Oracle-Centric Test Case Prioritization

Matt Staats*, Pablo Loyola*, Gregg Rothermel*†

*Division of Web Science and Technology
KAIST
Daejeon, South Korea
{staatsm,ployola}@kaist.ac.kr

†Department of Computer Science
University of Nebraska-Lincoln
Lincoln, NE
grother@cse.unl.edu

***Abstract*—Recent work in testing has demonstrated the benefits of considering test oracles in the testing process. Unfortunately, this work has focused primarily on developing techniques for generating test oracles, in particular techniques based on mutation testing. While effective for test case generation, existing research has not considered the impact of test oracles in the context of regression testing tasks. Of interest here is the problem of test case prioritization, in which a set of test cases are ordered to attempt to detect faults earlier and to improve the effectiveness of testing when the entire set cannot be executed. In this work, we propose a technique for prioritizing test cases that explicitly takes into account the impact of test oracles on the effectiveness of testing. Our technique operates by first capturing the flow of information from variable assignments to test oracles for each test case, and then prioritizing to "cover" variables using the shortest paths possible to a test oracle. As a result, we favor test orderings in which many variables impact the test oracle's result early in test execution. Our results demonstrate improvements in rate of fault detection relative to both random and structural coverage based prioritization techniques when applied to faulty versions of three synchronous reactive systems.**

***Keywords*-Software testing, software metrics.**

I. Introduction

In regression testing, software engineers execute an existing test suite over a program to attempt to detect whether new faults have been introduced into previously tested source code. In practice, regression testing can be an expensive process. For example, we are aware of one software development organization that has, for one of its primary products, a regression test suite containing over 30,000 functional test cases that require over 1000 machine hours to execute. Costs such as this have led to research on *test case prioritization* techniques, which order test cases to maximize the benefit of the testing process relative to the time required. By running more important test cases earlier, failures can be triggered earlier in the testing process, providing useful feedback and allowing debugging to begin earlier. Furthermore, if time constraints on testing exist, more important test cases are executed.

Recent work [1], [2], [3] has demonstrated the potential benefits of considering the impact of *test oracles* — the mechanism for determining whether test inputs pass or fail — on the testing process. This work has shown that by explicitly considering which program values are likely to contain faults, the effectiveness of test oracles, and thus the testing process, can be improved. Most work on test oracles, however, has focused on supporting or automating the creation of oracles as part of automated test case generation. Such work is of limited value in scenarios where test suites already exist, as in regression testing.

Many test case prioritization techniques have been proposed; most are based primarily on structural coverage metrics such as branch or statement coverage (e.g., [4], Section VII reviews other techniques), and none incorporate information on test oracles. In this paper, we propose a new *oracle-centric test case prioritization technique*, that explicitly considers information on test oracles. Our technique begins by dynamically capturing the relationships between computed program values during test execution using techniques similar to those employed in data flow analysis. Following this, we measure the degree to which each computed value impacts those values that are used as part of test oracles. This allows us to compute, for each test case, how well each portion of the program has been exercised relative to test oracles. Finally, using this computed information, we prioritize the test cases to maximize program coverage relative to the test oracles, thus bringing the benefits of explicitly considering test oracles to test case prioritization.

To evaluate the effectiveness of oracle-centric prioritization, we created a prototype implementation for Java and conducted a study using three Java case examples drawn from the domain of reactive systems. For each case example, we constructed 25 sets of randomly generated test suites, each with varying numbers of test inputs, test input lengths, and oracle sizes. Following this, we applied three test case prioritization techniques: random ordering, block coverage-based prioritization, and oracle-centric prioritization, to these test suites. We then computed the effectiveness of each approach at increasing the rate of fault detection (measured by a metric known as APFD) of the resulting test suites.

In our study, oracle-centric prioritization outperforms both alternative prioritization methods, with improvements in APFD of up to 6.52%, and with statistically significant improvements for all case examples. This improvement is comparable to the level of relative improvement observed when moving from random test suite orderings to block-based coverage prioritization, and thus represents a significant improvement. Furthermore, in instances where block-based prioritization does not yield statistically significant improvements, oracle-centric prioritization still yields improvements

over randomized test orderings. These results thus demonstrate the potential for oracle-centric prioritization to improve on coverage-based approaches, and again highlights the importance of considering test oracles in testing.

II. Background

A. Test Case Prioritization

Let P be a program, let P' be a modified version of P, and let T be a test suite for P. Regression testing attempts to validate P'. Researchers have studied various methods for improving the cost-effectiveness of regression testing; these include test case prioritization techniques, which order T to maximize the rate of fault detection during regression testing.

A wide range of prioritization techniques have been proposed and studied (e.g., [4], [5], [6], [7], [8], [9]; Section VII provides a comprehensive list). Most of these techniques depend primarily on code coverage information. Code coverage information is obtained by instrumenting a program such that portions of its source code (e.g, method entries, statements, or basic blocks) that are exercised by test cases can be measured. Given the results of running tests on instrumented code, a "total-coverage" prioritization technique [4] orders test cases in terms of the total number of code components (e.g., methods, statements, or blocks) that they cover. One way to improve "total-coverage" techniques is to add feedback, using an iterative greedy approach in which each "next" test case is placed in the prioritized order taking into account the effect of test cases already placed in the order. For example, an "additional-block-coverage" technique [4] prioritizes test cases in terms of the numbers of new (not-yet-covered) blocks they cover, by iteratively selecting the test case that covers the most not-yet-covered blocks until all blocks are covered, then repeats this process until all test cases have been prioritized.

B. Test Oracles

Work related to test oracles has largely focused on methods for constructing test oracles, often with an emphasis on leveraging specifications [10]. More recent work has explored the impact of test oracles on testing effectiveness, demonstrating the potential power of test oracles and proposing methods for selecting test oracles [2], [3], [11]. To date, techniques for selecting and evaluating the effectiveness of test oracles (e.g., [2], [3]) have typically used mutation testing to select oracles. In these techniques, large numbers of mutants (programs with small syntactic changes) are generated from a program, and test cases are executed over the original program and each mutant. The variables or assertions that appear highly effective at distinguishing mutants from the original program are selected for use in the test oracle. Results indicate that this approach is effective, but also expensive, requiring test cases to be executed over hundreds or thousands of mutants [2], [3].

III. Overview of Technique

Researchers creating prioritization techniques have often operated under the assumption that maximizing the rate at which testing achieves structural coverage will maximize the rate at which testing uncovers faults. However, merely executing faulty code is not enough to guarantee revealing faults; we must also ensure that the fault changes program state in such a way that fault state propagates to a test oracle [12]. Our goal is to construct a method for test case prioritization that better captures this, thus improving the effectiveness of prioritization.

To understand the potential impact of test oracles on prioritization effectiveness, consider the pseudocode shown in Figure 1, along with the two test cases shown in Figure 2. The first test case covers both branches, whereas the second test case covers only the first. However, the first test case's oracle checks the result of the computation on line 2, while the second test case checks the computation on line 4, which uses the result of the computation on line 2. By selecting the second test case, we implicitly check the result of both computations, and thus can capture faults on lines 2-4. Thus by executing test 2 first, we may increase the likelihood of catching a fault earlier. However, current prioritization techniques are incapable of making such oracle-based determinations.

```
1 def codeSnippet(in1, in2):
2    x = max(in1, in2)
3    if x < 0:
4       y = x * -1
5    if y > 1000:
6       z = log(x)
```

Fig. 1. Example Code Snippet

```
Test 1:
   codeSnippet(-2000, -1500)
   assert(x == -1500)
Test 2:
   codeSnippet(-200, -150)
   assert(y == 150)
```

Fig. 2. Example Test Cases

To support oracle-based test case prioritization, we have developed a technique for understanding how test inputs and test oracles together verify system behavior. This technique is based on dataflow analysis, in which the definitions and uses of variables (*defs* and *uses*) are tracked during system execution [13]. Typically, where testing is concerned, dataflow analysis is used to form the basis of various dataflow coverage criteria, such as *all-defs*, *all-du-paths*, and so forth [13]. However, these criteria generally view dataflow in terms of pairs of *defs* and *uses*, and thus do not explicitly consider how sequences of *defs* and *uses* propagate to test oracles.

In this work, we have developed a metric for measuring how well each computed variable V in program P is checked by a test oracle. This computation takes into account the number of intermediate definitions between the last definition of V and its propagation to a test oracle O, with the intuition that the shorter the number of intermediate definitions, the more likely it is that an incorrect value computed for V will result in O detecting a fault. By capturing this information during each test

input's execution (on a previous release of a system), we can estimate how well each variable computed is checked by each combination of test input and test oracle. This information is used to order our test cases on a new version of the system, attempting to maximize the rate of detection of faults by our test oracles during subsequent executions.

Our technique can be divided into two components. The first component relates to capturing data flow information during test case execution, and is reasonably straightforward. We describe this component in Algorithm 1. As shown, given a program *P* and a test input *T*, each assignment of a variable x computed with values $y_1 ... y_n$ results in setting the distance from each y_i to x to 1 (resetting the previous distances for x as appropriate.) Then, for all variables connected to $y_1 ... y_n$, the distance to x is the minimum distance to any value $y_i \in y_1 ... y_n$ plus one. Finally, after test case execution completes, the distance from each value computed to the values used in the test oracle (O) is measured, producing DM. (The use of *measure* is explained later.) Along with the set of variable assignment locations VS (unioned across all test input executions), this represents the data needed for prioritization.

Algorithm 1 Test Input Dataflow Recording

Require: Test input *T*
Require: Program *P*
Require: Oracle variables *O*
Require: *connectedSet*(v): set of elements connected to v
1: $vs = \emptyset$
2: **while** *T* runs over *P* **do**
3: **for all** y_i in x := y_1 op y_2 ... y_n **do**
4: **if** $y_i \neq x$ **then**
5: **for all** $z \in$ *connectedSet*(x) **do**
6: *updateDist*(x, z, ∞)
7: **end for**
8: **end if**
9: *updateDist*($x, y_i, 1$)
10: **for all** $z \in$ *connectedSet*(y_i) **do**
11: *updateDist*(x, z, *min*(*dist*(x, z), *dist*(y_i, z) + 1))
12: **end for**
13: $vs = vs \cup x$
14: **end for**
15: **end while**
16: map $DM = \{\}$
17: **for all** $v \in O$ **do**
18: **for all** $x \in vs$ **do**
19: $DM[x]$ = *measure*(*dist*(v, x)))
20: **end for**
21: **end for**
22: **return** DM

The second component of our technique relates to using captured data flow information to prioritize test cases and is a bit more complex. When prioritizing based on structural coverage criteria, we seek to quickly maximize our coverage over source code elements such as branches or statements. In such algorithms, each element is either "covered" or "uncovered", and thus straightforward, heuristics developed for the set covering problem can be used [14]. In contrast, when prioritizing based on test oracles, we seek not only to "cover" each occurrence of a variable, but also to minimize the distance from each variable *def* to a *use* in an oracle. Thus the problem is not to satisfy a checklist (e.g., coverage all branches), but to minimize a metric computed across many elements. This introduces additional factors such as how to weigh completely uncovered elements and how best to consider distance. To the best of our knowledge, there are no pre-existing algorithms for performing this computation.

We describe the components of our approach in Algorithms 2, 3, and 4. Algorithm 2 uses the set of distance maps produced by Algorithm 1 to produce a set of ordered test cases. The algorithm operates by tracking the minimum distance from all computed variables to an oracle used by some test case. Initially, the ordered list of test cases is empty, and thus the minimum distance to each variable is the maximum distance, MAX. Following initialization, a greedy approach for selection is used, with each test case considered in turn. The test case that reduces the cost the most is selected, and the minimum weight and distance are updated. (If no test case reduces the cost, a random test case is chosen.) Once all test cases have been selected, the process terminates. Algorithms 3 and 4 provide the methods for considering and adding test cases. When considering a test case, the distance to each variable is considered, and updated if the distance can be shortened. When adding a test case, we update the weight for each variable covered by the test case.

Algorithm 2 Oracle-Centric Prioritization

Require: Map of distance maps *DMS*
Require: Test suite *TS*
Require: Maximum cost *MAX*
Require: Set of compute variables *VS*
1: map $minWeight = \{\}$
2: $curVal = 0$
3: **for all** variable $v \in VS$ **do**
4: $minWeight[v] = MAX$
5: $curVal$+=MAX
6: **end for**
7: Ordered test suite *OTS* = []
8: **while** *size(TS)* > 0 **do**
9: $bestVal = curVal$
10: **for all** $t \in$ *TS* **do**
11: $newVal$ = *tryTest*($minWeight, curVal, DMS[t]$)
12: **if** $newVal < bestVal$ **then**
13: $bestVal = newVal$
14: $bestTest = t$
15: **end if**
16: **end for**
17: **if** $bestVal == curVal$ **then**
18: $bestTest$ = *randomTest(TS)*
19: **end if**
20: *removeTest*($bestTest$, *TS*)
21: *addTest*($minWeight, DMS[bestTest]$)
22: $OTS.append(TS)$
23: $curVal = bestVal$
24: **end while**
25: **return** *OTS*

Two factors should be considered when computing the benefit of adding a test case. First, we must consider the distance to completely uncovered variables (i.e., the penalty).

Algorithm 3 Compute Updated Test Suite Value (tryTest)

Require: Minimum weight map *MINWEIGHT*
Require: Current value *VAL*
Require: Distance map *DM*
1: **for all** variables $v \in DM.keys()$ **do**
2: $oldW = MINWEIGHT[v]$
3: $newW = DM[v]$
4: **if** $oldW > newW$ **then**
5: VAL-=$(oldW - newW)$
6: **end if**
7: **end for**
8: **return** VAL

Algorithm 4 Add Test Case (addTest)

Require: Minimum weight map *MINWEIGHT*
Require: Distance map *DM*
1: **for all** variable $v \in DM.keys()$ **do**
2: $oldW = MINWEIGHT[v]$
3: $newW = DM[v]$
4: **if** $oldW > newW$ **then**
5: $MINWEIGHT[v] = newW$
6: **end if**
7: **end for**

Eventually, all coverable variables will be covered, but when selecting test inputs early we must achieve a balance between covering all variables early, which may result in selecting tests cases to cover new variables, and minimizing the distance to previously covered variables, which may result in ignoring hard to cover variables. We define MAX as the distance assigned to uncovered variables; as MAX increases our technique will favor covering variables early.

The second factor to consider is the method used to compute distance. While we can simply use a linear measurement of distance, we believe that differences between short distances are very relevant (e.g., distance of 3 versus distance of 6), whereas differences in large distances (e.g., 20 versus 30) are less likely to relate to testing effectiveness. To understand why this may occur, consider a scenario in which a computed value is incorrect. If the probability of propagation at each assignment is X, then the probability of propagating an incorrect value for distance d is X^d, which quickly becomes indistinguishably small for large values of d. Thus in practice we may choose to use an alternative method of measuring distance, such as logarithmic functions. We represent the function used to compute distance as *measure*.

We have implemented our technique using source code instrumentation via the Eclipse platform [15]. By using source code instrumentation, we can capture the actual assignments as they appear in the code, and thus our measurements closely model the code as written by the developer. Furthermore, the use of source code instrumentation gives us a tool-agnostic platform for future work, as we can easily interface with other tools to leverage our core approach to address other problems such as test input generation and test suite reduction.

In practice, we need only monitor assignments involving primitive variables (i.e., int, float, etc.). An assignment involving an object simply creates another method of interacting with the object's members. Note that as our approach is dynamic, aliasing between objects is handled automatically. During an assignment, we simply look up the primitive member's parent object to find the correct primitive location.

IV. Evaluation

To better understand the performance of oracle-centric prioritization, we designed and performed an empirical study. We compare our approach with two alternative approaches for ordering test cases. The first, random order, acts as a baseline for comparison; it focuses on arbitrary test case orders and serves as a minimum threshold for effectiveness (while random order is not a prioritization technique per se, for simplicity, we refer to it henceforth as "random prioritization".) The second approach, additional-block-coverage, is the most commonly studied structural coverage based approach, and serves as a representative example of existing techniques. We explore the following research questions:

RQ1 *Is oracle-centric prioritization more effective than random prioritization?*

RQ2 *Is oracle-centric prioritization more effective than additional-block-coverage prioritization?*

A. Case Examples

The intuition behind this work was originally born out of work performed within the domain of critical reactive systems, in which we observed that the impact of test oracles on such systems was potentially stronger than the choice of test inputs, and later work in which we developed methods for supporting test oracle selection. Given the potentially high cost of retesting such systems [16], we wished to see whether we could leverage these results to improve test case prioritization for them.

Accordingly, for our initial study, we use three synchronous reactive case examples originally developed at the University of Minnesota in conjunction with various industrial partners, described below and listed in Table I. Each system was developed in Simulink. ASW was automatically translated to Java using tools developed at Vanderbilt University [17], while both WBS and FGS were translated to C using tools developed at Rockwell Collins and manually translated to Java. Table I also provides basic size measurements for the systems. The systems' lines of code (LOC) counts were obtained by CLOC [18].

TABLE I
Case Example Measurements

	# Simulink Nodes	# Java LOC
ASW	14	336
WBS	157	206
FGS	4510	1237

Altitude Switch (ASW): Altitude Switch (ASW) is a reusable component that turns power on to a Device Of Interest (DOI) when an aircraft descends below a threshold altitude. If the altitude cannot be determined for more than two seconds,

ASW indicates a fault. The detection of a fault turns on an indicator lamp within the cockpit.

Wheel Brake System (WBS): Wheel Brake System (WBS) is a Simulink model derived from the WBS case example found in ARP 4761 [19]. WBS is installed on the two main landing gears of an aircraft. Braking on the ground can be commanded automatically (autobrake) without the need for pedal application, and allows the pilot to pre-arm the deceleration rate prior to takeoff or landing. When the wheels have traction, the autobrake function controls brake pressure to provide a smooth and constant deceleration.

Flight Guidance System (FGS): The Flight Guidance System is a component of the overall Flight Control System (FCS) in a commercial aircraft. It compares the measured state of an aircraft (position, speed, and altitude) to the desired state and generates pitch and roll-guidance commands to minimize the difference between the measured and desired state. In this study we have used the model of the mode logic, which determines which lateral and vertical modes of operation are active and armed at any given time.

B. Experiment Design Overview

In this study, we investigate the impact of one independent variable, the prioritization technique applied, on one dependent variable, the rate of fault detection, using randomization to control for several factors.

1) Independent Variable: As noted earlier, we consider three different test case prioritization techniques: *random*, *oracle-centric*, and *additional-block-coverage*. The first approach, *random*, represents the experimental control. In this approach, we randomly order the test suite. We expect this to be, on average, the most ineffective method for ordering test cases, and thus it serves as a useful baseline for understanding the effectiveness of our approach. *Additional-block-coverage*, as noted in Section II, serves as a representative example of structural coverage based approaches. Here, we prioritize test input selection to maximize basic block coverage quickly. In using this technique, we seek to determine whether our approach can outperform existing methods of test case prioritization. Finally, our *oracle-centric* approach prioritizes with respect to information propagated to test oracles. Understanding its performance is the primary goal of this study. Note that with our oracle-centric approach, we apply *log* as our *measure* function with a maximum distance of 5.0 (though for these case examples, the selection of these parameters does not impact effectiveness).

When applying additional-block-coverage and oracle-centric prioritization, the random test suite ordering is initially supplied to the algorithms. The algorithms then begin to reorder the test suite using their respective approaches, but cease reordering once they can no longer improve their respective metrics. In other words, once the approaches cannot improve coverage (in the case of additional-block coverage) or decrease the distance from value computation to use in a test oracle (in the case of oracle-centric), all remaining test cases are appended to the ordered test suite in their original order. For example, if a test suite contains 10 test cases, and additional-basic block coverage need only select three test cases to achieve the maximum achievable basic block coverage; the other seven test cases will be appended in the order they originally appeared.

The foregoing procedure was followed in order to avoid spurious differences in the effectiveness metric when prioritizing the same test suite using different approaches. As each prioritization technique stops reordering when useful information is exhausted, when differences in effectiveness exist we can be sure the prioritization technique is the cause, not the random ordering of the test suite.

(for example) random ordering of the test suite after maximum coverage is achieved.

2) Dependent Variable: To investigate our research questions we need to measure the benefits of the various prioritization techniques in terms of rate of fault detection. To measure rate of fault detection we use the APFD (Average Percentage of Faults Detected) metric [4], which measures the weighted average of the percentage of faults detected over the life of a test suite. APFD values range from 0 to 100; higher numbers imply faster (better) fault detection rates. More formally, let T be a test suite containing n test cases, and let F be a set of m faults revealed by T. Let TF_i be the index of the first test case in ordering TO of T that reveals fault i. The APFD for test suite TO is given by the equation:

$$APFD = 1 - \frac{TF_1 + TF_2 + ...TF_N}{nm} + \frac{1}{2n}$$

Given this equation, the earlier a fault is detected (i.e., the smaller each TF_i is), the higher APFD will be. Thus, orderings in which faults tend to be detected earlier will be rated higher than orderings in which faults tend to be detected later.

C. Controlled Factors

We control for two factors that potentially impact the effectiveness of prioritization: test suites and faults. Both factors are controlled by using randomized construction of inputs/programs.

1) Test Suites: Each case example was developed in conjunction with industrial partners at an earlier date; this particular study had not yet been planned, and thus no test suites were constructed during those collaborations. Accordingly, we needed to generate test suites (with test oracles) for use in prioritization. As this is an initial study, we do not know what test suite factors contribute to the effectiveness of our prioritization approach. We therefore wished to generate many test suites with varying properties. In this study, we used random test suite generation, as this allowed us to (1) arbitrarily vary test suite properties and (2) easily generate large numbers of test suites.

Synchronous reactive systems operate in three distinct steps: (1) accepting input, (2) updating internal state, and (3) producing output. These steps are repeated indefinitely until the systems terminate. Inputs are restricted to scalar values (floating point numbers, integers, and so forth). As a result of

this behavior, construction of test inputs is straightforward: one or more steps are defined, followed by one or more assertions of expected behavior.

To generate test suites, we used a simple naive random test suite construction approach for creating JUnit tests, allowing us to vary (1) individual test case length, (2) test suite size, and (3) the number of assertions used in each test cases's oracle. Each test oracle is an *expected value test oracle*, specifying one or more concrete expected values for each test input. Each expected value is derived from the original system, and the oracle is constructed using *assertEquals* from JUnit. We present an example test input from WBS in Figure 3.

```
@Test
public void test1() {
  WBS wbsObj = new WBS();
  wbsObj.update(-207, true, false);
  wbsObj.update(1621, true, false);
  assertEquals(1,wbsObj.Nor_Pressure);
}
```

Fig. 3. Example WBS JUnit Test

In our study, for each case example we generated 25 sets of test suites, each containing between 10 and 30 test cases. Each test case contained between 1 and 5 steps (i.e. inputs), with 1 to N assertions comprising the test oracle, where N is the number of variables computed by the system. These numbers were chosen to yield a large number of test suites with varying properties; if our approach's relative effectiveness varies according to some test suite property, we may be able to understand how.

2) Faults: To conduct our study, we required a large pool of faulty program versions. These program versions are needed in order to determine how effective each prioritization technique is. To generate faults we used Sofya, [1] a code analysis platform that provides a bytecode-based mutation generation tool. Sofya implements the following mutation operators:

Arithmetic Operator Change (AOP) Replaces an arithmetic operator with other arithmetic operators.

Logical Connector Change (LCC) Replaces a logical connector with other logical connectors.

Relational Operator Change (ROC) Replaces a relational operator with other relational operators.

Argument Order Change (AOC) Changes the order of arguments in a method invocation, if there is more than one argument. The change is applied only if arguments have the appropriate type.

In our study, for each case example, we randomly generated a large set of mutants; 250 for WBS, 500 for ASW, and 1000 for FGS. Larger sets of mutants were used with larger systems to ensure an even sampling across mutation types and expressions.

Naturally, during testing, we do not expect to detect 1000 faults, and thus to use the entire set of generated mutants would be unrealistic. Therefore, for each case example, we generated 50 subsets of mutants, with each subset containing 5-15 mutants. When constructing subsets we selected only mutants that were detected by at least one test input (from any test suite). This ensured that for each mutant subset, each mutant could be detected. This does not impact our conclusions: we are comparing the relative effectiveness of prioritization techniques, and removing mutants that cannot be detected does not impact relative APFD scores.

3) Experimental Steps: We performed the following steps for each case example:

1) Generate the mutant set.
2) Generate 50 randomly sampled mutant subsets.
3) Generate 25 random test suites.
4) Apply each prioritization technique to each test suite.
5) Compute the mutants detected by each test input, for each sampled mutant subset, and record this in a fault matrix that plots test inputs against mutants they detect.
6) Measure the APFD for each combination of a prioritized test suite and a mutant subset (1250 per technique).

D. Threats to Validity

External: Our study is limited to three synchronous reactive systems translated from Simulink. Nevertheless, we believe these systems are representative of an important class of software systems. In our study, we have used randomly generated JUnit test suites. Random test suite generation allows us to directly control for factors such as test suite size and oracle construction. However, there are many other methods for constructing test suites that could be used (structural test generation, manual generation, etc.) and such methods could lead to different conclusions. We thus avoid generalizing our results to other types of test suites and leave that for future work. The test oracles used in our study are expected value test oracles. Our results may differ when using different test oracles, such as formal specifications. However, in practice, expected value tests oracles are commonly used in testing approaches, particularly within our domain of interest.

Internal: Our greatest concern is problems with our instrumentation, and thus we have manually cross-validated our analysis programs on small examples and manually validated random selections from the real results.

Construct: Our measurement of effectiveness is APFD. It is possible that other metrics such as more comprehensive economic models [20] could yield different results. However, APFD intuitively captures the goal of prioritization: rapid fault detection over time. As this is an initial study, understanding how well our technique compares to other approaches in terms of rapid fault detection is our primary concern.

Conclusion: For each case example and prioritization technique, we have created 1250 APFD measurements. It is possible that larger numbers of measurements might yield statistically significant insights that are currently undetected (e.g, the relationship between test input length and relative effectiveness), but as shown in Section V, this number of measurements is sufficient to achieve statistical significance by a wide margin for our main research questions.

[1] sofya.unl.edu

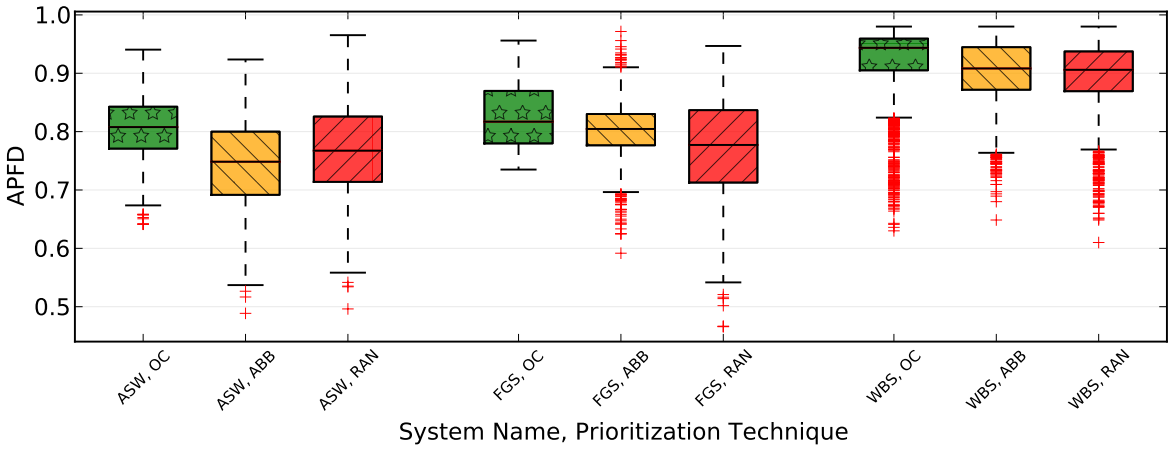

Fig. 4. APFD Boxplot for Each System, Prioritization Method

TABLE II
APFD RESULTS

	Oracle-Centric		Block Coverage		Random			
	μ APFD	σ AFPD	μ APFD	σ AFPD	μ APFD	σ AFPD	Oracle-Block % Imp.	Oracle-Random % Imp.
ASW	80.5	5.43	74.4	7.50	76.6	8.13	6.52%	4.08%
WBS	82.7	5.26	82.1	8.49	82.1	9.30	1.10%	1.21%
FGS	82.3	5.55	80.1	4.94	76.9	8.64	2.71%	6.25%

V. RESULTS AND ANALYSIS

We now discuss the results of our study in the context of our two research questions: *RQ1*, "Is oracle-centric prioritization more effective than random prioritization?", and *RQ2*, "Is oracle-centric prioritization more effective than additional-block-coverage prioritization?".

In Figure 4, we plot the computed APFD values for each test suite and mutant subset combination as a boxplot. In Table II, for each case example, we list the mean and standard deviation in computed APFD for each prioritization technique, and the average relative improvement in APFD values observed when using oracle-centric prioritization instead of random or additional-block coverage prioritization.

As the results show, on average, oracle-centric prioritization outperforms both random and additional-block-coverage prioritization for each case example, with improvements ranging from 1.10% to 6.52%. This is clearly seen in Figure 4, in which the median APFD for each case example is highest when using oracle-centric prioritization.

Our results suggest that, for the systems considered, the oracle-centric approach can be a generally preferable method for prioritizing test cases. However, as seen in Figure 4, for each case example, a fair amount of overlap exists in the APFD distribution across each prioritization method. We thus wished to determine whether our results were statistically significant, i.e., if our results were likely due to chance. We began by restating our research questions as statistical hypotheses[2]:

H_1 The APFD score when using oracle-centric prioritization is higher than the APFD score when using random prioritization.

H_2 The APFD score when using oracle-centric prioritization is higher than the APFD score when using additional-basic-block coverage.

Our observations are drawn from an unknown distribution; therefore, we cannot fit our data to a theoretical probability distribution. To evaluate our hypotheses without any assumptions about the distribution of our data, we use the two-tailed bootstrap paired permutation test (a non-parametric test with no distribution assumptions [21]) with 250,000 samples, with median as the test statistic. Per our experimental design, each APFD score produced by a combination of a set of mutants, a test suite, and a prioritization technique can be paired with the APFD produced by the same combination of mutants and test cases and a different technique. Thus, for each example, to evaluate H_1 and H_2, we construct 1250 pairs (50 mutant sets * 25 test suites) for each hypothesis. We then apply the statistical test at $\alpha = 0.05$.

Table III provides the results of our statistical tests. As shown, our results indicate that we can reject the null hypotheses for all case examples with $p < 0.05$ (and often much lower).[3] Because the mean APFD for the oracle-centric technique is always higher than the mean APFDs for random and additional-block-coverage techniques, we accept H_1 and H_2 for each case example.

Thus, with respect to our original research questions, we conclude that our results statistically support the assertion that oracle-centric prioritization is, with respect to APFD, more

[2]The null hypotheses are straightforward and omitted for reasons of space.

[3]Note that we do not generalize across case examples as the appropriate statistical assumption — random selection from the population of case examples and coverage criteria — is not met.

TABLE III
PERMUTATION TEST P-VALUES

	Oracle vs./ Block	Oracle vs./ Random
ASW	< 0.001	< 0.001
WBS	0.008	0.019
FGS	< 0.001	< 0.001

effective than random or additional-block-coverage techniques. In the next section, we discuss the implications of our results and factors that may contribute to them.

VI. DISCUSSION

A. Impact of Test Suite and Oracle Size

Our results demonstrate that oracle-centric prioritization is an effective method of prioritization with respect to APFD, outperforming additional-block-coverage and random prioritization with statistical significance. Obviously, when prioritizing test cases, we should select the technique that is likely to produce the best results, and thus generally we should select oracle-centric prioritization for this class of systems. However, if we can understand when oracle-centric prioritization can be expected to outperform other approaches, we can refine our selection.

As noted in Section IV, during test suite construction two key factors were varied: the length of the test input and the number of expected values defined. To examine the impact of these factors on relative effectiveness, we computed the Spearman correlation [21] between each factor and the relative effectiveness of oracle-centric prioritization over additional-basic-block and randomized prioritization (defined as the difference in APFD). However, our results were unable to detect a clear pattern in the randomly generated data, with the correlation between APFD and average test input length or oracle size being weak ($< |0.3|$).

We then generated 80 new test suites: 40 suites in which oracle size, which means the number of variables used in the oracle data set, was controlled (i.e., the same test oracle size was used for each input), and 40 suites where test input length was controlled. (Other factors remained randomized, e.g., random test suite lengths were used when constructing the suites controlled for oracle size.) Using these test suites, we reran our study to compute the relative improvement in APFD for each test suite. Finally, we recalculated our measurement of correlation between relative improvement and our factors of interest, test input length and test oracle size. Again, we found little correlation ($< |0.3|$) between relative APFD and our factors of interest.

Based on these results, we believe that oracle-centric prioritization's relative effectiveness does not relate to average test input length or oracle size, and thus — somewhat to our surprise — using larger or smaller test oracles does not lead to improvements in the technique. This is a somewhat unintuitive results, but hints that the obvious properties of the test oracle such as the number of assertions, the type of the assertions, etc., may not strongly impact the effectiveness of oracle-centric prioritization. Additional studies controlling for other factors, such as program domain, program complexity, and test input construction method will be required to better understand when oracle-centric prioritization is likely to outperform other approaches.

B. Effectiveness of Random Prioritization

As shown in Figure 4 and Table II, in some circumstances random prioritization fared well relative to additional-basic-block prioritization. This was most noticeable for ASW for which the average relative improvement over random prioritization is slightly less than the improvement over additional-basic-block coverage (4.08% versus 6.52%). (Note that these differences in average effectiveness are not statistically significant.)

Initially, we were surprised by this result, as based on previous prioritization studies such as those described in [4], we expected additional-block-coverage to be better than simple randomized prioritization. We then reviewed the coverage data for each test case, and computed several metrics, namely: the average basic block coverage achieved by a test case; the standard deviation in basic block coverage across test cases; the average percentage of inputs covering each basic block; and the standard deviation in the percentage of inputs covering each basic block. We list these results in Table IV.

Examining these results, we see that the average percentage of basic blocks covered by each test case is generally quite high (up to 64.4%), while the standard deviation is generally low (as low as 3.1%). Furthermore, we see that within the set of coverable basic blocks, each block is covered by a large number of test cases (average 82.2-95.2%).[4] To understand the ease with which high coverage is achieved, and the larger percentage of test cases covering each block, remember that these systems are constructed as a sequence of computations. Thus, with the exception of blocks in deeply nested branches, most blocks are easily covered by arbitrary test cases.

In short, most test cases achieve high levels of coverage, and most basic blocks are easily covered. Thus, when prioritizing test cases, coverage-based approaches like additional-block-coverage often have few options, as test cases are difficult to distinguish using test coverage information alone.

We draw two observations from this. First, these results somewhat reflect previous results within this domain find that random testing is often just as powerful as structural coverage-based test case generation approaches [22], [23]. Random testing, contrary to intuition, can often achieve high levels of structural coverage easily.[5]

Second, and more important, these results highlight the need to explicitly considering test oracles on the testing process, and on prioritization in particular. Structural coverage-based methods have long been used as a proxy for testing effectiveness,

[4]We observed a larger standard deviation due to a non-normal distribution. Most basic blocks are covered by nearly all test cases (90%+), while a small number of blocks are covered by most test cases (50%+). Due to space limitations we cannot present this data.

[5]These previous results explored the impact of the method of test case generation, but the underlying reason — the ease of achieving high levels of coverage for this class of systems — remains the same.

TABLE IV
BREAKDOWN OF COVERAGE INFORMATION

	μ % TI Block Cov.	σ % TI Block Cov.	μ % Covering TI	σ % Covering TI
ASW	64.4%	3.1%	95.2%	13.1%
WBS	43.0%	3.5%	91.8%	16.6%
FGS	57.5%	6.9%	82.2%	24.2%

but the information conveyed by these criteria is only part of effectiveness, and in some scenarios (such as this one) can yield little information. By moving beyond these approaches to those that better consider *how* faults can be detected — in other words, by explicitly considering the impact of the test oracle — we can improve the effectiveness not only of prioritization approaches, but potentially many other testing tasks.

VII. RELATED WORK

A wide range of prioritization techniques have been proposed and studied. As mentioned in Section II, initially, most techniques depended on code coverage information to drive the prioritization [4], [5], [6], [7], [24], [25], [26], [27]. Restricting the foregoing techniques to consider coverage of changed components has also been explored [4], [6].

More recently, several prioritization techniques that go beyond the use of code coverage information have also been proposed. Leon and Podgurski [28] present prioritization techniques based on distributions of execution profiles. Li et al. [29] present search-based prioritization algorithms. Korel et al. [8], [30] propose prioritization techniques based on coverage of system models. Yoo et al. [9] study the use of expert knowledge for prioritization and propose clustering test cases into similar groups to facilitate the process. Hou et al. [31] study prioritization of test cases when testing web service software and Sampath et al. [32] study prioritization strategies for web applications. Sherriff et al. [33] use change history to gather change impact information and prioritize test cases accordingly. Malishevsky et al. [34] present a prioritization technique that uses coverage information along with data on test execution times and estimated fault severities, and Park et al. [35] propose a technique for estimating these times and severities using historical information. Mirarab and Tahvildari [36] present techniques that use Bayesian networks to prioritize test cases. Walcott et al. [37] present a technique that combines information on test execution times with code coverage information, and utilizes a genetic algorithm to obtain test case orderings. Zhang et al. [38] use similar input data and utilize integer linear programming for ordering test cases. Alspaugh et al. [39] study the application of several knapsack solvers to prioritization. Finally, Rummel et al. [40] present the only dataflow based approach we are aware of, using definition-use coverage to prioritize test inputs. In each of the foregoing cases, the research considers aspects of code and executions without taking oracles into account. We have chosen to extend a basic coverage-based technique to achieve our goals, but it is likely that analogous extensions could be applied to these prior techniques as well.

The prioritization approaches most similar to ours are those of Fraser and Wotawa [41] and Jeffrey and Gupta [42]. Fraser and Wotawa prioritize in terms of property relevance, in which a model checker is used to determine how relevant each property is to each test case. As this approach is designed for use with a model-checker, it may be quite expensive in practice. Jeffery and Gupta use relevant slices to determine the portion of the code relevant to the output (i.e., the oracle), and then prioritize to favor large slices where most executed statements are relevant to the output. Their approach uses total weighting to order test inputs, as opposed to the additional-coverage used by our oracle-centric prioritization, and thus does not consider how test cases interact. Furthermore, their study also omits a comparison with additional-coverage based approaches, as well as any analysis of statistical significance.

As noted in Section II, work empirically considering test oracles in testing is comparatively sparse (though some work discussing the importance does exist [43]). To the best of our knowledge, existing metrics for oracle effectiveness rely on expensive mutation testing based approaches (Fraser et al. [2], Staats et al. [3], and Voas [12]). Schuler and Zeller et al. [44] demonstrate the potential for dynamic slicing to measure test oracle effectiveness) but no firm method for employing this has been proposed. To date, none of this work has been connected with test case prioritization.

VIII. CONCLUSION AND FUTURE WORK

We have proposed a method for prioritizing a test suite based on the impact of each test input on the test oracle, as opposed to traditional structural-coverage metric base approaches. This approach brings the benefits of explicitly considering test oracles, recently explored in test generation work, to test case prioritization. We have conducted a study using three systems from the critical avionics domain, and have demonstrated that oracle-centric test prioritization can yield improvements in APFD, with relative improvements up to 6.52% over additional-basic block coverage.

While our results are positive, they raise additional questions concerning how test oracles can best be leveraged to improve test case prioritization. In particular, a better understanding of how our proposed method's parameters impact effectiveness, as well as the impact of test suite properties such as oracle power, test input length, and so forth, may yield insights concerning how to further improve test case prioritization. We intend to explore these questions in future work.

IX. ACKNOWLEDGEMENTS

We thank Michael Whalen for discussions concerning fault propagation behavior when testing critical reactive systems.

This work is supported in part by the 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), the National Science Foundation through award CNS-0720757, and the Air Force Office of Scientific Research through award FA9550-10-1-0406.

REFERENCES

[1] M. Staats, M. Whalen, and M. Heimdahl, "Programs, tests, and oracles: The foundations of testing revisited," in *Proc. of the 33rd Int'l Conf. on Software Engineering*. IEEE, 2011.

[2] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," in *Proc. of the 19th Int'l Symp. on Software Testing and Analysis*. ACM, 2010.

[3] 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. of the 34rd Int'l Conf. on Software Engineering*. IEEE, 2012.

[4] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Trans. on Software Engineering*, vol. 27, no. 10, 2001.

[5] H. Do, G. Rothermel, and A. Kinneer, "Prioritizing JUnit test cases: An empirical assessment and cost-benefits analysis," *Empirical Software Engineering: An Int'l Journal*, vol. 11, 2006.

[6] A. Srivastava and J. Thiagarajan, "Effectively prioritizing tests in development environment," *ACM SIGSOFT Software Engineering Notes*, vol. 27, 2002.

[7] W. Wong, J. Horgan, S. London, and H. Agrawal, "A study of effective regression testing in practice," in *Proc. of the Int'l Symp. on Software Reliability Engineering*, 1997.

[8] B. Korel, G. Koutsogiannakis, and L. Tahat, "Application of system models in regression test suite prioritization," in *Proc. of the Int'l Conf. on Software Maintenance*, 2008.

[9] S. Yoo, M. Harman, P. Tonella, and A. Susi, "Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge," in *Proc. of the Int'l. Symp. on Software Testing and Analysis*, 2009.

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

[11] M. Staats, M. Whalen, and M. Heimdahl, "Nier 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.

[12] J. Voas, "PIE: A dynamic failure-based technique," *IEEE Trans. on Software Engineering*, vol. 18, no. 8, 1992.

[13] P. Frankl and E. Weyuker, "An applicable family of data flow testing criteria," *IEEE Trans. on Software Engineering*, vol. 14, no. 10, pp. 1483–1498, 1988.

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

[15] E. Gamma and K. Beck, *Contributing to Eclipse: Principles, Patterns, and Plugins*. Addison Wesley Longman Publishing Co., Inc., 2003.

[16] J. Jones and M. Harrold, "Test-suite reduction and prioritization for modified condition/decision coverage," *Software Engineering, IEEE Trans. on*, vol. 29, no. 3, pp. 195–209, 2003.

[17] J. Sztipanovits and G. Karsai, "Generative programming for embedded systems," in *Proc. of Int'l. Conference on Principles and Practice of Declarative Programming*, vol. 6, no. 08. Springer, 2002.

[18] A. Danial, "Cloc–count lines of code," *http://cloc.sourceforge.net/*.

[19] SAE-ARP4761, *Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*. SAE International, December 1996.

[20] H. Do and G. Rothermel, "Using sensitivity analysis to create simplified economic models for regression testing," in *Proc. of the 17th Int'l Symp. on Software Testing and Analysis*, 2008.

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

[22] M. Staats, G. Gay, M. Whalen, and M. Heimdahl, "On the danger of coverage directed test case generation," in *Proc. of Fundamental Approaches to Software Engineering*, 2012.

[23] A. Arcuri, M. Z. Z. Iqbal, and L. C. Briand, "Formal analysis of the effectiveness and predictability of random testing," in *Proc. of the 19th Int'l Symp. on Software Testing and Analysis*, 2010.

[24] H. Do, G. Rothermel, and A. Kinneer, "Empirical studies of test case prioritization in a JUnit testing environment," in *Proc. of the Int'l Symp. on Software Reliability Engineering*, 2004.

[25] S. Elbaum, A. Malishevsky, and G. Rothermel, "Prioritizing test cases for regression testing," in *Proc. of the Int'l. Symp. on Software Testing and Analysis*, 2000.

[26] ——, "Test case prioritization: A family of empirical studies," *IEEE Transactions on Software Engineering*, vol. 28, pp. 159–182, 2002.

[27] J. Jones and M. Harrold, "Test suite reduction and prioritization for modified condition/decision coverage," *IEEE Trans. on Software Engingeering*, vol. 29, 2003.

[28] D. Leon and A. Podgurski, "A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases," in *Proc. of the Int'l Symp. on Software Reliability Engineering* , 2003.

[29] Z. Li, M. Harman, and R. Hierons, "Search algorithms for regression test case prioritization," *IEEE Trans. on Software Engingeering*, vol. 33.

[30] B. Korel, L. Tahat, and M. Harman, "Test prioritization using system models," in *Proc. of the Int'l Conf. on Software Maintenance*, 2005.

[31] S. Hou, L. Zhang, T. Xie, and J. Sun, "Quota-constrained test case prioritization for regression testing of service-centric systems," in *Proc of the Int'l Conf. on Software Maintenance*, 2008.

[32] S. Sampath, R. Bryce, G. Viswanath, V. Kandimalla, and A. Koru, "Prioritizing user-session-based test cases for web applications testing," in *Proc. of the Int'l Conf. on Software Testing, Verification, and Validation*, 2008.

[33] M. Sherriff, M. Lake, and L. Williams, "Prioritization of regression tests using singular value decomposition with empirical change records," in *Proc. of the Int'l Symposium on Software Reliability Engineering*, 2007.

[34] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization," in *Proc. of Int'l Conf. on Software Engineering*, 2001.

[35] H. Park, J. Ryu, and J. Baik, "Historical value-based approach for cost-cognizant test case prioritization to improve the effectiveness of regression testing," in *Proc. of the Int'l Conf. on Secure System Integration and Reliability Improvement*, 2008.

[36] S. Mirarab and L. Tahvildari, "A prioritization approach for software test cases on Bayesian Networks," in *Proc. of the Int'l Conf. on Fundamental Approaches to Software Engineering , LNCS 4422-0276*, 2007.

[37] A. Walcott, M. Soffa, G. Kapfhammer, and R. Roos, "Time-aware test suite prioritization," in *Proc. of the Int'l Symposium on Software Testing and Analysis*, 2006.

[38] L. Zhang, S. Hou, C. Guo, T. Xie, and H. Mei, "Time-aware test-case prioritization using integer linear programming," in *Proc. of the 18th Int'l Symp. on Software Testing and Analysis*. ACM, 2009.

[39] S. Alspaugh, K. Walcott, M. Belanich, G. Kapfhammer, and M. Soffa, "Efficient time-aware prioritization with knapsack solvers," in *Proc. of the ACM Int'l Workshop on Empirical Assessment of Software Engineering Languages and Technologies*, 2007.

[40] M. Rummel, G. Kapfhammer, and A. Thall, "Towards the prioritization of regression test suites with data flow information," in *Proc. of the 2005 ACM Symp. on Applied computing*. ACM, 2005.

[41] G. Fraser and F. Wotawa, "Test-case prioritization with model-checkers," in *25th Conf. on IASTED Intl.*, 2007.

[42] D. Jeffrey and N. Gupta, "Test case prioritization using relevant slices," in *Computer Software and Applications Conf., (COMPSAC) 2006.*, vol. 1. IEEE, 2006.

[43] E. Weyuker, "On testing non-testable programs," *The Computer Journal*, 1982.

[44] D. Schuler and A. Zeller, "Assessing oracle quality with checked coverage," in *Software Testing, Verification and Validation (ICST), 2011*. IEEE, 2011.