Automated Oracle Data Selection Support

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

Abstract—The choice of test oracle—the artifact that determines whether an application under test executes correctly—can significantly impact the effectiveness of the testing process. However, despite the prevalence of tools that support test input selection, little work exists for supporting oracle creation. We propose a method of supporting test oracle creation that automatically selects the *oracle data*—the set of variables monitored during testing—for expected value test oracles. This approach is based on the use of mutation analysis to rank variables in terms of fault-finding effectiveness, thus automating the selection of the oracle data. Experimental results obtained by employing our method over six industrial systems (while varying test input types and the number of generated mutants) indicate that our method—when paired with test inputs generated either at random or to satisfy specific structural coverage criteria—may be a cost-effective approach for producing small, effective oracle data sets, with fault finding improvements over current industrial best practice of up to 1,435 percent observed (with typical improvements of up to 50 percent).

Index Terms—Testing, test oracles, oracle data, oracle selection, verification

1 Introduction

There are two key artifacts to consider when testing software: the *test inputs* and the *test oracle*, which determines if the system executes correctly. Substantial research has focused on supporting the creation of test inputs but little attention has been paid to the creation of oracles [1]. However, existing research indicates that the choice of oracle has a significant impact on the effectiveness of testing [2], [3], [4], [5]. Therefore, we are interested in the development of automated tools that support the creation of a test oracle.

Consider the following testing process: (1) the tester selects inputs using some criterion—structural coverage, random testing, or engineering judgement, among others; (2) the tester then defines concrete, anticipated values for these inputs for one or more variables (internal variables or output variables) in the program. This type of oracle is known as an *expected value test oracle* [3]. Experience with industrial practitioners indicates that such test oracles are commonly used in testing critical systems, such as avionics or medical device systems.

Our goals here are twofold. First, the current practice when constructing expected value test oracles is to define expected values for only the outputs, a practice that can be suboptimal when faults that occur inside the system fail to propagate to these observed variables. Second, manually defining expected values is a time-consuming and, consequently, expensive process. It is simply not feasible to monitor everything.[1] Even in situations where an executable software specification can be used as an oracle—for instance, in some model-based development scenarios—limited visibility into embedded systems or the high cost of logging often make it highly desirable to have the oracle observe only a small subset of all variables.

To address these goals, we present and evaluate an approach for automatically selecting *oracle data*—the set of variables for which expected values are defined [4]—that aims at maximizing the fault finding potential of the testing process relative to cost. This oracle data selection process, illustrated in Fig. 1, is completely automated. First, we generate a collection of mutants from the system under test. Second, an externally generated test suite is run against the mutants using the original system as the oracle and logs of the values of a candidate set of variables are recorded after certain points in execution (i.e., at a certain timing granularity or after pre-defined execution points). Third, we use these logs to measure how often each variable in the candidate set reveals a fault in a mutant and, based on this information, we rank variable effectiveness. Finally, based on this ranking, we estimate which variables to include in the oracle data. The underlying hypothesis is that, as with mutation-based test data selection, oracle data that is likely to reveal faults in the mutants will also be likely to reveal faults in the actual system under test. Once this oracle data is selected, the tester defines expected values for each element of the oracle data. Testing then commences with a small, and potentially highly effective, oracle.

In previous work, we proposed this approach and applied it to a fixed number of mutants and test inputs generated to satisfy two specific structural coverage criteria [6]. Although the results were promising, the initial study was limited, and

- *G. Gay is with the Department of Computer Science & Engineering, University of South Carolina. E-mail: greg@greggay.com.*
- *M. Staats is with Google, Inc. E-mail: staatsm@gmail.com.*
- *M. Whalen and M. Heimdahl are with the Department of Computer Science and Engineering, University of Minnesota. E-mail: {whalen, heimdahl}@cs.umn.edu.*

Manuscript received 12 Aug. 2013; revised 20 Apr. 2015; accepted 25 Apr. 2015. Date of publication 21 May 2015; date of current version 13 Nov. 2015. Recommended for acceptance by P. Tonella.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2015.2436920

1. In some scenarios, such as regression testing, we can automate definition of expected values. Comparing large numbers of expected values is still potentially expensive, and our approach is still of value in such a scenario.

0098-5589 © 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

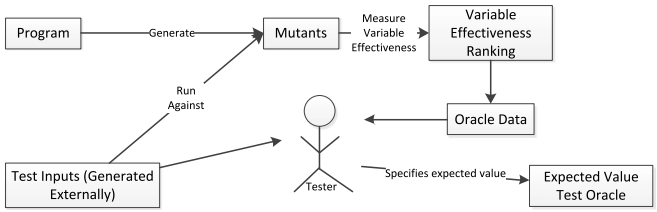


Fig. 1. Supporting expected value test oracle creation.

did not yield enough evidence to identify the specific conditions in which our approach would be useful. Specifically, we did not know how the use of structural coverage criteria—as compared to, for example, tests generated to assess compliance of a program to its specification—or the number of mutants used during training impacted our results.

To better evaluate our hypothesis and find answers to these additional questions, we have evaluated our approach using four commercial sub-systems from the civil avionics domain and two medical device systems, and using four different types of test input data—tests generated to satisfy two structural coverage metrics, tests generated to satisfy coverage of the requirements, and tests generated purely at random. We perform a comparison against two common *baseline approaches*: (1) *current practice*, favoring the outputs of the system under test as oracle data, and (2) *random selection* of the oracle data set. We also compare to an idealized scenario where the seeded faults in the test suites and mutants used for training are identical to the test suites and mutants in the final evaluation set; thus, providing an estimate of the maximum fault finding effectiveness we could hope to achieve with a mutation-based oracle data selection support method. We repeat the experiment using varying numbers of mutants in order to determine the amount of data needed to construct powerful, stable sets of oracle data.

We draw from our results three key conclusions:

- Our approach generally produces more effective test oracles than the alternative methods explored—in particular, outperforming output-based test oracles with observed improvements in fault finding of up to 1,435 percent, and consistent improvements of up to 50 percent. Even in cases where our approach is not effective, the results achieved are similar to using output-based test oracles.
- Our approach is the most effective when paired with test suites generated to exercise the internal structure of the program under test. When our approach is applied to randomly generated test inputs, improvements are more subdued (3-43 percent), and improvements are nearly non-existent when applied to requirements-based tests.
- Finally, the choice of test inputs impacts the number of mutants required to derive effective sets of oracle data. In our study, for a given system, the oracle data for structural test suites is improved by the use of a large number of mutants (up to 125 mutants). For requirements based tests suites, no improvements are generally observed after the number of mutants exceeds 50.

We therefore conclude that our approach may be a cost effective method of supporting the creation of an oracle data set—particularly in scenarios where test suites are generated to satisfy structural coverage criteria.

2 Background & Related Work

In software testing, a *test oracle* is the artifact used to determine whether the software is working correctly [7]. There are many types of test oracles, ranging from program invariants to "no crash oracles" [1]. In our experience with industrial partners developing critical software systems, one commonly used class of test oracles are *expected value test oracles*—test oracles that, for each test input, specify concrete values the system is expected to produce for one or more variables (internal state and/or output). Thus, in designing an expected value test oracle, two questions must be answered: (1) *"what variables should be monitored?"* and (2), *"how often should the values of those variables be checked?"*.

During testing, the oracle compares the actual values produced by the program against the expected values at the selected points in execution. In current practice, testers generally must manually select these expected values without the aid of automation; this process is naturally dependent on the skill of the tester. Our goal is therefore to develop tools and techniques to support and optimize the selection of the oracle data. While others have focused on the latter question—how often valuess should be checked [8]—we focus on the former question: what variables should be monitored for faults?

In Richardson et al.'s definition of a test oracle, an oracle is composed of two parts—the oracle *information* and oracle *procedure* [7]. The oracle procedure renders a pass/fail verdict on an executed test at selected points in execution using the data collected in the oracle information. For expected value oracles, the oracle information must contain two distinct sets of information: the oracle *value set* and the oracle *data set*. The oracle data set is the subset of variables (internal state and outputs) for which expected values are specified; i.e., what variables the oracle must monitor. The oracle value set is then the set of expected values for those variables [3], [6].

For example, an oracle may specify expected values for all of the outputs; we term this an *output-only* oracle. This type of oracle appears to be the most common expected value oracle used in testing critical systems. Other types of test oracles include, for example, *output-base* oracles, whose oracle data set contains all the outputs, followed by some number of internal state variables, and *maximum* oracles, whose oracle data set contains *all* of the outputs and internal state variables.

It has been empirically shown that larger oracle data sets are generally more powerful than smaller oracle data sets [2], [3], [5]. (In the remainder of this paper the *size* of an oracle is the number of variables used in the oracle data set.) This is due to *fault propagation*—faults leading to incorrect states usually propagate and manifest themselves as failures for only a subset of program variables; larger oracle data sets increase the likelihood that such a variable will be monitored by the test oracle. This limited fault propagation is particularly a concern in the avionics community, where complex Boolean expressions can mask out faults and prevent incorrect values from affecting output variables. Common practice dictates focusing on the output variables, but the effects of this masking can delay or prevent the propagation of faults to these variables. It would be desirable to identify the

variables whose definition creates the potential for masking and check the behavior of these internal "bottleneck" points.

Naturally, one solution is to use the maximum oracle, an oracle that observes and checks the behavior of all variables in the system. This is always the most effective expected value test oracle, but using it is often prohibitively expensive. This is particularly the case when (1) expected values must be manually specified—a highly labor intensive process, especially when results are checked at multiple execution points or time steps during a single test—or (2) when the cost of monitoring a large oracle data set is prohibitive, e.g., when testing embedded software on the target platform.

One might ask at this point if a "fault" is really a fault if it fails to propagate to an output variable? Frequently the system under test may be capable of *absorbing* the fault, as the corrupt program state fails to impact the final behavior of the software. Indeed, in some cases these "faults" cannot ever propagate to an output. However, in practice (as we will see later in Section 5) such faults typically do propagate to the output under some circumstance, and thus these faults should still be identified and corrected. This is particularly true for safety-critical systems, such as avionic or medical devices, where undiscovered faults can lead to loss of equipment or harm to human operators. This motivates the use of internal variables, which allow testers to correct more issues in the system without incurring the cost of the maximum oracle.

2.1 Related Work

Work on test oracles often focuses on methods of constructing oracles from other software engineering artifacts, such as formal software requirements [1]. In our work, we are not constructing the entire oracle; rather, we are identifying effective oracle data sets, from which effective expected value oracles can be built. We are not aware of any work proposing or evaluating alternative methods of selecting the oracle data.

Voas and Miller have proposed the PIE approach, that—like our work—relies on a form of mutation analysis [9]. Their approach could be used to select internal variables for monitoring, though evaluation of this idea is lacking. More recent work has demonstrated how test oracle selection can impact the effectiveness of testing, indicating a need for effective oracle selection techniques [3], [4].

Memon and Xie have applied mutation testing in order to optimize the oracle procedure in an expected value test oracle for event-driven graphical user interfaces [8]. The authors' goal was to construct cheaper—but still effective—test oracles by considering *how often* to invoke the potentially expensive oracle procedure (i.e., to compare expected and actual values). The authors found that (1) transient faults are as common as persistent ones, and (2) the majority of transient faults were linked to two event types. Thus, by comparing values only after those events, the oracle can catch the majority of faults while remaining computationally affordable.

Memon and Xie's goals are similar to ours—they are supporting the construction of efficient expected value test oracles. However, our respective domains require different approaches to this task. In the avionics systems that we have studied, complexity stems from the hundreds or thousands of variables that can potentially be monitored, and there are not necessarily discrete events that we can choose to ignore (especially when considering critical systems). Thus, even if we could execute the oracle procedure less often, we are still left with the question of which variables to monitor.

Several tools exist for automatically generating invariant-based test oracles for use in regression testing, including Eclat [10], DiffGen [11], and work by Evans and Savoia [12]. However, Briand et al. demonstrate for object-oriented systems that expected value oracles outperform state-based invariants, with the former detecting faults missed by the latter [2].

Fraser and Zeller use mutation testing to generate both test inputs and test oracles [13] for Java programs. The test inputs are generated first, followed by generation of post-conditions capable of distinguishing the mutants from the program with respect to the test inputs. Unlike our work, the end result of their approach is a *complete* test case, with inputs paired with expected results (in this case, assertions). Such tests, being generated from the program under test, are guaranteed to pass (except when the program crashes). Accordingly, the role of the user in their approach is to decide, for each input and assertion pair, if the program is working correctly. Thus, in some sense their approach is more akin to invariant generation than traditional software testing. The most recent version of this work attempts to generalize the result of their approach to simplify the user's task [14]. However, this creates the possibility of producing *false positives*, where a resulting parameterized input/assertion can indicate faults when none exist—further changing the user's task.

With respect to evaluation, no comparisons against baseline methods of automated oracle selection are performed; Generated tests and assertions are compared against developer-produced tests and assertions, but the cost—i.e., number of developer tests/assertions—is not controlled. Thus, relative cost-effectiveness cannot accurately be assessed. Additionally, their approach selects enough tests and assertions to detect *all* generated mutations, and thus allows no method of controlling for the human cost.

Our work chiefly differs from other approaches in that we are trying to *support* creation of a test oracle, rather than automate it. Oracle creation support can be considered an approach to addressing the *human oracle cost problem*—that is, the problem of alleviating the burden on the human tester when no means exist to completely automate the generation of a test oracle [1]. Other approaches to addressing the human oracle problem include the automated generation of human-readable test inputs [15], test suite reduction aimed at reducing human workload [16], and incorporating knowledge from project artifacts in order to generate test cases that human testers can easily understand [17].

This report is an extension of previously published work [6], and improves upon it in two key ways. First, we explore the effectiveness of our approach with respect to a variety of test input types. Second, we investigate how the number of mutants generated impacts the process. These analyses help us to better understand the effectiveness of our approach in different testing contexts—in particular, how structural coverage may be helped by good oracle data selection—and the cost and scalability of our approach.

3 Oracle Data Selection

Our approach for selecting the oracle data set is based on the use of mutation testing [18]. In mutation testing, a large

set of programs—termed *mutants*—are created by seeding faults (either automatically or by hand) into a system. Test input capable of distinguishing the mutant from the original is said to *kill* the mutant. In our work, we adopt this approach for oracle creation support. Rather than generate test inputs that kill the mutants, however, we use mutants to automatically generate an oracle data set that—when used in an expected value oracle, and with a fixed set of test inputs—kills the mutants. To construct this data set, we perform the following:

1) Generate several mutants, called the *training set*, from our system under test.
2) Run test inputs over the training set and the original system, collecting logs of the values of a set of candidate variables at pre-defined observation points.
3) Use these logs to determine which variables distinguish each mutant from the original system.
4) Process this information to create a list of variables ordered in terms of apparent fault finding effectiveness, the *variable ranking*.
5) Examine this ranking, along with the mutants and test inputs, to estimate (as x) how large the oracle data set should be. Alternatively, the tester can specify x based on the testing budget.
6) Select the top x variables in the ranking for use in the final oracle data set.

While conceptually simple, there are several relevant parameters to be considered for each step. The following sections will outline these parameters, as well as the rationale for the decisions that we have made.

3.1 Mutant Generation and Test Input Source

During mutation testing, *mutants* are created from an implementation of a system by introducing a single fault into the program. Each fault results from either inserting a new operator into the system or by replacing an operator or variable with a different operator or variable. This mutation generation is designed such that no mutant will "crash" the system under test. The mutation testing operators used in this experiment include changing an arithmetic operator, changing a relational operator, changing a Boolean operator, introducing the Boolean $\neg$ operator, using the value of a variable from the previous computation cycle, changing a constant expression by adding or subtracting 1 from int and real constants (or by negating Boolean constants), and substituting a variable reference with another variable of the same type.

The type of faults used to create mutants may impact the effectiveness of the selected oracle data when used to test the actual system under test. Note that the type of mutants used in the evaluation in this report are similar to those used by Andrews et al., where the authors found that generated mutants are a reasonable substitute for actual failures in testing experiments [19]. Additionally, recent work from Just et al. suggests a significant correlation between mutant detection and real fault detection [20]. This offers evidence that mutation-based techniques will be useful for supporting the creation of oracles for real-world systems.

Our approach can be used with any set of test inputs. In this work, we assume the tester is equipped with an existing set of test inputs and wishes to determine what oracle data is likely to be effective with said test inputs. This assumption allows the numerous existing methods of test input selection to be paired with our approach for oracle data selection. This scenario is likely within our domain of interest.

3.2 Variable Ranking

Once we have generated mutants, we then run suites of test inputs over both the mutants and the original program (with the execution over the original program serving as a golden run [21]). The user chooses a set of internal and output variables to serve as candidates for oracle data selection. Then, during execution of these inputs, we collect the value of every variable in that candidate set at various points during execution. A variable has detected a fault when the variable value in the original "correct" system differs from the variable value produced by a mutant, for some test. We track the mutants killed by each variable.

In order to form an oracle data set, snapshots of the state of the candidate variables must be collected. *When and how often* those snapshots are logged will help determine what data is available to our variable ranking approach. Options can generally be divided into *event-driven* logging or *time-driven* logging. In the event-driven case, the state of the candidate variables might be logged at certain pre-defined points in execution. This could include, for example, after discrete computational cycles, when new input is passed to the system, when new output is issued from the system, or when certain types of events occur. In a time-driven process, snapshots could be taken at a certain timing granularity—say, every X seconds.

The more often snapshots are taken, the more data the variable ranking algorithm has to work with. However, this logging also incurs a computational overhead on the execution of the system. Therefore, the selection of logging frequency depends on the project that an oracle is being produced for.

Once we have collected these traces, we can produce a set of variables ranked according to effectiveness. One possible method of producing this ranking is simply to order variables by the number of mutants killed. However, the effectiveness of individual variables can be highly correlated. For example, when a variable v_a is computed using the value of a variable v_b: if v_b is incorrect for some test input, it is highly probable that v_a is also incorrect. Thus, while v_a or v_b may be highly effective when used in the oracle data set, the combination of both is likely to be only marginally more effective than the use of either alone.

To avoid selecting a set of dependent variables that are individually effective—but duplicative as a group—we make use of a greedy algorithm for solving the *set covering problem* [22] to produce a ranked set of variables. In the set covering problem, we are given several sets with some elements potentially shared between the sets. The goal is then to select the minimum set of elements such that one element from each set has been selected. In this problem, each set represents a mutant, and each element of the set is a variable capable of detecting the mutant for at least one of the test inputs. Calculating the smallest possible set covering is an NP-complete problem [23]. Thus, we employ a well-known effective greedy algorithm to solve the problem [24]: (1) select the element covering the largest number of sets, (2) remove from consideration all sets covered by said element, and (3) repeat until all sets are covered.

In our case, each element removed corresponds to a variable. These variables are placed in a ranking in the order they were removed (with the most effective variables being removed first). The resulting ranking can then be used to produce an oracle data set of size n by simply selecting the top n variables from the list.

In the case examples studied, all variables are *scalar* and cannot be a heap object or pointers. Thus, comparison is straightforward. As our approach requires only that expected and actual values differ (rather than *how* they differ), mutation-based oracle optimization should be effective when making comparisons of any data structure, as long as an accurate and efficient method of comparing for equality exists.

Additionally, the systems explored in this work contain no nested loops. For a "step" of the system, every variable is declared and assigned exactly once. Thus conceptually, there exists no difference between output variables and internal variables in terms of how expected values should be defined.

3.3 Estimating Useful Oracle Data Size

If the testers would—by default—use the output variables of the system as their oracle data set, then a natural use of this mutation-based technique would be to select a new oracle data set of the same size. That is, if the testers would have used n output variables in their oracle data set, then they could use our technique to select n variables from the full set of internal and output variables.

However, one potential strength of our technique is that it can produce oracle data sets of any size, granting freedom to testers to choose an oracle data set to fit their budget, schedule, monitoring limitations, or other testing concerns. Once we have calculated the ranked list of variables, we can select an oracle data set of size 1, 2, 3, etc. up to the maximum number of variables in the system.

In some scenarios, the tester may have little guidance as to the appropriate size of the oracle data. In such a scenario, it would be ideal to offer a recommendation to the tester. One would like to select an oracle data set such that the size of the set balances cost and effectiveness—that is, not so small that potentially useful variables are omitted, and not so large that a significant number of variables contribute little to performance.

To accomplish this, testers could examine the fault finding effectiveness of oracle data sets of size 1, 2, 3, etc. The effectiveness of these oracles will increase with the oracle's size, but the increases will likely diminish as the oracle size increases. As a result, it is generally possible to define a natural cutoff point for recommending an oracle size; if the fault finding improvement between an oracle of size n and size $n + 1$ is less than some threshold, we recommend an oracle of size n.

In practice, establishing a threshold will depend on factors specific to the testing process. In our evaluation, we examine oracle sizes up to $max(10, (2 * \#\ output\ variables))$ and explore two potential thresholds: 5 and 2.5 percent.

4 Evaluation

We wish to evaluate whether our approach yields effective oracle data sets. While it would be preferable to directly compare against existing algorithms for selecting oracle data, to the best of our knowledge, no such methods exist. We therefore compare our technique against two *baseline approaches* for oracle data set selection, detailed later, as well as against an idealized *best case* application of our own approach.

We also would like to determine the impact of the choice of test input generation criteria on our approach. In particular, we are interested if the effectiveness varies when moving from tests generated to satisfy structural coverage criteria—which tend to be short and targeted at specific code constructs—to requirements-based test inputs, which tend to be longer and are directly concerned with showing the relationship between the inputs and outputs (i.e., they attempt to cause values to propagate through the system). Finally, we are interested in cost and scalability, specifically how the number of mutants used to select the oracle data impacts the effectiveness of the resulting oracle.

We have explored the following research questions:

Research Question 1 (RQ1): *Is our approach more effective in practice than baseline approaches to oracle data selection?*

Research Question 2 (RQ2): *What is the maximum potential effectiveness of the mutation-based approach, and how effective is the realistic application of our approach in comparison?*

Research Question 3 (RQ3): *How does the choice of test input data impact the effectiveness of our approach?*

Research Question 4 (RQ4): *What impact does the number of training mutants have on the effectiveness of our approach?*

Research Question 5 (RQ5): *What is the ratio of output to internal variables in the generated oracle data sets?*

4.1 Experimental Setup Overview

We have used four industrial systems developed by Rockwell Collins Inc. engineers and two additional subsystems of an infusion pump created for medical device research [25]. The Rockwell Collins systems were modeled using the Simulink notation from Mathworks Inc. [26] and the remaining systems using Stateflow [26], [27]. The systems were automatically translated into the Lustre programming language [28] to take advantage of existing automation. In practice, Lustre would then be automatically translated to C code. This translation is a simple transformation, and if applied to C, the case study results would be identical.

The four Rockwell Collins systems represent sizable, operational modules of industrial avionics systems. Two systems, *DWM1* and *DWM2*, represent distinct subsystems of a display window manager (DWM) for a commercial cockpit display system. Two other systems, *Vertmax_Batch* and *Latctl_Batch*, describe the vertical and lateral mode logic for a flight guidance system. The remaining two systems, *Infusion_Mgr* and *Alarms*, represent the prescription management and alarm-induced behavior of an infusion pump. Both systems come with a set of real faults that we can use to assess real-world fault-finding.

Information related to these systems is provided in Table 1. Subsystems indicates the number of Simulink subsystems presents, while blocks represents the number of blocks used. Outputs and internals indicates the number of output and internal variables present. For the examples developed in Stateflow, we list the number of Stateflow states, transitions, and internal and output variables. As we

TABLE 1
Case Example Information

	# Subsystems	# Blocks	# Output Variables	# Internal Variables
DWM_1	3,109	11,439	7	569
DWM_2	128	429	9	115
Vertmax	396	1,453	2	415
Latctl	120	718	1	128
	# States	**# Transitions**	**# Output Variables**	**# Internal Variables**
Infusion_Mgr	27	50	5	107
Alarms	78	107	5	182
Infusion_Mgr (faulty)	30	47	5	86
Alarms (faulty)	81	101	5	155

have both faulty and corrected versions of *Infusion_Mgr* and *Alarms*, we list information for both.

For the purposes of the study conducted in this work, we automatically generate tests—both randomly and with a coverage-directed search algorithm—that are effectively *unit tests* for modules of synchronous reactive systems. Computation for synchronous reactive systems takes place over a number of execution "cycles." That is, when input is fed to the system, there is a corresponding calculation of the internal variables and outputs. A single cycle can be considered a sequence of assignments of values to variables. A loop would be considered as a series of computational cycles. This naturally answers the question of "when" to log variable values for oracle generation—after each computational cycle, we can log or check the current value of the candidate variables.

For each case example, we performed the following steps:

1) *Generated test input suites.* We created 10 test suites satisfying decision coverage, and 10 test suites satisfying MC/DC coverage, and—for systems with known requirements—10 test suites satisfying UFC (requirements) coverage using automatic counterexample-based test generation. We also produced randomly constructed test suites of increasing size. (Section 4.2).
2) *Generated training sets.* We randomly generated 10 sets of 125 mutants to be used to construct oracle data sets, each containing a single fault. (Section 4.3.)
3) *Generated evaluation sets.* For each training set, we randomly generated a corresponding evaluation set of 125 mutants, each containing a single fault. Each mutant in a evaluation set is guaranteed to *not* be in the training set. (Section 4.3.)
4) *Ran test suite on mutants.* We ran each mutant (from both training and evaluation sets) and the original case example using every test suite and collected the internal state and output variable values produced after each computation cycle. This yields raw data used for the remaining steps of our study. (Section 4.5.)
5) *Generated oracle data sets.* We used the information gathered to generate oracle data sets using the algorithm detailed in Section 3. Data sets were generated for each training set *and* for each evaluation set (in order to calculate an idealized ceiling performance). We also generated random and output-based baseline rankings. These rankings are used to generate oracles of various sizes. (Section 4.5.)
6) *Assessed fault finding ability of each oracle and test suite combination.* We determined how many mutants were detected by every oracle, using each test suite. For oracles generated using a training set, the corresponding evaluation set was used; for oracles generated using an evaluation set, the same evaluation set was used. For the *Infusion_Mgr* and *Alarms* systems, we also assess the performance of each oracle and test suite combination on the set of real faults (Section 4.6.)

4.2 Test Suite Generation

As noted previously, we assume the tester has an existing set of test inputs. Consequently, our approach can be used with any method of test input selection. As we are studying the effectiveness using avionics systems, two structural coverage criteria are likely to be employed: decision coverage and modified condition/decision coverage (MC/DC) [29].

Decision coverage is a criterion concerned with exercising the different outcomes of the Boolean decisions within a program. Given the expression, `((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 automated 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/DCcoverage 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 [30].

We are also interested in test suites designed to satisfy criteria that are not focused on the internal structure of the

system under test, such as *Unique First Cause (UFC)*—a black-box criterion that measures coverage of a set of requirements encoded as temporal logic properties [31]. Adapted from MC/DC a test suite satisfies UFC coverage over a set of requirements—encoded as LTL formulas—if executing the test cases in the test suite guarantees that every basic condition in each formula has taken on all possible outcomes at least once, and each basic condition in each expression has been shown to independently affect the outcome of the expression. As requirement were not available for the *Infusion_Mgr* and *Alarms* systems, we only produce UFC-satisfying tests for the four Rockwell Collins systems.

We used counterexample-based test generation to generate tests satisfying the three coverage criteria [32], [33]. In this approach, each coverage obligation is encoded as a temporal logic formula and the 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 property of 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 DSE/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 [29]. We have used the JKind model checker [34], [35] in our experiments because we have found that it is efficient and produces tests that are easy to understand [36].

Counterexample-based test generation results in a separate test for each coverage obligation. This results in a large amount of redundancy in the tests generated, as each test likely covers several coverage obligations. Such an unnecessarily large test suite is unlikely to be used in practice. We therefore reduce each generated test suite while maintaining coverage. We use a simple randomized greedy algorithm. It begins by determining the coverage obligations satisfied by each test generated, and initializing an empty test set *reduced*. The algorithm then randomly selects a test input from the full set of tests; if it satisfies obligations not satisfied by test input already in *reduced*, it is added to the set. The algorithm continues until all tests have been removed from the full set of tests.

We produce 10 test suites for each combination of case example and coverage criterion to control for the impact of randomization. We also produced 10 suites of random tests—increasing in size from 10 to 100 tests—in order to determine the effectiveness of our approach when applied to test suites that were not designed to fulfill a coverage criterion.

For the systems with real faults, we generate coverage-satisfying 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 obviously been generated to provide coverage over the code as it currently exists.

4.3 Mutant Generation

For each case example, 250 mutants are created by introducing a single fault into the correct implementation (using the approach discussed in Section 3.1). We then produce 10 *training sets* by randomly selecting 10 subsets of 125 mutants. For *each* training set, the 125 mutants *not* selected for the training set are used to construct an evaluation set.

Mutants can be divided into *weak mutants*—mutations that infect the program state, but where the result of that corruption does not propagate to a variable checked by the oracle—and *strong mutants*—mutants where state is corrupted and that corruption does propagate. The mutants as they are generated are weak mutants, because there is no a-priori way without analysis of determining whether a mutant is functionally equivalent to the original program. However, we perform a post-processing analysis (described below) to remove functionally equivalent mutants, so the mutants used for testing are *strong mutants*.

We remove *functionally equivalent* mutants from the evaluation set using the JKind model checker [34], [35]. This is possible due to the nature of the systems in our study—each system is finite; thus, determining equivalence is decidable and fast.[2] This removal is done for the evaluation sets because equivalent mutants represent a potential threat to validity in our evaluation. No mutants are removed from the training sets.

In practice, one would *only* generate a training set of mutants for use in building an oracle data set. We generate both training and evaluation sets in order to measure the performance of the proposed generation approach for research purposes. Thus, while it is possible we select the oracle data based partly on equivalent mutants which cannot affect the output, our evaluation measures the ability of the approach to detect provably faulty systems.

To address Question 4, we also produced four subsets of each training set. These subsets, respectively, contained 10, 25, 50, and 75 percent of the training mutants. These subsets allow us to determine the effectiveness of oracle data generated with fewer mutants.

4.4 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 *Alarms* 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. Most are non-trivial faults—these were not mere typos or operand mistakes, they require specific conditions to trigger, and extensive verification efforts were required to identify these faults.

2. Equivalence checking is fairly routine in the hardware domain; a good introduction can be found in [37].

TABLE 2
Real Faults for Infusion Pump Systems

	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.

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

4.5 Oracle Data Set Generation

For each given case example, we ran the test suites against each mutant and the original version of the program. For each execution of the test suite, we recorded the value of every internal variable and output at each step of every test using an in-house Lustre interpreter. This raw trace data is then used by our algorithm and our evaluation.

For each combination of set of mutants (training sets and evaluation sets) and test suite, we generated an oracle ranking using the approach described in Section 3. The rankings produced from training sets reflect how our approach would be used in practice; these sets are used in evaluating our research questions. The rankings produced from evaluation sets represent an idealized testing scenario, one in which we already know the faults we are attempting to detect. Rankings generated from the evaluations sets, termed *idealized rankings*, hint at the maximum potential effectiveness of our approach and are used to address Question 2.

Each ranking was limited to m variables (where m is 10 or twice the number of output variables, whichever was larger) since oracles significantly larger than output-only oracles were deemed unlikely to be used in practice. Note that the time required to produce a single ranking—generate mutants, run tests, and apply the greedy set cover algorithm—is less than one hour for each pairing of case example and test suite.

To answer Questions 1 and 3, we compare against two baseline rankings. First, to provide an unbiased ranking for comparison, the *random approach* creates completely random oracle rankings. The resulting rankings are simply a random ordering of the output and internal variables. Second, the *output-base* approach creates rankings by first selecting output variables—ordered at random—and then randomly selecting internal state variables until the size of the oracle matches the pre-determined threshold. Thus, the output-base rankings always lists the outputs first (i.e., more highly ranked) followed by the randomly-selected internal state variables. The output-based ranking reflects a common industrial approach to oracle data selection: focus on the output variables. However, there are two differences between the output-base oracles used in our evaluation and common practice: (1) we randomly vary the order of the output variables to avoid any particular biasing of the results (in the real world, no one order would typically be favored), and (2), we add randomly chosen internal variables to the oracle data set after prioritizing the outputs so that we can compare larger oracle sizes than would typically be employed.

4.6 Oracle Evaluation

To determine the fault finding effectiveness of a test suite t and oracle o on a case example, we simply compare the values produced by the original case example against every mutant using test suite t and the subset of variables corresponding to the oracle data for oracle o (the original "correct" system fulfills the role of the user in our approach, specifying expected values for the oracle data).

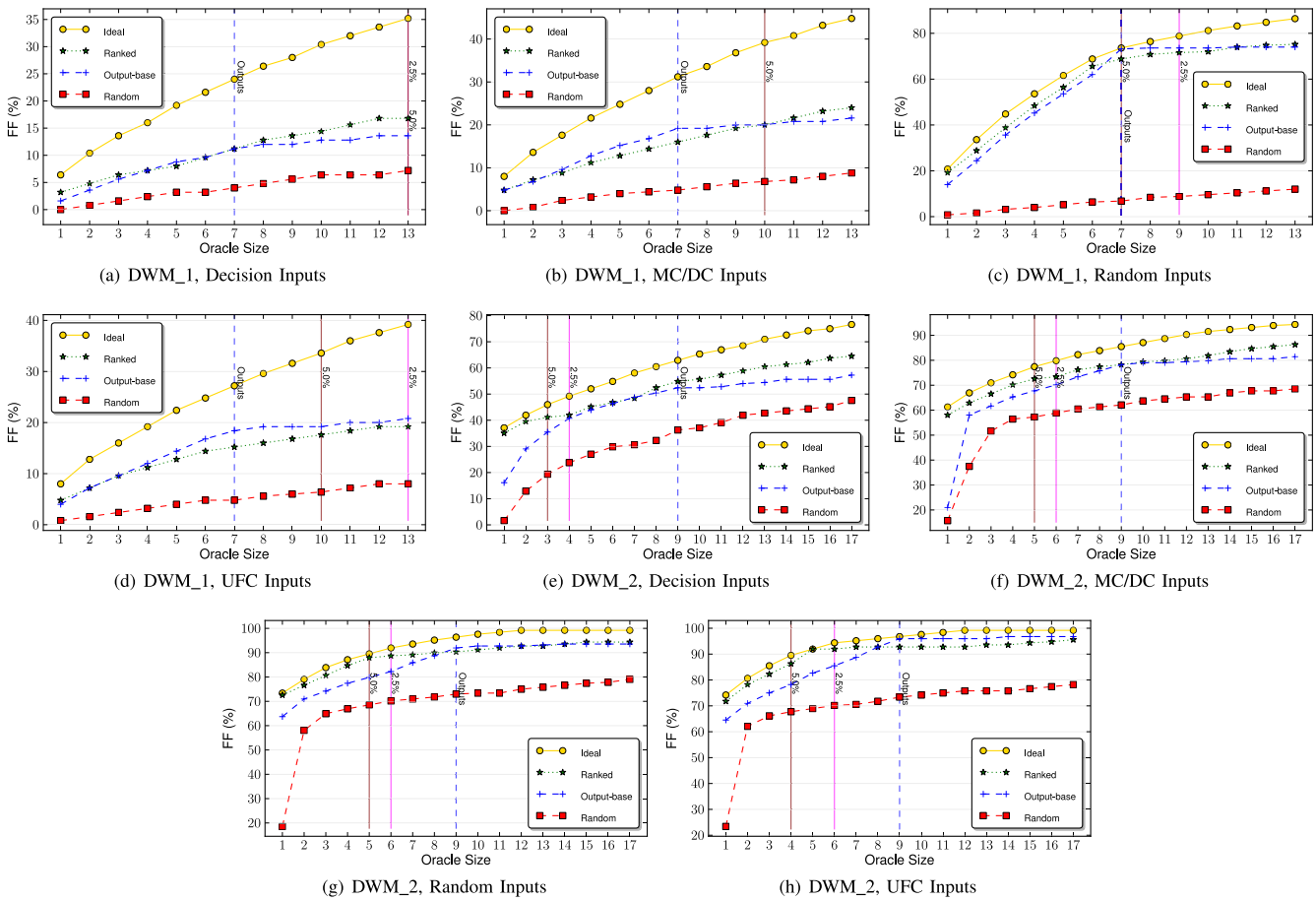
For all six case examples, we measure the fault finding effectiveness of oracles generated from the training sets using the corresponding evaluation sets, and we measure the effectiveness of the idealized oracles generated from evaluation sets using the same evaluation sets. The fault finding effectiveness of an oracle is computed as the percentage of mutants killed versus the total number of mutants in the evaluation set. We perform this analysis for each oracle and test suite for every case example, and use the produced results to evaluate our research questions.

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. Note that we do not generate separate "idealized" oracles when evaluating on real faults, as these would simply be oracles generated from additional mutants, and do not represent the same performance ceiling.

5 Results & Discussion

In this section, we discuss our results in the context of our four research questions. We begin by plotting the median fault finding effectiveness of the produced test oracles for increasing oracle sizes in Figs. 2, 3, 4, and 5.[3] Four ranking methods are plotted: both baseline rankings, our mutation-based approach, and an idealized mutation-based

3. For readability, we do not state "median" relative improvement, "median" fault finding, etc. in the text, though this is what we are referring to.



(a) DWM_1, Decision Inputs (b) DWM_1, MC/DC Inputs (c) DWM_1, Random Inputs

(d) DWM_1, UFC Inputs (e) DWM_2, Decision Inputs (f) DWM_2, MC/DC Inputs

(g) DWM_2, Random Inputs (h) DWM_2, UFC Inputs

Fig. 2. Median effectiveness of various approaches to oracle data selection for *DWM_1* and *DWM_2*. Yellow circles = Ideal; Green * = Ranked; Blue + = Output-base; Red squares = Random.

approach. For each subfigure, we plot the number of outputs as a dashed, vertical line. This line represents the size of an output-only oracle; this is the oracle size that would generally be used in practice. We also plot the 5 and 2.5 percent thresholds for recommending oracle sizes as solid lines (see Section 3.3). Note that the 2.5 percent threshold is not always met for the oracle sizes explored.

In Tables 3, 4, and 5, we list the median relative improvement in fault finding effectiveness using our proposed oracle data creation approach versus the output-base ranking. In Tables 6 and 7, we list the median relative improvement in fault finding effectiveness using the idealized mutation-based approach (an oracle data set built and evaluated on the same mutants) versus our mutation-based approach. As shown in Figs. 2, 3, 4, and 5, random oracle data performs poorly; thus, detailed comparisons were deemed uninteresting and are omitted.

5.1 Statistical Analysis

Before discussing the implications of our results, we would like to first determine which differences observed are statistically significant. With regard to *RQ1* and *RQ2*, we would like to determine with significance at what oracle sizes, and for which case examples, (1) the idealized performance of a mutation-based approach outperforms the actual performance of the mutation-based approach, and (2) the mutation-based approach outperforms the baseline ranking approaches. We evaluated the statistical significance of our results using a two-tailed bootstrap permutation test. We begin by formulating the following statistical hypotheses:[4]

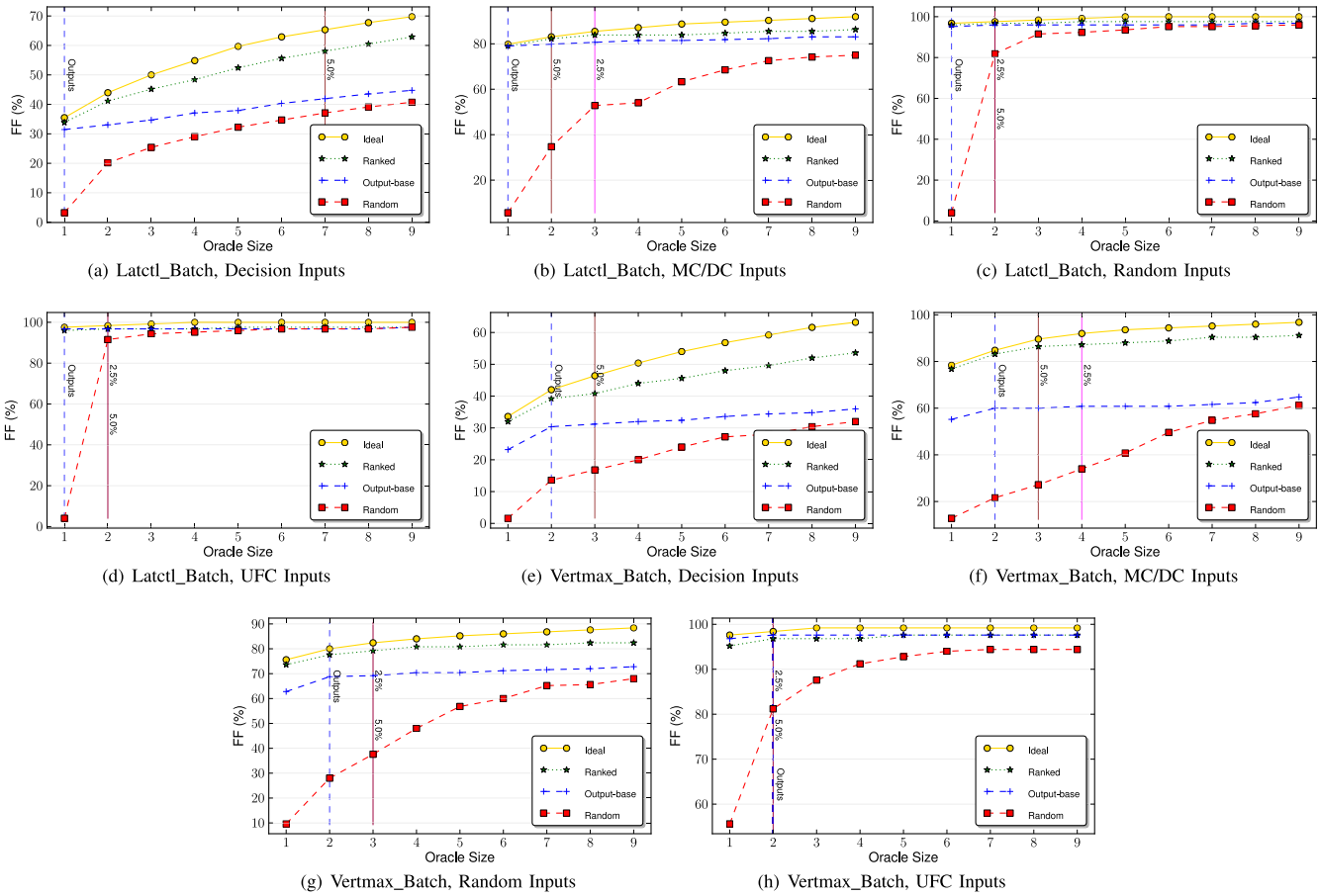
H_1: For a given oracle size m, the standard mutation-based approach outperforms the output-base approach.
H_2: For a given oracle size m, the standard mutation-based approach outperforms the random approach.
H_3: For a given oracle size m, the idealized approach outperforms the standard mutation-based approach.

Towards *RQ4*, we would also like to quantify—with significance—the number of mutants needed to train the oracle data. We repeated the same experiment, varying the number of mutants used to train the oracle—making use of training sets containing 10, 25, 50, and 75 percent of the mutants used to train oracles in the initial experiment. We have generated fault finding results using the same evaluation sets, and formulated the following hypothesis.

H_{4-7}: For a given oracle size m, the standard mutation-based approach, generated using a full training set, outperforms the standard mutation-based approach, generated with $N\%$ of the same training set.

The null hypothesis $H0_X$ for each hypothesis H_X above is that each set of values are drawn from the same distribution. To evaluate our hypotheses without any assumptions on the

4. As we evaluate each hypothesis for each case example and oracle size, we are essentially evaluating a set of statistical hypotheses.



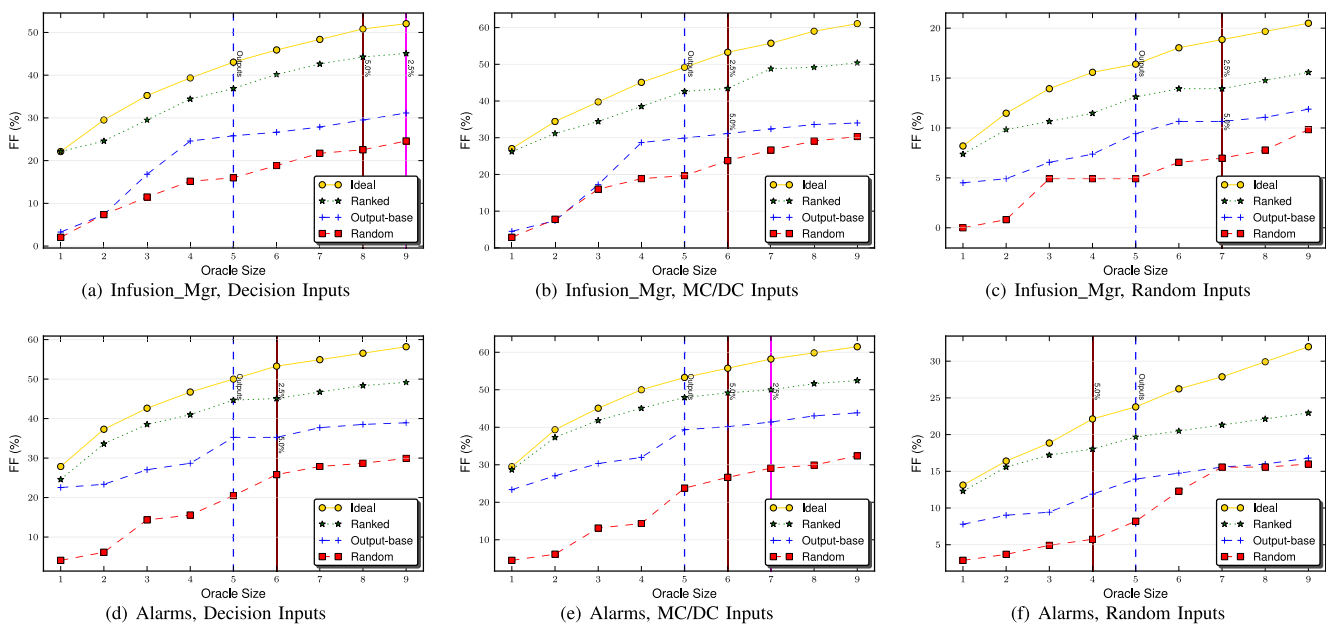
(a) Latctl_Batch, Decision Inputs (b) Latctl_Batch, MC/DC Inputs (c) Latctl_Batch, Random Inputs

(d) Latctl_Batch, UFC Inputs (e) Vertmax_Batch, Decision Inputs (f) Vertmax_Batch, MC/DC Inputs

(g) Vertmax_Batch, Random Inputs (h) Vertmax_Batch, UFC Inputs

Fig. 3. Median effectiveness of various approaches to oracle data selection for *Latctl_Batch* and *Vertmax_Batch*. Yellow circles = Ideal; Green * = Ranked; Blue + = Output-base; Red Squares = Random.

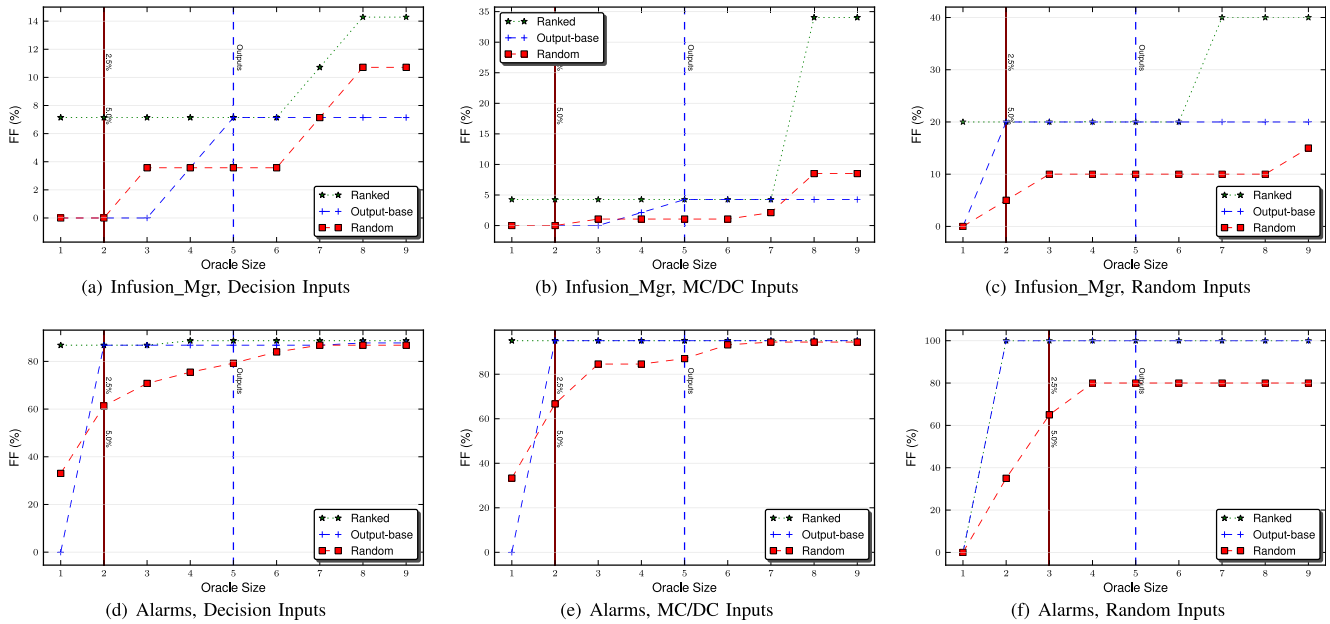
distribution of our data, we use the two-tailed bootstrap paired permutation test (a non-parametric test with no distribution assumptions [38]), with median as the test statistic. Per our experimental design, each evaluation set has a paired training set, and each training set has paired baseline rankings (output-base and random). Thus, for each combination of case example and coverage criterion, we can pair each test suite T + training set ranking with T + random or



(a) Infusion_Mgr, Decision Inputs (b) Infusion_Mgr, MC/DC Inputs (c) Infusion_Mgr, Random Inputs

(d) Alarms, Decision Inputs (e) Alarms, MC/DC Inputs (f) Alarms, Random Inputs

Fig. 4. Median effectiveness of various approaches to oracle data selection for *Infusion_Mgr* and *Alarms*, evaluated on mutants. Yellow circles = Ideal; Green * = Ranked; Blue + = Output-base; Red squares = Random.



(a) Infusion_Mgr, Decision Inputs (b) Infusion_Mgr, MC/DC Inputs (c) Infusion_Mgr, Random Inputs

(d) Alarms, Decision Inputs (e) Alarms, MC/DC Inputs (f) Alarms, Random Inputs

Fig. 5. Median effectiveness of various approaches to oracle data selection for *infusion_mgr* and *Alarms*, evaluated on real faults. Yellow circles = Ideal; Green * = Ranked; Blue + = Output-base; Red squares = Random.

output-base ranking ($H0_1, H0_2$), each test suite T + idealized ranking with T + training set ranking (for $H0_3$), and finally each test suite T + training set with T + training set subset ranking ($H0_4 - H0_7$). We then apply and evaluate our null hypotheses for each case example, coverage criteria, and oracle size with $\alpha = 0.05$.[5] We discuss the results of our statistical tests below in the context of the questions they address.

5.2 Evaluation of Practical Effectiveness (Q1)

When designing a method of supporting oracle creation, the obvious question to ask is, *"Is this better than current best practice?"* In Tables 3 and 4, we list the median relative improvement in fault finding effectiveness on seeded faults using our proposed oracle data creation approach versus the output-base ranking—the standard practice of checking the values of the output variables, with additional random variables added to ensure the same number of variables across all oracle types. Relative improvements not statistically significant at $\alpha = 0.05$ level are marked with a $*$. Almost all of the oracles generated outperform the random approach with statistical significance, often by a wide margin.[6]

From these two tables, we can see that for both structural coverage criteria, nearly every oracle generated for five of six systems (*Latctl_Batch*, *Vertmax_Batch*, *DWM_2*, *Infusion_Mgr*, and *Alarms*) outperforms the output-base approaches with statistical significance. A common pattern can be seen: for oracles smaller than the output-only oracle, our approach tends to perform well compared to output-base, with improvements of up to 1,435 percent. This reflects the strength of prioritizing variables: we generally select more effective variables for inclusion earlier than the output-base approach. Even in cases where output variables are the most effective, our approach is able to order them in terms of effectiveness. As the test oracle size grows closer in size to the output-only oracle, the relative improvement decreases, but our approach often still outperforms the output-only oracle, up to 45.2 percent. Finally, as the test oracle grows in size beyond the output-only oracle incorporating (by necessity) internal variables, our relative improvement when using our approach again grows, with improvements of 3.64-52.33 percent for the larger oracles.

A similar trend can be seen when random tests are used to train the oracle, although the acutal improvements are more subdued. For oracles smaller than the output-base approach, improvements of up to 85.71 percent can be seen. As middle sizes are approached, our approach performs between an identical performance (plus or minus roughly 2 percent—our method does demonstrate a higher level of variance, as it depends on both a set of training mutants and a particular test suite) and improvements of up to 35.14 percent. For the largest oracle sizes, modest improvements can be seen—up to 38.46 percent for the *Infusion_Mgr* system.

In particular, the *Infusion_Mgr* and *Alarms* systems, we see a large improvement in fault-finding effectiveness from using our oracle creation method. This is explained by the structure of these systems. Both systems work to determine the behavior of an infusion pump by checking sensor readings against a series of complex Boolean conditions. The state spaces of these models are both *deep* and *narrow*, meaning that to reach large portions of the state space, a specific sequence of input values that meet certain combinations of those conditions must occur. Thus, output-based oracles may not detect faults due to *masking*—some expressions in the systems can easily be prevented from influencing the outputs. If masking prevents the effect of a fault from

5. Note that we do not generalize across case examples or coverage criteria as the appropriate statistical assumption—random selection from the population of case examples and coverage criteria—is not met. Furthermore, we do not generalize across oracle sizes as it is possible our approach is statistically significant for some sizes, but not others.

6. Exceptions being output-base vs random on *Infusion_Mgr* with MC/DC and decision inputs at size 2 and *Alarms* with random inputs at size 7.

TABLE 3
Median Relative Improvement Using Mutation-Based Selection over Output-Base Selection for Structural Coverage-Satisfying Tests

	Decision					
Oracle Size	DWM_1	DWM_2	Vertmax_Batch	Latctl_Batch	Infusion_Mgr	Alarms
1	66.67%*	109.17%	34.48%	4.89%	1307.14%	11.62%
2	26.79%*	14.09%	29.76%	24.10%	222.22%	41.38%
3	12.50%*	11.11%	31.82%	28.73%	88.49%	44.28%
4	0.00%*	7.50%	35.90%	27.47%	41.94%	42.35%
5	2.94%*	9.45%	43.39%	31.91%	44.99%	27.84%
6	0.00%*	8.21%	45.79%	35.98%	48.49%	28.06%
7	0.00%*	6.55%	46.51%	38.84%	52.10%	20.78%
8	0.00%*	6.49%	49.44%	37.73%	45.95%	19.19%
9	8.01%*	7.96%	46.94%	37.15%	44.59%	21.90%
10	10.82%*	9.03%	52.33%	36.60%	45.17%	23.96%
11	15.83%	9.42%				
12	21.24%	9.46%				
13	23.13%	10.60%				
14	23.53%	10.53%				
15		12.19%				
16		13.46%				
17		13.46%				
18		14.08%				
	MC/DC					
Oracle Size	DWM_1	DWM_2	Vertmax_Batch	Latctl_Batch	Infusion_Mgr	Alarms
1	0.00%*	168.85%	37.50%	0.00%	1435.00%	21.11%
2	9.72%*	7.95%	37.33%	3.12%	305.00%	38.27%
3	-12.92%	7.59%	43.42%	3.16%	125.83%	42.11%
4	-13.96%	7.70%	44.00%	3.96%	35.29%	40.73%
5	-20.00%	7.23%	46.05%	3.96%	45.20%	21.78%
6	-19.23%	5.88%	44.45%	3.98%	46.34%	24.00%
7	-15.69%	4.40%	43.30%	4.40%	48.78%	20.79%
8	-12.00%	2.15%	43.30%	3.96%	51.22%	20.19%
9	-4.81%*	1.02%*	38.27%	3.96%	43.90%	21.83%
10	0.00%*	1.02%*	37.13%	3.64%	38.54%	23.47%
11	3.85%	1.01%*				
12	10.62%	2.00%				
13	12.02%	2.93%				
14	16.95%	3.02%				
15		3.88%				
16		4.77%				
17		4.17%				
18		4.17%				

TABLE 4
Median Relative Improvement Using Mutation-Based Selection over Output-Base Selection for Randomly-Generated and UFC-Satisfying Tests

	Random					
Oracle Size	DWM_1	DWM_2	Vertmax_Batch	Latctl_Batch	Infusion_Mgr	Alarms
1	24.27%	10.07%	15.50%	0.00%*	60.0%	70.33%
2	15.52%	7.95%	13.07%	0.84%	85.71%	80.00%
3	8.79%	8.29%	15.09%	0.85%*	83.33%	80.91%
4	8.11%	8.71%	16.56%	0.85%*	55.55%	53.14%
5	6.73%	9.54%	17.54%	0.91%*	33.33%	35.14%
6	1.46%*	7.48%	17.20%	0.87%*	33.33%	29.51%
7	-1.11%	4.61%	17.03%	1.64%	37.09%	28.57%
8	0.00%*	0.88%*	16.04%	0.86%	38.46%	27.33%
9	0.00%*	-1.72%	15.47%	0.86%*	28.57%	31.58%
10	1.03%*	-0.93%	15.47%	0.84%*	28.57%	32.58%
11	2.02%	-0.84%				
12	2.39%	0.00%*				
13	4.10%	0.00%*				
14	4.19%	0.85%*				
15		0.88%*				
16		0.88%*				
17		0.95%*				
18		0.97%*				
	UFC					
Oracle Size	DWM_1	DWM_2	Vertmax_Batch	Latctl_Batch		
1	0.00%*	8.59%	-0.81%	0.00%*		
2	0.00%*	6.98%	-0.81%	0.00%*		
3	0.00%*	9.53%	-0.81%	0.00%*		
4	-10.56%	9.52%	-0.81%	0.00%*		
5	-11.76%	9.55%	0.00%*	0.83%		
6	-14.29%	6.42%	0.00%*	0.83%		
7	-17.32%	2.73%	0.00%*	0.83%		
8	-13.96%	-0.87%	0.00%*	0.83%		
9	-12.02%	-3.36%	0.00%*	0.83%		
10	-11.11%	-3.36%	0.00%*	0.82%		
11	-10.91%	-2.59%				
12	-6.48%*	-2.56%				
13	-6.56%	-2.52%				
14	-5.96%*	-2.46%				
15		-1.65%				
16		-0.86%				
17		-0.83%				
18		-0.82%				

reaching an output variable, then the effectiveness of an output-based oracle will naturally decrease. Our oracle creation approach can select the important output variables and certain internal bottlenecks to observe.

For the structural criteria and random test suites, one key exception can be seen when examining the *DWM_1* system. For this case example, mutation-based oracles tend to be roughly equivalent in effectiveness to output-base oracles (we generally cannot reject $H0_1$ at $\alpha = 0.05$), and at times (particularly for oracles generated using MC/DC satisfying test suites) produce results that are up to 20 percent worse. It is only for small or large oracles (approximately +-6 variables from the output-only oracle) that our approach does well, with up to a 66.67 percent improvement at smallest sizes and 4.19-23.53 percent improvement at the largest recorded oracle size. Examining the composition of these oracles reveals why: the ranking generated using our approach for this case example begins mostly with output variables, and thus oracles generated using our approach are very similar to those generated using the output-base approach. The performance gain at small sizes suggest that certain output variables are far more important than others, but what is crucially important at all levels up to the total number of outputs is *to choose output variables*.

As noted previously, in a small number of instances, our approach in fact does worse than the output-base approach. The issue appears to be that the greedy set-coverage algorithm is overfitting to the training data. If the trace data indicates that there is a highly effective internal state variable, the algorithm will prevent a computationally-related output variable from being selected later in the process. However, it is possible that overall, faults occur more prevalently in that output variable, and that the internal variable is only more fault-prone for the selected set of training data. Given a more optimal set cover algorithm or additional overfitting avoidance improvements, this issue would likely be circumvented. However, for larger oracle sizes, this issue is generally corrected, with statistically significant improvements again being demonstrated.

Very different results are observed when our technique is applied with test inputs generated to satisfy the UFC requirements coverage criterion. When these test suites are used, our technique is, on occasion, modestly successful (up to 9.55 percent improvement), but more often demonstrates either no improvement (*Vertmax_Batch* and *Latctl_Batch*) or worse results (for oracle sizes surrounding the number of output variables on the *DWM_1* system). We will elaborate on reasons for this difference shortly.

In Table 5, we list the median improvement in fault finding effectiveness on the set of real faults for the *Infusion_Mgr* and *Alarms* systems. On the *Infusion_Mgr* system, we observe the same trends that we saw when evaluating

TABLE 5
Median Relative Improvement Using Mutation-Based Selection over Output-Base Selection over Real Faults

Decision		
Oracle Size	**Infusion_Mgr**	**Alarms**
1	Inf (0.00% → 7.14%)	Inf (0.00% → 86.78%)
2-3	Inf (0.00% → 7.14%)	0.00%*
4	100%	2.17%
5-6	0.00%*	2.17%
7	0.00%	2.17%
8	9.09%	0.00%*
9-10	99.99%	0.00%*
MC/DC		
Oracle Size	**Infusion_Mgr**	**Alarms**
1	Inf (0.00% → 4.25%)	Inf (0.00% → 86.78%)
2-3	Inf (0.00% → 4.25%)	0.00%*
4	700.0%	0.00%*
5-7	0.00%*	0.00%*
8	700.0%	0.00%*
9-10	349.99%	0.00%*
Random		
Oracle Size	**Infusion_Mgr**	**Alarms**
1	Inf (0.00% → 20.00%)	0.00%*
2-6	0.00%*	0.00%*
7-10	100.00%	0.00%*

"Inf" = output-base oracle failed no *tests—we note the median percentage of tests failed by the generated oracle.*

TABLE 6
Median Relative Improvement in Idealized Performance over Standard Performance of Mutation-Based Selection for Structural Coverage-Satisfying Tests

	Decision					
Oracle Size	**DWM_1**	**DWM_2**	**Vertmax_Batch**	**Latctl_Batch**	**Infusion_Mgr**	**Alarms**
1	100.00%	3.54%	5.00%	0.00%*	0.00%*	13.79%
2	133.33%	8.70%	5.28%	7.55%	13.79%	3.35%
3	133.33%	11.24%	13.73%	12.50%	21.62%	9.00%
4	143.65%	14.29%	16.03%	16.67%	12.50%	11.59%
5	124.04%	15.76%	15.25%	15.38%	16.67%	12.61%
6	123.21%	17.24%	17.24%	10.71%	14.58%	15.69%
7	116.67%	18.64%	20.34%	11.27%	14.00%	18.80%
8	112.50%	17.24%	17.46%	13.89%	16.67%	21.67%
9	109.82%	17.95%	19.40%	14.86%	18.18%	21.55%
10	106.46%	18.87%	21.54%	14.81%	14.29%	21.43%
11	105.41%	17.65%				
12	109.26%	19.05%				
13	109.31%	19.18%				
14	115.00%	20.00%				
15		19.72%				
16		17.72%				
17		18.18%				
18		17.33%				
	MC/DC					
Oracle Size	**DWM_1**	**DWM_2**	**Vertmax_Batch**	**Latctl_Batch**	**Infusion_Mgr**	**Alarms**
1	66.67%	5.48%	2.08%	0.97%	0.00%*	3.23%
2	82.86%	6.41%	2.80%	0.99%	10.53%	7.14%
3	91.29%	6.69%	4.55%	2.04%*	15.91%	11.11%
4	85.71%	7.06%	3.67%	3.77%	16.00%	4.51%
5	100.00%	7.74%	4.07%	4.85%	12.96%	8.27%
6	97.37%	8.84%	5.41%	5.66%	14.29%	11.57%
7	94.87%	8.74%	6.14%	5.63%	15.52%	13.59%
8	100.00%	10.05%	6.19%	6.60%	16.13%	14.71%
9	95.65%	8.91%	6.55%	7.55%	20.00%	15.50%
10	92.15%	10.21%	7.02%	8.37%	21.67%	16.91%
11	88.89%	11.00%				
12	87.10%	11.00%				
13	87.10%	11.00%				
14	90.31%	11.59%				
15		11.43%				
16		10.90%				
17		11.21%				
18		11.21%				

against seeded mutations. At small oracle sizes, we see an improvement in the percentage of the test suite that fails from 0 percent of the tests with an output-base oracle up to 20 percent with a generated oracle. As we approach the number of output variables, the two approaches converge. Finally, at large oracle sizes, our approach improves on the output-base oracle by up to 349 percent.

Examining the plots for *Infusion_Mgr* in Fig. 5 offers insight into the importance of the outputs in finding the real faults embedded into the system—and shows why focusing on the output variables is not always a wise idea. Only two of the five output variables are relevant for finding any of the known faults. This can be seen in the plots, as the median percentage of tests that fail rises exactly twice for the output-base oracle for sizes 1-5. As a result, for the structure-based test inputs, choosing oracle *completely at random* sometimes results in a more effective oracle. If some care is taken in selecting an oracle, time may not be wasted in specifying the expected behavior of outputs that rarely, if ever, are useful for finding faults. Our approach is able to select the important output variables early and automatically suggest additional bottlenecks in the state space.

The results on the version of *Alarms* with real faults are more subdued. At size one, our approach typically outperforms the output-base approach by a large margain—from no failing tests to failure results in a median of 86.78 percent of the tests. However, from that point, the largest improvement from our approach is by an additional 2.17 percent. The reason for this can be clearly seen in Fig. 5—no matter what variables are chosen, by an oracle of size 10, 90-100 percent of the tests will have failed. This can be explained by examining the faults 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—explains the observed results. The majority of the system outputs deal explicitly with signaling alarms in the presence of particular input conditions. If a test triggers that particular fault, then the results of that fault will be obvious at the output level. Our approach is able to select the most important output first, while an unordered approach may not. However, unlike on the *Infusion_Mgr* system, additional gains from observing internal variables are rare.

The major observations made when evaluating on seeded mutations are confirmed by the evaluation against the real faults, indicating the applicability of our approach in practice. However, the performance benefits were more subdued. A likely reason for this effect is due to the potentially substantial differences between the faulty and corrected models (or faulty system and any source of expected values)—in the case of *Alarms*, fixing the model involved adding 24 new transitions and changing the guard conditions on time-related transitions, meaning that any expected value oracle is *very likely* to detect a fault. Examining models that are less substantially different would likely yield different results—for instance, the fixes to the *Infusion_Mgr* model only required two new transitions and changes to a small number of guard conditions.

In addition, the changes imposed by the mutation operators may not be similar to the real mistakes made by the system developers. In particular, mutations do not replicate errors of omission—leaving out functionality is very different from making a mistake in the implemented functionality. While examining the traces of mutated systems may help us discover important internal variables for identifying faults caused by incorrect implementation, those bottlenecks may not assist in observing errors that stem from missing execution paths (as several of the real faults, listed in Table 2, are). This indicates that—while the core tenants of our mutation-based approach seem correct—there is room for improvement in the sources of the traces used to generate an oracle with our approach. For instance, in addition to seeded faults, it may be possible to train oracles using past revisions of a system (as long as there is enough overlap in the internal structure of the system). The combination of seeded mutations and corrected faults may yield even more effective oracles.

While our approach is of little use when paired with UFC-satisfying test inputs, it does seem clear that our approach can be effective in practice when paired with either randomly-generated test inputs or test suites generated to satisfy a structural coverage criterion. For those test suites, we can consistently generate oracle data sets that are effective over *different* faults, generally outperforming existing ranking approaches. Our approach is able to highlight the most important variables, allowing developers to craft smaller, more effective oracle data sets.

5.3 Potential Effectiveness of Oracle Selection (Q2)

In Tables 6 and 7, we list the median improvement in fault finding effectiveness between the idealized performance (oracle data set built and evaluated on the same mutants) and the real-world performance of our approach. The results which are not statistically significant at $\alpha = 0.05$ are marked with a $*$.

As noted, there is limited empirical work on test oracle effectiveness. Consequently, it is difficult to determine what constitutes effective oracle data selection—clearly performing well relative to a baseline approach indicates our approach is effective, but it is hard to argue the approach is effective in the absolute sense. We therefore posed *Q2*: what is the *maximum* potential effectiveness of a mutation-based approach? To answer this question, we applied our approach to the same mutants used to evaluate the oracles in *Q1* (as opposed to generating oracles from a disjoint training set). This represents an idealized testing scenario in which we already know what faults we are attempting to find; thus, this scenario is used to estimate the maximum potential of our approach.

The results can be seen in Figs. 2, 3, 4 and Tables 6 and 7. We can observe from these results that while the *potential* performance of a mutation-based oracle is (naturally) almost always higher than the actual performance of our method, the gap between the actual implementation of our approach and the ideal scenario is often quite small. In some cases, such as when using UFC-satisfying test inputs with the *Vertmax_Batch* system or at small oracle sizes on *Infusion_Mgr*, the difference in results is statistically insignificant. Thus we can conclude that while there is clearly room for improvement in oracle data selection methods, our approach appears to often be quite effective in terms of *absolute* performance.

TABLE 7
Median Relative Improvement in Idealized Performance over Standard Performance of Mutation-Based Selection for Random and UFC-Satisfying Tests

	Random					
Oracle Size	**DWM_1**	**DWM_2**	**Vertmax_Batch**	**Latctl_Batch**	**Infusion_Mgr**	**Alarms**
1	8.89%	1.12%	2.06%	0.82%	12.50%	7.14%
2	20.26%	3.23%	1.96%	1.65%	10.00%	11.11%
3	12.61%	3.86%	3.33%	1.67%	26.97%	16.23%
4	9.74%	2.87%	3.90%	2.48%	28.57%	22.73%
5	9.67%	0.98%*	4.68%	2.48%	30.77%	25.00%
6	5.55%	2.78%	5.10%	2.48%	31.25%	31.91%
7	4.18%	4.41%	5.91%	2.48%	29.41%	38.39%
8	5.82%	5.31%	6.12%	2.48%	27.78%	41.38%
9	7.96%	6.28%	6.90%	2.48%	31.41%	42.86%
10	8.75%	7.11%	7.33%	2.48%	30.00%	36.36%
11	10.64%	6.96%				
12	12.56%	6.60%				
13	12.25%	6.06%				
14	12.37%	5.98%				
15		5.19%				
16		5.08%				
17		5.08%				
18		4.27%				

	UFC			
Oracle Size	**DWM_1**	**DWM_2**	**Vertmax_Batch**	**Latctl_Batch**
1	60.00%	3.37%	0.85%	0.83%*
2	72.08%	4.17%	1.64%	1.67%
3	76.92%	3.96%	1.65%	2.50%*
4	81.25%	3.88%	1.65%	3.33%*
5	79.47%	0.85%	1.64%	2.48%
6	82.29%	2.54%	1.64%	2.48%
7	89.18%	3.45%	0.82%	2.48%*
8	88.24%	4.31%	0.82%	2.48%*
9	94.12%	4.46%	0.82%	2.48%*
10	100.00%	5.22%	0.82%	2.48%*
11	100.00%	6.03%		
12	100.00%	6.03%		
13	103.85%	5.24%		
14	103.94%	5.19%		
15		5.13%		
16		4.68%		
17		4.22%		
19		3.36%		

5.4 Impact of Coverage Criteria (Q3)

Our technique relies on pre-existing suites of test inputs. It follows that we would like to investigate the impact of varying the *type* of test suite on the effectiveness of oracle selection.

Intuitively, when examining the results of the oracle data sets generated using the two structural coverage criteria, using test suites satisfying the stronger criterion (MC/DC) should have a lower potential for improving the testing process via oracle selection, as the test inputs should do a better job of exercising the code. However, as shown in Figs. 2, 3, 4, and 5 for each case example, the gap between the output-base and generated oracles for both decision and MC/DC test suites seems to be roughly the same. For example, for the *DWM_2* system, we can see that despite overall higher levels of fault finding when using the MC/DC test suites, the general relationships between the output-base baseline approach, our approach, and the idealized approach remain similar. We see a rapid increase in effectiveness for small oracles, followed by a decrease in the improvement of our approach versus the output-base baseline as we approach oracles of size 10 (corresponding to an output-only oracle), followed by a gradual increase in the improvement. In

TABLE 8
Median Number of Steps per Test

	Decision	MC/DC	UFC	Random
DWM_1	2.0	2.0	4.0	6.0
DWM_2	1.5	2.0	7.0	6.0
Vertmax_Batch	1.0	2.0	5.0	6.0
Latctl_Batch	1.0	2.0	5.0	6.0
Infusion_Mgr	3.0	3.0		6.0
Infusion_Mgr (real faults)	3.0	3.5		6.0
Alarms	2.0	2.0		6.0
Alarms (real faults)	2.0	3.0		6.0

some cases, relative improvements are higher for decision coverage (*Latctl_Batch*) and in others they are higher for MC/DC (*Vertmax_Batch*). Relative improvements even vary between oracle sizes—as can be seen on *Infusion_Mgr* and *Alarms*, where MC/DC-satisfying tests tend to lead to larger improvements than decision-satisfying tests at small oracle sizes, but decision-satisfying tests lead to larger improvements at larger sizes.

The results in Tables 3, 4, 5 and Figs. 2, 3, 4, 5—particularly those for DWM_2—reveal three key observations about the oracles generated using random test inputs. First, the oracles largely exhibit the same trends as the oracles generated for the structure-based test suites—a sharp rise at small sizes, performance comparable to the output-base oracle around middle sizes, and further gains at the end. Second—however, the improvements from using our method over an output-base oracle are more modest. Finally, On the Rockwell Collins systems, *all oracle selection methods* achieve higher levels of fault finding over random tests than they do when applied to test inputs generated to satisfy structural coverage criteria.

Observations 2 and 3 can be partially explained by examining the *individual tests*. As seen in Table 8, the median length of each randomly-generated test—as measured in number of recorded test steps—is significantly longer than the length of the tests in the coverage satisfying tests. In fact, the random tests tend to be three to six times longer than their structure-based counterparts. This addresses the third observation in particular, because longer tests allow more time for corrupted internal states to propagate to the output variables. Tests generated to satisfy structural coverage criteria tend to focus on exercising particular syntactic elements of the source code, and thus, tend to be short—just long enough to exercise that particular obligation. As a result, tests generated to satisfy structural coverage obligations may be very effective at locating faults, but may not be long enough to propagate the fault to the output level.

The third observation is not true for *Infusion_Mgr* and *Alarms*—coverage-satisfying tests outperform randomly-generated tests. This is due to the complex structure of these systems. Deep exploration of their state spaces requires particular combinations of Boolean conditions, and random tests are less likely to hit some of these combinations. However, the second observation—that the gains are more modest—is still true. Even if the state space is less thoroughly explored, the longer test lengths still allow more time for the effects of the triggered faults to propagate to the output variables.

As shown in Figs. 2 and 3—despite different fault finding numbers—the plots for test oracles that make use of test suites generated to satisfy the UFC requirements coverage criterion look very similar to the plots for the oracles that use other test input types. However, as with random tests, (1) *all oracle selection methods* tend to do very well, and (2), the improvement from using our method tends to be small.

There is some wisdom in the common approach of monitoring the output variables, as some faults will always be caught by monitoring them. Ultimately, the recommendation of whether or not to make use of our oracle data selection method is biased by the choice of test input—more specifically, *the probability of a fault propagating to the output variables*. As UFC coverage obligations tend to take the form of explicit relationships between inputs and outputs, it is unsurprising that our approach often fails to outperform the output-base oracle. While faults can still be masked in UFC tests, they are far more likely to propagate to an output variable than when running tests providing structural coverage. Therefore, it is unlikely that much improvement will be seen when working with test inputs that satisfy a requirements coverage criterion. On the other hand, when working with tests that explicitly exercise the internal structure of the software (as is mandated for certification for the avionics domain), masking of faults is common and our approach can significantly improve the effectiveness of the testing process. Regardless of the differing level of fault finding, the consistent trends across case examples exhibited for generated oracles indicates that, perhaps more that the test inputs used, characteristics of the system under test are the primary determinant of the effectiveness of our approach.

5.5 Impact of Number of Training Mutants (Q4)

For our initial experiment, we chose to use 125 mutants for training the oracle data. This number was chosen because earlier studies yielded evidence that results tend to stabilize after 100 mutants are used. That said, it may be possible to train sufficiently powerful oracles with fewer mutants—in fact, this would be an ideal situation, as examining fewer mutants will save time in the oracle generation process.

In Fig. 6, we have summarized the results for statistical tests $H4 - H7$. We have omitted the full set of values, instead opting to determine the smallest training set size where we *cannot* reject the null hypothesis—the smallest training set that produces test oracles of effectiveness not statistically different from the full training set. The selected training set for each combination of case example, coverage criteria, and oracle size is plotted in the figure. (Note small vertical and horizontal offsets are present for readability. Points above $X\%$ *and below* $Y\%$ should be interpreted as training set size of exactly $Y\%$. Point above oracle size X and Y should be interpreted as oracle sizes of Y.) For example, for the *DWM_1* system, training set sizes of 50 percent of those typically used provided statistically equivalent performance for oracle sizes between one to seven, while the full training set is required for oracles of sizes eight and larger.

Each point represents the plateau at which we cease to observe significant improvements from adding additional mutants to the training process. Based on the figure, we conclude that, for the most part, our initial estimate of 125 mutants was reasonable. For several of the combinations of

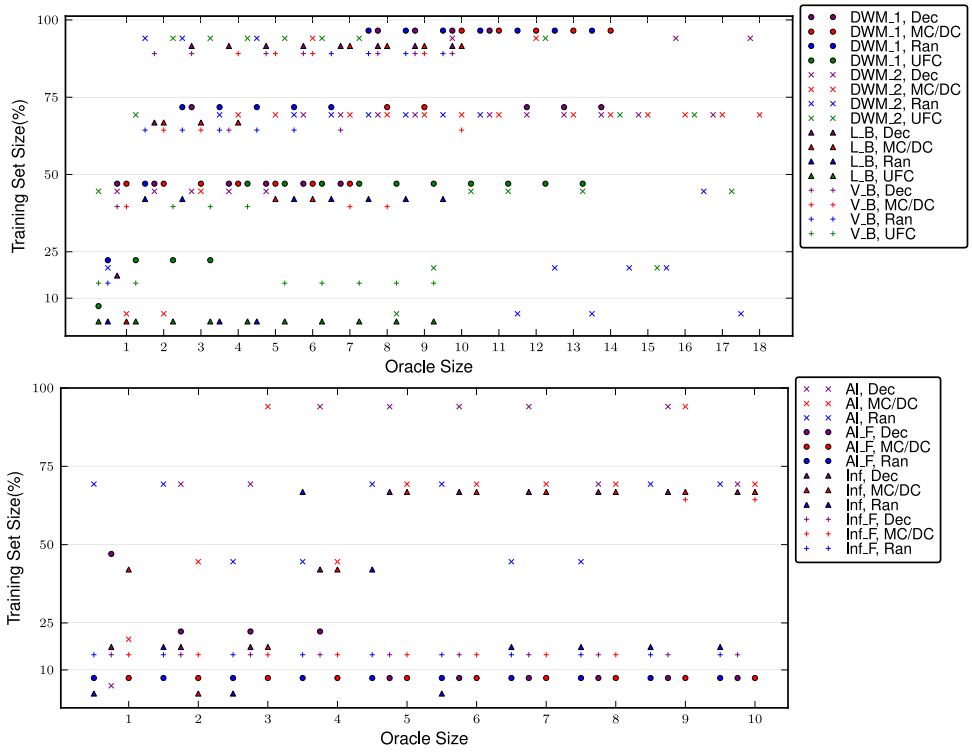


Fig. 6. Smallest effective training sets (V_B = *Vertmax_Batch*, L_B = *Latctl_Batch*, Al = *Alarms*, Inf = *Infusion_Mgr*, F = version with real faults).

case example and test input, we fail to observe this plateau for several oracle sizes—we can say with statistical certainty that oracle effectiveness will diminish with fewer training mutants. These points typically correspond to areas where the ideal approach has a moderate (7-15 percent) and statistically significant difference over our approach, for example the *DWM_1* system with random, MC/DC, and decision coverage for oracle sizes between 7-13. For these systems, our approach constructs test oracles that are effective, but can clearly be improved.

Often, however, smaller training set sizes are sufficient. We frequently observe a plateau in fault finding results after 75 or 50 percent of the training mutants are used to train the oracle (though *some* improvement is often seen when using the full training set). Thus, for the majority of combinations of test input type and case example, we can conclude that 62 to 100 mutants are needed to produce effective test oracles.

When using real faults as an evaluation criterion, dimishing returns are seen at smaller training set sizes—often at 25 percent for *Infusion_Mgr* and 10 percent for *Alarms*. As shown by our earlier results, ranking variables using information learned from seeded faults leads to improvements in effectiveness. However, when the mutation operators are dissimilar to the real mistakes made by the developers, we quickly exhaust *what* we can learn from the seeded faults. This again indicates that—while our mutation-based approach yields benefits—there is a need to expand the sources of the traces used to generate the oracle data set.

Although results vary between case example, it seems that one of the largest determinations of how many mutants are needed is the choice of test input type. Test suites generated to satisfy a structural coverage metric tend to require a large number of mutants. In fact, when using suites designed to satisfy decision coverage, it may be possible to improve results further by adding *more mutants* than we did. Decision and MC/DC coverage obligations exercise pieces of the internal structure of a system, and thus, it is hard to predict exactly where a corrupt internal state will propagate to. The more mutants we examine, the more evidence there is for which specific internal variables we can observe to catch faults.

In contrast, UFC test obligations are expressed in terms of the relationship between inputs and outputs to the system under test, and thus, the tests are very likely to propagate faults to the output variables. As a side effect, our oracle selection method reaches a plateau very quickly. For two case examples, *Vertmax_Batch* and *Latctl_Batch*, as few as 10 percent of the training mutants used are necessary to reach stable conclusions. In fact, using more training mutants with test suites that satisfy UFC coverage can occasionally be slightly detrimental, as overfitting at larger training set sizes leads to worse median fault finding results at certain oracle sizes.

5.6 Oracle Composition (Q5)

In addition to performance, we are also interested in the *composition* of the generated data sets. Are they similar in structure to the current industrial practice of favoring the output variables, or, are they more heavily constructed from the many internal variables of the examined systems?

The average composition of the generated oracle data sets are listed in Table 9. For each system and test type, we list the size of the oracle data, the average number of chosen output variables, the total number of output variables, the average number of chosen internal variables, and the total number of internal variables. Recall that the oracle size was chosen to be the larger of twice the number of output variables or 10 variables. Therefore, while a generated oracle

TABLE 9
Average Composition of Generated Oracles

	Test Type	Oracle Size	# Chosen Outputs	Total Outputs	# Chosen Internals	Total Internals
DWM_1	Decision	14	0.37	7	13.63	569
	MC/DC		0.43		13.57	
	UFC		0.84		13.16	
	Random		2.28		11.72	
DWM_2	Decision	18	3.10	9	14.90	115
	MC/DC		4.33		13.67	
	UFC		7.84		10.16	
	Random		8.30		9.70	
Vertmax _Batch	Decision	10	1.50	2	8.50	415
	MC/DC		1.51		8.49	
	UFC		2.00		8.00	
	Random		0.96		9.04	
Latctl _Batch	Decision	10	0.00	1	10.00	128
	MC/DC		0.00		10.00	
	UFC		1.00		9.00	
	Random		0.97		9.03	
Infusion _Mgr	Decision	10	1.88	5	8.12	107
	MC/DC		1.73		8.27	
	Random		2.72		7.28	
Infusion _Mgr (RF)	Decision	10	2.40	5	7.60	86
	MC/DC		2.29		7.71	
	Random		3.34		6.66	
Alarms	Decision	10	2.02	5	7.98	182
	MC/DC		2.27		7.73	
	Random		1.03		8.97	
Alarms (RF)	Decision	10	1.71	5	8.29	155
	MC/DC		1.54		8.46	
	Random		0.88		9.12	

RF = real faults.

data set could be entirely composed of internal variables, none of the generated data sets will be composed entirely of output variables.

As with the performance of the generated oracle data, the composition seems to be largely determined by the type of test suite used to generate the data. The oracles created from structure-based test suites saw a large efficacy improvement from the oracle generation process because faults did not tend to propagate to the output variables in these tests. It follows, that the generated oracle data sets favor internal variables and largely ignore the output variables—for every system except *Vertmax_Batch*, fewer than half of the output variables are used in the oracles generated from structure-based tests. The UFC tests, on the other hand, are written specifically to propagate certain behaviors to the output variables, and the oracle data generated from the UFC tests tends to make use of more of those output variables. The composition of the oracle data sets generated from random tests varies depending on the case example—for the *Vertmax_Batch* system, oracle data generated from random tests only uses one of the two output variables. Similarly, for the *Alarms* system, the oracles generated from random tests made use of fewer output variables than those generated from structure-based tests. However, for the other systems, the oracle data sets generated from the random tests did more commonly choose output variables.

If the majority of the selected variables are internal variables, the effort cost of producing expected values for those variables must be considered. Not all variables can have their values specified with equal difficulty. Two key factors must be considered—*how often the value of the variable is checked* and *how the variable is used within the system*.

When and how often correctness is checked will help determine how effective the test oracle will be at catching faults. It is common to check behaviors after particular events or at regular points in time—for example, following the completion of a discrete computational cycle, as is the case with the systems used in our evaluation. A reasonable hypothesis might be that the more frequently values are checked, the easier it will be to spot problems. However, if expected values must be specified manually, there will be an increase in the required effort to produce the expected values for these result checks. Balancing the effort-to-volume ratio is important.

How a variable is used within the structure of the system may make specifying the value of those variables quite complicated. For example, if a variable is assigned a value in a triple-nested loop, unrolling the loops and selecting the value may be difficult. Similarly, an assignment requiring a complicated calculation might also require specifying the values of other dependent variables.

Therefore, even if our approach suggests a particular internal variable, testers may ignore the advice if specifying values for that variable is too difficult. In such situations, we recommend two courses of action—either remove difficult internal variables from consideration or weight those variables by the difficulty of specification.

Our approach starts with a candidate set of oracle variables and prunes it down into a recommended subset. While we have used all of the internal variables in our candidate set, that is not required. Testers could simply remove certain variables from consideration before generating an oracle. This is the easiest solution to the problem of specifying expected values for more difficult internal variables.

However, that solution does carry a risk of causing the tester to miss out on valuable oracle information. Variables that are difficult to specify expected values for might be worth considering if they also correspond to valuable monitoring points in the system. Therefore, another option is that, before generating the oracle data set, the tester could go through the list of variables in the system and apply weights to some or all of them to represent the cost of producing expected output for that variable. Weights can easily be incorporated into the set-covering algorithm used to generate the oracle data—the weight can be factored against the expected fault-finding improvement from the use of that variable. Unless the improvement from using that variable is quite high, an expensive to test internal variable will not be chosen.

6 Threats to Validity

6.1 External Validity

Our study is limited to six synchronous reactive critical systems. Nevertheless, we believe these systems are representative of the avionics systems in which we are interested and our results are therefore generalizable to other systems in the domain.

We have used Lustre [28] as our implementation language rather than a more common language such as C or C++. However, systems written in Lustre are similar to traditional imperative code produced in embedded systems development. A simple syntactic transformation suffices to translate Lustre code into C code.

We have generated approximately 250 mutants for each case example, with 125 mutants used for training sets and up to 125 mutants used for evaluation. These values are chosen to yield a reasonable cost for the study. It is possible the

number of mutants is too low. Nevertheless, based on past experience, we have found results using less than 250 mutants to be representative [39].

6.2 Internal Validity

We have used the JKind model checker to generate test cases. This generation approach provides the shortest test cases that provide the desired coverage. It is possible that test cases produced through some other method would yield different oracle data sets.

6.3 Construct Validity

We measure the fault finding of oracles and test suites over seeded faults, rather than real faults encountered during development of the software, for four of the examined systems. Given that our approach to selecting oracle data is also based on the mutation testing, it is possible that using real faults would lead to different results. As mentioned earlier, Andrews et al. and Just et al. have shown that the use of seeded faults leads to conclusions similar to those obtained using real faults in similar fault finding experiments [19], [20]. For the systems with real faults, our general results held.

Yao et al. have found that certain mutation operators can result in more equivalent mutants than other operators, thus skewing the results of testing experiments [40]. While we did remove equivalent mutants from the evaluation set (about 3 percent of the mutants for each system), in practice, these did not disproportionately result from particular mutation operators. Therefore, we do not feel that our results could have been impacted by biasing in the mutation operators.

7 Conclusion

In this study, we have explored a mutation-based method for supporting oracle creation. Our approach automates the selection of oracle data, the set of variables monitored by the test oracle—a key component of expected value test oracles.

Experimental results indicate that our approach, when paired with test suites generated to satisfy structural coverage criteria or random tests, is successful with respect to alternative approaches for selecting oracle data, with improvements up to 1,435 percent over output-base oracle data selection and improvements of up to 50 percent relatively common. Even in cases where our approach is not more effective, it appears to be comparable to the common practice of monitoring the output variables. We have also found that our approach performs within an acceptable range from the expected maximum performance.

However, we also found that our approach was not effective when paired with test inputs generated to satisfy requirements-based metrics. These tests, expressed in terms of the relationships between inputs and outputs, are highly likely to propagate faults to the output variables, reducing the potential gains from selecting key internal states to monitor.

Thus, we recommend the use of our approach when test suites that exercise structures internal to the system under test are employed in the testing process (such test suites are required by standards in the avionics domain) [29].

Acknowledgments

This work has been supported by NASA Ames Cooperative Agreement NNA06CB21A, NSF grants CCF-0916583, CNS-0931931, and CNS-1035715, an US National Science Foundation (NSF) graduate fellowship, and the Fonds National de la Recherche, Luxembourg (FNR/P10/03). The authors would additionally like to thank Rockwell Collins Inc. for their support.

References

[1] M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. (2013). A comprehensive survey of trends in oracles for software testing," Univ. of Sheffield, Dept. of Comput. Sci., Tech. Rep. CS-13-01. [Online]. Available: http://philmcminn.staff.shef.ac.uk/publications/pdfs/2013-techrep.pdf

[2] L. Briand, M. DiPenta, and Y. Labiche, "Assessing and improving state-based class testing: A series of experiments," *IEEE Trans. Softw. Eng.*, vol. 30, no. 11, pp. 770–793, Nov. 2004.

[3] M. Staats, M. Whalen, and M. Heimdahl, "Better testing through oracle selection (nier track)," in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 892–895.

[4] M. Staats, M. Whalen, and M. Heimdahl, "Programs, testing, and oracles: The foundations of testing revisited," in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 391–400.

[5] Q. Xie and A. Memon, "Designing and comparing automated test oracles for gui-based software applications," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 1, p. 4, 2007.

[6] 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 *Proc. Int. Conf. Softw. Eng.*, 2012, pp. 870–880.

[7] D. J. Richardson, S. L. Aha, and T. O'Malley, "Specification-based test oracles for reactive systems," in *Proc. 14th Int. Conf. Softw. Eng.*, May 1992, pp. 105–118.

[8] A. Memon and Q. Xie, "Using transient/persistent errors to develop automated test oracles for event-driven software," in *Proc. 19th Int. Conf. Autom. Softw. Eng.*, Sept. 2004, pp. 186–195.

[9] J. Voas and K. Miller, "Putting assertions in their place," in *Proc. 5th Int. Symp. Softw. Rel. Eng.*, 1994, pp. 152–157.

[10] C. Pacheco and M. Ernst, "Eclat: Automatic generation and classification of test inputs," in *Proc. 19th Eur. Conf. Object-Oriented Programm.*, 2005, pp. 504–527.

[11] K. Taneja and T. Xie, "Diffgen: Automated regression unit-test generation," in *Proc. 23rd IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2008, pp. 407–410.

[12] R. Evans and A. Savoia, "Differential testing: A new approach to change detection," in *Proc. 6th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.: Companion Papers*, 2007, pp. 549–552.

[13] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," in *Proc. 19th Int. Symp. Softw. Testing Anal.*, 2010, pp. 147–158.

[14] G. Fraser and A. Zeller, "Generating parameterized unit tests," in *Proc. ACM Int. Symp. Softw. Testing Anal.*, 2011, pp. 147–158.

[15] S. Afshan, P. McMinn, and M. Stevenson, "Evolving readable string test inputs using a natural language model to reduce human oracle cost," in *Proc. Int. Conf. Softw. Testing, Verif. Validation*, 2013, pp. 352–361.

[16] M. Harman, K. Sung, K. Lakhotia, P. McMinn, and S. Yoo, "Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem," in *Proc. 3rd Int. Conf. Softw. Testing, Verif. Validation Workshops*, 2010, pp. 182–191.

[17] P. McMinn, M. Stevenson, and M. Harman, "Reducing qualitative human oracle costs associated with automatically generated test data," in *Proc. 1st Int. Workshop Softw. Test Output Validation*, 2010, pp. 1–4.

[18] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649–678, Sep. 2011.

[19] J. Andrews, L. Briand, Y. Labiche, and A. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Trans. Softw. Eng.*, vol. 32, no. 8, pp. 608–624, Aug. 2006.

[20] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proc. Symp. Found. Softw. Eng.*, 2014, pp. 654–665.
[21] M. Hiller, A. Jhumka, and N. Suri, "An approach for analysing the propagation of data errors in software," in *Proc. Int. Conf. Depend. Syst. Netw.*, Washington, DC, USA, 2001, pp. 161–172.
[22] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*. Cambridge, MA, USA: MIT Press, 2001.
[23] M. Garey and M. Johnson, *Computers and Intractability*. New York, NY, USA: Freeman, 1979.
[24] V. Chvatal, "A greedy heuristic for the set-covering problem," *Math. Oper. Res.*, vol. 4, no. 3, pp. 233–235, 1979.
[25] A. Murugesan, S. Rayadurgam, and M. Heimdahl, "Modes, features, and state-based modeling for clarity and flexibility," in *Proc. Workshop Modeling Softw. Eng.*, 2013, pp. 13–17.
[26] (2015). Mathworks Inc. Simulink. [Online]. Available: http://www.mathworks.com/products/simulink
[27] (2015). MathWorks Inc. Stateflow. [Online]. Available: http://www.mathworks.com/stateflow
[28] N. Halbwachs, *Synchronous Programming of Reactive Systems*. Norwell, MA, USA: Kluwer, 1993.
[29] RTCA DO-178, *Software Considerations in Airborne Systems and Equipment Certification*. RTCA and EUROCAE, 2011.
[30] J. Chilenski, "An investigation of three forms of the modified condition decision coverage (MCDC) criterion," Office of Aviation Res., Washington, D.C., Tech. Rep. DOT/FAA/AR-01/18, Apr. 2001.
[31] M. Whalen, A. Rajan, M. Heimdahl, and S. Miller, "Coverage metrics for requirements-based testing," in *Proc. Int. Symp. Softw. Testing Anal.*, 2006, pp. 25–36.
[32] A. Gargantini and C. Heitmeyer, "Using model checking to generate tests from requirements specifications," *Softw. Eng. Notes*, vol. 24, no. 6, pp. 146–162, Nov. 1999.
[33] S. Rayadurgam and M. Heimdahl, "Coverage based test-case generation using model checkers," in *Proc. 8th IEEE Int. Conf. Workshop Eng. Comput. Based Syst.*, Apr. 2001, pp. 83–91.
[34] G. Hagen, "Verifying safety properties of lustre programs: An SMT-based approach," Ph.D. dissertation, Univ. of Iowa, Iowa, IA, USA, Dec. 2008.
[35] A. Gacek. (2015). JKind-a Java implementation of the KIND model checker. [Online]. Available: https://github.com/agacek
[36] M. Heimdahl, G. Devaraj, and R. Weber, "Specification test coverage adequacy criteria = specification test generation inadequacy criteria?" in *Proc. 8th IEEE Int. Symp. High Assurance Syst. Eng.*, Tampa, FL, USA, Mar. 2004, pp. 178–186.
[37] C. Van Eijk, "Sequential equivalence checking based on structural similarities," *IEEE Trans. Comput.-Aided Des. Integrated Circuits Syst.*, vol. 19, no. 7, pp. 814–819, Jul. 2002.
[38] R. Fisher, *The Design of Experiment*. New York, NY, USA: Hafner, 1935.
[39] A. Rajan, M. Whalen, and M. Heimdahl, "The effect of program and model structure on MC/DC test adequacy coverage," in *Proc. 30th Int. Conf. Softw. Eng.*, 2008, pp. 161–170.
[40] X. Yao, M. Harman, and Y. Jia, "A study of equivalent and stubborn mutation operators using human analysis of equivalence," in *Proc. 36th Int. Conf. Softw. Eng.*, New York, NY, USA, 2014, pp. 919–930.

Gregory Gay received the PhD degree from the University of Minnesota, working with the Critical Systems research group, and the MS degree from West Virginia University. He 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.

Matt Staats received the PhD degree from the University of Minnesota-Twin Cities. He 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. His 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. He 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 more than 40 papers 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. He is a senior member of the IEEE.

Mats P.E. Heimdahl received the MS degree in computer science and engineering from the Royal Institute of Technology (KTH) in Stockholm, Sweden and the PhD degree in information and computer science from the University of California at Irvine. He 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. His research interests include software engineering, safety critical systems, software safety, testing, requirements engineering, formal specification languages, and automated analysis of specifications. He received the US National Science Foundation (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. He is a senior member of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.