Automated Oracle Creation Support, or: How I Learned to Stop Worrying About Fault Propagation and Love Mutation Testing

Matt Staats
Division of Web Science Technology
Korea Advanced Institute of Science & Technology
staatsm@gmail.com

Gregory Gay, Mats P.E. Heimdahl
Department of Computer Science and Engineering
University of Minnesota
greg@greggay.com, heimdahl@cs.umn.edu

***Abstract*—In testing, the test oracle is the artifact that determines whether an application under test executes correctly. The choice of test oracle can significantly impact the effectiveness of the testing process. However, despite the prevalence of tools that support the selection of test inputs, little work exists for supporting oracle creation.**

In this work, we propose a method of supporting test oracle creation. This method 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. Experiments over four industrial examples demonstrate that our method may be a cost-effective approach for producing small, effective oracle data, with fault finding improvements over current industrial best practice of up to 145.8% observed.

***Keywords*-testing, test oracles, oracle data, oracle selection, verification**

I. Introduction

There are two key artifacts to consider when testing software: the *test data*—the inputs given to the application under test—and the *test oracle*, which determines if the application executes correctly. Substantial research has focused on supporting the creation of effective test inputs but relatively little attention has been paid to the creation of oracles. We are interested in the development of automated tools that support the creation of part or all of a test oracle.

Of particular interest in our work are *expected value test oracles* [27]. Consider the following testing process for a software system: (1) the tester selects test inputs using some criterion (structural coverage, random testing, engineering judgement, etc.), potentially employing automated test generation techniques; (2) the tester then defines concrete, expected values for these inputs for one or more variables (internal state variables or output variables) in the program, thus creating an expected value test oracle. The set of variables for which expected values are defined is termed the *oracle data set* [28]. Our focus is on critical systems; past experience with industrial practitioners indicates that expected value test oracles are commonly used in testing such systems.

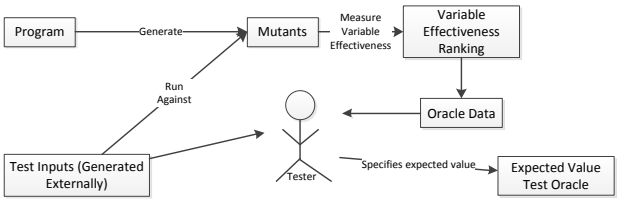


Figure 1. Supporting Expected Value Test Oracle Creation

We present an approach to automatically select the oracle data for use by expected value oracles. Our motivation is twofold. First, existing research indicates that the choice of oracle has a significant impact on the effectiveness of the testing process [32], [3], [28], [27]. However, current widespread practice is to define expected values for only the outputs, a practice that can be suboptimal. Second, manually defining expected values is a potentially time-consuming and consequently, expensive process. Thus in practice the naive solution of "monitor everything" is not feasible because of the high cost of creating the oracle. Even in situations where an executable software specification can be used as an oracle, e.g., 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.

Our goal is to support the creation of test oracles by selecting oracle data such that the fault finding potential of the testing process is maximized with respect to the cost. We illustrate our proposed approach in Figure 1. First, we generate a collection of mutants from the system under test. Second, the test suite (generated externally) is run against the mutants using the original system as the oracle (fully automated back-to-back testing). Third, we measure how often each variable in the system reveals a fault in a mutant and—based on this information—we rank variable effectiveness in terms of fault finding. Finally, we estimate—based on this ranking—which variables to include in the oracle data for an expected value oracle. 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. This oracle data selection process is completely automated and requires no manual intervention. Once this

oracle data is selected, the tester defines expected values for each element of the oracle data. Testing then commences with a—hopefully—small and highly effective oracle.

We hypothesize that this oracle creation support process will produce an oracle data set that is more effective at fault finding than an oracle data set selected with a standard approach, such as the monitoring of all output variables. To evaluate our hypothesis, we have evaluated our approach using four commercial sub-systems from the civil avionics domain. To our knowledge, there are no alternative approaches to oracle data selection discussed in the literature. To provide an evaluation in the absence of related work, 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) simple random selection of the oracle data set. In addition, we also compare to an idealized scenario where the seeded faults in the mutants are identical to the faults in the actual system under test; thus, providing an estimate of the maximum fault finding effectiveness we could hope to achieve with any oracle data selection support method.

Our results indicate that our approach is generally successful with respect to the baseline approaches, performing as well or better in almost all scenarios, with best-case improvement of 145.8%, and consistent improvements in the 5-30% range. Furthermore, we have also found that our approach often performs almost as well as the estimated maximum. We therefore conclude that our approach may be a cost effective method of supporting the creation of an oracle data set.

II. Background & Related Work

In software testing, a *test oracle* is the artifact used to determine whether the software is working correctly [25]. We are interested in test oracles defined as expected values; 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). During testing, the oracle compares the actual values against the expected values[1]. We term such oracles *expected value oracles*. In our experience with industrial partners such test oracles are commonly used when testing critical software systems.

Our goal is to support the selection of the *oracle data* [27] or *oracle data set*[2]. The oracle data is the subset of internal state variables and outputs for which expected values are specified. 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 *output-base* oracles, whose oracle data contain all the outputs, followed by some number of internal states, and *maximum* oracles, whose oracle data contain *all* of the outputs and internal state variables. In the remainder of this paper, we will refer to the *size* of an oracle, where *size* refers to the number of variables used in the oracle data set.

Larger oracle data sets are generally more powerful than smaller oracle data sets [27], [32], [3]. This phenomenon is due to *fault propagation*—faults leading to incorrect states do not always propagate and manifest themselves as failures in a variable in the oracle data set. By using larger oracle data, we can improve the likelihood we will detect faults.

While the maximum oracle is always (provably) the most effective expected value test oracle, it is often prohibitively expensive to use. This is the case when (1) expected values must be manually specified (consulting requirements documents as needed), a highly labor intensive process, or (2) when the cost of monitoring a large oracle data set is prohibitive, e.g., when testing embedded software on the target platform. In current practice, testers must manually select the oracle data set without the aid of automation; this process is naturally dependent on the skill of the tester. We therefore wish to automatically construct small, but effective oracle data sets for expected value oracles.

Work on test oracles often consists of methods of constructing test oracles from other software engineering artifacts [2]. In our work, we are not constructing the entire test oracle; rather, we are identifying an effective oracle data set for an expected value oracle. We are not aware of any work proposing or evaluating alternative methods of selecting the oracle data set.

Voas and Miller suggest that the PIE approach, which—like our work—relies on a form of mutation analysis, could be used to select internal variables for monitoring [31], 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 [28], [27]. Xie and Memon explore methods of constructing test oracles specifically for GUI systems, yielding several recommendations [32]. 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 [3]. Several tools exist for automatically generating invariant-based test oracles for use in regression testing, including Eclat [20], DiffGen [29], and work by Evans and Savoy [8]. Note that these tools do not quantify the potential effectiveness of invariants, and thus—unlike our approach—no prioritization can be performed.

Fraser and Zeller use mutation testing to generate both test inputs and test oracles [10] 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,

[1]For our case examples, all variables are *scalar* and cannot be a heap object or pointers. Thus, comparison is straightforward.

[2]Oracle data roughly corresponds to Richardson et al.'s concept of *oracle information* [25].

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 (excepting program crashes). Accordingly, the role of the user in their approach differs: the user must 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 parameterize (generalize) the result of their approach to simplify the user's task [11]. However, this creates the possibility of producing *false positives*, where a resulting parameterized input/assertion can indicate faults when none exist (5.9% to 12.7% during evaluation), further changing the user's task. With respect to evaluation, no comparisons against baseline methods of automated oracle selection are performed; developer tests+assertions are compared against, but the cost, i.e., number of developer tests/assertions, is not controlled, and thus relative cost-effectiveness cannot accurately be assessed.

Our work chiefly differs in that we are trying to *support* creation of a test oracle, rather than completely automate it. The domain and type of oracles generated also differ, as does the nature of the test inputs used in our evaluation.

Mutation analysis was originally introduced as a method of evaluating the effectiveness of test input selection methods [7]. Subsequent work also explored the use of mutation analysis as a means of generating tests. In [16], Jia and Harman summarize both the technical innovations and applications related to mutation testing. To the best of our knowledge, only Fraser and Zeller have leveraged this to address test oracles.

III. Oracle Data Selection

Our approach for selecting the oracle data set is based on the use of mutation testing for selecting test inputs [16]. In mutation testing, a large set of programs, termed *mutants*, are created by seeding various faults (either automatically or by hand) into a system. A test input capable of distinguishing the mutant from the original program 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 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 accomplish this, we perform the following basic steps:

1) Generate several mutants, called the *training set*, from our system under test.
2) Run test inputs, provided by the tester, over the training set and the original system, determining which variables distinguish each mutant from the original system.
3) Process this information to create a list of variables ordered in terms of apparent fault finding effectiveness, the the *variable ranking*.
4) 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.
5) Select the top X variables in the ranking as the oracle data.

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

A. 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 is created by 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 all mutants produced are both syntactically and semantically valid: no mutant will "crash" the system under test. The mutation testing operators used in this experiment are similar to those used by other researchers, for example, arithmetic, relational, and boolean operator replacement, boolean variable negation, constant replacement, and delay introduction (that is, use the stored value of the variable from the previous computational cycle rather than the newly computed value). A detailed description is available in [21].

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 in [1] where the authors found that generated mutants are a reasonable substitute for actual failures in testing experiments. This offers evidence that our use of mutation testing will support the creation of oracles useful for real 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. Furthermore, this scenario is the most likely within our domain of interest.

B. Variable Ranking

Once we have generated mutants, we then run the test inputs over both the mutants and the original program. During execution of these inputs, we collect the values for every variable, at every step of every test (i.e., the complete state at every point of the execution). We term this resulting data as the *trace data*. A variable is said to have 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 which mutants are killed by which variables. Note that duplicate detections, in which a variables detects the same mutant in multiple tests, are not counted.

Once we have computed this information, 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 and 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 variables that are individually effective, but ineffective as a group, we have elected to use a greedy algorithm for solving the *set covering problem* [6] 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. Our goal is 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 [12]. Consequently, we employ a well-known effective greedy algorithm to solve the problem [5]: (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. The resulting ranking can then be used to produce an oracle data set of size n—simply select the top n elements of the list.

C. Estimating Useful Oracle Data Size

Once we have a calculated the ranked list of variables, we can select an oracle data set of size 1, 2, etc. up to the maximum number of variables in the system, with the choice of size likely made according to some testing budget. In some scenarios, the tester may have little guidance as to the appropriate size of the oracle data. In such a scenario, we would like to offer a recommendation to the tester; we would like to select an oracle data set such that the size of the set is justifiable, i.e., not so small that potentially useful variables are omitted, and not so large that a significant number of variables not adding value are selected.

To accomplish this, we determine the fault finding effectiveness of oracle data sets of size 1, 2, 3, etc. over our training set of mutants. The fault finding effectiveness of these oracles will increase with the oracle's size, but the increases will diminish as the oracle size increases. Consequently, it is 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. Therefore, in our evaluation, we explore two potential thresholds: 5% and 2.5%.

IV. Evaluation

We wish to evaluate if our oracle creation support approach yields effective oracle data sets. Preferably, we would directly compare against existing algorithms for selecting oracle data; however, to the best of our knowledge, no such methods exist. We therefore compare our technique against two potential *baseline approaches* for oracle data set selection, detailed later, as well as an idealized version of our own approach. We wish to explore the following research questions:

Question 1: *Is our approach more effective in practice than current baseline approaches to oracle data selection?*

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

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

A. Experimental Setup Overview

In this research, we have used four industrial systems developed by Rockwell Collins engineers. All four systems were modelled using the Simulink notation from Mathworks Inc. [18] and were automatically translated into the Lustre synchronous programming language [14] in order to take advantage of existing automation[3] .

Two systems, *DWM1* and *DWM2*, represent distinct portions 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.

	Subsyst.s	Blocks	Outputs	Internals
DWM1	128	429	9	115
DWM2	3109	11,439	7	569
Vertmax_Batch	396	1,453	2	415
Latctl_Batch	120	718	1	128

Table I
Case Example Information

All four systems represent sizable, operational systems. Information related to these systems is provided in Table I.

For each case example, we performed the following steps: (1) generated structural test input suites (detailed in Section IV-B below), (2) generated training sets (Section IV-C), (3) generated evaluation sets (Section IV-C), (4) ran test suite

[3]In practice, Lustre would then be automatically translated to C code, but this is a syntactic transformation. Our usage of Lustre is purely for convenience; if applied to C the results here would be identical.

on mutants (Section IV-E), (5) generated oracle rankings (Section IV-D), and (6) assessed fault finding ability of each oracle and test suite combination using evaluation sets (Section IV-E).

B. 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. Since we are studying the effectiveness using avionics systems, two structural coverage criteria are likely to be employed: branch coverage and MC/DC coverage [26]. Several variations of MC/DC exist—for this study, we use Masking MC/DC, as it is frequently used within the avionics community [4].

We use a counterexample-based test generation approach to generate tests satisfying these coverage criteria [13], [24]. This approach is guaranteed to generate a test suite that achieves the maximum possible coverage. We have used the NuSMV model checker in our experiments [19] for its efficiency and we have found the tests produced to be both simple and short [15].

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. We begin by determining the coverage obligations satisfied by each test generated, and initialize an empty test set $reduced$. We then randomly select a test input from the full set of tests; if it satisfies obligations not satisfied by any test input in $reduced$, we add it to $reduced$. We continue until all tests have been removed from the full set of tests. We produce 10 different test suites for each case example/coverage criterion to control for the impact of randomization.

C. Mutant Generation

For each case example, 250 mutants are created by introducing a single fault into the correct implementation. 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. We then remove functionally equivalent mutants from each evaluation set, resulting in a reduction of 0-9 mutants.

We remove *functionally equivalent* mutants using the NuSMV model checker. This is possible due to the nature of the systems in our study—each system is finite, and thus determining equivalence is decidable, and in practice fast[4]. The removal of equivalent mutants is done for the evaluation sets as it represents a potential threat to validity in our evaluation; our method is effective only if it detects faults which can impact the external behavior. Note that the removal of equivalent mutants for training sets is possible for the case examples examined (and would likely improve the effectiveness of our approach), but is *not* done as it may not be practical for sufficiently large systems and is generally undecidable for systems that are not finite.

[4]Equivalence checking is fairly routine in the hardware domain; a good introduction can be found in [30].

D. Oracle Data Set Generation

For each set of mutants generated (training sets and evaluation sets), we generated an oracle ranking using the approach described in Section III. 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 the maximum potential effectiveness of oracle data selection in general, and are used to address Question 2.

We limited each ranking to contain m variables (where m is 10 or twice the number of output variables, whichever was larger) since test oracles significantly larger than output-only oracles were deemed unlikely to be used in practice, and using larger oracles makes visualizing the more relevant and thus interesting oracle sizes difficult.

To answer Questions 1 and 3, we compare against two baseline rankings. First, the *output-base* approach creates rankings by first randomly selecting output variables, and then randomly selecting internal state variables. Thus, the output-base rankings always list the outputs first (i.e., more highly ranked) followed by the randomly-selected internal state variables. This ranking was chosen to reflect what appears to be the current approach to oracle data selection: select the outputs first, and if resources permit, select one or more of the internal state variables. Second, to provide an unbiased method, the *random approach* creates completely random oracle rankings. The resulting rankings are simply a random ordering of the internal state and output variables.

E. Data Collection

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 simulator. Note that the time required to produce a single oracle ranking—generate mutants, run tests, and apply the greedy set cover algorithm—is quite small, less than one hour for every case example. This raw trace data is then used by our algorithm and our evaluation.

The fault finding effectiveness of an oracle is computed as the percentage of mutants killed versus the number of mutants in the evaluation set used. We perform this analysis

for each oracle and test suite for every case example, and use the information produced by this analysis to evaluate our research questions.

V. Results & Discussion

In this section, we discuss our results in the context of our three research questions: (Q1) *"Is our approach more effective than existing baseline approaches for oracle data selection?"*, (Q2) *"What is the maximum potential effectiveness of our mutation-based technique, and how does the real-world effectiveness compare?"*, and (Q3) *"How does the choice of test input data impact the effectiveness of our approach?"*.

In Figure 2, we plot the median fault finding effectiveness of expected value test oracles for increasing oracle sizes[5]. Four ranking methods are plotted: both baseline rankings, our mutation-based approach, and the idealized mutation-based approach. Median values were used for these plots as plotting all test suite/training set combinations (using boxplots, scatter-plots, etc.) yields figures that are very difficult to interpret. For each subfigure, we plot the number of outputs as a dashed, vertical blue 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% thresholds for recommending oracle sizes as solid orange lines (see Section III-C). Note that the 2.5% threshold is not always met for the oracle sizes explored.

In Table III, we list the median relative improvement in fault finding effectiveness using our proposed oracle data creation approach versus the output-base ranking. In Table IV 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 Figure 2, random oracle data performs poorly, and detailed comparisons were thus deemed less interesting and are omitted.

A. Statistical Analysis

Before discussing the implications of our results, we would like to first determine which differences observed are statistically significant. That is, we would like to determine with statistical significance, at what oracle sizes and for which case examples, (1) the idealized performance of a mutation-based approach outperforms the real-world 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 [6]:

H_1: For a given oracle size m, the idealized approach outperforms the standard mutation-based approach.

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

H_3: For a given oracle size m, the standard mutation-based approach outperforms the random approach.

We then formulate the appropriate null hypotheses:

$H0_1$: For a given oracle size m, the fault finding numbers for the idealized approach are drawn from the same distribution as the fault finding numbers for the standard mutation-based approach.

$H0_2$: For a given oracle size m, the fault finding numbers for the standard mutation-base approach approach are drawn from the same distribution as the fault finding numbers for the output-base approach.

$H0_3$: For a given oracle size m, the fault finding numbers for the standard mutation-base approach approach are drawn from the same distribution as the fault finding numbers for the random approach.

Our observations are drawn from an unknown distribution; therefore, we cannot fit our data to a theoretical probability distribution. To evaluate our null hypotheses without any assumptions on the distribution of our data, we use the two-tailed bootstrap paired permutation test (a non-parametric test with no distribution assumptions [9], [17]) with 250,000 samples, 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 case example and coverage criteria combination, we can pair each test suite T + idealized ranking with T + training set ranking (for $H0_1$), and each test suite T + training set ranking with T + random or output-base ranking ($H0_2, H0_3$). We then apply the statistical test for each case example, coverage criteria, and oracle size with $\alpha = 0.05$[7].

Our results indicate that null hypotheses $H0_1$ and $H0_3$ can be rejected for all combinations of case examples, coverage criteria, and oracle sizes, with $p < 0.001$. We therefore *accept* H_1 and H_3 for all combinations of case examples, coverage criteria, and oracle sizes. In the case of $H0_2$, there exist combinations in which the differences are not statistically significant, or marginally statistically significant (near α). These combinations mostly correspond to points of little to no relative improvements over output-base oracles. We list these p-values results in Table II[8] and

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

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

[7]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. The statistical tests are employed to where observed differences between oracle data selection methods are unlikely to be due to chance.

[8]Also, for *Latctl_Batch*, when using MC/DC tests and an oracle of size one, $H0_2$ cannot be rejected as $p = 1.0$.

	Branch	MC/DC	
Oracle Size	**DWM_2**	**DWM_2**	**DWM_1**
1	0.104	0.013	< 0.001
2	1.0	0.203	< 0.001
3	1.0	1.0	< 0.001
4	0.009	1.0	< 0.001
5	0.047	0.408	< 0.001
6	0.293	0.002	< 0.001
7	0.063	0.063	< 0.001
8	0.475	0.470	1.0
9	0.081	1.0	0.407
10	0.133	0.003	0.248
11	1.0	0.008	0.407
12	1.0	< 0.001	1.0
13	0.294	< 0.001	1.0
14	0.004	< 0.001	0.246

Table II
$H0_2$ P-VALUES

highlight all results without statistical significance with a * in Table III. P-values not specified are < 0.001.

B. Evaluation of Practical Effectiveness (Q1)

When designing an oracle creation support method, the obvious question to ask is, *"Is this better than current best practice?"* As we can see from Figure 2, every oracle generated outperforms the random approach with statistical significance, often by a wide margin. We thus can immediately discard random oracle data selection as a useful method of oracle data selection, as both our approach and the output-base approach outperform it in all scenarios. We do not discuss the random approach further.

We can also see that for both coverage criteria and every case example, nearly every oracle generated for three of four systems outperforms the output-base approaches with statistical significance. The pattern goes as follows: for oracles smaller than the output-only oracle, our approach tends to perform relatively well compared to the output-base approach, with improvements ranging from 0.0 to 145.8%. This reflects the strength of prioritizing variables: we generally select more effective variables for inclusion into the oracle data earlier than the output-base approach. As the test oracle size grows closer in size to the output-only oracle, the relative improvement decreases, but often (4 of 8 case example/coverage combinations) still outperforms the output-only oracle, up to 26.4%. Finally, as the test oracle grows in size beyond the output-only oracle, our relative improvement when using our approach again grows, with improvements of 2.2-45% for largest oracles.

This observation is illustrated best using the *DWM_1* case example. Here we see that, for both coverage criteria, while the output-base ranking method performs relatively well for small oracle sizes, the set-covering approach nevertheless ranks the *most* effective variable first, locating roughly 80.% and 145.8% more faults than the variable chosen by the output-base technique for the branch and MC/DC coverage criteria, respectively. The set-covering oracle continues on to select a handful of variables that find additional faults, but as additional outputs are added to the output-base approach the relative improvement decreases, becoming statistically insignificant in the case of MC/DC coverage. This indicates the wisdom of the current approach, but also demonstrates room for improvement: for branch coverage, the proposed approach is, at worst, 5.9% better than the output-oracle. For the MC/DC approach, we show no improvement, though we achieve similar fault finding to the output-only oracle using *smaller* oracles. Finally, as the oracle grows, incorporating (by necessity) internal state variables, our approach fares well, as effective internal state variables continue to be added, reaching up to 14.2% relative improvement.

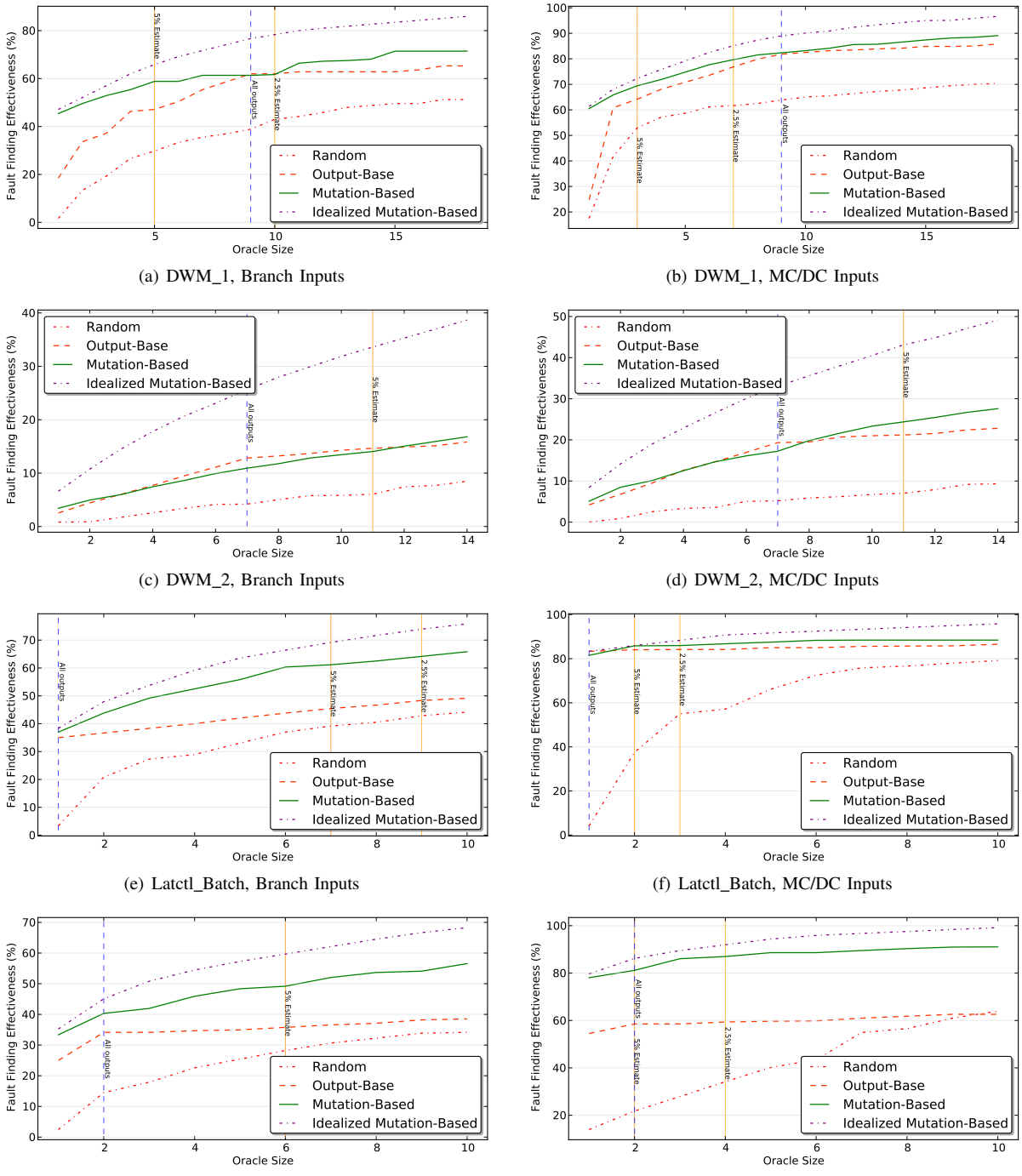
The only exception occurs for the *DWM_2* system. For this case example, although mutation-based oracles tend to be roughly equivalent in effectiveness as output-base oracles (we generally cannot reject $H0_2$ at $\alpha = 0.05$), only for small or large oracles (approximately +-6 sizes from the output-only oracle) does our approach do relatively well. Examining the composition of these oracles indicates why: the ranking generated using our approach for this case example begins mostly with output variables, and thus oracles of generated using our approach are very similar to those generated using the output-base approach. The performance gain of mutation-base oracles 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 for the oracle*.

However, in a few instances, our approach in fact does worse (with statistical significance) than the output-base approach. The issue appears to be the greedy set-coverage algorithm: for *DWM_2*, the approach tends to select a highly effective internal state variable first, which prevents a computationally related output variable from being selected for larger oracle data. Given an optimal set cover algorithm, this issue would likely be avoided. However, eventually, for larger oracle sizes, this issue is corrected, with statistically significant improvements of 11.7% and 18.1% observed.

Despite this inconsistency, it seems clear that our approach can be effective in practice. We can consistently generate oracle data sets that are effective over *different* faults, generally with higher effectiveness than existing ranking approaches. In cases where our approach does not do better, the difference observed tends to be small, if present. Furthermore, we can provide the tester with recommendations of the cost effective oracle sizes, thus avoiding the need for testers to manually estimate an effective oracle size.

C. Potential Effectiveness of Oracle Data Selection Approach (Q2)

As noted previously, 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



(a) DWM_1, Branch Inputs

(b) DWM_1, MC/DC Inputs

(c) DWM_2, Branch Inputs

(d) DWM_2, MC/DC Inputs

(e) Latctl_Batch, Branch Inputs

(f) Latctl_Batch, MC/DC Inputs

(g) Vertmax_Batch, Branch Inputs

(h) Vertmax_Batch, MC/DC Inputs

Figure 2. Median Effectiveness of Various Approaches to Oracle Data Selection

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, and thus is used to estimate the maximum potential of our approach.

The results can be seen in Figure 2 and Table IV. We can observe from these results that while the *potential* performance of a mutation-based oracle is (naturally) almost always higher than the real-world performance of our method, the gap between a realistic implementation of our approach and the ideal scenario is often quite small. Indeed, apart from the *DWM_2* system, for most case examples and oracle sizes, the difference between the idealized and

	Branch				MC/DC			
Oracle Size	**DWM_1**	**DWM_2**	**Vertmax_Batch**	**Latctl_Batch**	**DWM_1**	**DWM_2**	**Vertmax_Batch**	**Latctl_Batch**
1	80.0%	50.0%*	34.6%	2.7%	145.8%	40.0%	44.4%	0.0%*
2	17.1%	0.0%*	26.4%	17.6%	9.3%	25.0%*	44.4%	3.8%
3	12.5%	0.0%*	30.5%	25.0%	7.5%	0.0%*	48.3%	3.8%
4	11.1%	-10.0%	30.5%	27.5%	5.7%	0.0%*	46.6%	3.8%
5	10.5%	-10.0%	38.2%	30.2%	5.3%	0.0%*	49.0%	3.8%
6	10.5%	-11.1%*	41.1%	32.6%	5.2%	-10.0%	47.5%	3.6%
7	8.1%	-10.0%*	44.4%	34.7%	2.6%	-11.7%*	47.4%	3.5%
8	5.9%	-8.3%*	43.2%	32.5%	0.0%*	-4.5%*	46.6%	3.5%
9	7.3%	0.0%*	40.0%	33.3%	-1.2%*	0.0%*	44.2%	3.4%
10	7.2%	0.0%*	42.1%	32.6%	-1.1%*	8.6%	45.0%	3.4%
11	7.3%	0.0%*	–	–	-1.1%*	12.0%	–	–
12	8.8%	0.0%*	–	–	0.0%*	17.3%	–	–
13	10.6%	6.6%*	–	–	1.1%*	17.3%	–	–
14	12.1%	11.7%	–	–	2.2%*	18.1%	–	–
15	14.4%	–	–	–	2.3%	–	–	–
16	14.5%	–	–	–	2.2%	–	–	–
17	14.2%	–	–	–	2.2%	–	–	–
18	14.2%	–	–	–	2.2%	–	–	–

Table III
MEDIAN RELATIVE IMPROVEMENT USING MUTATION-BASED SELECTION OVER OUTPUT-BASE SELECTION

realistic scenarios is *less than* the gap between the output-base approach and our approach. 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.

D. Impact of Coverage Criteria (Q3)

Our final question concerns the impact of varying the coverage criteria—and thus the test inputs—on the relative effectiveness of oracle selection. In this study, we have used two coverage criteria of varying strength. Intuitively, it seems likely that when using stronger test suites (those satisfying MC/DC in this study), the potential for improving the testing process via oracle selection would be less, as the test inputs should do a better job of exercising the code.

Precisely quantifying likeness is difficult; however, as shown in Figure 2, for each case example the general relationship seems (to our surprise) to be roughly the same. For example, for the *DWM_1* system, we can see that despite the 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 relative 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 said relative improvement. In some cases relative improvements are higher for branch coverage (*Latctl_Batch*) and in others they are higher for MC/DC (*Vertmax_Batch*).

These results indicate that, perhaps more that the test inputs used, characteristics of the system under test are the primary determinant of the relative effectiveness of our approach. Unfortunately, it is unclear exactly what these characteristics are. Testers can, of course, estimate the effectiveness of our applying approach by automating our study, applying essentially the same approach used to provide the user a suggested oracle size[9]. However, this is potentially expensive and provides no insight concerning why our approach is very good (relative to baseline approaches, and in an absolute sense) for some systems and merely equivalent for others. Developing metrics that can allow us to a priori estimate the effectiveness of our approach is an area for potential future work.

VI. THREATS TO VALIDITY

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

We have used Lustre [14] as our implementation language rather than a more common language such as C or C++. However, as noted previously, systems written in Lustre are similar in style to traditional imperative code produced by code generators used in embedded systems development. A simple syntactic transformation suffices to translate Lustre code to C code that would be generally be used.

In our study, we have used test cases generated to satisfy two structural coverage criteria. These criteria were selected as they are particularly relevant to the domain of interest. However, there exist many methods generating test inputs, and it is possible that tests generated according to some other methodology would yield different results. For example, requirements-based black-box tests may be effective at propagating errors to the output, reducing the potential improvements for considering internal state variables. We therefore avoid generalizing our results to other methods of test input selection, and intend to study how our approach generalizes to these methods in future work.

[9]Indeed, automating our study to estimate the effectiveness of our approach is akin to applying mutation testing to estimate test input/oracle data effectiveness

	Branch				MC/DC			
Oracle Size	**DWM_1**	**DWM_2**	**Vertmax_Batch**	**Latctl_Batch**	**DWM_1**	**DWM_2**	**Vertmax_Batch**	**Latctl_Batch**
1	3.7%	101.6%	3.6%	4.7%	2.5%	68.0%	2.9%	0.6%
2	8.0%	152.1%	19.5%	8.2%	2.9%	76.9%	6.4%	1.1%
3	12.5%	156.4%	15.4%	15.0%	5.9%	94.9%	6.2%	3.0%
4	14.6%	158.6%	16.1%	13.7%	7.1%	96.0%	7.4%	5.0%
5	16.2%	152.1%	21.0%	13.6%	7.5%	94.6%	7.1%	5.5%
6	17.5%	146.9%	21.9%	12.4%	8.3%	93.5%	8.3%	6.4%
7	19.7%	140.4%	20.9%	17.7%	9.1%	94.3%	9.2%	7.0%
8	22.3%	139.3%	21.7%	20.9%	8.3%	85.7%	10.2%	7.9%
9	22.5%	140.8%	19.6%	14.2%	8.8%	84.2%	9.8%	8.8%
10	20.2%	143.5%	20.2%	16.1%	9.5%	80.8%	10.5%	8.9%
11	21.0%	147.9%	–	–	9.8%	82.5%	–	–
12	18.6%	144.2%	–	–	9.8%	81.0%	–	–
13	19.3%	144.2%	–	–	9.8%	83.9%	–	–
14	17.0%	139.3%	–	–	9.8%	83.0%	–	–
15	17.3%	–	–	–	9.7%	–	–	–
16	17.3%	–	–	–	9.5%	–	–	–
17	18.0%	–	–	–	9.7%	–	–	–
18	19.0%	–	–	–	10.1%	–	–	–

Table IV
MEDIAN RELATIVE IMPROVEMENT USING IDEALIZED MUTATION-BASED SELECTION OVER REALISTIC MUTATION-BASED SELECTION

The tests used are effectively unit tests for a Lustre module, and the behavior of oracles may change when using tests designed for large system integration. Given that we are interested in testing systems at this level, we believe our experiment represents a sensible testing approach and is applicable to practitioners.

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 [22], [23]. Furthermore, pilot studies showed that the results change little when using more than this 100 mutants for training or evaluation sets.

Construct Validity: In our study, we measure the fault finding of oracles and test suites over seeded faults, rather than real faults encountered during development of the software. 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. This is especially likely if the fault model used in mutation testing is significantly different than the faults we encounter in practice. Nevertheless, as mentioned earlier, Andrews 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 [1].

VII. CONCLUSION AND FUTURE WORK

In this study, we have explored a mutation-based method for supporting oracle creation. Our approach automates the selection of oracle data, a key component of expected value test oracles. Our results indicate that our approach is successful with respect to alternative approaches for selecting oracle data, with improvements up to 145.8% over output-base oracle data selection, with improvements in the 10%-30% range relatively common. In cases where our approach is not more effective, it appears to be comparable to the output-base approach. We have also found that our approach performs within an acceptable range from the theoretical maximum.

While our results are encouraging, and demonstrate that our approach can be effective using real-world avionics systems, there are a number of questions that we would like to explore in the future, including:

- **Estimating Training Set Size:** Our evaluation indicates that for our case examples, a reasonable number of mutants are required to produce effective oracle data sets. However, when applying our approach to very large systems, or when using a much larger number of test inputs, we may wish to estimate the needed training set size.
- **Oracle Data Selection Without Test Inputs:** In our approach, we assume the tester has already generated a set of test inputs before generating oracle data. We may wish to generate oracle data without knowing the test inputs that will ultimately be used. Is oracle data generated with one set of test inputs effective for other, different test inputs?

VIII. ACKNOWLEDGEMENTS

This work has been partially supported by NASA Ames Research Center Cooperative Agreement NNA06CB21A, NSF grants CCF-0916583 and CNS-0931931, CNS-1035715, an NSF graduate research fellowship, and the WCU (World Class University) program under the National Research Foundation of Korea and funded by the Ministry of Education, Science and Technology of Korea (Project No: R31-30007).

REFERENCES

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

[2] L. Baresi and M. Young. Test oracles. In *Technical Report CIS-TR-01-02, Dept. of Computer and Information Science, Univ. of Oregon*.

[3] L. Briand, M. DiPenta, and Y. Labiche. Assessing and improving state-based class testing: A series of experiments. *IEEE Trans. on Software Engineering*, 30 (11), 2004.

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

[5] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of operations research*, 4(3):233–235, 1979.

[6] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2001.

[7] R. DeMillo, R. Lipton, and F. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE computer*, 11(4):34–41, 1978.

[8] R. Evans and A. Savoia. Differential testing: a new approach to change detection. In *Proc. of the 6th Joint European Software Engineering Conference and Foundations of Software Engineering*, pages 549–552. ACM, 2007.

[9] R. Fisher. *The Design of Experiment*. New York: Hafner, 1935.

[10] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. In *Prc. of the 19th Int'l Symp. on Software Testing and Analysis*, pages 147–158. ACM, 2010.

[11] G. Fraser and A. Zeller. Generating parameterized unit tests. In *Prc. of the 20th Int'l Symp. on Software Testing and Analysis*, pages 147–158. ACM, 2011.

[12] M. Garey and M. Johnson. *Computers and Intractability*. Freeman, New York, 1979.

[13] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. *Software Engineering Notes*, 24(6):146–162, November 1999.

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

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

[16] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, (99):1, 2010.

[17] P. Kvam and B. Vidakovic. *Nonparametric Statistics with Applications to Science and Engineering*. Wiley-Interscience, 2007.

[18] Mathworks Inc. Simulink product web site. http://www.mathworks.com/products/simulink.

[19] The NuSMV Toolset, 2005. Available at http://nusmv.irst.itc.it/.

[20] C. Pacheco and M. Ernst. Eclat: Automatic generation and classification of test inputs. *ECOOP 2005-Object-Oriented Programming*, pages 504–527, 2005.

[21] A. Rajan, M. Whalen, and M. Heimdahl. The Effect of Program and Model Structure on MC/DC Test Adequacy Coverage. In *Proceedings of 30th International Conference on Software Engineering (ICSE)*, 2008. Available at http://crisys.cs.umn.edu/ICSE08.pdf.

[22] A. Rajan, M. Whalen, and M. Heimdahl. The effect of program and model structure on MC/DC test adequacy coverage. In *Proc. of the 30th Int'l Conference on Software engineering*, pages 161–170. ACM New York, NY, USA, 2008.

[23] A. Rajan, M. Whalen, M. Staats, and M. Heimdahl. Requirements coverage as an adequacy measure for conformance testing. In *Proc. of the 10th Int'l Conf. on Formal Methods and Software Engineering*, pages 86–104. Springer, 2008.

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

[25] D. J. Richardson, S. L. Aha, and T. O'Malley. Specification-based test oracles for reactive systems. In *Proc. of the 14th Int'l Conference on Software Engineering*, pages 105–118. Springer, May 1992.

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

[27] M. Staats, M. Whalen, and M. Heimdahl. New ideas and emerging results track: Better testing through oracle selection. In *Proc. of NIER Workshop, Int'l. Conf. on Software Engineering (ICSE) 2011*. ACM New York, NY, USA, 2011.

[28] M. Staats, M. Whalen, and M. Heimdahl. Programs, testing, and oracles: The foundations of testing revisited. In *Proc. of Int'l. Conf. on Software Engineering (ICSE) 2011*. ACM New York, NY, USA, 2011.

[29] K. Taneja and T. Xie. Diffgen: Automated regression unit-test generation. In *Automated Software Engineering, 2008*, pages 407–410. IEEE, 2008.

[30] C. Van Eijk. Sequential equivalence checking based on structural similarities. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Trans. on*, 19(7):814–819, 2002.

[31] J. Voas and K. Miller. Putting assertions in their place. In *Software Reliability Engineering, 1994., 5th Int'l Symposium on*, pages 152–157, 1994.

[32] Q. Xie and A. Memon. Designing and comparing automated test oracles for gui-based software applications. *ACM Trans. on Software Engineering and Methodology (TOSEM)*, 16(1):4, 2007.