2014 IEEE International Conference on Software Testing, Verification, and Validation

An Empirical Analysis of Test Oracle Strategies for Model-based Testing

Nan Li and Jeff Offutt
Software Engineering
George Mason University
{nli1,offutt}@gmu.edu

***Abstract*—Model-based testing is a technique to design abstract tests from models that partially describe the system's behavior. Abstract tests are transformed into concrete tests, which include test input values, expected outputs, and test oracles. Although test oracles require significant investment and are crucial to the success of the testing, we have few empirical results about how to write them. With the same test inputs, test oracles that check more of the program state have the potential to reveal more failures, but may also cost more to design and create.**

This research defines six new test oracle strategies that check different parts of the program state different numbers of times. The experiment compared the six test oracle strategies with two baseline test oracle strategies. The null test oracle strategy just checks whether the program crashes and the state invariant test oracle strategy checks the state invariants in the model.

The paper presents five main findings. (1) Testers should check more of the program state than just runtime exceptions. (2) Test oracle strategies that check more program states do not always reveal more failures than strategies that check fewer states. (3) Test oracle strategies that check program states multiple times are slightly more effective than strategies that check the same states just once. (4) Edge-pair coverage did not detect more failures than edge coverage with the same test oracle strategy. (5) If state machine diagrams are used to generate tests, checking state invariants is a reasonably effective low cost approach. In summary, the state invariant test oracle strategy is recommended for testers who do not have enough time. Otherwise, testers should check state invariants, outputs, and parameter objects.

I. Introduction

A primary goal of software testing is to find faults by running tests. Whether tests can find faults depends on two key factors: *test inputs* and *test oracles*. In our context, test inputs consist of method calls to a system under test (SUT) and necessary test values. A *test oracle* determines whether a test passes. An example of a test oracle is an assertion in JUnit tests. Exhaustively enumerating all test inputs is effective at finding faults, but is prohibitively expensive. As a compromise, tests are usually created to satisfy a *coverage criterion*. A *coverage criterion* is a rule or a set of rules that are applied to software artifacts (source code, models, etc.) to create a set of test requirements that have to be covered by tests [1]. A more effective test coverage criterion often results in more test inputs to detect more faults than a weaker criterion. When tests are executed, a fault may be triggered to produce an error state, which then propagates to be revealed as a failure that can be observed by checking program states (outputs and internal state variables). Testers are likely to observe more failures by checking more program states. As suggested by Briand et al. [3], we define a *test oracle strategy* (abbreviated as OS) as a rule or a set of rules to specify which program states to check. The theory is the more program states are checked, the more faults an OS is likely to reveal [3], [19], [22], [23].

In model-based testing (MBT), a model (for example, a UML state machine diagram) partially specifies the behaviors of a system. Abstract tests are generated to cover test requirements imposed by a coverage criterion. For instance, edge coverage requires all transitions in a UML state machine diagram to be covered. Thus, an abstract test may look like: "transition 1, state invariant 1, ..., transition n, state invariant n." These abstract tests need to be converted into concrete tests. Properties of models such as state invariants in a state machine diagram can be used for OSes. If test oracle data, including expected test values, is very well specified by some specification language and additional information used to transform abstract test oracle data to executable code has been provided, the concrete test oracles can be generated automatically. Such test oracles are called *specified test oracles* [8] because the specification of a system is used as a source to generate test oracles, including expected values.

As pointed out by Harman et al. [8], automated test oracles are not available in many situations. For model-based testing, transformation from abstract tests to concrete tests requires that a model has to be very well specified using additional information. The information may include specification languages such as the object constraint language (OCL) and other additional diagrams and mapping tables to map abstract information to executable code. Such complicated requirements are usually not easy to realize in practice. Thus, most practitioners cannot use automated test input and oracle generation [12]. This is particularly true when agile processes are used. Therefore, most testers have to provide expected values manually for test oracles.

A test oracle must address observability. *Observability* is how easy it is to see a program's internal state variables and outputs [6]. If a test oracle checks more program states, the observability of the program states is increased, and more faults may be revealed. However, writing test oracles can be costly because testers usually provide expected values manually. Model-based testing can use state invariants from state machine diagrams as test oracles. This paper starts with that premise and asks the following questions. Is checking only the state invariants good enough? Should testers also check class variables? What is the cost of checking more of the program state?

978-0-7695-5185-2/14 $31.00 © 2014 IEEE
DOI 10.1109/ICST.2014.49

We previously developed a test automation framework to transform abstract tests to concrete tests [9], [10], [12]. This paper extends that work by studying OSes for system-level tests in model-based testing. This paper proposes six novel OSes. Each OS checks different outputs and internal program states such as class variables after each transition or at the end of a test. The new OSes are defined in section III.

We evaluated the effectiveness and cost of the new OSes based on the same test inputs. 16 open source and example programs, with UML state machine diagrams, were used. Test inputs were generated to satisfy edge coverage (EC), which covers all transitions, and edge-pair coverage (EPC), which covers all pairs of transition [1]. EC and EPC differ when at least one node as indegree and outdegree greater than one. Then by generating test oracle data for all OSes, we designed 16 sets of tests for each program (2 coverage criteria * 8 OSes). Then we ran the tests against faulty versions of the programs.

This research had five conclusions for using OSes in model-based testing. First, just checking runtime exceptions misses many faults and wastes much of the testing effort. Second, OSes that check more program states were not always more effective at finding faults than OSes that check fewer program states. Third, OSes that check program states multiple times were only slightly more effective than OSes that check the same program states once. Fourth, with the same program states, a test set that satisfies a stronger coverage criterion was not more effective at finding faults than tests from a weaker coverage criterion. Fifth, if state machine diagrams are used to generate tests, checking state invariants is a reasonably effective low cost approach. To achieve higher effectiveness, testers can check outputs and parameter objects.

The contents of the paper are as follows. Section II introduces the background of test oracles and discusses related work. Section III presents all eight OSes and how tests were created. Section IV shows the experimental design, subjects, procedure, and results and also has a discussion of the results and possible threats to validity. Finally, section V presents conclusions and discusses future work.

II. Background and Related Work

An important property of an OS is its *precision* [3]. The precision of an OS refers to the degree to which the internal state variables and outputs of a program are checked by this OS. The more internal state variables and outputs an OS checks, the more precise the OS is. The precision of an OS also refers to the OS's tolerance for errors because if an OS checks more internal state variables and outputs, it is possible that the OS can reveal more faults. The null OS (NOS) is considered as the least precise OS since it does not check any variable or output explicitly. It only reports a failure when a program terminates abnormally, thus, it tolerates many faults. In contrast, the most precise OS checks all possible internal state variables and outputs whenever they appear during the execution of test cases, so this OS is called *the very precise OS* [3] or *the maximum OS* [22].

How and which internal state variables and outputs should be checked by an OS depends on the cost-effectiveness of OSes. Although a more precise OS can reveal more faults (effectiveness), it needs more assertions (cost) than a less precise OS. Assertions usually are written by hand, each assertion requires expected results, and adds to execution time. Keep in mind that automated test oracle generation is often not available for many situations in industry.

The frequency of checking the program state is a factor when considering the cost-effectiveness of an OS. Given two OSes that check the same internal state variables and outputs, if they find the same faults, the one that checks the states less frequently will be more cost-effective.

To our knowledge, only a few papers have studied the test oracle problem empirically. Briand et al. [3] compared the *very precise OS* with the *state invariant OS* (SIOS) based on the statecharts of three classes of less than 500 lines of code apiece. The very precise OS checks all the class attributes and outputs after each operation and is considered to be the most accurate verification possible. In contrast, SIOS only checks the invariants of states reached after each transition. They found that the very precise OS is more effective at finding faults than SIOS. They also found that the cost of the very precise OS is higher than SIOS in terms of the number of test cases, the CPU execution time, and the lines of code.

Xie and Memon [23] considered what to evaluate and how often to check program states from GUIs. They found that the variations of the two factors affect the fault-detection ability and cost of test cases. They proposed six OSes that check a widget, a window, and all windows after every event and after the last event of a test. They concluded that "weak" OSes detect fewer faults and a "thorough" OS at the end of a test case yields the best cost-effectiveness ratio in many cases.

Staats et al. [22] found that an OS that checks outputs and internal state variables can find more defects than an OS that only checks the outputs. They also concluded that the number of variables checked by the *maximum OS* is bigger than that checked by the *output-only OS* and only a small portion of the added internal state variables contributes to improving fault-detection ability. To evaluate how internal state variables affect the fault-detection ability, some *less precise OS*es were compared. A *less precise OS* checks outputs and some internal state variables but not all. In their experiment, internal variables were chosen randomly for these less precise OSes. Therefore, which internal variables contribute to improving the effectiveness is unclear.

Shrestha and Rutherford [19] empirically compared NOS and the *pre and post-condition OS* (PPCOS) using the *Java Modeling Language* [4]. They found that the latter can find more faults than the former. They suggested that test engineers should move beyond NOS and use more precise OSes.

The *test oracle comparator problem* for web applications is how to determine automatically if a web application gets correct outputs given a test case and expected results. Sprenkle et al. [20] developed a suite of 22 automated oracle comparators to check *HTML* documents, contents, and tags. They found that the best comparator depends on applications' behaviors and the faults.

Yu et al. [24] studied the effectiveness of the *output-only OS* and six other OSes that check more internal state variables to detect special faults that appear in six concurrent programs.

They found that these six more precise OSes detected more faults than the *output-only OS*.

This paper presents a comprehensive experiment to study the effectiveness and cost of OSes and give guidelines about which OS should be used. A comparison between this paper and others are in table I. The first column shows the metrics and the other columns represent others' work.

Shrestha et al. used nine small programs, with the biggest having 263 statements. Others studied no more than six programs. This research used 16 programs with lines of code (LOC) ranging from 52 to 14,155. The subjects include general libraries, GUIs, and web applications. In contrast, Xie and Memon only studied GUIs and Sprenkle et al. studied web applications. Staats et al. worked on synchronous reactive systems [7], which do not have OO classes.

This research considers several OSes: NOS, SIOS, and six more precise OSes, which is more comprehensive than the other studies. We also studied which internal state variables contribute to the effectiveness of the OSes. This research also studied the frequency of checking program states, which only Xie and Memon studied before. Eight OSes that have different precisions were used in our experiment while both Shrestha et al. and Briand et al. studied two OSes.

Staats et al. [21] and Mateo and Usaola [17] proposed similar approaches that use mutation analysis to select which program states to check automatically. But this approach could be even more costly because users have to apply mutation analysis before providing test oracle data. Thus, this approach needs further study. Fraser and Zeller [5] also used mutation testing to derive test inputs and test oracles but they did not study the effectiveness or cost of OSes.

III. Test Inputs and Oracles Generation

This section presents how tests were generated at the system level and discusses the six new and two baseline OSes.

A. Test Sequence Generation

This research evaluated different OSes with the same tests. The tests were generated from UML state machine diagrams of 16 Java programs using the structured test automation language framework (STALE) [10]. STALE can read UML state machine diagrams and transform them into general graphs. Given a graph coverage criterion, STALE can generate abstract test paths to satisfy the coverage criterion. The abstract tests are composed of transitions and state invariants. To transform the abstract tests to concrete tests, users need to use the structured test automation language [12] to provide mappings. A mapping is a data structure that includes test inputs from distinguished elements (transitions and state invariants) to implementation. Each distinguished element from a diagram can have more than one mapping because testers need to provide as many mappings as possible to satisfy all the state invariants for a specific coverage criterion. Each mapping for an element only needs to be written once. When an element appears again in an abstract test, an appropriate mapping is selected automatically to satisfy the necessary state invariants. The concrete code of a mapping for a transition is a sequence of method calls.

The concrete test code for state invariants can be transformed to JUnit assertions, allowing each assertion to be evaluated at run-time. If an assertion evaluates to false, it means the state invariant is not satisfied by the concrete test sequences of the currently used mapping for a transition between this state and a preceding state. Therefore, the concrete test code of another mapping for the transition will be used and the state invariant is re-evaluated. This process continues until the state invariant is satisfied. If no existing mappings can satisfy a state invariant, STALE reports errors and asks the tester to provide more mappings.

Since the concrete test code of state invariants can be evaluated as JUnit assertions, the assertions can be used directly as test oracles. This is what we call SIOS. Additionally, testers can use STALE to write more assertions to check other internal state variables such as class variables and outputs. For instance, if the executable test code of a transition has a method call: "*boolean sign = classObjectA.doActionB();*", testers can write assertions to evaluate the return value of the method call *sign* and *classObjectA*'s class member variables by providing the expected test values.

STALE uses a prefix-graph based solution [11] to reduce the number of tests as well as the number of times transitions appear in the tests, which is considered as a type of *quantitative human oracle cost reduction* [8].

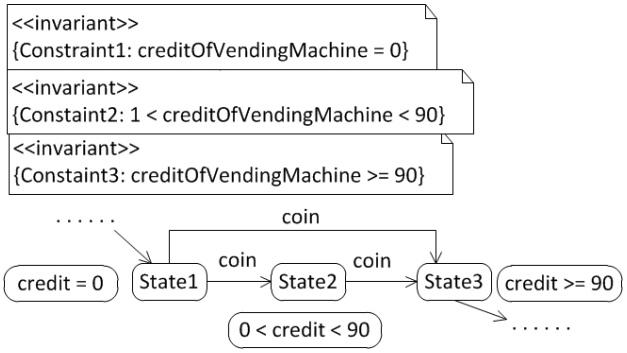

Fig. 1. Part of a state machine diagram of the vending machine example

Below we use an example of a vending machine program to show how to use STALE to generate tests and add test oracle data. The vending machine has been simplified as follows: customers insert coins to purchase chocolates; only dimes, quarters, and dollars are accepted; and the price for all chocolate is 90 cents. Figure 1 shows three states and associated transitions of a UML state machine diagram for the vending machine program, which describes how the system behaves when customers insert coins. Figure 2 shows part of the implementation of the class *VendingMachine*. This simplified state machine diagram only considers one state invariant, *credit*. The comments next to each state document the state status. Thus, *Constraint1* is in *State1*, *Constraint2* is in *State2*, and *Constraint3* is in *State3*.

STALE can read the state machine diagram and generate abstract tests. If testers choose EC, then the test requirements "State1, coin, State2," "State2, coin, State3," and "State1, coin, State3" need to be covered by the abstract tests. Testers need to provide mappings so the abstract tests can become executable

Metric	This research	Briand et al.	Xie et al.	Staats et al.	Shrestha et al.	Sprenkle et al.	Yu et al.
Number of programs	16	3	5	4	9	4	6
Types of subjects	General, GUI, Web	General	GUI	Non-OO	General	Web	Concurrent
NOS used	Yes	No	No	No	Yes	No	No
PPCOS or SIOS used	Yes	Yes	No	No	Yes	No	No
Less precise OSes used	Yes	No	Yes	Yes	No	Yes	Yes
Internal state variables are used	Yes	No	Yes	Internal state variables picked randomly	No	Yes	Yes
Frequency of checking program states	Yes	No	Yes	No	No	No	No
The number of OSes used	8	2	6	3	2	22	7

TABLE I. A COMPARISON OF PREVIOUS TEST ORACLE RESEARCH PAPERS

```
public class VendingMachine
{
    private int credit; // Current credit in the machine.
    ...
    // Constructor: vending machine starts empty.
    public VendingMachine() {}

    // A coin is given to the vendingMachine.
    // Must be a dime, quarter or dollar.
    public void coin (int coin) {}

    // Get the current credit value.
    public int getCredit () {}
    ...
```

Fig. 2. Class VendingMachine (partial)

code. To satisfy state invariants *Constraint1*, *Constraint2*, and *Constraint3*, we may have to provide multiple mappings for the transition *coin*. The concrete test code of one mapping can be "vm.coin(10);", which inserts a dime into the vending machine. *vm* is an object of class *VendingMachine* and is defined in another mapping. STALE provides a mechanism to let other mappings use this object. To satisfy the state invariants in *State2* and *State3*, we provide another mapping whose test code is "vm.coin(100);", which inserts a dollar.

Testers also provide mappings for the state invariants to evaluate if they are satisfied. For instance, test code "vm.getCredit() $\geq$ 90;" is used to evaluate if Constraint3 is satisfied. If a state invariant is not satisfied by one mapping, another mapping is selected. If no mappings can satisfy this state invariant, STALE will ask testers to enter more mappings. Testers can also provide more test oracle data to check other fields of class *VendingMachine*.

B. Test Oracle Strategies

Two OSes were used as baselines in the experiment. One is NOS, which only checks for exceptions or abnormal termination, as implicitly provided by Java runtime systems [19]. In our experience, most faults do not cause runtime exceptions, so this OS sounds trivial. It is, however, often used in industry.

The second OS is SIOS, which checks the state invariants from the state machine diagram. After testers use STALE to provide proper test mappings from abstract model elements to concrete test code, all state invariants in the state machine diagram should be satisfied. Since all the state invariants can be transformed to executable code using the provided mappings, these state invariants can be added to the tests as assertions automatically. SIOS is more precise than NOS.

This research considers two dimensions when designing OSes: how often to check states (frequency), and how many internal state variables to check (precision). Regarding the frequency, testers can check states after each method call, each transition, or they can check states only once. For precision, this research defines four elements of the program state to check:

1) *State invariants*: Check the state invariants in the model
2) *Object members*: Check member variables of objects that call methods in a transition
3) *Return values*: Check return values for each invoked method
4) *Parameter members*: Check member variables of objects that were passed to a method call as parameters

Deep checking was used for objects, that is, if an object's member was also an object, it was checked recursively until primitive variables were found.

We propose six new OSes beyond SIOS. Each new OS satisfies all the state invariants in a model and explicitly writes the satisfied state invariants as assertions. Additionally, they check more program states.

- OS1: After each transition is executed, check all distinct object members in this transition only once
- OS2: After each transition is executed, check the return values of the distinct methods only once
- OS3: After each transition is executed, check all distinct object members in this transition and the return values of the distinct methods only once
- OS4: After each transition is executed, check all distinct parameter members and the return values of the distinct methods only once
- OS5: After each transition is executed, check all distinct object and parameter members and the return values of the distinct methods only once

OT: After the last transition, check all distinct object and parameter members and the return values of the distinct methods that appear in all transitions only once

OS5 is not the most precise oracle strategy possible because it does not check outputs and variables whenever they appear in the tests. OT is the same as OS5, but with a lower frequency (one check per test). We could have designed frequency-one versions of the other OSes, but evaluating frequency for one strategy seemed enough.

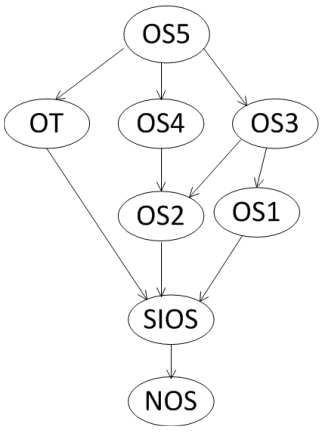

Fig. 3. Precision relationship among test oracle strategies

Figure 3 shows the precision relationships among the OSes. An arrow from one OS to another indicates that the higher OS is more precise.

IV. Experiments

The experiments address four questions:

RQ1: With the same test inputs, does a more precise OS reveal more faults than a less precise OS?

RQ2: With the same test inputs, does checking program states multiple times reveal more faults than checking the same program states once?

RQ3: With the same OS, do tests that satisfy a stronger coverage criterion reveal more faults than tests that satisfy a weaker coverage criterion?

RQ4: Which OS should be recommended when considering both effectiveness and cost?

Other researchers [3], [19], [22], [23] have studied RQ1, finding that more precise OSes are more effective than less precise OSes at revealing faults. However, they used different test coverage criteria and OSes on different types of programs, as discussed in section III.

RQ2 was evaluated by Xie and Memon [23], who found that checking variables after each GUI event can detect more faults than checking the same variables once after the last event of the test. However, their study only monitored states of GUIs such as windows. Our research checks entire program states of different kinds of programs.

Briand et al. [3] found that tests that satisfy a stronger coverage criterion can find more faults than a weaker criterion (RQ3), with the same test oracle strategy, for one program, but not the other two. Our experiment uses two different coverage criteria and 16 programs.

A very effective OS may be too costly for practical use. Thus, RQ4 considers the cost-effectiveness of the OSes. The rest of this section presents the experimental design, subjects, procedure, results, and threats to validity.

A. Experimental Design

The experiments compare the six new OSes (OS1, OS2, OS3, OS4, OS5, and OT) with the two baseline OSes, NOS and SIOS. All OSes were applied to edge-adequate and edge pair (EP)-adequate tests. Then the tests were run against faulty versions of the programs. The faults revealed and the cost of using the OSes were recorded.

Andrews et al. [2] found that synthetic faults generated using mutation testing can be used as real faults in experiments to predict the real fault detection ability of tests. In mutation testing, a *mutant* is a slight syntax change to the original program. A *mutation operator* is a rule or a set of rules that specifies how to generate mutants. If a test causes a mutated program (mutant) to produce different results from the original program, this mutant is said to be "killed." If a mutant cannot be killed by any tests, it is called *equivalent*.

The *mutation score* is the ratio of mutants that are killed over the killable (non-equivalent) mutants, which measures the effectiveness of a test set. Therefore, if different OSes are compared with the same test inputs and then the tests are run against mutants, the mutation score of each set of tests can reflect the relative effectiveness of each OS. A higher mutation score indicates the OS is more effective. A more precise OS can be expected to be at least as effective at revealing faults as a less precise OS. Thus, we expect OS1, OS2, OS3, OS4, OS5, and OT to reveal more faults than NOS and SIOS.

The experiment used muJava [16], a mutation analysis tool for Java, to generate synthetic faults. Each mutated program has only one mutant. Users can generate mutants, run tests against mutants, and view mutants, which helps testers recognize equivalent mutants. The latest version of muJava supports JUnit tests and all features of Java 1.6. So the JUnit tests that STALE generates are used in muJava directly. Mutants were generated by using the 15 selective method-level mutation operators of muJava [15].

This experiment used muJava to seed mutants, which are treated as faults. Each program has a different number of faults thus we measured the effectiveness of OSes as percentages of faults detected so that they would all be on the same scale.

Each test had eight different versions, one for each test oracle strategy. Generating and running the test oracles had three kinds of cost.

First, testers entered test oracle data as assertions by hand. Second, STALE generated tests based on these assertions. Note that each assertion was used many times in the concrete tests because the program states were checked after each transition. Third, assertions were executed as part of the tests. The second and third steps were automated, so the human cost in the first step dominates. Therefore, we used the number of

distinct assertions as a proxy for cost. This carries an implicit assumption that all assertions are equally difficult to write, a limited necessary to make this study practical.

We introduce a cost effectiveness ratio to measure the effect of writing fewer assertions on the mutation score. The cost-effectiveness of an OS is the ratio of the number of assertions over the percentage of faults detected. A smaller cost-effectiveness is better.

$$\textit{Cost-effectiveness} = \frac{\#assertionsCreatedByHand}{\%FaultsDetected} \quad (1)$$

To better specify how to measure the goals of the experiments, three groups of hypotheses are extracted from the first three research questions. The first group of hypotheses ($Hypotheses_A$) compares all pairs of OSes, OS_A and OS_B, where OS_B is more precise than OS_A. The null and alternative hypotheses are listed below.

Null hypothesis (H_0):
: There is no difference between the percentage of faults detected by OS_A and OS_B with the same test inputs.

Alternative hypothesis (H_1):
: OS_B detects more faults than OS_A with the same test inputs.

The test oracle strategy pairs that can be applied to $Hypotheses_A$ are: {NOS, SIOS}, {SIOS, OS1}, {SIOS, OS3}, {SIOS, OS5}, {OS1, OS3}, {OS3, OS5}, {OS1, OS5}, {OS2, OS5}, {OS4, OS5}, {SIOS, OS2}, {SIOS, OS4}, {OS2, OS3}, and {OS2, OS4} for both EC and EPC. We did not compare NOS with the other OSes because NOS is expected to be much less effective than other OSes. NOS and the other OSes are compared in section IV-D.

RQ2 asks if checking program states multiple times reveals more faults than checking the same program states once. OT checks the same parts of the state that OS5 checks, but OS5 checks after each transition and OT only checks once. Thus, the second hypothesis ($Hypotheses_B$) for RQ2 is:

Null hypothesis (H_0):
: There is no difference between the percentage of faults detected by OT and OS5 with the same test inputs.

Alternative hypothesis (H_1):
: OS5 detects more faults than OT with the same test inputs.

The third group of hypotheses ($Hypotheses_C$) for RQ3 takes two test coverage criteria CC_A and CC_B into consideration, where CC_B subsumes CC_A.

Null hypothesis (H_0):
: There is no difference between the percentage of faults detected by criterion CC_A and CC_B if both use the same OS.

Alternative hypothesis (H_1):
: CC_B detects more faults than CC_A if both use the same test oracle strategy.

$Hypotheses_C$ can be applied to edge-adequate and EP-adequate tests for any of the eight OSes used in this paper.

B. Experimental Subjects

We evaluated 16 Java programs and used STALE to generate tests from their UML state machine diagrams. First, we generated test inputs to satisfy both EC and EPC. Then we entered test oracle data for our six OSes (NOS did not need test oracle data, and the state invariant test oracle data were provided by STALE while generating test inputs). Finally, the eight OSes were applied to the two sets of tests that satisfy EC and EPC, resulting in 16 sets of tests for each program. Six of the 16 programs are open source projects, six are from textbooks, and the other four are from the coverage web application for Ammann and Offutt's book [1]. All programs are in Java and we generated the UML state machine diagrams by hand.

Table II shows some properties of the programs and tests. The column *LOC* shows the lines of code for each program. The columns *E* and *EPs* give the number of edges and edge-pairs for each program's FSM. The columns *Tests* show the number of tests for edge-adequate and EP-adequate tests. The columns *Trans* represent the number of transitions that appear in the tests and the columns *SI* provide the number of appearances of state invariants that are satisfied and also used as test oracles. The columns *Distinct Trans* and *Distinct SI* represent the number of distinct mappings of transitions and state invariants provided by hand.

As stated in section III-A, users need to provide mappings for transitions and state invariants so that abstract tests can be transformed to concrete tests. Since transitions and state invariants appear many times, the numbers of the columns *Trans* and *SI* are far more than those of the columns *Distinct Trans* and *Distinct SI*. By comparing the columns *Distinct Trans* for EC and EPC, we see that we only needed to provide more mappings for EPC than EC for three programs. That is, the mappings required to satisfy state invariants for EC also satisfy most of the state invariants for EPC. Thus, the results did not show much difference between EC and EPC.

C. Experimental Procedure

The experiment was carried out in the following steps:

1) For each program, the first author used STALE to create test inputs by hand to satisfy EC, and then generate additional test inputs to satisfy EPC.
2) STALE was used to enter expected results for the OSes. 16 tests were generated for each pair of combination for the two coverage criteria and eight OSes.
3) Generated faults for each program using muJava. Identified and removed equivalent mutants by hand.
4) Each set of tests was run against the faults for each program. The number of faults detected and the number of times the internal state variables and outputs are checked for each set of tests were recorded.
5) The cost-effectiveness of each OS was calculated and analyzed.

For efficiency, tests were only run against faults that appeared in methods called when the tests were run on the original program.

Programs	LOC	E	EPs	Properties of the Tests									
				Edge					Edge Pair				
				Tests	Trans	SI	Distinct Trans	Distinct SI	Tests	Trans	SI	Distinct Trans	Distinct SI
ATM	463	12	19	5	18	22	6	6	6	30	39	6	6
BlackJack	403	20	34	8	27	27	11	3	9	51	51	11	3
Calculator	2,919	76	403	14	167	167	11	9	39	893	893	11	9
CorssLexic	654	51	162	26	113	209	11	7	63	404	756	11	7
DFGraphCoverage	4,512	42	201	8	49	78	10	7	45	390	643	10	7
DynamicParser	1,269	65	213	20	116	408	13	15	26	385	1,233	14	15
GraphCoverage	4,480	59	187	16	122	207	14	11	23	359	605	14	11
JMines	9,486	28	91	9	60	6	7	1	22	201	21	7	1
LogicCoverage	1,808	62	259	30	115	94	12	8	94	561	483	12	8
MMCoverage	3,252	107	318	78	273	228	20	16	142	699	570	20	16
Poly	129	21	64	5	32	57	11	6	12	129	237	18	6
Snake	1,382	45	107	7	70	120	10	8	8	194	341	10	8
TicTacToe	665	12	20	5	24	7	6	3	7	46	16	6	3
Tree	234	24	74	6	35	48	6	3	8	99	146	6	3
Triangle	124	31	156	6	36	36	7	5	27	271	271	7	5
VendingMachine	52	26	61	7	44	88	6	6	9	105	210	7	6
Total	**31,832**	**681**	**2,369**	**250**	**1,301**	**1,802**	**161**	**114**	**540**	**4,817**	**6,515**	**170**	**114**

TABLE II. EXPERIMENTAL SUBJECTS

D. Experimental Results

We computed the faults detected by each OS for each program with both sets of tests. Then the number of faults detected by each OS were divided by the total number of faults for each program, producing percentages of faults detected by each OS for each program, as shown in Table III. The total number of the faults ("# Faults") is 9,627. So a total of 60,842,640 tests were executed ((8 OSes * 250 edge-adequate tests) + (8 OSes * 540 EP-adequate tests)) * 9,627).

Table III shows that NOS detected far fewer faults than the other OSes on average, indicating NOS is much less effective at finding faults. Since a more precise OS checks more program states than a less precise OS, we expected that the more precise OSes could find more faults. However, Table III shows that the percentages of the faults detected by OS1, OS2, OS3, OS4, OS5, and OT are very close, although slightly higher than that of SIOS.

We also wanted statistical evidence of the effectiveness difference between a less precise OS and a more precise OS. First, we used Qqplots [18] (not shown due to space) to determine that the percentages of faults detected by the OSes are not normally distributed. Then we used the one-tailed Wilcoxon signed-rank test (statistical significance level α = 0.05) [14] to compare the paired percentages of the faults detected by two different OSes for both EC and EPC. This is a non-parametric test to assess whether two population means differ when data are not normally distributed. We used this test because the paired data were from the same tests (either EC or EPC) with different OSes.

We compared the OS pairs {NOS, SIOS}, {SIOS, OS1}, {SIOS, OS3}, {SIOS, OS5}, {OS1, OS3}, {OS3, OS5}, {OS1, OS5}, {OS2, OS5}, {OS4, OS5}, {SIOS, OS2}, {SIOS, OS4}, {OS2, OS3}, and {OS2, OS4} for $Hypotheses_A$. We got the p-values from 0.0002 - 0.0027 for {NOS, SIOS}, {SIOS, OS1}, {SIOS, OS3}, {SIOS, OS5}, {OS1, OS5}, {OS2, OS5}, and {OS4, OS5}, so we can reject H_0. For these pairs, the effectiveness of a more precise OS is significantly greater than that of a less precise OS. Although we found statistical evidence of differences between OS1 and OS5, OS2 and OS5, and OS4 and OS5, these differences are very small, as shown in Table III. We also found that the differences between the OSes in three pairs {SIOS, OS2}, {SIOS, OS4}, and {OS2, OS3} are not due to chance.

For pairs {OS1, OS3}, {OS3, OS5}, and {OS2, OS4}, the number of paired data whose percentages of faults detected by OSes are different is fewer than five, so we could not perform the Wilcoxon signed-rank test. This also implies that there is no significant difference between the pairs {OS1, OS3}, {OS3, OS5}, and {OS2, OS4}. In summary, for $Hypotheses_A$, we reject H_0 for the pairs {NOS, SIOS}, {SIOS, OS1}, {SIOS, OS3}, {SIOS, OS5}, {OS1, OS5}, {OS2, OS5}, {OS4, OS5}, {SIOS, OS2}, {SIOS, OS4}, and {OS2, OS3}.

Although Table III tells us that a more precise OS can detect more faults than a less precise OS, the results of the Wilcoxon signed-rank test say that the percentage of faults detected by a more precise OS might not significantly differ from that of a less precise OS (such as {OS1, OS3}). Therefore, the answer to RQ1 is: for any two OSes that have different precision, the more precise OS might not always be more effective than the less precise OS.

We also used the Wilcoxon signed rank test for $Hypotheses_B$ to decide if the frequency of checking variables impacts the effectiveness of OSes. The results showed that we can reject H_0, and OS5 detects more faults than OT for EC and EPC at a significant level. So OS5 is more effective than OT for both EC and EPC. However, from TABLE III, we can see that the difference between the average percentages of faults detected by OT and OS5 is very small (0.59 vs. 0.61 for EC and 0.61 vs. 0.63 for EPC). OS5 only found more faults than OT for six programs (37.5% of all programs) for EPC. With

Programs	#Faults	Percentage of Faults detected by Test Oracle Strategies															
		Edge								Edge Pair							
		NOS	SIOS	OT	OS1	OS2	OS3	OS4	OS5	NOS	SIOS	OT	OS1	OS2	OS3	OS4	OS5
ATM	257	0.19	0.30	0.80	0.67	0.71	0.71	0.80	0.80	0.21	0.32	0.80	0.68	0.72	0.72	0.80	0.80
BlackJack	56	0.27	0.34	0.39	0.38	0.38	0.38	0.38	0.39	0.27	0.34	0.39	0.38	0.38	0.38	0.38	0.39
Calculator	494	0.19	0.41	0.45	0.46	0.43	0.46	0.43	0.46	0.24	0.45	0.49	0.50	0.47	0.50	0.47	0.50
CorssLexic	470	0.37	0.44	0.44	0.44	0.44	0.44	0.44	0.44	0.39	0.45	0.46	0.46	0.46	0.46	0.46	0.46
DFGraph Coverage	683	0.40	0.40	0.61	0.61	0.40	0.61	0.40	0.61	0.40	0.40	0.61	0.61	0.40	0.61	0.40	0.61
Dynamic Parser	3,378	0.51	0.68	0.68	0.68	0.68	0.68	0.68	0.68	0.51	0.68	0.68	0.68	0.68	0.68	0.68	0.68
Graph Coverage	385	0.49	0.55	0.71	0.78	0.55	0.78	0.55	0.78	0.49	0.55	0.71	0.78	0.55	0.78	0.55	0.78
JMines	263	0.25	0.25	0.26	0.30	0.25	0.30	0.25	0.30	0.77	0.77	0.78	0.78	0.77	0.78	0.77	0.78
Logic Coverage	436	0.50	0.86	0.87	0.87	0.86	0.87	0.86	0.87	0.50	0.86	0.87	0.87	0.86	0.87	0.86	0.87
MM Coverage	845	0.17	0.30	0.31	0.31	0.30	0.31	0.30	0.31	0.17	0.30	0.31	0.31	0.30	0.31	0.30	0.31
Poly	259	0.49	0.96	0.96	0.97	0.97	0.97	0.97	0.97	0.51	0.97	0.97	0.97	0.97	0.97	0.97	0.97
Snake	572	0.29	0.39	0.50	0.74	0.70	0.74	0.70	0.74	0.38	0.40	0.50	0.74	0.70	0.74	0.70	0.74
TicTacToe	1,045	0.05	0.44	0.49	0.44	0.47	0.47	0.49	0.49	0.05	0.49	0.50	0.49	0.49	0.49	0.50	0.50
Tree	113	0.21	0.53	0.62	0.62	0.57	0.62	0.42	0.62	0.29	0.59	0.62	0.62	0.62	0.62	0.62	0.62
Triangle	263	0.02	0.49	0.60	0.53	0.63	0.64	0.63	0.64	0.02	0.49	0.63	0.53	0.65	0.65	0.65	0.65
Vending Machine	108	0.00	0.60	0.77	0.69	0.83	0.83	0.83	0.83	0.00	0.61	0.78	0.71	0.84	0.84	0.84	0.84
Average	**9,627**	**0.34**	**0.54**	**0.59**	**0.60**	**0.58**	**0.61**	**0.58**	**0.61**	**0.37**	**0.56**	**0.61**	**0.62**	**0.60**	**0.63**	**0.60**	**0.63**

TABLE III. EFFECTIVENESS OF TEST ORACLE STRATEGIES

regard to RQ2, checking program states multiple times was only slightly more effective than checking the same program states once.

Table III shows that each OS for the edge-adequate tests reveals almost the same number of faults as the same OS for the EP-adequate tests. We used the one-tailed Mann-Whitney test (statistical significance level α = 0.05) [14] to look for statistical evidence that EPC is more effective than EC. The reason that we used one-tailed Mann-Whitney test rather than the Wilcoxon signed rank test was that the compared data were from two independent tests (one from EC and the other from EPC). We applied each OS to the edge-adequate and EP-adequate tests for each program and then compared the percentages of the faults detected by the two paired sets of tests that have the same OS. The p-values are between 0.3953 and 0.5823 for all the OSes, so we cannot reject H_0 for $Hypotheses_C$. Therefore, the answer to RQ3 is that EPC did not reveal more faults than EC with the same OS, although this could be because there were not significantly more edge pairs than edges.

Table IV shows how many distinct assertions were created by hand for each OS. Since a more precise OS checks more program states than a less precise OS, more distinct assertions were generated for the more precise OSes. Thus, the cost of OS5 is greater than that of any other OS except OT. Although OT checks program states less frequently than OS5, it uses the same number of distinct assertions as OS5. Furthermore, OS1, OS3, OS5, and OT require similar numbers of distinct assertions but have far more assertions than OS2 and OS4. This is because object members checked by OS1, OS3, OS5, and OT produced lots of assertions since we checked the member variables of objects recursively. For the same OS, the EP-adequate tests do not have many more distinct assertions than the edge-adequate tests. This is because we did not need to create many more mappings to satisfy EPC than the mappings created for EC, as discussed in section IV-B.

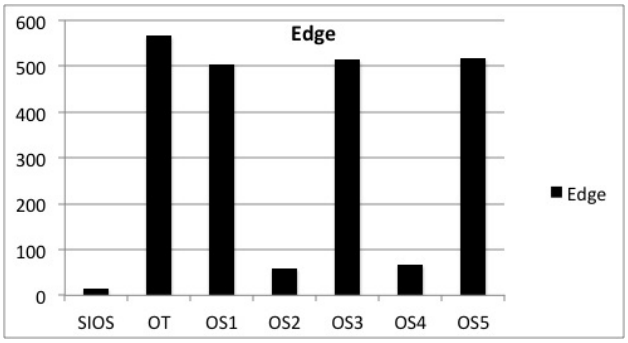

Fig. 4. The averages of cost-effectiveness of edge-adequate tests

Figure 4 gives the average cost-effectiveness from formula 1 over all the programs. The cost-effectiveness was computed for EC, not EPC, because EPC is not significantly more effective than EC for the programs and the cost of each OS for EPC is almost the same as that for EC. Since the cost of NOS is 0, we excluded it from this comparison. Figure 4 shows that SIOS is the most cost-effective, followed by OS2 and OS4.

E. Discussion and Recommendations

We found statistical evidence that the more precise OS is more effective in terms of the percentage of faults detected than the less precise OS for the following pairs: {NOS, SIOS}, {SIOS, OS1}, {SIOS, OS3}, {SIOS, OS5}, {OS1, OS5}, {OS2, OS5}, {OS4, OS5}, {SIOS, OS2}, {SIOS, OS4}, and {OS2, OS3}, but not for {OS1, OS3}, {OS3, OS5}, and

Programs	Cost of Test Oracle Strategies															
	Edge								Edge Pair							
	NOS	SIOS	OT	OS1	OS2	OS3	OS4	OS5	NOS	SIOS	OT	OS1	OS2	OS3	OS4	OS5
ATM	0	6	74	39	44	49	60	74	0	6	74	39	44	49	60	74
BlackJack	0	3	552	515	40	552	40	552	0	3	552	515	40	552	40	552
Calculator	0	9	123	123	20	123	20	123	0	9	123	123	20	123	20	123
CorssLexic	0	7	127	127	15	127	15	127	0	7	127	127	15	127	15	127
DFGraph Coverage	0	7	163	163	18	163	18	163	0	7	163	163	18	163	18	163
Dynamic Parser	0	15	23	23	15	23	15	23	0	15	23	23	15	23	15	23
Graph Coverage	0	11	156	156	26	156	26	156	0	11	156	156	26	156	26	156
JMines	0	1	792	792	82	792	82	792	0	1	792	792	82	792	82	792
Logic Coverage	0	8	181	181	10	181	10	181	0	8	181	181	10	181	10	181
MM Coverage	0	16	587	584	20	587	20	587	0	16	587	584	20	587	20	587
Poly	0	6	12	10	12	12	12	12	0	6	14	12	14	14	14	14
Snake	0	8	463	463	55	463	55	463	0	8	463	463	55	463	55	463
TicTacToe	0	3	90	32	7	36	61	90	0	3	90	32	7	36	61	90
Tree	0	3	35	23	15	35	15	35	0	3	35	23	15	35	15	35
Triangle	0	5	59	50	14	59	14	59	0	5	59	50	14	59	14	59
Vending Machine	0	6	18	17	9	17	9	18	0	6	21	19	9	20	9	21
Total	**0**	**114**	**3,455**	**3,298**	**402**	**3,375**	**472**	**3,455**	**0**	**114**	**3,460**	**3,302**	**404**	**3,380**	**474**	**3,460**

TABLE IV. COST OF TEST ORACLE STRATEGIES

{OS2, OS4}. This shows that if the precision difference of two OSes is not large enough, there is no difference between the effectiveness of the OSes.

In Table III, the percentage of faults detected (mutation score) by the EP-adequate tests for the most precise OS (OS5) in this paper is only 0.63. This is a low score considering that 90% mutation is considered a good test set. The test inputs were generated to satisfy state invariants in the state machine diagrams while mutation-adequate tests usually require more test inputs. This implies that mutation coverage is generally more effective at finding faults than EC and EPC on the model. We previously [13] found that mutation can find more faults than EPC at the unit testing level. Furthermore, the system tests generated in this paper could only call methods that are mapped to the models at a high level.

Table III shows that NOS is not very good at revealing faults. This suggests that checking runtime exceptions is not enough. We also noticed that SIOS can reveal more than 80% of the faults detected by OS5 but with many fewer assertions. Test inputs were generated to satisfy state invariants, thus checking the limited number of outputs and internal state variables used in the state invariants can reveal many faults. In contrast, checking more program states (as OS5 does) that are not affected by the test inputs are not likely to reveal more faults. If the tester wants to check more program states, checking return values or parameter members (as with OS2 and OS4) yields better cost-effectiveness than checking object members (as with OS1, OS3, OS5, and OT), because they require more cost with little increase in effectiveness.

The results show that checking program states multiple times was only slightly more effective than checking the same program states once for both EC and EPC. Our belief is that program states are changed when execution enters a different state of a state machine diagram. Therefore, most faults should be revealed if all program states are checked for each distinct diagram state, as done by OT. Checking the same program states multiple times does not seem to help much.

Moreover, we found statistical evidence that EP-adequate tests are not significantly more effective than edge-adequate tests. This may be because EPC did not require many more mappings on the state machine diagrams, even though EPC had more tests.

Checking program states frequently could require too many assertions in tests. OT results in 7,534 assertions for EC (16,782 for EPC) but OS5 produces 24,084 (83,821 for EPC). These numbers were not shown in the tables due to lack of space. Checking lots of program states could cause the size of a JUnit test method to exceed 65,536 bytes, resulting in a compiler error. Since testers must split these methods by hand, this results in additional hidden costs.

Figure 4 shows that SIOS is the most cost-effective for both EC and EPC. So if testers need to save time, SIOS is a good choice. Otherwise, testers should choose OS2 or OS4 because they are almost as effective as OS5 but require fewer assertions than OS5.

F. Threats to Validity

As in most software engineering studies, we cannot be sure that the subject programs are representative. The results may differ with other programs and models. Another threat to external validity is that we generated tests to satisfy EC and EPC. The results may differ if we used different tests. One construct validity threat is that we used muJava to generate synthetic faults. Using real faults or another mutation tool may yield different results. Another is that we approximated cost by

the number of assertions, a simplification that was required to make the experiment practical. Another internal threat is that the first author identified equivalent mutants by hand, thus, mistakes could affect the results in small ways.

V. Conclusions and Future Work

This paper proposes six new test oracle strategies to help find more faults with less cost in model-based testing. We compared these OSes with two baseline OSes: NOS and SIOS. We generated test inputs to satisfy EC and EPC from UML state machine diagrams for 16 programs. Then the eight OSes were applied to the edge-adequate and EP-adequate tests, resulting in 16 sets of tests for each program. These sets of tests were run against faults. The effectiveness and cost of each OS were recorded and the cost-effectiveness was calculated in section IV.

We had several findings. First, NOS is not very effective and testers need to check program states, not just runtime exceptions. Second, the more precise OSes did not always detect more failures than the less precise OSes. Third, an OS that checks program states multiple times was only slightly more effective than OSes that check the same states just once. Fourth, with the same OS, a stronger test coverage criterion was not more effective at finding faults than a weaker criterion. In summary, checking only runtime exceptions will result in many failures not being noticed. Checking only state invariants (SIOS) is recommended for testers who are willing to sacrifice a little quality for time; otherwise, testers should check state invariants, outputs and parameter objects (OS2 or OS4).

In the future, we plan to improve SIOS, and develop new OSes such as checking the program states in OS1, OS2, OS3, OS4 once. Using mutation analysis to select which program states to check seems promising [17], [21], but could be costly because testers have to run mutation analysis before providing test oracle data. We intend to seek a way to select OSes that are effective but less costly.

References

[1] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, Cambridge, UK, 2008.

[2] James H. Andrews, Lionel C. Briand, and Yvan Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th International Conference on Software Engineering, (ICSE 2005)*, pages 402–411, St. Louis, Missouri, May 2005. IEEE Computer Society.

[3] Lionel C. Briand, Massimiliano Di Penta, and Yvan Labiche. Assessing and improving state-based class testing: A series of experiments. *IEEE Transaction on Software Engineering*, 30(11):770–793, November 2004.

[4] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7:212–232, June 2005.

[5] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering*, 38(2):278–292, 2012.

[6] Roy S. Freedman. Testability of software components. *IEEE Transactions on Software Engineering*, 17(6):553–564, 1991.

[7] Nicolas Halbwachs. Synchronous programming of reactive systems - a tutorial and commented bibliography. In *Tenth International Conference on Computer-Aided Verification, CAV98, Vancouver (B.C.), LNCS 1427*, pages 1–16. Springer Verlag, 1998.

[8] Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. A comprehensive survey of trends in oracles for software testing. Technical Report CS-13-01, University of Sheffield, Department of Computer Science, 2013.

[9] Nan Li. A smart structured test automation language (SSTAL). In *The Ph.D. Symposium of 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)*, ICST '12, pages 471–474, Montreal, Quebec, Canada, April 2012.

[10] Nan Li. The structured test automation language framework. Online, 2013. http://cs.gmu.edu/∼nli1/stale/, last access May 2013.

[11] Nan Li, Fei Li, and Jeff Offutt. Better algorithms to minimize the cost of test paths. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 280–289, Montreal, Quebec, Canada, April 2012. IEEE Computer Society.

[12] Nan Li and Jeff Offutt. A test automation language for behavioral models. Technical Report GMU-CS-TR-2013-7, Department of Computer Science, George Mason University, Fairfax, VA, USA, 2013.

[13] Nan Li, Upsorn Praphamontripong, and Jeff Offutt. An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. In *Fifth Workshop on Mutation Analysis (Mutation 2009)*, Denver CO, April 2009.

[14] Richard Lowry. *Concepts and Applications of Inferential Statistics*. Cambridge University Press, Cambridge, UK, 2008.

[15] Yu-Seung Ma and Jeff Offutt. Description of method-level mutation operators for Java. Online, 2005. http://cs.gmu.edu/∼offutt/mujava/mutopsMethod.pdf, last access August 2013.

[16] Yu-Seung Ma, Jeff Offutt, Yong-Rae Kwon, and Nan Li. muJava home page. Online, 2013. http://cs.gmu.edu/∼offutt/mujava/, last access August 2013.

[17] Pedro Reales Mateo and Macario Polo Usaola. $Bacterio^{ORACLE}$: An oracle suggester tool. In *Proceedings of the 25th International Conference on Software Engineering and Knowledge Engineering*, SEKE 2013, June 2013.

[18] Jeremy Miles and Mark Shevlin. *Applying Regression and Correlation: A Guide for Students and Researchers*. SAGE Publications Ltd, 2000.

[19] Kavir Shrestha and Matthew Rutherford. An empirical evaluation of assertions as oracles. In *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 110–119, Berlin, Germany, March 2011. IEEE Computer Society.

[20] Sara Sprenkle, Lori Pollock, Holly Esquivel, Barbara Hazelwood, and Stacey Ecott. Automated oracle comparators for testing web applications. In *The 18th IEEE International Symposium on Software Reliability Engineering*, pages 117–126, Trollhattan, Sweden, November 2007.

[21] Matt Staats, Gregory Gay, and Mats P. E. Heimdahl. Automated oracle creation support, or: how I learned to stop worrying about fault propagation and love mutation testing. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 870–880, Piscataway, NJ, USA, 2012. IEEE Press.

[22] Matt Staats, Michael W. Whalen, and Mats P.E. Heimdahl. Better testing through oracle selection (NIER track). In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 892–895, Waikiki, Honolulu, HI, USA, May 2011. ACM.

[23] Qing Xie and Atif Memon. Designing and comparing automated test oracles for GUI-based software applications. *ACM Transaction on Software Engineering and Methodology*, 16(1), February 2007.

[24] Tingting Yu, Witawas Srisa-an, and Gregg Rothermel. An empirical comparison of the fault-detection capabilities of internal oracles. In *The 24th IEEE International Symposium on Software Reliability Engineering*, ISSRE '13, Pasadena, CA, USA, November 2013.