteria fare poorly when used as targets for test generation. We have identified several factors that contribute to this, including the formulation of structural coverage criteria; the behavior of the test generation mechanism (in this case, software model checkers); and structural properties of the case examples.

First and—given the differences observed with OMC/DC coverage—foremost, we note that traditional coverage criteria are formulated over specific elements in the source code. For each element, (1) execution must reach the element and (2) exercise the element in a specific way. This type of formulation falls short in two ways. First, it is possible to change the number and structure of each element by varying the structure of the program, which we have previously seen can significantly impact the number of tests required to satisfy the MC/DC coverage criterion [12], [51]. This is linked partly to the second issue, *masking*—some expressions in the systems can easily be prevented from influencing the outputs. When covered expressions do not propagate to an observable output (i.e., do not reach a test oracle variable), faults cannot be observed. This reduces the effectiveness of any testing process based on structural coverage criteria, as we can easily satisfy coverage obligations for internal expressions without allowing resulting errors to propagate to the output.

Issues related to masking are in turn exacerbated by automated test generation, bringing us to our second factor. We have found that test inputs generated using counterexample-based generation (including those in this study) tend to be short, and manipulate only a handful of input values, leaving other inputs at default values (in our case, `false` or `0`) [8]. Such tests tend to exercise the program just enough to satisfy the coverage obligations for which they were generated and do not consider the propagation of values to the outputs. In contrast, random tests can vary arbitrarily in length (up to 10 steps in this study) and vary all input values; such test inputs may be more likely to overcome any masking present in the system. Rather than pointing to this as a strength of random testing, we would like to emphasize that this is a *weakness* of the coverage-directed test generation method. The use of coverage cannot be assumed to guarantee effective tests—the particulars of the method of test generation appear to have a greater impact at present.

Finally, the structure of the case examples themselves—being fairly representative of case examples within this domain—is also partly at fault. Recall that when testing the Docking_Approach, Alarms, and Infusion_Mgr systems, tests generated to satisfy structural coverage criteria sometimes dramatically outperform random test generation. This is due to the structure of these systems: large portions of these system's behavior are activated only when very specific conditions are met. As a result the state space is both deep and narrow at multiple points, and exploration of these deep states requires relatively long tests with specific combinations of input values. Random testing is therefore highly unlikely to reach much of the state space, and indeed, less than 50% of the MC/DC obligations were covered for Docking_Approach. In contrast, the Rockwell Collins systems (while stateful) have a state space that is shallow and highly interconnected; these systems are therefore easier to cover with random testing and, thus, the potential benefits of structural coverage metrics are diminished.

It is these issues—particularly the first two issues—that motivated the development of the OMC/DC coverage criterion. Observable MC/DC coverage explicitly avoids issues related to masking by requiring in its test obligations both demonstrate the independent impact of a condition on the outcome of the decision statement, and follow a non-masking path to some variable monitored by the test oracle. Consequently, test suites generated via counterexample-based test generation to satisfy OMC/DC outperform purely randomly generated test suites of equal size by up to 42.5%.

This represents a major improvement over existing structural coverage criteria, though we still urge caution. The high cost and difficulty of generating OMC/DC satisfying test suites—relative to generating the weaker decision, condition, or MC/DC test suites—makes its use as a target for directed test suite generation less likely at this point in time. Additional work demonstrating the cost effectiveness (and perhaps specific improvements for test generation) will be necessary before OMC/DC coverage can replace MC/DC coverage.

We see three key implications in our results. First, with regard to **RQ1**, we can weakly conclude that using any of these four structural coverage criteria as an addition to another non-structure-based testing method—in this case, random testing—

can potentially yield improvements in the testing process. These results are similar to those of other authors, for example, results indicating MC/DC is an effective coverage criterion when used to check the adequacy of manual, requirement-driven test generation [32] and results indicating that reducing randomly generated tests with respect to decision coverage yields improvements over pure random test generation [19]. These results, in conjunction with the results for **RQ2**, reinforce the advice that coverage criteria are best applied after test generation to find areas of the source code that have not been tested. In the case of MC/DC this advice is already explicitly stated in regulatory requirements and by experts on the use of the criterion [2], [15].

Second, the dichotomy between the Docking_Approach, Alarms, and Infusion_Mgr examples and the Rockwell Collins systems highlights that, while the current methods of determining test suite adequacy in avionics systems are themselves largely inadequate, some method of determining testing adequacy is needed. While current practice recommends that coverage criteria should be applied after test generation, in practice, this relies on the honesty of the tester (it is not required in the standard). Therefore, it seems inevitable that at least some practitioners will use automated test generation to reduce the cost of achieving the required coverage.

The lack of consensus in the results across these varied systems is concerning in light of this inevitability. While coverage-directed test generation *can* lead to effective testing, there is no evidence that it can do so consistently. With the importance given to coverage criteria in the avionics industry, the temptation exists to rely on coverage as an assurance of reliable and thorough testing. However, we stress that blind faith in the power of code coverage is a risky proposition at best in light of the inconsistency of the results in this study.

Finally, our results with OMC/DC coverage indicate that it is possible to extend existing coverage to overcome some of the issues we have highlighted above. The primary problem with existing structural coverage criteria is that all effort is expended on covering structures internal to the system, and no further consideration is paid to how the effect of covered structure reaches an observable point in the system. OMC/DC considers the observability aspect for Boolean expressions (using MC/DC) by appending a path condition onto each test obligation in MC/DC. Similar extensions could be applied to a variety of other existing coverage metrics, e.g., boundary value testing.

5 Recommendations

Assuming our results generalize, we believe that these studies raise serious concerns regarding the efficacy of coverage-directed automated testing. The tools are not at fault: we have asked these tools to produce test inputs satisfying some form of structural coverage, and they have done so admirably; for example, satisfying MC/DC for the *Docking_Approach* example, for which random testing achieves a mere 37.7% of the possible coverage. However, our results—along with the existing body of research on this topic—lead us to conclude that it is not sufficient to simply maximize a structural coverage metric when automatically generating test inputs. In practice, the mechanism for achieving coverage is important. *How* tests are generated matters more than what is being maximized.

The key issues involve how structural coverage criteria are formulated and how automatic test generation tools operate. Traditional coverage criteria are formulated over specific elements in the source code. To cover an element, (1) execution must reach the element and (2) exercise the element in a specific way. However, commonly used structural coverage criteria typically leave a great deal of leeway to how the element is reached, and—more importantly, in our experience—place no constraints whatsoever on how the test should evolve after the element is exercised. Automatic test generation tools typically use this freedom to do just enough work to satisfy coverage criteria, without consideration of, for example, how the faults are to be detected by the test oracle.

First, let us consider the path to satisfy a coverage obligation, e.g., a branch of a complex conditional. Structural coverage criteria require only that the point of interest is reached and exercised. We have found that automatically generated tests often take a shortest-path approach to satisfying test obligations, and manipulate only a handful of input values, leaving other inputs at default values. This is a cost effective method of satisfying coverage obligations; why tinker with program values that do not impact the coverage achieved? However, we, and other authors, have observed that variations provided by (for example) simple random testing result in test suites that are nearly as effective in terms of coverage, and moreover produce more interesting behavior capable of detecting faults [25], [7]. As illustrated by our experiments, however, even with lower coverage achieved, randomly generated test inputs can outperform automatically generated test suites in terms of fault finding.

Second, for the most commonly used structural coverage criteria, there is no directive concerning how tests should evolve after satisfying the structural element. Given this lack of direction, automatic test generation tools typically do not consider the path from the covered element to an output/assertion/observable state when generating a test, and therefore test inputs may achieve high coverage but fail to demonstrate faults that exist within the code. One reason for this failure is *masking*, which occurs when expressions/computations in the system are prevented from influencing the observed output, i.e., do not reach a variable or assertion monitored by the test oracle. More generally, this is related to the distinction between incorrect program state and program output: just because a test triggers a fault, there is no guarantee this will manifest as a *detected* fault. Indeed, in our experience, care must be taken to ensure that this occurs.

These two high level issues result in the generation of test inputs that may indeed be effective at encountering faults, but may make actually observing them—that is, actually detecting the fault—very difficult or unlikely. This reduces the effectiveness of any testing process based on structural coverage criteria, as we can easily satisfy coverage obligations for internal expressions without allowing resulting errors to propagate to the output.

Furthermore, even when the generated tests are more effective than the randomly-generated tests, these tests may still not actually yield *good* fault-finding performance. On the Alarms, Infusion_Mgr, and—in particular—Docking_Approach, the generated tests commonly found fewer than half of the seeded faults when using a common output-only oracle, and less than 80% of the faults with the prohibitively expensive maximum oracle.

We believe that these issues raise serious concerns about the efficacy of coverage-directed automated testing. A focus in automated test generation work has been on efficiently achieving coverage without carefully considering how achieving coverage impacts fault detection. We therefore run the risk of producing tools that are satisfying the letter of what is expected in testing, but not the spirit. Nevertheless, the central role of coverage criteria in testing is unlikely to fade, as demonstrated by the emphasis on coverage criteria for certification.

Hence, the key is to improve upon the base offered by these criteria and existing technology. We have come to the conclusion that the research goal in automated test generation should not be developing methods of maximizing structural code coverage, but rather determining how to *maximize fault-finding effectiveness*. We propose that algorithms built for test generation must evolve to take into account factors beyond the naive execution of individual elements of the code—factors such as masking, program structure, and the execution points monitored by the test oracle.

To that end, we recommend three approaches. First, we could improve, or replace, existing structural coverage criteria, extending them to account for factors that influence test quality. Automated test generation has improved greatly in the last decade, but the targets of such tools have not been updated to take advantage in this increase in power. Instead, we continue to rely on criteria that were originally formulated when manual test generation was the only practical method of ensuring 100% achievable coverage.

Second, automated test generation tools could be improved to avoid pitfalls when using structural coverage criteria. This could take many forms, but one straightforward approach would be to develop heuristics or rules that could operate alongside existing structural coverage criteria. For instance, tools could be encouraged to generate longer test cases, increasing the chances that a corrupted internal state would propagate to an observable output (or other monitored variable).

Third, important factors specific to individual domains, e.g., web testing v.s. embedded systems, could be empirically identified and formalized as heuristics within a test generation algorithm.

5.1 Use More Robust Coverage Criteria

In our own work, the issues of criteria formulation and masking motivated the development of the Observable MC/DC coverage criterion employed in this case study [12]. OMC/DC coverage explicitly avoids issues related to masking by requiring that its test obligations both demonstrate the independent impact of a condition on the outcome of the decision statement, and follow a non-masking propagation path to some variable monitored by the test oracle. OMC/DC, by accounting for the program structure and the selection of the test oracle, can, in our experience, address some of the failings of traditional structural coverage criteria within the avionics domain, allowing for the generation of test suites achieving better fault detection than random test suites of equal size. Similarly, Vivanti et al. have demonstrated evidence that the use of data-flow coverage as a goal for test generation results in test suites with higher fault detection capabilities than suites generated to achieve branch coverage [52]. OMC/DC considers the observability aspect for Boolean expressions (using MC/DC) by appending a path condition onto each test obligation in MC/DC. Similar extensions could be applied to a variety of other existing coverage metrics, e.g., boundary value testing.

5.2 Algorithmically Improve Test Selection

Extensions to coverage criteria are not without downsides: for stronger metrics, programs will contain *unsatisfiable* obligations where there is no test that can be constructed to satisfy the obligation. Depending on the search strategy, the test generator may never terminate on such obligations. Further, the cost and difficulty of generating OMC/DC satisfying test suites—relative to generating the weaker MC/DC test suites—makes the use of strong coverage criteria as targets for test generation harder to universally recommend.

Instead, another possible method of ensuring test quality is to use a traditional structural coverage metric as the objective of test generation, and augment this by considering other factors empirically established to impact fault detection effectiveness. For example, in the context of a search-based test generation algorithm, this might mean adding additional objective functions to the search strategy, rather than adding additional constraints to the coverage criterion. For instance, an algorithm could both work to maximize an existing structural coverage criterion and minimize the propagation distance between the assignment of a value and its observation by the oracle (an algorithm that minimizes this distance for test prioritization purposes already exists [53]).

As an example, consider purely random testing. Random testing does not employ an objective function, but it is possible to use one to approximate how well the system's state space is being covered.We have seen systems containing bottlenecks (pinch-points) in the state space where randomly generated tests perform very poorly. Such bottlenecks require certain specific input sequences to reach a large portion of the state space. However, approaches such as concolic testing [4]—combining random testing with symbolic execution—are able to direct the generation of tests around such bottlenecks. Similar approaches could be employed to direct test generation towards, for example, propagation paths from variable assignment to oracle-monitored portions of the system.

By pursuing and balancing multiple objectives, we could potentially offer stronger tests that both satisfy MC/DC obligations and offer short propagation paths, even when it would be impossible to generate a test that satisfies the corresponding—but stricter—OMC/DC obligation. Embedding aspects of test

selection into the objective function—or even into the search algorithm itself—may allow for improved efficiency. The search method can use, say, masking as a pruning mechanism on paths through the system, and the algorithm would not have to track as much symbolic information related to the objective metric itself.

5.3 Tailor an Approach to the Domain

It is important to emphasize that there is no "one size fits all" solution to test generation. The size and shape of the state space of a system varies dramatically between domains and programming paradigms, and, as a result, it is difficult to tailor universal testing strategies. Much of our work has focused on embedded systems that run as cyclic processes. In this area, a common issue is that the impact of exercising a code path on the system's output is often delayed; only several cycles after a fault occurs can we observe it. If the goal of test generation is only to cause the code path to be executed, many of the tests will not cause a visible change in system behavior. In object-oriented systems, a central, but related issue is that a method call may change internal state that, again, is not visible externally until another method call produces output. As a result, choosing appropriate method sequences and their ordering becomes a major challenge.

Thus while general rules and heuristics for improving test generation are valuable, we believe there are large improvements to be found in tailoring the approach to the testing challenge at hand. For example, two often overlooked factors are the cost of generating tests and the cost of running tests. It is hard to outperform random testing in terms of the cost of generating tests, because doing so requires very little computation. If it is also cheap to run tests, then for many systems it is difficult to outperform straight random testing. On the other hand, if it is expensive to run tests, e.g., for embedded systems, this may require access to a shared hardware "rig." In this case, using search-based techniques to generate tests for a specific strong coverage criterion (such as OMC/DC) may be sensible because the number of required tests can be dramatically smaller than the number of random tests required to achieve the same level of fault finding.

Another overlooked factor is the "reasonableness" of generated tests. Automated test generation methods should deliver tests that not only find faults, but are also meaningful to the domain of interest. Coverage-based techniques take the path of least resistance when generating tests, but can produce tests that make little sense in the context of the domain. Techniques that can generate inputs with meaning to the human testers are valuable in reducing the "human oracle cost" associated with checking failing test results [54].

Addressing factors such as these must be done on a per domain level, and indicate that code coverage should be one of several goals in test generation. We suspect that, in the long run, effective test generation tools will consist of both general techniques and heuristics, and additional *test generation profiles* for each domain. Determining the *correct* objective functions for test generation for each domain is an open research question, one requiring both technical advancements in search-based test generation and empirical studies.

6 Threats to Validity

External Validity: Our study has focused on a relatively small number of systems but, nevertheless, we believe the systems are representative of the critical systems domain, and our results are generalizable to other systems in that domain.

We have used two methods for test generation (random generation and counterexample-based). There are many methods of generating tests and these methods may yield different results. Counterexample-based testing is used to produce coverage-directed test cases because it is a method used widely in testing safety-critical systems. Random testing is used as a contrasting baseline because it is one of the most simple test generation methods in existance. Because the length of test inputs and the values selected for the input variables are chosen at random, we believe that random generation is a fair baseline—it has not been tuned to systems in this domain and has no particular strengths or result guarantees.

For all coverage criteria, we have examined 50 test suites reduced using a simple greedy algorithm. It is possible that larger sample sizes may yield different results. However, in previous studies, smaller numbers of reduced test suites have been seen to produce consistent results [51].

Construct Validity: In our study, we primarly measure fault finding over seeded faults, rather than real faults encountered during development. However, Andrews et al. showed that seeded faults lead to similar conclusions to those obtained using real faults [55] for the purpose of measuring test effectiveness. We have assumed these conclusions hold true in our domain/language for several case examples. We have, however, made use of two systems containing real faults in order to widen our pool of observations.

Our generation of mutants was randomized to avoid bias in mutant selection. A large pool of mutants was used to avoid generated a set of mutants particularly skewed toward or against a coverage criteria. In our experience, mutants sets greater than 100 result in very similar fault finding; we generated 250 to further increase our confidence no bias was introduced. In addition, we have also used one case example which has an associated set of real faults, which yields results comparable to those found when using seeded faults.

We measure the cost of test suites in terms of the number of steps. Other measurements exist, e.g., the time required to generate and/or execute tests [56]. We chose size as a metric that favors directed test generation. Thus, conclusions concerning the inefficacy of directed test generation are reasonable.

Conclusion Validity: When using statistical analyses, we have attempted to ensure the base assumptions beyond these analyses are met, and have favored non-parametric methods. In cases in which the base assumptions are clearly not met, we have avoided using statistical methods. (Notably, we have avoided statistical inference across case examples.)

7 Conclusion

Our results indicate that the use of structural coverage as a supplement to an existing testing method—such as random testing—may result in more effective tests suites than random testing alone. However, the results for the use of coverage

criteria as a *target* for directed, automatic test case generation are mixed. For three of the systems, automatic test case generation yielded effective tests. However, for the remaining systems, randomly generated tests often yielded similar—or more effective—fault-finding results. These results lead us to conclude that, while coverage criteria are potentially useful as test adequacy criteria, the use of coverage-directed test generation is more questionable as a means of creating tests within the domain of avionics systems. If simple random test generation can yield equivalently sized—but more effective test suites—for more traditional coverage criteria such as decision, condition or MC/DC coverage, then more research must be conducted before automated test generation can be recommended.

We do not wish to condemn a particular test generation method or recommend another. Instead, we want to shine a light on the risks of relying on structural coverage criteria as an assurance of effective testing. Given the important role of structural coverage criteria in the verification and validation of safety-critical avionics systems, we find these results quite troublesome. We believe that structural coverage criteria are, for the domain explored, potentially unreliable, and thus, unsuitable, as a target for determining the adequacy of automated test suite generation. Our observations indicate a need for methods of determining test adequacy that (1) provide a reliable measure of test quality and (2) are better suited as targets for automated techniques. At a minimum, such coverage criteria must, when satisfied, indicate that our test suites are better than simple random test suites of equal size. Such criteria must account for all of the factors influencing testing, including the program structure, the test oracle used, the nature of the state space of the system under test, and the method of test generation. Towards this goal, the OMC/DC criterion is an improvement in this regard, but we believe there is still much work to be done.

Until the challenges of determining the efficacy of generated test suites are overcome, we urge caution when automatically generating test suites: code coverage does not guarantee test quality. While automated test generation is an alluring possibility, savings of time and cost may not be worth the trade-off in the safety of the released software.

REFERENCES

[1] H. Zhu and P. Hall, "Test data adequacy measurement," *Software Engineering Journal*, vol. 8, no. 1, pp. 21–29, 1993.

[2] RTCA, *DO-178B: Software Considerations In Airborne Systems and Equipment Certification*. RTCA, 1992.

[3] S. Rayadurgam and M. Heimdahl, "Coverage based test-case generation using model checkers," in *Proc. of the 8th IEEE Int'l. Conf. and Workshop on the Engineering of Computer Based Systems*, pp. 83–91, IEEE Computer Society, April 2001.

[4] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," 2005.

[5] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," *PLDI05: Proc. of the 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2005.

[6] S. Anand, E. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn, "An orchestrated survey on automated software test case generation," *Journal of Systems and Software*, vol. 86, pp. 1978–2001, August 2013.

[7] M. Staats, G. Gay, M. W. Whalen, and M. P. Heimdahl, "On the danger of coverage directed test case generation," in *15th Int'l Conf. on Fundamental Approaches to Software Engineering (FASE)*, April 2012.

[8] M. Heimdahl, G. Devaraj, and R. Weber, "Specification test coverage adequacy criteria = specification test generation inadequacy criteria?," in *Proc. of the Eighth IEEE Int'l Symp. on High Assurance Systems Engineering (HASE)*, (Tampa, Florida), March 2004.

[9] M. P. Heimdahl and G. Devaraj, "Test-suite reduction for model based tests: Effects on test quality and implications for testing," in *Proc. of the 19th IEEE Int'l Conf. on Automated Software Engineering (ASE)*, (Linz, Austria), September 2004.

[10] M. Staats, M. Whalen, and M. Heimdahl, "Better testing through oracle selection (nier track)," in *Proceedings of the 33rd Int'l Conf. on Software Engineering*, pp. 892–895, 2011.

[11] M. Staats, G. Gay, and M. Heimdahl, "Automated oracle creation support, or: how I learned to stop worrying about fault propagation and love mutation testing," in *Proceedings of the 2012 Int'l Conf. on Software Engineering*, pp. 870–880, 2012.

[12] M. Whalen, G. Gay, D. You, M. Heimdahl, and M. Staats, "Observable modified condition/decision coverage," in *Proceedings of the 2013 Int'l Conf. on Software Engineering*, ACM, May 2013.

[13] G. Hagen, *Verifying safety properties of Lustre programs: an SMT-based approach*. PhD thesis, University of Iowa, December 2008.

[14] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, no. 99, p. 1, 2010.

[15] J. J. Chilenski and S. P. Miller, "Applicability of Modified Condition/Decision Coverage to Software Testing," *Software Engineering Journal*, pp. 193–200, September 1994.

[16] N. Juristo, A. Moreno, and S. Vegas, "Reviewing 25 years of testing technique experiments," *Empirical Software Engineering*, vol. 9, no. 1, pp. 7–44, 2004.

[17] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria," 1994.

[18] P. Frankl and S. N. Weiss, "An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria," in *Proc. of the Symposium on Testing, Analysis, and Verification*, 1991.

[19] A. Namin and J. Andrews, "The influence of size and coverage on test suite effectiveness," 2009.

[20] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, (New York, NY, USA), pp. 435–445, ACM, 2014.

[21] E. Weyuker and B. Jeng, "Analyzing partition testing strategies," *IEEE Trans. on Software Engineering*, vol. 17, no. 7, pp. 703–711, 1991.

[22] T. Chen and Y. Yu, "On the expected number of failures detected by subdomain testing and random testing," *IEEE Transactions on Software Engineering*, vol. 22, no. 2, 1996.

[23] D. Hamlet and R. Taylor, "Partition testing does not inspire confidence," *Software Engineering, IEEE Transactions on*, vol. 16, pp. 1402–1411, Dec 1990.

[24] W. Gutjahr, "Partition testing vs. random testing: The influence of uncertainty," *IEEE Transactions on Software Engineering*, vol. 25, no. 5, pp. 661–674, 1999.

[25] A. Arcuri, M. Z. Z. Iqbal, and L. C. Briand, "Formal analysis of the effectiveness and predictability of random testing," in *ISSTA*, pp. 219–230, 2010.

[26] A. Arcuri and L. C. Briand, "Adaptive random testing: An illusion of effectiveness?," in *ISSTA*, 2011.

[27] A. Gargantini and C. Heitmeyer, "Using model checking to generate tests from requirements specifications," *Software Engineering Notes*, vol. 24, pp. 146–162, November 1999.

[28] R. Gopinath, C. Jensen, and A. Groce, "Code coverage for suite evaluation by developers," in *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, (New York, NY, USA), pp. 72–82, ACM, 2014.

[29] R. Majumdar and K. Sen, "Hybrid concolic testing," in *ICSE*, pp. 416–426, 2007.

[30] Y. Yu and M. Lau, "A comparison of MC/DC, MUMCUT and several other coverage criteria for logical decisions," *Journal of Systems and Software*, vol. 79, no. 5, pp. 577–590, 2006.

[31] S. Kandl and R. Kirner, "Error detection rate of MC/DC for a case study from the automotive domain," *Software Technologies for Embedded and Ubiquitous Systems*, pp. 131–142, 2011.

[32] A. Dupuy and N. Leveson, "An empirical evaluation of the MC/DC coverage criterion on the hete-2 satellite software," in *Proc. of the Digital Aviation Systems Conf. (DASC)*, (Philadelphia, USA), October 2000.

[33] A. Gacek, "JKind - a Java implementation of the KIND model checker." https://github.com/agacek, 2015.

[34] G. Fraser, F. Wotawa, and P. Ammann, "Issues in using model checkers for test case generation," *Journal of Systems and Software*, vol. 82, no. 9, pp. 1403–1418, 2009.

[35] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Network Distributed Security Symposium (NDSS)*, Internet Society, 2008.

[36] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson, "Model-based testing of object-oriented reactive systems with spec explorer.," in *Formal Methods and Testing* (R. M. Hierons, J. P. Bowen, and M. Harman, eds.), vol. 4949 of *Lecture Notes in Computer Science*, pp. 39–76, Springer, 2008.

[37] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Comput. Surv.*, vol. 43, pp. 11:1–11:29, Feb. 2011.

[38] P. McMinn, "Search-based software test data generation: A survey," *Software Testing, Verification and Reliability*, vol. 14, pp. 105–156, 2004.

[39] "Reactive systems inc. Reactis Product Description." http://www.reactive-systems.com/index.msp.

[40] RTCA/DO-178C, "Software considerations in airborne systems and equipment certification."

[41] A. Murugesan, S. Rayadurgam, and M. Heimdahl, "Modes, features, and state-based modeling for clarity and flexibility," in *Proceedings of the 2013 Workshop on Modeling in Software Engineering*, 2013.

[42] "Mathworks Inc. Simulink." http://www.mathworks.com/products/simulink, 2015.

[43] "MathWorks Inc. Stateflow." http://www.mathworks.com/stateflow, 2015.

[44] N. Halbwachs, *Synchronous Programming of Reactive Systems*. Klower Academic Press, 1993.

[45] A. Rajan, M. Whalen, M. Staats, and M. Heimdahl, "Requirements coverage as an adequacy measure for conformance testing," 2008.

[46] J. Andrews, L. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?," *Proc of the 27th Int'l Conf on Software Engineering (ICSE)*, pp. 402–411, 2005.

[47] C. Van Eijk, "Sequential equivalence checking based on structural similarities," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,*, vol. 19, no. 7, pp. 814–819, 2002.

[48] J. Chilenski, "An investigation of three forms of the modified condition decision coverage (MCDC) criterion," Tech. Rep. DOT/FAA/AR-01/18, Office of Aviation Research, Washington, D.C., April 2001.

[49] M. Staats, G. Gay, and M. Heimdahl, "Automated oracle creation support, or: how I learned to stop worrying about fault propagation and love mutation testing," in *Proceedings of the 2012 Int'l Conf. on Software Engineering*, pp. 870–880, IEEE Press, 2012.

[50] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. pp. 80–83, 1945.

[51] A. Rajan, M. Whalen, and M. Heimdahl, "The effect of program and model structure on MC/DC test adequacy coverage," in *Proc. of the 30th Int'l Conf. on Software engineering*, pp. 161–170, ACM, 2008.

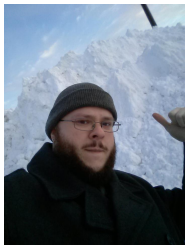
[52] M. Vivanti, A. Mis, A. Gorla, and G. Fraser, "Search-based data-flow test generation," in *ISSRE'13: Proceedings of the 24th IEEE Int'l Symposium on Software Reliability Engineering*, IEEE Press, Nov. 2013.

[53] M. Staats, P. Loyola, and G. Rothermel, "Oracle-centric test case prioritization," in *ISSRE*, pp. 311–320, 2012.

[54] P. McMinn, M. Stevenson, and M. Harman, "Reducing qualitative human oracle costs associated with automatically generated test data," in *Proceedings of the First International Workshop on Software Test Output Validation*, STOV '10, (New York, NY, USA), pp. 1–4, ACM, 2010.

[55] J. Andrews, L. Briand, Y. Labiche, and A. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *Software Engineering, IEEE Transactions on*, vol. 32, pp. 608 –624, aug. 2006.

[56] G. Devaraj, M. Heimdahl, and D. Liang, "Coverage-directed test generation with model checkers: Challenges and opportunities," *Computer Software and Applications Conf., Annual Int'l*, vol. 1, pp. 455–462, 2005.



Gregory Gay is an Assistant Professor of Computer Science & Engineering at the University of South Carolina. His research interests include automated testing and analysis—with an emphasis on test oracle construction—and search-based software engineering. Greg received his Ph.D. from the University of Minnesota, working with the Critical Systems research group, and an M.S. from West Virginia University.

Matt Staats has worked as a research associate at the Software Verification and Validation lab at the University of Luxembourg and at the Korean Advanced Institute of Science and Technology in Daejeon, South Korea. He received his Ph.D. from the University of Minnesota-Twin Cities. Matt Staats's research interests are realistic automated software testing and empirical software engineering. He is currently employed by Google, Inc.

Michael Whalen is a Program Director at the University of Minnesota Software Engineering Center. Dr. Whalen is interested in formal analysis, language translation, testing, and requirements engineering. He has developed simulation, translation, testing, and formal analysis tools for Model-Based Development languages including Simulink, Stateflow, SCADE, and $RSML^{-e}$, and has published extensively on these topics. He has led successful formal verification projects on large industrial avionics models, including displays (Rockwell-Collins ADGS-2100 Window Manager), redundancy management and control allocation (AFRL CerTA FCS program) and autoland (AFRL CerTA CPD program). He has recently been researching tools and techniques for scalable compositional analysis of system architectures.

Mats P.E. Heimdahl is a Full Professor of Computer Science and Engineering at the University of Minnesota, the Director of the University of Minnesota Software Engineering Center (UMSEC), and the Director of Graduate Studies for the Master of Science in Software Engineering program. He earned an M.S. in Computer Science and Engineering from the Royal Institute of Technology (KTH) in Stockholm, Sweden and a Ph.D. in Information and Computer Science from the University of California at Irvine.

His research interests are in software engineering, safety critical systems, software safety, testing, requirements engineering, formal specification languages, and automated analysis of specifications.

He is the recipient of the NSF CAREER award, a McKnight Land-Grant Professorship, the McKnight Presidential Fellow award, and the awards for Outstanding Contributions to Post-Baccalaureate, Graduate, and Professional Education at the University of Minnesota