Test Oracle Strategies for Model-based Testing

Nan Li, *Member, IEEE,* and Jeff Offutt, *Member, IEEE*

Abstract—Testers use model-based testing to design abstract tests from models of the system's behavior. Testers instantiate the abstract tests into concrete tests with test input values and test oracles that check the results. Given the same test inputs, more elaborate test oracles have the potential to reveal more failures, but may also be more costly. This research investigates the ability for test oracles to *reveal* failures. We define ten new test oracle strategies that vary in amount and frequency of program state checked. We empirically compared them with two baseline test oracle strategies. The paper presents several main findings. (1) Test oracles must check more than runtime exceptions because checking exceptions alone is not effective at revealing failures. (2) Test oracles do not need to check the entire output state because checking partial states reveals nearly as many failures as checking entire states. (3) Test oracles do not need to check program states multiple times because checking states less frequently is as effective as checking states more frequently. In general, when state machine diagrams are used to generate tests, checking state invariants is a reasonably effective low cost approach to creating test oracles.

Index Terms—Test Oracle, RIPR Model, Test Oracle Strategy, Test Automation, Subsumption, Model-Based Testing

1 Introduction

A Primary goal of software testing is to reveal failures by running tests. Whether tests can reveal failures 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 input values. A *test oracle* determines whether a test passes. An example of a test oracle is an assertion in JUnit tests.

When tests are executed, a fault may be triggered to produce an error state, which then propagates to an external failure. This is known as the Reachability, Infection, and Propagation (*RIP*) model [12], [33], [34], [35]. But even if a fault propagates to a failure, it is only useful if the failing part of the output state is *revealed* to the tester. With manual testing, it can be assumed that all failures are revealed, but when automated, the test may not check the part of the output that is erroneous. Therefore, this paper extends the traditional RIP model to the Reachability, Infection, Propagation, and Revealability (RIPR) model. The RIPR model is discussed in detail in Section 2.2. This paper introduces a precise definition for the term *test oracle strategy* (abbreviated as OS) in Section 2.3. The informal usages of "oracle strategy" and " test oracle strategy" from previous papers (for example, reference [8]) were not sufficient for our work. The theory behind test oracle strategies is the more program states are checked, the more faults an OS is likely to reveal [8], [40], [43], [45].

When this paper talks about checking program state, or checking outputs, it uses the term in a broad sense. At the system testing level, outputs include everything that is sent to the screen, written to a file or database, or sent as messages to a separate program or hardware device. At the unit testing level, outputs include return values, method parameters whose values can be changed, and objects and variables that are shared among the method under test and other parts of the program.

In model-based testing (MBT), a model (for example, a UML state machine diagram) partially specifies behaviors of a system [36]. Abstract tests are generated to cover test requirements imposed by a coverage criterion. For example, 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." Note that the number of transitions could be different from the number of state invariants. These abstract tests need to be converted into concrete tests. State invariants in abstract tests are checked at that point in the corresponding concrete test. Properties of models such as state invariants in a state machine diagram can be used for OSes[1]. 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* [5] because the specification of a system is used as a source to generate test oracles, including expected values. Although the new concepts in this paper are independent of testing context, the experimental studies in Section 5 are in the context of model-based testing.

As pointed out by Barr et al. [5], automated test oracles are often not available. For model-based testing, automatic transformation from abstract tests to concrete tests requires that a model be very well specified using additional information. The information may include specification languages such as the object constraint language (OCL) [15] and other additional diagrams and mapping tables to map abstract information to executable code. Unfortunately, such detailed and precise requirements are often not available in practice. Thus, most practitioners cannot use automated test input and oracle generation. This is particularly true when agile processes are used because requirements are

- *N. Li is with the research and development division at Medidata solutions, New York, NY, USA. E-mail: nli@mdsol.com. Most of this work was done while the first author was at George Mason University.*
- *J. Offutt is a Professor of Software Engineering at George Mason University, Fairfax, VA, USA. E-mail: offutt@gmu.edu.*

1. "OSes" is the plural of OS.

changed often and software is released frequently. This makes it hard for testers to maintain consistency among the models. Therefore, this research assumes testers must provide expected values manually for test oracles.

An OS must address observability. *Observability* is how easy it is to see a program's internal state variables and outputs [1], [14]. If an OS checks more program states (for example, more member variables of objects, more objects, or more frequently), the observability of the program states is increased, and more failures may be revealed. However, writing test oracles can be costly when testers 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 more of the program state, such as class variables? What is the cost (in terms of writing test oracles) and benefit (in terms of finding faults) of checking more of the program state?

We defined six OSes in our previous conference paper [23], and define four additional OSes in this paper. In this research, we compared all ten OSes with two baseline OSes in the context of model-based testing. These OSes will be defined in Section 4.2. We evaluated the effectiveness and cost of the OSes based on the same test inputs. Six open source projects, six example programs, four other web applications, and one large industrial web services program with UML state machine diagrams were used. Test inputs were generated using a model-based testing tool, the Structured Test Automation Language Framework (STALE) [20], [21], [24], to satisfy edge coverage (EC), which covers all transitions, and edge-pair coverage (EPC), which covers all pairs of transition [1]. EC and EPC are defined in Section 2.1. Then by generating test oracle data for all OSes, we designed 24 sets of tests for each program (2 coverage criteria * 12 OSes). Then we ran the tests against faulty versions of the programs.

This paper extends our previous work [23] in five significant ways. First, this paper extends the venerable RIP model to RIPR [1], [33], [35] in Section 2.2. Second, this paper provides a precise definition for test oracle strategy in Section 2.3. Third, this paper formally defines *subsumption* among test oracle strategies in Section 2.4. Fourth, in Section 4.2, this paper defines four additional novel OSes that check program states less frequently and empirically compares them with the OSes that were used in the previous paper. Fifth, this paper significantly extends the empirical evaluation, including the use of a large industrial web services program in Section 5.

This provides five recommendations for using OSes in model-based testing. First, just checking runtime exceptions misses many faults and wastes much of the testing effort. Second, with the same test inputs, OSes that check more program states were not always more effective at revealing failures than OSes that check fewer program states. Third, OSes that check program states multiple times were not always more effective than OSes that check the same program states once. Fourth, with the same OSes, a test set that satisfies a stronger coverage criterion (EPC) was not more effective at revealing failures than tests from a weaker coverage criterion (EC). Whether a stronger coverage criterion can reveal more faults than a weaker coverage criterion depends on the model and the coverage criteria used. In this paper we say a criterion is *stronger* if it subsumes a weaker 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.

This paper is organized as follows. Section 2 introduces some fundamental testing concepts, and new theory related to the test oracle problem. Section 3 discusses related work, focusing on empirical studies about the test oracle problem. Section 4 presents all twelve OSes and how tests were created. Section 5 presents the experimental design, subjects, procedure, results, discussions, and possible threats to validity. Finally, Section 6 presents conclusions and discusses future work.

2 Definitions and Foundations for the Test Oracle Theories

This research has three general types of foundations. Section 2.2 introduces the RIPR model to show how a fault can be revealed by writing high quality test oracles. Section 2.3 then presents the key properties of test oracle strategies, and Section 2.4 defines the *subsumption* relationship for test oracle strategies. Before discussing the new concepts in Sections 2.2, 2.3, and 2.4, Section 2.1 introduces several fundamental software testing terms used in this research.

2.1 Fundamental Testing Definitions

Terms used in this paper come from standard textbooks such as Ammann and Offutt [1] and Young and Pezzè [38]. Some definitions are slightly adapted to fit in the context of this paper.

Definition 1 (Test Inputs). The inputs necessary to complete some execution of the system under test (SUT).

In our context, test inputs are sequences of method calls to the SUT, including all necessary objects, parameters, and resources.

Definition 2 (Expected Results). The results that will be produced when executing the test inputs if the program satisfies its intended behavior.

Definition 3 (Test Oracle). A test oracle provides expected results for some program states as specified by the test oracle strategy (formally defined later in Section 2.3). The test oracle determines whether a test passes by comparing expected with actual results.

Test oracles usually consists of assertions that compare expected results with actual results.

Definition 4 (Test). A test consists of test inputs and test oracles.

Definition 5 (Fault). A fault is a static defect in the software.

Definition 6 (Failure). A failure is an external, incorrect behavior with respect to the requirements or other description of the expected behavior.

When a test executes, a statement that has a fault may be reached. Under the right circumstances, a test that reaches the fault may cause a software failure that testers could observe. This is discussed further in Section 2.2.

Definition 7 (Unit Testing). Unit testing assesses software by designing and executing tests for methods in classes.

Definition 8 (System Testing). System testing assesses software by designing and executing tests that capture the system behaviors.

In industry, developers often write unit tests and testers write system tests. In addition, the distinction among integration testing, system testing, and user acceptance testing often blurs.

Definition 9 (Test Requirements (TRs)). Test Requirements (TRs) are specific elements of software artifacts that must be satisfied or covered.

Exhaustively enumerating all test inputs is effective at finding failures, but is prohibitively expensive. As a compromise, tests can be 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]. Test requirements guide testers to design effective tests just as functional requirements guide developers to design effective software. Coverage criteria also give testers a "stopping point" at which testing can be considered complete. For example, given the test criterion "cover every node," a specific test requirement would be "cover the initial node." This research uses graph coverage criteria to generate tests. First we define a graph.

A graph G is [1]:

- a set N of *nodes*, where $N \neq \emptyset$
- a set N_0 of *initial nodes*, where $N_0 \subseteq N$ and $N_0 \neq \emptyset$
- a set N_f of *final nodes*, where $N_f \subseteq N$ and $N_f \neq \emptyset$
- a set E of *edges*, where E is a subset of $N \times N$

For a graph to be useful in testing, it must have at least one initial and final node, but may have more than one initial and final nodes.

Edge coverage requires each edge to be covered by at least one test. The formal definition of edge coverage is:

Definition 10 (Edge Coverage (EC)). The set of test requirements TR contains each edge, in G.

Definition 11 (Edge-adequate Tests). A test set is edge-adequate if at least one test in the set covers every edge in the graph.

When defined on individual methods represented by their control flow graphs, node coverage corresponds to the familiar statement coverage and edge coverage corresponds to branch coverage. Less well known is edge-pair coverage, which requires that each pair of edges be covered. If the length of a path is defined as the number of edges in the path, edge-pair coverage is defined precisely as:

Definition 12 (Edge-Pair Coverage (EPC)). TRs contains each reachable path of length up to 2 in G.

For example, consider the partial graph in Figure 1. EC would have five test requirements on this graph, to cover edges $[1, 4]$, $[2, 4]$, $[3, 4]$, $[4, 5]$, and $[4, 6]$. EPC would have six test requirements, to cover subpaths $[1, 4, 5]$, $[1, 4, 6]$, $[2, 4, 5]$, $[2, 4, 6]$, $[3, 4, 5]$, and $[3, 4, 6]$. Following the definition strictly, EPC would also include the five edges, but it is customary to omit edges for EPC if they appear in one of the edge-pairs.

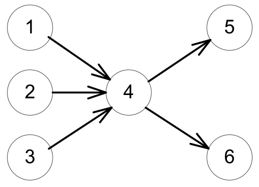

Fig. 1. Example of Edge-Pair Coverage

The test requirements for **edge-pair** coverage are subpaths of length 0, 1, or 2. The "length up to 2" is included to ensure that edge-pair coverage subsumes edge coverage (paths of length one) and node coverage (paths of length zero). That is, if a graph only has one edge or a node with no edges (as is common in, for example, getter and setter methods), tests would need to cover the edge (or node) to satisfy EPC.

Definition 13 (Edge-Pair (EP)-adequate Tests). Edge-Pair (EP)-adequate tests cover all TRs of edge-pair coverage.

The empirical portion of this research uses mutation analysis to place faults into software. 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 test, it behaves identically to the original program and is thus called *equivalent*. The *mutation score* is the ratio of mutants that are killed over the killable mutants (non-equivalent mutants), which serves as a measure of the effectiveness of a test set.

Definition 14 (Mutation-adequate Tests). Mutation-adequate tests kill all non-equivalent mutants.

2.2 The RIPR Model

The distinction between fault and failure, particularly in the context of mutation, led to the development of the reachability, infection, and propagation model in the 1980s. The RIP model was independently developed by Morell and Offutt in their dissertations [33], [35] and published as Propagation, Infection, and Execution [34], and Reachability, Necessity, and Sufficiency [12]. Current literature combines the terms as Reachability, Infection, and Propagation [1].

At that time, tests were almost invariably run by hand and testers examined the results manually. It was a reasonable assumption that if an error propagated to the output, the tester would notice and mark the run as failing. In modern times, however, test automation has negated this assumption. An automated test must include an explicit comparison of outputs with expected results; called *checking*, or the *oracle*. Because it is often impractical and unnecessary to check all outputs, automated tests typically check certain specific parts of the output state. However, if the oracle does not check the particular portion of the state that contains an erroneous value, the oracle will not see the failure. That is, the test oracle must also *reveal* the failure.

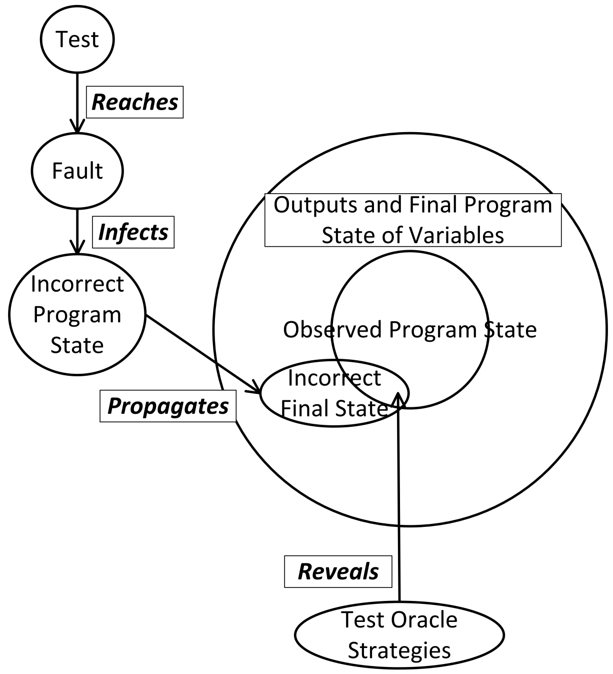

Fig. 2. The Reachability, Infection, Propagation, and Revealability Model

Figure 2 illustrates how test oracles are used to reveal faults by observing the *program state*. The *program state* during execution is the current value of all program variables and the current statement being executed. To detect a fault, a test has to *reach* the location of the fault. This is illustrated in Figure 2 where the **Test** reaches the **Fault**. The execution of the statements in the faulty location must cause an incorrect internal program state, that is, the state must be *infected*, as illustrated by the arrow from **Fault** to **Incorrect Program State** in Figure 2. For example, a *for* loop (int i = 0; i $<$ 4; i++) is supposed to be executed four times. If a developer makes a mistake and uses $\leq$ instead of $<$, the *for* loop is executed one more time. Thus, the program state for the faulty version is different from the correct version and is incorrect. However, testers do not look for failing behavior by inspecting internal program states. The incorrect internal program state then must *propagate* to an **Incorrect Final State (failure)**. Figure 2 illustrates this on the right. The large circle represents the complete output state of the program. The erroneous portion of the state is shown as a smaller circle, **Incorrect Final State (failure)**. Testers use a test oracle strategy write test oracles such as assertions to observe the final program state. This is shown in Figure 2 as **Observed Final Program State**. Failures are only *revealed* if the **Observed Final Program State** includes part of the **Incorrect Final State**. Since test oracles are almost always written by humans, testers need guidance for devising an OS that will maximize the chance of writing them such that an incorrect portion of the final state, if it exists, can be observed by the test oracle.

2.3 Designing Test Oracle Strategies

When testing beyond the unit level, the output state is too large to observe completely. Thus, testers must decide what parts of the program state to evaluate to determine if the test passes or fails. At the system level, this may be outputs to a screen, a database, or messages sent to other systems. This can be such a complicated problem that some practitioners simply take the cheapest possible solution: does the program terminate or not? This is called the *null test oracle* strategy (NOS) [40] and is used by some testers. Integration testing may need to evaluate states from objects spread throughout the program, as well as data sent to external destinations such as screens, files, databases, and sensors. Therefore, it is difficult for testers to know which states should be evaluated and when to check the states.

This research considers two dimensions when designing test oracle strategies (OSes): which internal state variables and outputs to check (*precision*) and how often to check the states (*frequency*). Briand et al. defined the *precision* of a test oracle strategy as the degree to which internal state variables and outputs are checked by the OS [8]. This paper refines the *precision* of a test oracle strategy to be how many output values and internal state variables are checked by the OS. The more internal state variables and outputs a test oracle strategy checks, the more precise the test oracle strategy is. A test oracle strategy (OS) is more expensive if more variables are checked (more precise) or the variables are checked more frequently (more frequent). States can be checked after each method call or each transition (more than one method call can be included in a transition) or testers can just check the states once per test or test suite.

The definitions of precision and frequency lead to several questions that are important for test engineers. Which variables should testers check? Should the member variables of all objects be checked? Should the return values of all method calls be checked? The OSes used in this paper are differentiated by precision and frequency, as defined in Section 4.2.

We conclude this section with a formal definition of test oracle strategy (OS), adding precision to previous uses of the term:

Definition 15 (Test Oracle Strategy (OS)). A test oracle strategy (OS) is a rule or a set of rules, defined by a precision P and a frequency F, that specify which program states to check. P defines a set of internal state variables V and outputs O to be checked by each oracle. F defines how often to check the variables and outputs specified by P.

2.4 Subsumption

This paper defines the *subsumption* relationship for test oracle strategies (OSes). The goal of defining this term is to help testers better understand how much of the program state OSes check. A subsumption hierarchy among OSes can help decide which OS to use when writing test oracles. A formal definition of subsumption is given below.

Definition 16 (Subsumption). A test oracle strategy OS_A *subsumes* another test oracle strategy OS_B if OS_A checks all of the program states that OS_B checks.

Both the precision and frequency properties of OSes have to be considered. If OS_A is more precise but checks program states less frequently than OS_B, OS_A may not subsume OS_B. For example, assume that OS_A checks all return values and member variables once and OS_B checks all return values of all method calls after every method call. OS_A is more precise than OS_B but has lower frequency. Because they check different program states, OS_A does not subsume OS_B. As another example, if OS_A checks all internal state variables and outputs after every method call and OS_B checks all internal state variables and outputs at the end of a test, we can say that OS_A has the same precision but higher frequency than OS_B. When we apply these two OSes to a test suite, the program states checked by OS_B are always examined by OS_A, thus, OS_A subsumes OS_B.

3 Related Work

The related work section has two subsections. Section 3.1 discusses related work about test oracle theories and test oracle strategies, as well as empirical studies on the effectiveness and cost-benefit tradeoff among OSes. Section 3.2 compares this research with previous studies that evaluated OSes empirically in similar ways to our evaluation.

3.1 Test Oracles

The test oracle problem was first defined by Howden [18]. Hierons et al. suggested using formal specifications to develop oracles [17]. Our research targets engineers who use finite state machines that are defined operationally, as opposed to using denotational formal specifications. Barr et al. summarized the test oracle problem in a survey in four broad categories: specified oracles, derived oracles, implicit oracles, and no test oracle [5]. *Specified oracles* get oracle information from specifications such as models and contracts. When specifications are not available, oracles (derived oracles) can be derived from other artifacts such as documents. *Implicit oracles* do not require domain knowledge or specifications. They can be used for all programs. One example is abnormal termination (exception). *No test oracle* happens when testers do not have any artifacts for writing test oracles. In this research, NOS is an implicit test oracle strategy (defined in Section 2.3) and other OSes are derived from specifications.

Regarding test oracle theories, Staats et al. [44] studied how test oracles affect the *propagation estimate* metric of testability. The propagation estimate measures failure probabilities, that is, how likely an error state created by an infection is to propagate to an output, that is, fail. Our paper directly extended the widely known RIP model to the RIPR model to illustrate that not only must an erroneous value propagate to an output, the test oracle must also reveal that failure to the tester. Revealability is a crucial aspect of test oracles that has not been previously considered. If the test oracle does not reveal a failure, all the work of designing and running the test is lost.

Barr et al. [6] created a repository of scientific publications on the test oracle problem. Even with this resource, we were only able to find a few papers that studied the test oracle problem empirically. Briand et al. [8] compared the *very precise OS* with the *state invariant OS* (SIOS) based on the statecharts. They studied four classes from three programs, each with less than 500 lines of code. 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 revealing 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. Briand et al. calculated cost-effectiveness using their definitions of effectiveness (number of faults found) and cost (number of tests, CPU execution time, lines of code). With a weaker coverage criterion (*round-trip path coverage* [7]), using the very precise OS was found to be more cost-effective. With a stronger coverage criterion (*disjunct coverage* [8]), using the state invariant OS was found to be more cost-effective. We also use the faults found to measure effectiveness. However, we use the number of assertions created by hand as a more accurate reflection of cost. The machine cost of running assertions that have more checks is orders of magnitude less than the human cost of creating additional checks. Our measurement is described in detail in Section 5.1.

Xie and Memon [45] considered precision and frequency when designing OSes for GUIs. They found that the variations of the two factors affected 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 found that more precise OSes detected more faults than less precise OSes. They also found that an OS with higher frequency (checking after every event) detected more faults than an OS with lower frequency (checking only after the last event of a test). Xie and Memon defined effectiveness as the number of faults found and the cost as the number of comparisons that oracles perform. Given the same precision, two OSes with lower frequency had better cost-effectiveness and one OS with higher frequency had better cost-effectiveness.

Staats et al. [43] found that an OS that checks outputs and internal state variables can reveal 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 [40] empirically compared NOS and the *pre and post-condition OS* (PPCOS) using the *Java Modeling Language* [9]. They found that the latter can reveal more failures 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. [41] 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. [46] studied the effectiveness of the *output-only*

OS and six other OSes that check 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*.

3.2 Similar Studies

This paper presents a comprehensive experiment to study the effectiveness and cost of OSes and gives guidelines about which OS should be used. A comparison between this paper and the others is shown in Table 1. The first column shows the metrics and the other columns represent others' work.

Shrestha and Rutherford [40] used nine small programs, with the biggest having 263 statements. Others studied six programs or fewer. This paper used 17 programs with lines of code (LOC) ranging from 52 to 15,910. The total LOC of the 17 programs is 47,742 while both Shrestha et al. and Briand et al. used fewer than 2,000 LOC. The subjects of this research 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 [16], which do not have OO classes. Because Staats et al. used a non-OO language, they measured their LOC in terms of number of blocks (*#blocks*), not number of statements (*#statements*), as reported in Staats et al. [42]

This research considers twelve OSes: NOS, SIOS, and ten 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. Twelve OSes that have different precision values were used in our experiment while both Shrestha et al. and Briand et al. studied two OSes.

Staats et al. [42] and Mateo and Usaola [32] 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 [13] also used mutation testing to derive test inputs and test oracles but they did not study the effectiveness or cost of OSes.

4 Test Inputs and Oracle Generation

This section presents how tests were generated at the system level and defines the ten new and two baseline OSes that were applied to the tests.

4.1 Test Input Generation

This research evaluated different OSes with the same tests. The tests were generated from UML state machine diagrams of 17 Java programs using the structured test automation language framework (STALE) [21], [24]. STALE reads UML state machine diagrams and transforms 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 constraints (based on state invariants). To transform the abstract tests to concrete tests, testers use the structured test automation language [24] to provide mappings. A *mapping* is a data structure that describes how to translate test inputs from model elements (transitions and state invariants in this research) to the implementation. Each model 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 called the *state invariant oracle strategy* (SIOS). Additionally, testers can use STALE to write more assertions to check other internal state variables such as class variables and parameter objects. 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 and *classObjectA*'s class member variables by providing the expected test values.

STALE uses a prefix-graph based solution [22] to reduce the number of tests as well as the number of times transitions appear in the tests. Barr et al. called this a type of *quantitative human oracle cost reduction* [5]. The quantitative human oracle cost reduction reduces the effort of generating test oracles by decreasing the size of test cases or test suites.

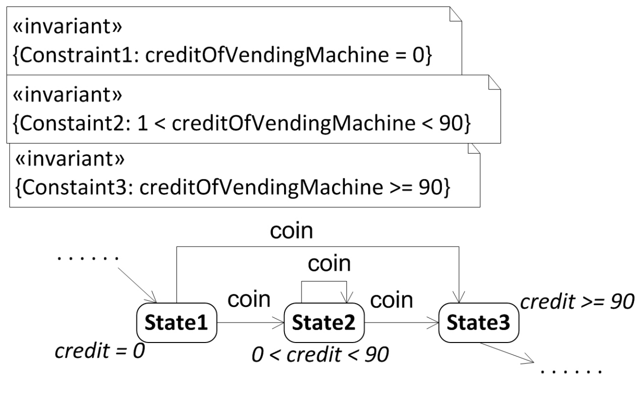
«invariant»
{Constraint1: creditOfVendingMachine = 0}
«invariant»
{Constaint2: 1 < creditOfVendingMachine < 90}
«invariant»
{Constaint3: creditOfVendingMachine >= 90}
coin
coin
coin
coin
State1
State2
State3
credit = 0
0 < credit < 90
credit >= 90
......
......

Fig. 3. Part of a State Machine Diagram of the Vending Machine Example. Constraint1 applies to State1, Constraint2 applies to State2, and Constraint3 applies to State3.

We use an example of a vending machine program to show how to use STALE to generate tests and add test

TABLE 1
A comparison of empirical test oracle research papers. (LOC is Lines of Code, NOS is the Null Oracle Strategy, PPCOS is the Pre and Post-Condition Oracle Strategy, and SIOS is the State Invariant Oracle Strategy.)

Metric	**This research**	**Briand et al.** [8]	**Xie and Memon** [45]	**Staats et al.** [43]	**Shrestha and Rutherford** [40]	**Sprenkle et al.** [41]	**Yu et al.** [46]
Where published	submitted	TSE	TOSEM	ICSE	ICST	ISSRE	ISSRE
Number of programs	17	2	5	4	9	4	6
Total LOC	47,742	1,552	25,767	14,039	1,049	39,793	7,877
Types of subjects	General, GUI, Web Application, and Web Service	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	12	2	6	3	2	22	7

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 chocolates is 90 cents. Figure 3 shows part of a UML state machine diagram for the vending machine–three states and their transitions. The figure also shows three invariants in boxes above the state diagram, defined as *constraints*. Each constraint applies to a particular state. The invariant $credit = 0$ applies to **State1**, $0 < credit < 90$ applies to **State2**, and $credit \geq 90$ applies to **State3**. Part of the implementation of class *VendingMachine* is in Figure 4.

```
1: public class VendingMachine
2: {
3:   private int credit; // Current credit in the machine.
     ...
4:   // Constructor: vending machine starts empty.
5:   public VendingMachine() {}

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

9:   // Get the current credit value.
10:   public int getCredit () {}
     ...
11: }
```

Fig. 4. Class VendingMachine (partial)

STALE reads the state machine diagram and generates abstract tests. If testers choose edge coverage, then the three 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 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. Method call "*vm.coin(10);*" uses the method on line 8 in Figure 4. *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. Similarly, "*vm.coin(100);*" uses the method on line 8 in Figure 4.

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 asks testers to enter more mappings. Testers can also provide more test oracle data to check other fields of the class.

In addition to the state invariants, testers can also check other fields of class *VendingMachine*. STALE provides a mechanism for testers to enter test oracle data for different test oracle strategies. When a mapping is used in a test, the corresponding test oracle has to be inserted after the test code of the mapping. The location where a mapping appears inside tests depends on the coverage criteria, constraints, test inputs, and test generation algorithms. Thus testers cannot write fixed expected values in test oracles. If the test code of a mapping for transition "coin" is *vm.coin(10);*, an assertion after this transition (such as "assertEquals(10, vm.getCredit());") could return false since the credit before adding the dime may not be 0 in a preceding state. Therefore, to automate test oracle generation, the test code of element mappings has to be changed. The call *"vm.getCredit();"* uses the method on line 10 in Figure 4. The mapping *coinTen* adds a statement "credit = vm.getCredit();" to get the value of *credit* before inserting a dime into the vending machine. Note that variable *credit* needs to be declared in the test initialization to avoid duplicate variable definitions. Then the test oracle code for the mapping *coinTen* can check whether the assertion returns true: "assertEquals(credit + 10, vm.getCredit());".

4.2 Test Oracle Strategies

Two OSes were used as baselines in the experiments. NOS only checks for uncaught runtime exceptions or abnormal termination, as implicitly provided by Java runtime systems [40]. In our experience, most faults do not cause runtime exceptions, so this OS sounds trivial to academic researchers. However, in our experience, both consulting and teaching part-time industrial students, NOS is often used in industry.

The second baseline OS was SIOS, which checks the state invariants from the state machine diagrams. 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 and thus subsumes NOS.

As stated in the introduction, this research considers two dimensions when designing OSes: *precision* (how many internal state variables and outputs to check), and *frequency* (how often to check states). Regarding the frequency, testers can write test oracles that check states after each method call, after each transition, or they can check states only once at the end of the test. The checks are automated, that is, each test includes explicit comparisons of actual values from the state with expected values that are unique to the test and based on requirements, specifications, or domain knowledge of the software's intended behavior. We use the JUnit test framework, which is very widely used in industry, although any test automation framework could be used. Note that frequency does not apply to NOS since no explicit assertions are included in NOS. SIOS checks state invariants after each transition. In terms of 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*: The mappings for transitions define methods on specific objects that must be called to trigger that transition. These objects contain member variables. These *object members* are checked.
3) *Return values*: Check return values of each method invoked in a transition.
4) *Parameter members*: Check member variables of objects that are passed as parameters in a method call.

In OO software, a *deep comparison* compares the values of every member of two objects recursively until primitive variables are found. This research used deep comparisons. For the vending machine example in Figures 3 and 4, constraints 1, 2, and 3 are state invariants. *vm* is a *VendingMachine* object member. The actual credit returned by *"vm.getCredit();"* (line 10 in Figure 4) is a return value. The parameter *coin* of the method *coin* (line 8 in Figure 4) is a parameter member. In executable tests, testers need to write assertions in STALE to verify these four elements associated with the transitions and constraints.

We propose ten new OSes beyond SIOS and NOS. Each new OS satisfies all the state invariants in a model and explicitly writes the satisfied state invariants as assertions. Thus, they explicitly subsume SIOS, and by transitivity, NOS. A test is based on a sequence of transitions through the model, and each transition represents one or more method calls that have the potential to modify the four program state elements listed above. The strategies check different elements of the program state, generally increasing precision as the numbers get larger from OS1 through OS5. The first five, OS1 through OS5, have higher frequency, that is, they check state after each transition. Paralleling these strategies are five that have the same precision, but only check state once. We call these OT1 through OT5 for "**O**ne **T**ime checking." For convenience in writing, we sometimes refer to all twelve collectively as oracle strategies, or OSes.

The definitions of the five high frequency OSes refer to different combinations of the four elements of the state listed above. Each OS is defined with the precision P and frequency F. The OS strategies check values after each transition.

OS1: Check *object members*: P is defined to check all *object members* in transitions. F is defined to check them immediately after each transition is executed.

OS2: Check *return* values: P is defined to check all *return values* in transitions. F is defined to check them immediately after each transition is executed.

OS3: Check *object members* and *return values*: P is defined to check all *object members* and *return values* in transitions. F is defined to check them immediately after each transition is executed.

OS4: Check *parameter members* and *return values*: P is defined to check all *parameter members* and *return values* in transitions. F is defined to check the them immediately after each transition is executed.

OS5: Check *object members*, *parameter members*, and *return values*: P is defined to check all *object members*, *parameter members*, and *return values* in transitions. F is defined to check them immediately after each transition is executed.

For each OSi[2], OTi has the same precision (checks the same elements of the state) but with a lower frequency (only

2. OSi refers to one of the five specific OSes (OS1, OS2, OS3, OS4, OS5), and OTi refers to one of the five specific OTs.

one check per test). The practical difference between the OS strategies and the OT strategies is that the OS strategies check the values after each transition and the OT strategies check after the last transition. Note that the OT strategies check all the values from all the transitions, not just the last transition, even if they perform all checks after the last transition. Additionally, they can only check objects and variables whose scope is still active after the last transition.

OT1: P is defined to check all *object members* in **all** transitions in the test; F is defined to check the *object members* defined in P after the last transition is executed

OT2: P is defined to check all *return values* in **all** transitions in the test; F is defined to check the *return values* defined in P after the last transition is executed

OT3: P is defined to check all *object members* and *return values* in **all** transitions in the test; F is defined to check the *object members* and *return values* defined in P after the last transition is executed

OT4: P is defined to check all *parameter members* and *return values* in **all** transitions in the test; F is defined to check the *parameter members* and *return values* defined in P after the last transition is executed

OT5: P is defined to check all *object members*, *parameter members*, and *return values* in **all** transitions in the test; F is defined to check the *object members*, *parameter members*, and *return values* defined in P after the last transition is executed

All the OSi and OTi strategies subsume NOS and SIOS. The five OSi strategies and $OT5$ were defined in our previous paper [23]. The other OTi strategies are new to this paper.

Although OS5 is the most precise test oracle strategy in our study, it would be possible to design an even more precise strategy. For example, an oracle could check logs, files, or other offline storage locations. Test oracle strategies OT1, OT2, OT3, OT4, and OT5 mimic a common programmer habit of writing assertions at the end of tests. The difference is that programmers often only write a few ad-hoc assertions, while OT1, OT2, OT3, OT4, and OT5 systematically check various outputs and internal state variables after the last transition. We wanted to see if each OT can be as good as the corresponding OS in terms of revealing failures.

Checking all object members, return values, and parameter members that appear in all the transitions at the end of tests could cause the OTis to check different program states from the OSis. For instance, if OS2 checks a return value *Object a* after the system initialization in a test (*a* has an initial value during the system initialization), then OS2 does not check *a* in the rest of the tests because *a* is not used as a return value in other transitions. However, *a*'s state could be changed if *a* is used as a parameter or makes method calls. Therefore, OT2 can check different program states (check *a* after the last transition) that would not be checked by OS2 (check *a* only after the first transition). Moreover, no matter when *a*'s status is changed, OS5 is able to monitor the change since OS5 checks object members, return values and parameter members for every transition. Thus, OS5 subsumes OTi, where $1 \leq i \leq 5$. So OTi is as precise as OSi but may detect faults that cannot be revealed by OSi, where $1 \leq i \leq 4$.

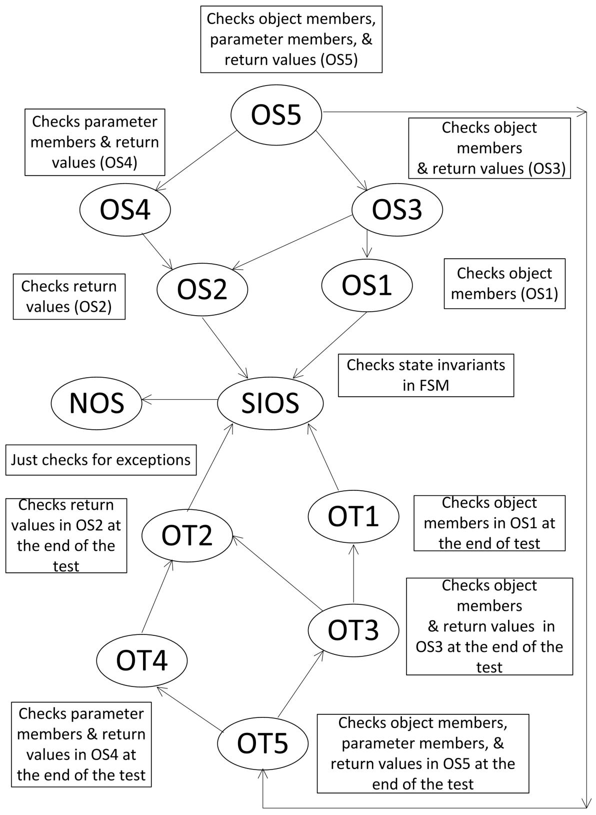
Checks object members, parameter members, & return values (OS5)
OS5
Checks parameter members & return values (OS4)
Checks object members & return values (OS3)
OS4
OS3
Checks return values (OS2)
OS2
OS1
Checks object members (OS1)
Checks state invariants in FSM
NOS
SIOS
Just checks for exceptions
Checks return values in OS2 at the end of the test
OT2
OT1
Checks object members in OS1 at the end of test
Checks object members & return values in OS3 at the end of the test
OT3
OT4
Checks parameter members & return values in OS4 at the end of the test
OT5
Checks object members, parameter members, & return values in OS5 at the end of the test

Fig. 5. Subsumption Relationships among Test Oracle Strategies. Subsumption is transitive, thus, OS5 subsumes OS2, OS1, SIOS, and NOS by transitivity.

Figure 5 shows the subsumption relationships among the OSes and OTs. An arrow from one strategy to another indicates that the former strategy subsumes the latter. The OS subsumption relationships are transitive. Thus, if OS5 subsumes OS3 and OS3 subsumes OS1, then OS5 subsumes OS1. Because OS5 subsumes SIOS and OT5 subsumes SIOS based on transitivity, we do not show arrows from OS5 to SIOS and from OT5 to SIOS.

5 Experiments

The experiments address four research questions:

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

RQ2: With the same test inputs, does a higher frequency, that is, 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 [8], [40], [43], [45] 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 3.

RQ2 was evaluated by Xie and Memon [45] for GUIs, who found that checking variables after each event (events in GUI testing represent user actions such as button clicks) 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. Our research checked more outputs and internal state variables of different kinds of programs.

Briand et al. [8] found that with the very precise test oracle strategy, a stronger coverage criterion (*disjunct coverage* [8]) found the same faults as a weaker criterion (*round-trip path coverage* [7]) for one class, but not the other three. This is related to our RQ3. Our experiment used two different coverage criteria and 17 programs.

A very effective OS may be too costly for practical use. Thus, RQ4 considers the cost-effectiveness of the OSes, which to our knowledge has not been studied before. The rest of this section presents the experimental design, subjects, procedure, results, discussions, and threats to validity.

5.1 Experimental Design

The experiments compared the ten new strategies (OS1, OS2, OS3, OS4, OS5, OT1, OT2, OT3, OT4, and OT5) with the two baseline OSes, NOS and SIOS. All OSes were applied to edge-adequate and 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. [3] found that synthetic faults generated using mutation testing can be used as faults in experiments to predict the real fault detection ability of tests. This research used synthetic faults generated with mutation testing. Because programs have different numbers of faults, to compare the effectiveness of the tests in the same scale, we used proportions of faults (mutation scores) detected by the tests to measure the effectiveness of the tests for each program. Furthermore, if tests have the same test inputs but different OSes, 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. If one OS subsumes another OS, the subsuming OS is expected to be at least as effective at revealing failures than the subsumed OS. Thus, OS1, OS2, OS3, OS4, OS5, OT1, OT2, OT3, OT4, and OT5 were expected to reveal the same or more faults than NOS and SIOS.

The experiments used muJava [30], [31], a mutation analysis tool for Java, to generate synthetic faults. Each mutant is the result of only one application of a single mutation operator to the original program. Users can use muJava to generate mutants, run tests against mutants, and view mutants. The latest version of muJava supports JUnit tests and all features of Java 1.6. So the JUnit tests that STALE generated were used in muJava directly. Mutants were generated by using the 15 selective method-level mutation operators of muJava [29].

All OSes were applied with the same sets of edge-adequate and EP-adequate tests. In the experimental process, the test oracle generation and execution had three kinds of cost. First, testers entered test oracle data (assertions) by hand. Second, STALE generated tests based on the provided test oracle data. Thus, the concrete tests include the test oracles. The assertions provided in the first step may appear multiple times in the concrete tests. When EC or EPC is applied, every transition of a UML state machine diagram is part of the test requirements. STALE uses the prefix-graph based algorithm to generate a set of test paths to cover the test requirements. Each test path may have multiple distinct transitions and the distinct transitions may appear in different test paths. By using STALE, testers need to provide mappings for each distinct transition only once, even though each transition may appear multiple times in the test paths. STALE selects appropriate mappings to satisfy state invariants in the state machine diagram. Then STALE converts the test paths to executable concrete tests based on the selected mappings. Likewise, testers use STALE to write distinct assertions for each mapping of distinct transitions. Since OS1, OS2, OS3, OS4, and OS5 check the program states after each transition, distinct assertions may appear multiple times in the concrete tests, along with the selected mapping of distinct transitions. Third, as part of tests, assertions must be executed.

The first step was manual and the second and third were automated. In our study, the execution time of test oracles was tiny, indeed, almost impossible to measure. Not only was it many orders of magnitude less than human effort, it was orders of magnitude less than the rest of the test execution. Even if run thousands of times, few test oracles would significantly impact the execution time. (An exception would be a very complex test oracle.) Thus, the cost of an OS is primarily the cost of creating test oracle data (assertions) by hand. Although human cost is far more difficult to measure than execution cost (as previous studies did), we elected to measure human cost so as to increase the value of the results.

To make this research study practical, we assumed that the cost of writing each assertion was constant. While there would be some variability, the differences would be relatively slight and average out over all assertions. The cost of creating a test oracle was the sum of the costs of creating each assertion in the oracle.

We gave both state invariants and normal assertions the same weight for four reasons. First, writing an assertion needs testers to understand the program. Thus, writing any assertion takes the same amount of time for understanding the program. Second, this research required testers to write test oracles for each transition. This step required test oracles to be generated automatically no matter where transitions appear in tests. Thus, testers have to change the test code of the transitions, as shown in Section 4.1. Our experience told us that the first two steps took the most time for creating an assertion.

Third, designing state invariants is only part of designing a state machine diagram and the diagram was mainly used for generating tests. Moreover, a group of assertions (state invariants in different states) was created within the diagram. Thus, the time to create each state invariant only takes a fraction of the total time to generate tests. Fourth, when designing normal assertions, testers may have to spend extra time to look for assertions. For instance, when checking member variables of a class, if a member variable is also an object, testers have to check its member variables, and so on until all member variables are primitive. Therefore, the cost of each assertion was treated equally and we used the number of *distinct assertions* as an approximation for the cost. Since many tests include the same assertion, the total number of assertions may be much greater than the number of distinct assertions, yet we only have to design each distinct assertion once.

The cost-effectiveness of an OS is the ratio of the proportion of faults detected by the OS over the number of assertions. The assertions were provided by hand to check the internal state variables and outputs. A bigger cost-effectiveness is better. The cost-effectiveness ratio can be interpreted as: how many more faults can be detected by adding additional assertions?

This cost-effectiveness ratio was applied to SIOS, OS1, OS2, OS3, OS4, OS5, OT1, OT2, OT3, OT4, and OT5, but not to NOS. NOS is a special case. Since there are no assertions, there is no cost by our measure. However, the cost measure does not include the cost of designing and generating tests. This is so that we can compare OSes strictly on the basis of their oracles, without regard to the test generation cost. A disadvantage of this approach is that NOS, in effect, means the testers only use tests that are likely to result in a runtime exception. All other tests are useless, even if they result in a failure. Thus we do not consider NOS in our cost-effectiveness ratio.

$$\textit{Cost-effectiveness} = \frac{ProportionOfFaultsDetected}{\#DistinctAssertionsCreated} \quad (1)$$

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 but with the same frequency. (Frequency does not apply to NOS and SIOS, so they are not compared on the basis of frequency.) However, we compared NOS and SIOS since they are baseline OSes and other OSes are more precise. These hypotheses focus exclusively on the precision, so we do not compare OSes with different frequencies. The null and alternative hypotheses are listed below.

Null hypothesis (H_0):
: There is no difference between the proportion of failures revealed by OS_A and OS_B with the same test inputs and frequencies (if applicable).

Alternative hypothesis (H_1):
: The proportion of failures revealed by OS_B is greater than OS_A with the same test inputs and frequencies (if applicable).

The test oracle strategy pairs that were 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}, {OS2, OS4}, {SIOS, OT1}, {SIOS, OT3}, {SIOS, OT5}, {OT1, OT3}, {OT3, OT5}, {OT1, OT5}, {OT2, OT5}, {OT4, OT5}, {SIOS, OT2}, {SIOS, OT4}, {OT2, OT3}, and {OT2, OT4} for both edge coverage (EC) and edge-pair coverage (EPC). This research did not compare NOS with other OSes (OS1, OS2, OS3, OS4, OS5, OT1, OT2, OT3, OT4, and OT5) because NOS was expected to be much less effective than other OSes. The comparison between the effectiveness of NOS and other OSes is shown in Section 5.4.

RQ2 asks if an OS with higher frequency is more effective at revealing failures than another OS that has the same precision but lower frequency. OTi checks the same object members, return values, and parameter members that OSi checks. For each i, $1 \leq i \leq 5$, OSi checks after each transition and OTi only checks once after the last transition. Thus, the second group of hypotheses ($Hypotheses_B$) for RQ2 were:

Null hypothesis (H_0):
: There is no difference between the proportion of failures revealed by OTi and OSi with the same test inputs, where $1 \leq i \leq 5$.

Alternative hypothesis (H_1):
: The proportion of failures revealed by OSi is greater than OTi with the same test inputs, where $1 \leq i \leq 5$.

The test oracle strategy pairs that were applied to $Hypotheses_B$ are: {OT1, OS1}, {OT2, OS2}, {OT3, OS3}, {OT4, OS4}, and {OT5, OS5}. The third group of hypotheses ($Hypotheses_C$) for RQ3 took two test coverage criteria CC_A and CC_B into consideration (CC_B subsumes CC_A).

Null hypothesis (H_0):
: There is no difference between the proportion of failures revealed by criterion CC_A and CC_B if both use the same test oracle strategy.

Alternative hypothesis (H_1):
: The proportion of failures revealed by criterion CC_B is greater than CC_A if both use the same test oracle strategy.

$Hypotheses_C$ were applied to edge-adequate and EP-adequate tests for each of the twelve OSes used in this research.

5.2 Experimental Subjects

We evaluated 17 Java programs, creating UML statecharts by hand, and using STALE to generate tests. The experiment involved multiple manual steps, which limited our ability to use large subjects. In actual practice, the UML models would be already available, or at least generated by the developers. With our less than perfect knowledge of the software, this task was extremely time consuming. For similar reasons, previous experiments on test oracles have also focused on small subjects, as shown in Table 1.

Based on our experience, testers often need a lot of time to understand requirements and acquire domain knowledge for the software under test. A large project may take years for dozens or hundreds of software engineers to develop. Thus, it would take many months for us to generate meaningful tests for large and complex projects. To add additional realism to the experiment, the first author applied STALE to a product that he has been working on at his project; that is, he already understood the requirements.

Six of the 17 programs (Calculator[3], Snake[4], TicTacToe[5], CrossLexic[6], Jmines[7], and DynamicParser[8]) are open source projects from SourceForge. Six are from textbooks: VendingMachine [1], ATM [11], Tree [4], BlackJack [19], Triangle [39], and Poly [27]. Four others are part of the coverage web application for Ammann and Offutt's book [2]. The last, Roc, is a real-world product from Medidata. Roc is a service that wraps the *Elastic MapReduce (EMR)* of *Amazon Web Services (AWS)*. *EMR* is a web service that processes large amounts of data efficiently using *Hadoop*[9]. Since Roc is a proprietary Medidata project, we cannot provide a link to its code base.

All programs are in Java and we generated the UML state machine diagrams by hand for all the program except for Roc. The UML state machine diagram for Roc already existed; created by the engineers.

To reduce the threat to validity by having subjects that are overly similar, we chose subjects that had a variety of purposes, type of deployment, and size. Calculator, Snake, CrossLexic, Jmines, BlackJack, and DynamicParse are GUIs. GraphCoverage, DFCoverage, LogicCoverage, and MinMCCoverage are web applications. Roc is a web service. TicTacToe is a command-line program. The other five programs are software components that do not have explicit user interfaces. The programs were intentionally chosen to vary in size. As shown in Table 2, the lines of code (LOC) varied from 52 to 15,910, measured by a line counter, *Cloc* version 1.6.2 [10][10]. Strengths of this study are the precision of measurement and the realism of the process. To achieve these goals, quite a bit of work had to be carried out by hand, including deriving state machine diagrams, measurements, and creating the mappings. These tasks are needed in an experimental context, but either would not be necessary in a practical setting or would be spread across the entire development process. In our context, it would have taken many months for programs with tens of thousands of lines of code, although as shown in Table 1, the size of programs in this study compares favorably with previous similar studies.

At the right abstraction level, we were able to generate UML state machine diagrams to cover important behaviors of complex systems. For example, Roc has a service and a client. The service contains the logic to process user requests such as starting an *EMR* cluster and users use the client to send requests to the server. At the user acceptance testing level, we focus on testing the system from the user's perspective. We generated a UML state machine diagram to cover all important user actions provided by the client. The diagram did not need to cover the logic in the server. We tested the server at the unit testing and component testing levels, which are out of the scope of this paper.

3. http://jcalcadvance.sourceforge.net/
4. http://sourceforge.net/projects/javasnakebattle/
5. http://sourceforge.net/projects/tttnsd/
6. http://crosslexic.sourceforge.net/
7. http://jmines.sourceforge.net/
8. http://dynamic-parser.sourceforge.net/
9. Apache Hadoop processes large data sets over clusters of computers using *Hadoop Distributed File System (HDFS)*.
10. The total LOC includes configuration files and scripts.

First, we generated test inputs to satisfy both EC and EPC on the state diagram. Then we entered test oracle data for our ten 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 twelve OSes were applied to the two sets of tests that satisfied EC and EPC, resulting in 24 sets of tests for each program.

Table 2 shows properties of the programs and tests. The column *LOC* shows the lines of code for each program. The columns *E* and *EP* give the number of test requirements for edge and edge-pair coverage. The columns *Tests* show the number of tests for edge-adequate and EP-adequate tests. The columns *Trans* represent the number of transitions that appeared in the tests and the columns *SI* provide the number of appearances of state invariants that were satisfied and also used as test oracles. We counted the number of test oracle assertions that have state invariants when all the tests passed.The columns *Distinct Trans* and *Distinct SI* represent the number of distinct mappings of transitions and state invariants provided by hand. (Recall that we only count each assertion once for the purposes of cost, even though they appear in the tests many times.)

As stated in Section 4.1, 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. Because of this, the results did not show much difference between EC and EPC.

5.3 Experimental Procedure

The experiment was carried out in the following steps:

1) We created a UML state machine diagram for each program. Since the designs were not available, this was done by hand by the first author (except for Roc).
2) STALE read the state machine diagram, recognized all identifiable elements in the diagram, and generated abstract tests to satisfy EC and EPC. This step was completely automated by our tool.
3) We used STALE to create abstract-to-concrete mappings for each element in the finite state machines, as described in Section 4.1. This step requires significant domain knowledge of the software, and is well known to be difficult to automate with informal specifications. Even with formal specifications, these mapping values can only be partially automatically generated. This step is normally the most human-intensive portion of model-based testing. The primary strength of the language TAL [24] is in reducing the labor associated with creating mappings. Test inputs were created first to satisfy EC, then augmented to satisfy EPC.
4) Using the mappings, STALE automatically generated concrete edge-adequate and EP-adequate tests.

TABLE 2
Experimental Subjects

Programs	LOC	E	EP	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
Roc	15,910	30	62	13	63	88	13	11	26	155	206	13	11
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	**47,742**	**711**	**2,431**	**263**	**1,364**	**1,890**	**174**	**125**	**566**	**4,972**	**6,721**	**183**	**125**

5) We used STALE to enter expected results for the OSes. 24 tests were generated for each pair of combination for the two coverage criteria and twelve OSes.
6) We used muJava to generate faults for each program, then identified and removed equivalent mutants by hand.
7) We ran each set of tests 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.
8) We calculated and analyzed the cost-effectiveness of each OS.

We needed each test set to have the same input values across all test oracle strategies. If the values differed, that would introduce a possible confounding variable. Our goal of the study was to compare the test oracle strategy, so to a large extent, the quality of the test set is irrelevant. The main consideration is that the tests caused enough faults to propagate to failure to measure differences among the OSes in revealability.

As is usual with empirical fault studies, tests were only run against faults that appeared in methods called when the tests were run on the original program.

5.4 Experimental Results

The results are divided into four parts. Section 5.4.1 presents the effectiveness of the OSes in terms of revealing failures. Section 5.4.2 analyzes the RQs statistically based on the effectiveness of the OSes. Section 5.4.3 presents results on the costs of the OSes. Section 5.4.4 presents cost-effectiveness results. For readers who want more detail, all of the experimental subjects and results are available online at *https://cs.gmu.edu/∼nli1/TSE_TestOracle*.

5.4.1 Effectiveness of Test Oracle Strategies

Tables 3 and 4 show the number of faults and failures revealed by each OS for each program with both the EC and EPC test sets. Table 3 contains the data for the higher frequency strategies (OSes) and Table 4 contains the data for the lower frequency strategies (OTs). Both Tables 3 and 4 show the total numbers of faults in each program and the number of faults that were revealed as failures. The largest subject, Roc, has only 95 faults because we only generated faults for the methods the tests used. Since the tests used only client actions, we did not generate faults for the logic on the server code. Columns NOS and $SIOS$ are the same in both tables. Tables 5 and 6 have similar columns. They show the number of faults that are revealed as failures by each OS divided by the number of faults in each program, producing proportions of failures revealed by each OS for each program. The total number of the faults ("# Faults," that is, non-equivalent mutants) is 9,722. So a total of **96,714,456** tests were executed (((12 OSes * 263 edge-adequate tests) + (12 OSes * 566 EP-adequate tests)) * 9,722).

Note that OT1 revealed more failures than OS1 for the subjects "TicTacToe" and "Triangle," and OT3 revealed more failures than OS3 for the subject "TicTacToe." As discussed in Section 4.2, OTi can check different program states from OSi, where $1 \leq i \leq 4$. Thus, OTi can sometimes reveal more failures than OSi. Tables 5 and 6 show that NOS revealed far fewer failures than the other OSes on average, indicating NOS is much less effective at revealing faults. Since with the same frequency and test inputs, a more precise OS checks more program states than a less precise

OS, we expected the more precise OS to reveal more faults, and checking outputs and internal state variables more frequently can reveal more failures than checking the same program states less frequently. However, Tables 5 and 6 show that the proportions of the faults detected by OS1, OS2, OS3, OS4, OS5, OT1, OT2, OT3, OT4, and OT5 were very close, and 3% to 8% higher than that of SIOS. Figure 6 presents the averages of the proportions of faults detected by each OS with EC and EPC. These numbers are derived from Tables 5 and 6.

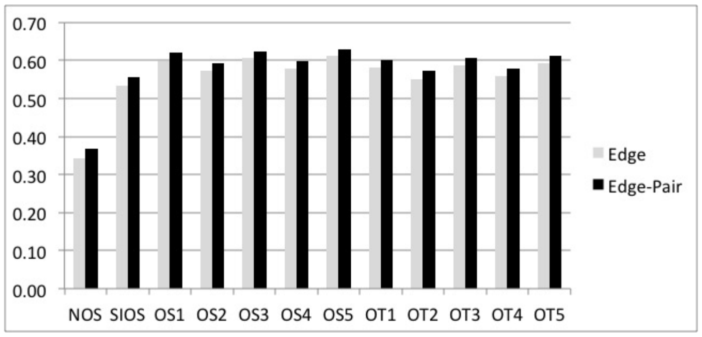

Fig. 6. Effectiveness of Test Oracle Strategies. Effectiveness is the average of the proportions of faults detected by each OS with EC and EPC.

5.4.2 Statistical Comparison of OSes in the RQs

To analyze the RQs statistically, we used Qqplots [26] to determine that the proportions of failures revealed by the OSes for both EC and EPC were not normally distributed. Figure 7 shows one Qqplot for the effectiveness of the EC tests for NOS. We do not show the others because they all look similar. Because these data deviate from a straight line, the proportions of faults detected by the OSes for EC and EPC were not normally distributed.

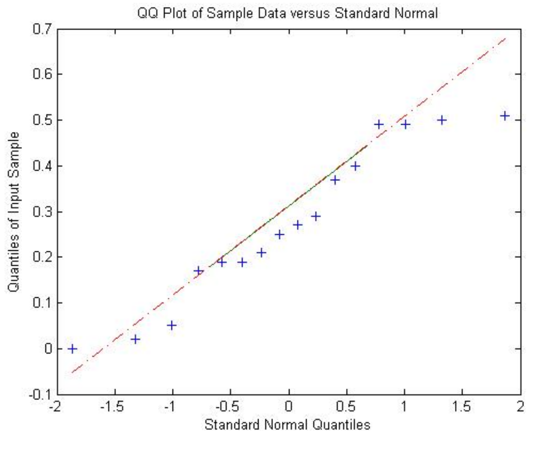

Fig. 7. Qqplot for NOS on Edge Coverage. The deviation from the straight line indicates the proportions of failures revealed were not normally distributed.

To get statistical evidence of the effectiveness difference between a less precise OS and a more precise OS with the same frequency and test inputs, we used the one-tailed Wilcoxon signed-rank test (statistical significance level α = 0.05) [28] to compare the paired proportions of the faults detected by two different OSes for both EC and EPC. We used the one-tailed Wilcoxon signed-rank test because the compared data were paired and came from the same tests (EC or EPC tests). For instance, SIOS for EC was compared with OS1 for EC. This is a non-parametric test to assess whether two population means differ when data are not normally distributed. This test first finds the absolute difference for each pair and gets the number of the pairs that are different, N. Then this test ranks the pairs and calculates the test statistic W. If N is greater than 9, the sampling distribution of W is a reasonably close approximation of the normal distribution and the one-tail probability p can be calculated. According to Lowry [28], if N is less than or equal to 9, but greater than 4, the calculated W value is compared to a $W_{critical}$ from the separate table of critical values of W.

For $Hypotheses_A$, 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}, {OS2, OS4}, {SIOS, OT1}, {SIOS, OT3}, {SIOS, OT5}, {OT1, OT3}, {OT3, OT5}, {OT1, OT5}, {OT2, OT5}, {OT4, OT5}, {SIOS, OT2}, {SIOS, OT4}, {OT2, OT3}, and {OT2, OT4} were compared for EC and EPC using the Wilcoxon signed rank test. Table 7 shows the detailed results, with p-values and effect sizes, for $Hypotheses_A$. We got p-values from 0.0003 - 0.0018 for {NOS, SIOS}, {SIOS, OS1}, {SIOS, OS3}, {SIOS, OS5}, {OS2, OS5}, {SIOS, OT1}, {SIOS, OT3}, {SIOS, OT5}, {OT2, OT5}, {OT4, OT5}, and {OT2, OT3} for both EC and EPC as well as {OS2, OS3} and {OS4, OS5} for EC because the N values of these pairs were greater than 9. 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. Pairs {SIOS, OS2}, {SIOS, OS4}, {SIOS, OT4} and {OT1, OT5} for EC and EPC as well as {SIOS, OT2} for EC and {OS2, OS3} and {OS4, OS5} for EPC had N less than 10 but greater than 4, thus, the table of $W_{critical}$ values was used. Because $|W| > W_{critical}$, we concluded that the differences between the OSes in these pairs were not due to chance.

The N values for {OS1, OS5} for EC and EPC as well as {SIOS, OT2} for EPC were less than 10 but greater than 4. Because the calculated $|W| \leq W_{critical}$, we concluded that there were no significant differences between these pairs. For pairs {OS1, OS3}, {OS3, OS5}, {OS2, OS4}, {OT1, OT3}, {OT3, OT5}, and {OT2, OT4} for EC and EPC, the N values of OSes were less than five, so we could not perform the Wilcoxon signed-rank test. This also implied that there was no significant difference between these pairs.

Some OS pairs had inconsistent results for EC and EPC. For pair {SIOS, OT2}, the EC tests showed that OT2 is more effective than SIOS using the Wilcoxon signed-rank test but the EPC tests showed that OT2 is as effective as SIOS.

In summary, for $Hypotheses_A$, we reject H_0 for the pairs {NOS, SIOS}, {SIOS, OS1}, {SIOS, OS3}, {SIOS, OS5}, {OS2, OS5}, {OS4, OS5}, {OS2, OS3}, {SIOS, OT1}, {SIOS, OT3}, {SIOS, OT5}, {OT2, OT5}, {OT4, OT5}, {OT2, OT3}, {SIOS, OS2}, {SIOS, OS4}, {SIOS, OT4} and {OT1, OT5} for both EC and EPC. In addition, we reject H_0 for the pair {SIOS, OT2} for EC only. Table 7 shows the rejected pairs in bold. If a pair is rejected for both EC and EPC, the pair is in bold. If a pair is rejected for either EC or EPC, but not both,

TABLE 3
Numbers of Faults Found by Test Oracle Strategies, Part 1

Programs	#Faults	# Faults Found by Test Oracle Strategies													
		Edge							Edge-Pair						
		NOS	**SIOS**	**OS1**	**OS2**	**OS3**	**OS4**	**OS5**	**NOS**	**SIOS**	**OS1**	**OS2**	**OS3**	**OS4**	**OS5**
ATM	257	49	76	182	172	182	201	206	53	81	184	175	184	201	206
BlackJack	56	15	19	22	19	22	19	22	15	19	22	19	22	19	22
Calculator	494	94	205	228	205	228	205	228	205	228	246	223	246	223	246
CrossLexic	470	176	206	209	209	209	209	209	182	211	214	214	214	214	214
DFGraph Coverage	683	274	274	415	274	415	274	415	274	274	415	274	415	274	415
Dynamic Parser	3,378	1,723	2,300	2,300	2,300	2,300	2,300	2,300	1,724	2,301	2,301	2,301	2,301	2,301	2,301
Graph Coverage	385	187	210	301	210	301	210	301	187	210	301	210	301	210	301
JMines	263	66	66	79	66	79	66	79	202	202	205	202	205	202	205
Logic Coverage	436	218	375	381	375	381	375	381	217	375	381	375	381	375	381
MM Coverage	845	143	251	262	251	262	251	262	143	251	262	251	262	251	262
Poly	259	128	249	250	250	250	250	250	131	250	250	250	250	250	250
Roc	95	21	26	26	30	30	41	41	21	26	26	30	30	41	41
Snake	572	164	225	421	400	421	400	421	216	226	422	401	422	401	422
TicTacToe	1,045	56	464	464	486	486	509	509	56	507	507	507	507	523	523
Tree	113	24	60	70	64	70	64	70	33	67	70	70	70	70	70
Triangle	263	4	128	140	166	168	166	168	4	128	140	171	172	171	172
Vending Machine	108	0	65	75	90	90	90	90	0	66	77	91	91	91	91
Total	**9,722**	**3,342**	**5,199**	**5,825**	**5,567**	**5,894**	**5,630**	**5,952**	**3,577**	**5,417**	**6,023**	**5,764**	**6,073**	**5,817**	**6,122**

TABLE 4
Numbers of Faults Found by Test Oracle Strategies, Part 2

Programs	#Faults	# Faults Found by Test Oracle Strategies													
		Edge							Edge-Pair						
		NOS	**SIOS**	**OT1**	**OT2**	**OT3**	**OT4**	**OT5**	**NOS**	**SIOS**	**OT1**	**OT2**	**OT3**	**OT4**	**OT5**
ATM	257	49	76	175	171	178	198	206	53	81	177	173	180	201	206
BlackJack	56	15	19	21	19	21	19	22	15	19	21	19	21	19	22
Calculator	494	94	224	205	224	205	224	240	119	223	240	223	240	223	240
CrossLexic	470	176	206	209	206	209	206	209	182	211	214	211	214	211	214
DFGraph Coverage	683	274	274	415	274	415	274	415	274	274	415	274	415	274	415
Dynamic Parser	3,378	1,723	2,300	2,300	2,300	2,300	2,300	2,300	1,724	2,301	2,301	2,301	2,301	2,301	2,301
Graph Coverage	385	187	210	275	210	275	210	275	187	210	275	210	275	210	275
JMines	263	66	66	68	66	68	66	68	202	202	204	202	204	202	204
Logic Coverage	436	218	375	381	375	381	375	381	217	375	381	375	381	375	381
MM Coverage	845	143	251	260	251	260	251	260	143	251	260	251	260	251	260
Poly	259	128	249	249	249	249	249	249	131	250	250	250	250	250	250
Roc	95	21	26	26	30	30	41	41	21	26	26	30	30	41	41
Snake	572	164	225	286	225	286	225	286	216	226	286	226	286	226	286
TicTacToe	1,045	56	464	489	486	489	512	512	56	507	510	508	510	527	527
Tree	113	24	60	70	64	70	64	70	33	67	70	70	70	70	70
Triangle	263	4	128	150	148	158	148	158	4	128	158	154	166	154	166
Vending Machine	108	0	65	65	83	83	83	83	0	66	66	84	84	84	84
Total	**9,722**	**3,342**	**5,199**	**5,663**	**5,362**	**5,696**	**5,426**	**5,759**	**3,577**	**5,417**	**5,854**	**5,561**	**5,887**	**5,619**	**5,942**

TABLE 5
Effectiveness of Test Oracle Strategies, Part 1

Programs	**#Faults**	**Proportions of Faults Detected by Test Oracle Strategies**													
		Edge							**Edge-Pair**						
		NOS	**SIOS**	**OS1**	**OS2**	**OS3**	**OS4**	**OS5**	**NOS**	**SIOS**	**OS1**	**OS2**	**OS3**	**OS4**	**OS5**
ATM	257	0.19	0.30	0.71	0.67	0.71	0.78	0.80	0.21	0.32	0.72	0.68	0.72	0.78	0.80
BlackJack	56	0.27	0.34	0.39	0.34	0.39	0.34	0.39	0.27	0.34	0.39	0.34	0.39	0.34	0.39
Calculator	494	0.19	0.41	0.46	0.41	0.46	0.41	0.46	0.24	0.45	0.50	0.45	0.50	0.45	0.50
CorssLexic	470	0.37	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
DFGraph Coverage	683	0.40	0.40	0.61	0.40	0.61	0.40	0.61	0.40	0.40	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	0.51	0.68	0.68	0.68	0.68	0.68	0.68
Graph Coverage	385	0.49	0.55	0.78	0.55	0.78	0.55	0.78	0.49	0.55	0.78	0.55	0.78	0.55	0.78
JMines	263	0.25	0.25	0.30	0.25	0.30	0.25	0.30	0.77	0.77	0.78	0.77	0.78	0.77	0.78
Logic Coverage	436	0.50	0.86	0.87	0.86	0.87	0.86	0.87	0.50	0.86	0.87	0.86	0.87	0.86	0.87
MM Cov-erage	845	0.17	0.30	0.31	0.30	0.31	0.30	0.31	0.17	0.30	0.31	0.30	0.31	0.30	0.31
Poly	259	0.49	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
Roc	95	0.22	0.27	0.27	0.32	0.32	0.43	0.43	0.22	0.27	0.27	0.32	0.32	0.43	0.43
Snake	572	0.29	0.39	0.74	0.70	0.74	0.70	0.74	0.38	0.40	0.74	0.70	0.74	0.70	0.74
TicTacToe	1,045	0.05	0.44	0.44	0.47	0.47	0.49	0.49	0.05	0.49	0.49	0.49	0.49	0.50	0.50
Tree	113	0.21	0.53	0.62	0.57	0.62	0.57	0.62	0.29	0.59	0.62	0.62	0.62	0.62	0.62
Triangle	263	0.02	0.49	0.53	0.63	0.64	0.63	0.64	0.02	0.49	0.53	0.65	0.65	0.65	0.65
Vending Machine	108	0.00	0.60	0.69	0.83	0.83	0.83	0.83	0.00	0.61	0.71	0.84	0.84	0.84	0.84
Average	**9,722**	**0.34**	**0.53**	**0.60**	**0.57**	**0.61**	**0.58**	**0.61**	**0.37**	**0.56**	**0.62**	**0.59**	**0.62**	**0.60**	**0.63**

only the rejected coverage column is in bold.

In addition to the p-values, W values, and $W_{critical}$ values, Table 7 also reports the effect sizes for each pair when the number of differences is greater then 9. The effect size r is computed by the z-ratio Z over the square root of N_{total}, where N_{total} is the total number of all samples. In this experiment, N_{total} = 34. Because the Wilcoxon signed rank test cannot calculate Z values when $N \leq 9$, we were not able to compute the effect sizes for the pairs whose $N \leq 9$. The thresholds for small effect, medium effect, and large effect are 0.1, 0.3, and 0.5, respectively. Therefore, all pairs whose $N > 9$ have large effect sizes.

Although Tables 5 and 6 show that a more precise OS can detect a higher average proportion of faults than a less precise OS, the results of the Wilcoxon signed-rank test showed that the proportion of faults detected by a more precise OS might not be significantly different from that of a less precise OS (such as {OS1, OS5}). Therefore, **the answer to RQ1** is: for any two OSes that have different precision, the more precise OS is not necessarily more effective than the less precise OS, with the same frequency and test inputs.

We also used the Wilcoxon signed rank test for $Hypotheses_B$ to decide if the frequency of checking variables impacts the effectiveness of OSes for five pairs {OT1, OS1}, {OT2, OS2}, {OT3, OS3}, {OT4, OS4}, and {OT5, OS5} for EC and EPC. The results in Table 8 showed that we can reject H_0 for four pairs: {OT1, OS1}, {OT3, OS3}, {OT5, OS5} for EC and {OT3, OS3} for EPC. We were not able to reject H_0 for four other pairs: {OT4, OS4} for EC and {OT1, OS1}, {OT2, OS2}, and {OT5, OS5} for EPC. The numbers of different pairs of {OT2, OSs} for EC and {OT4, OS4} for EPC were less than 5, thus the Wilcoxon signed rank test could not be applied to these two pairs. In Table 8, we mark the rejected pairs in bold, similar to Table 7. From Tables 5 and 6, the difference between the average proportions of faults detected by OT_i and OS_i ($1 \leq i \leq 5$) for both EC and EPC is very small. Therefore, **the answer to RQ2** is: checking program states multiple times was not always significantly more effective than checking the same program states once.

Tables 5 and 6 show that the edge-adequate tests reveal almost the same number of failures as the EP-adequate tests with the same OS. We used the one-tailed Mann-Whitney test (statistical significance level α = 0.05) [28] to look for statistical evidence that EPC is more effective than EC. We used the Mann-Whitney test because the comparison was between two independent tests, EC and EPC, with the same OS. We applied each OS to the edge-adequate and EP-adequate tests for each program and then compared the proportions of the faults detected by the two paired sets of tests that have the same OS. Table 9 reports the detailed results. Because the U values are between the lower and upper limits and $p - values$ are much greater than p (0.05). we cannot reject H_0 for $Hypotheses_C$. Therefore, no pairs are marked in bold. The effect sizes are computed using the same formula above. Since all the effect sizes are less than

TABLE 6
Effectiveness of Test Oracle Strategies, Part 2

Programs	**#Faults**	**Proportions of Faults Detected by Test Oracle Strategies**													
		Edge							**Edge-Pair**						
		NOS	**SIOS**	**OT1**	**OT2**	**OT3**	**OT4**	**OT5**	**NOS**	**SIOS**	**OT1**	**OT2**	**OT3**	**OT4**	**OT5**
ATM	257	0.19	0.30	0.68	0.67	0.69	0.77	0.80	0.21	0.32	0.69	0.67	0.70	0.78	0.80
BlackJack	56	0.27	0.34	0.38	0.34	0.38	0.34	0.39	0.27	0.34	0.38	0.34	0.38	0.34	0.39
Calculator	494	0.19	0.41	0.45	0.41	0.45	0.41	0.45	0.24	0.45	0.49	0.45	0.49	0.45	0.49
CorssLexic	470	0.37	0.44	0.44	0.44	0.44	0.44	0.44	0.39	0.45	0.46	0.45	0.46	0.45	0.46
DFGraph Coverage	683	0.40	0.40	0.61	0.40	0.61	0.40	0.61	0.40	0.40	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	0.51	0.68	0.68	0.68	0.68	0.68	0.68
Graph Coverage	385	0.49	0.55	0.71	0.55	0.71	0.55	0.71	0.49	0.55	0.71	0.55	0.71	0.55	0.71
JMines	263	0.25	0.25	0.26	0.25	0.26	0.25	0.26	0.77	0.77	0.78	0.77	0.78	0.77	0.78
Logic Coverage	436	0.50	0.86	0.87	0.86	0.87	0.86	0.87	0.50	0.86	0.87	0.86	0.87	0.86	0.87
MM Coverage	845	0.17	0.30	0.31	0.30	0.31	0.30	0.31	0.17	0.30	0.31	0.30	0.31	0.30	0.31
Poly	259	0.49	0.96	0.96	0.96	0.96	0.96	0.96	0.51	0.97	0.97	0.97	0.97	0.97	0.97
Roc	95	0.22	0.27	0.27	0.32	0.32	0.43	0.43	0.22	0.27	0.27	0.32	0.32	0.43	0.43
Snake	572	0.29	0.39	0.50	0.39	0.50	0.39	0.50	0.38	0.40	0.50	0.40	0.50	0.40	0.50
TicTacToe	1,045	0.05	0.44	0.47	0.47	0.47	0.49	0.49	0.05	0.49	0.49	0.49	0.49	0.50	0.50
Tree	113	0.21	0.53	0.62	0.57	0.62	0.57	0.62	0.29	0.59	0.62	0.62	0.62	0.62	0.62
Triangle	263	0.02	0.49	0.57	0.56	0.60	0.56	0.60	0.02	0.49	0.60	0.59	0.63	0.59	0.63
Vending Machine	108	0.00	0.60	0.60	0.77	0.77	0.77	0.77	0.00	0.61	0.61	0.78	0.78	0.78	0.78
Average	**9,722**	**0.34**	**0.53**	**0.58**	**0.55**	**0.59**	**0.56**	**0.59**	**0.37**	**0.56**	**0.60**	**0.57**	**0.61**	**0.58**	**0.61**

0.20, all pairs have small effect sizes. Therefore, **the answer to RQ3**: is that the stronger coverage criterion (EPC) was not found to be more effective than the weaker criterion (EC) with the same OS.

5.4.3 Costs of Test Oracle Strategies

Table 10 shows how many distinct assertions were created by hand. Because each OTi uses the same number of distinct assertions as OSi, where $1 \leq i \leq 5$, we put the costs of both OSi and OTi in the same table. Since with the same frequency, a more precise OS checks more outputs and internal state variables than a less precise OS, more distinct assertions were created for the more precise OSes. Thus, the cost of OS5 was greater than any other OSi and the cost of OT5 was greater than OT1 through OT4. Figure 8 shows the costs of each OS. The numbers are derived from Table 10. This figure also shows the number of distinct assertions on each column since the differences between EC and EPC are very small.

Since testers need to write test oracles for all outputs and internal state variables that appear in all transitions for each OT test oracle strategy, testers have to analyze the effects of each transition by the end of each test. For the experimental subjects, we had to analyze each transition and write the same test oracles as for OSi, $1 \leq i \leq 5$, because each distinct transition produced different program states by the end of the tests. Thus, the cost of OTi is equivalent to that of OSi. Furthermore, OS1, OS3, OS5, OT1, OT3,

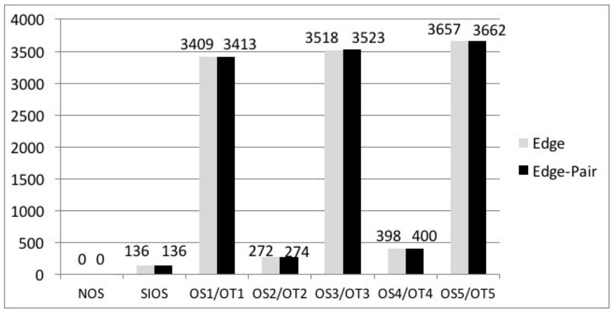

Fig. 8. Costs of Test Oracle Strategies. The numbers are derived from Table 10.

and OT5 required similar numbers of distinct assertions but have far more assertions than OS2, OS4, OT2, and OT4. This was because object members checked by OS1, OS3, OS5, OT1, OT3, and OT5 produced lots of assertions since we checked the member variables of objects recursively (a deep comparison). For the same OS, the EP-adequate tests did not have many more distinct assertions than the edge-adequate tests. This was because we did not need to create many more mappings to satisfy EPC than the mappings created for EC, as discussed in Section 5.2.

For completeness, Tables 11 and 12 show the total numbers of assertions used for each OS. As stated in Section 5.1, we use the number of distinct assertions for cost, not the numbers in Tables 11 and 12. As before, Table 11

TABLE 7
Experimental Results for Hypotheses$_A$. Compares all pairs of OSes in terms of precision, with frequency held constant.

Hypotheses$_A$ Pairs	**Edge**		**Edge-Pair**	
	p-values / W values	**Effect Sizes**	**p-values / W values**	**Effect Sizes**
{NOS, SIOS}	0.0003	0.58	0.0003	0.58
{SIOS, OS1}	0.0008	0.54	0.0008	0.54
{SIOS, OS3}	0.0003	0.58	0.0005	0.56
{SIOS, OS5}	0.0003	0.58	0.0003	0.58
{OS1, OS3}	N = 4	N/A	N = 3	N/A
{OS3, OS5}	N = 3	N/A	N = 3	N/A
{OS1, OS5}	$\lvert W\rvert = 15 \leq W_{critical} = 15$	N/A	$\lvert W\rvert = 15 \leq W_{critical} = 15$	N/A
{OS2, OS5}	0.0008	2	0.0018	0.50
{OS4, OS5}	0.0018	0.50	$\lvert W\rvert = 45 > W_{critical} = 29$	N/A
{SIOS, OS2}	$\lvert W\rvert = 36 > W_{critical} = 26$	N/A	$\lvert W\rvert = 28 > W_{critical} = 22$	N/A
{SIOS, OS4}	$\lvert W\rvert = 36 > W_{critical} = 26$	N/A	$\lvert W\rvert = 36 > W_{critical} = 26$	N/A
{OS2, OS3}	0.0018	0.50	$\lvert W\rvert = 45 > W_{critical} = 29$	N/A
{OS2, OS4}	N = 3	N/A	N = 3	N/A
{SIOS, OT1}	0.0012	0.52	0.0012	0.52
{SIOS, OT3}	0.0005	0.56	0.0005	0.56
{SIOS, OT5}	0.0005	0.56	0.0003	0.58
{OT1, OT3}	N = 4	N/A	N = 4	N/A
{OT3, OT5}	N = 4	N/A	N = 4	N/A
{OT1, OT5}	$\lvert W\rvert = 21 > W_{critical} = 17$	N/A	$\lvert W\rvert = 21 > W_{critical} = 17$	N/A
{OT2, OT5}	0.0008	0.54	0.0008	0.54
{OT4, OT5}	0.0018	0.50	0.0018	0.50
{SIOS, OT2}	$\lvert W\rvert$ **= 21** $> W_{critical}$ **= 17**	N/A	$\lvert W\rvert = 15 \leq W_{critical} = 15$	N/A
{SIOS, OT4}	$\lvert W\rvert = 21 > W_{critical} = 17$	N/A	$\lvert W\rvert = 21 > W_{critical} = 17$	N/A
{OT2, OT3}	0.0018	0.50	0.0018	0.50
{OT2, OT4}	N = 3	N/A	N = 3	N/A

TABLE 8
Experimental Results for $Hypotheses_B$ Compares all pairs of OSes in terms of frequency, with precision held constant.

Hypotheses$_B$ Pairs	**Edge**		**Edge-Pair**	
	p-values / W values	**Effect Sizes**	**p-values / W values**	**Effect Sizes**
{OT1, OS1}	**0.0969**	**0.28**	$\lvert W\rvert = 19 \leq W_{critical} = 24$	N/A
{OT2, OS2}	N = 4	N/A	$\lvert W\rvert = 15 \leq W_{critical} = 15$	N/A
{OT3, OS3}	$\lvert W\rvert$ **= 45** $> W_{critical}$ **= 29**	N/A	$\lvert W\rvert$ **= 28** $> W_{critical}$ **= 22**	N/A
{OT4, OS4}	$\lvert W\rvert = 15 \leq W_{critical} = 15$	N/A	N = 4	N/A
{OT5, OS5}	$\lvert W\rvert$ **= 28** $> W_{critical}$ **= 22**	N/A	$\lvert W\rvert = 15 \leq W_{critical} = 15$	N/A

TABLE 9
Experimental Results for Hypotheses$_C$. Compares pairs of test coverage criteria with the OS held constant.

Hypotheses$_C$ Pairs	**Lower Limit**	**Upper Limit**	U_{value}	$p - value$	$Z - value$	**Effect Sizes** $r - value$
{NOS (EC), NOS (EPC)}	96	193	123	0.2358	-0.72	0.12
{SIOS (EC), SIOS (EPC)}	96	193	120.5	0.2090	-0.81	0.14
{OS1 (EC), OS1 (EPC)}	96	193	126	0.2676	-0.62	0.11
{OS2 (EC), OS2 (EPC)}	96	193	126	0.2676	-0.62	0.11
{OS3 (EC), OS3 (EPC)}	96	193	127	0.2776	-0.59	0.10
{OS4 (EC), OS4 (EPC)}	96	193	127	0.2776	-0.59	0.10
{OS5 (EC), OS5 (EPC)}	96	193	129	0.2981	-0.53	0.09
{OT1 (EC), OT1 (EPC)}	96	193	124	0.2451	-0.69	0.12
{OT2 (EC), OT2 (EPC)}	96	193	124	0.2451	-0.69	0.12
{OT3 (EC), OT3 (EPC)}	96	193	122	0.2236	-0.76	0.13
{OT4 (EC), OT4 (EPC)}	96	193	122	0.2236	-0.76	0.13
{OT5 (EC), OT5 (EPC)}	96	193	124	0.2451	-0.69	0.12

shows the number of assertions for the high frequency test oracle strategies (OS1-OS5), plus NOS and SIOS, and Table 12 shows the number of assertions for the low frequency strategies (OT1-OT5) as well as NOS and SIOS.

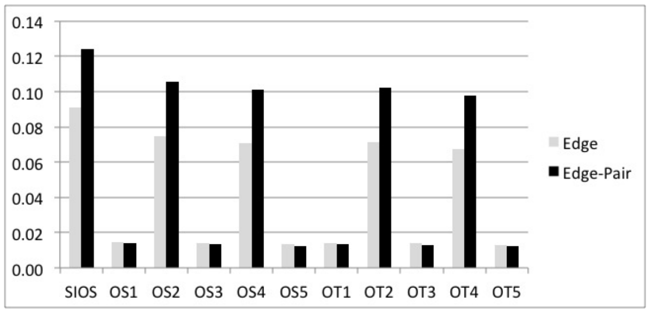

Fig. 9. Averages of Cost-effectiveness. Based on formula 1 over all programs.

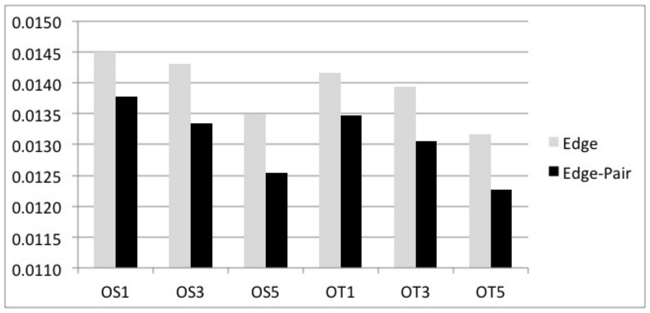

Fig. 10. Averages of Cost-effectiveness Below 0.02. This figure expands six entries from Figure 9 to show them on a larger scale.

5.4.4 Cost-Effectiveness of Test Oracle Strategies

Figure 9 gives the average cost-effectiveness from formula 1 over all the programs for EC and EPC. The figure shows that the strategies that check "all object members" (the odd numbered OSes and OTs) are simply not cost-effective. Checking all object members is clearly expensive, and the data show that they do not reveal significantly more failures. The other strategies are shown on a different scale in Figure 10 to better see their differences. Figure 9 shows that SIOS is the most cost-effective, followed by OS2, OT2, OS4, and OT4. The details about the cost-effectiveness of OSes for EC and EPC are shown in Tables 13 and 14. As stated in Section 5.1, the cost-effectiveness of NOS is not considered. Since we only measure the number of assertions generated, and NOS needs no assertions, its cost-effectiveness would be incomparable. We omit the cost of generating tests because it is a constant across all OSes. However, because of NOS's extremely low effectiveness, its cost really should include the cost of generating the two-thirds of the failure-causing tests that are wasted by not checking their results at all. More discussion is in Section 5.5.

5.5 Discussion and Recommendations

This section discusses the experimental results from five aspects. Section 5.5.1 compares the subsumption relationship among OSes from theoretical and empirical perspectives. Section 5.5.2 generally discusses the effectiveness of OSes based on the experimental results. Sections 5.5.3 and 5.5.4 discusses RQ2 and RQ3. Section 5.5.5 presents general guidance for selecting OSes.

5.5.1 Comparing Theoretical and Empirical Results for OS Subsumption Relationships

We found statistical evidence that the more precise OS was more effective in terms of the proportion of failures revealed than the less precise OS for several pairs, with the same test inputs and frequencies. According the definition of OS, more precise OSes subsume less precise OSes. Figures 11 and 12 show the empirical results of these subsumption relationships among the OSes that have different precision but the same frequency. An arrow from strategy OS_A to OS_B indicates that our data showed that OS_A is *more effective* that OS_B. For example, Figure 11 shows that OS3 is more effective than OS2, which is more effective than SIOS. All strategies are more effective than NOS. Figure 11 shows the *more effective* relationships among just the OS strategies, plus SIOS and NOS. Figure 12 shows the *more effective* relationships among the OT strategies, plus SIOS and NOS.

Figure 5 gave the theoretical subsumption relationships among different OSes and OTs. In theory, OS5 subsumes OS3, which subsumes OS1. OS4 subsumes OS2. OT5 subsumes OT3, which subsumes OT1. OT4 subsumes OT2, which subsumes SIOS. By comparing Figure 5 to Figure 11 and 12, however, statistical analysis shows that an OS_A that subsumes another OS_B in theory is not necessarily more effective than OS_B in practice. Generally, this demonstrates that although subsumption indicates a theoretical difference, subsumption does not always lead to a difference in practice. While it is probably safe, on average, to assume that if OS_A subsumes OS_B, OS_A will be *at least as* effective as OS_B, OS_A is not necessarily *more* effective. It is possible that if the difference in precision between two OSes is small, there will be little or no difference in the effectiveness of the OSes.

Note that in Figures 11 and 12, we did not use transitivity for most OS pairs. We explicitly ran the tests among all the pairs except the comparisons between NOS and the other OSes. For an effective relationship like OS5-OS2-SIOS-NOS in Figure 11, we ran tests on the pairs {NOS, SIOS}, {SIOS, OS2}, {SIOS, OS5}, and {OS2, OS5}. In the experiments, we did not compare NOS with other OSes because NOS is much less effective than, and incomparable with, the others. Moreover, we can use transitivity to determine that NOS is statistically different from the other OSes. For the first group of hypotheses, we used the *one-tailed (not two-tailed)* Wilcoxon signed-rank test. This means that for *each* subject, the values for OSi ($1 \leq$ i ≤ 5) are greater than or equal to the values for SIOS, and the values for SIOS are greater than or equal to the values for NOS. If OSi ($1 \leq$ i ≤ 5) is statistically different from SIOS and SIOS is statistically different from NOS, OSi ($1 \leq$ i ≤ 5) is statistically different from NOS because the differences between the values for OSi ($1 \leq$ i ≤ 5) and those for NOS are even greater than the differences between the values for OSi ($1 \leq$ i ≤ 5) and those for SIOS.

TABLE 10
Cost of Test Oracle Strategies–Total Number of Distinct Assertions

Programs	Cost of Test Oracle Strategies													
	Edge							Edge-Pair						
	NOS	SIOS	OS1 OT1	OS2 OT2	OS3 OT3	OS4 OT4	OS5 OT5	NOS	SIOS	OS1 OT1	OS2 OT2	OS3 OT3	OS4 OT4	OS5 OT5
ATM	0	6	49	34	49	50	74	0	6	49	34	49	50	74
BlackJack	0	3	552	3	552	3	552	0	3	552	3	552	0	552
Calculator	0	9	123	9	123	9	123	0	9	123	9	123	9	123
CorssLexic	0	7	127	7	127	7	127	0	7	127	7	127	7	127
DFGraph Coverage	0	7	163	7	163	7	163	0	7	163	7	163	7	163
Dynamic Parser	0	15	23	15	23	15	23	0	15	23	15	23	15	23
Graph Coverage	0	11	156	11	156	11	156	0	11	156	11	156	11	156
JMines	0	1	792	82	792	82	792	0	1	792	82	792	82	792
Logic Coverage	0	8	181	8	181	8	181	0	8	181	8	181	8	181
MM Coverage	0	16	587	16	587	16	587	0	16	584	16	587	16	587
Poly	0	6	12	10	12	10	12	0	6	14	12	14	12	14
Roc	0	22	66	108	152	164	211	0	22	66	108	152	164	211
Snake	0	8	463	8	463	8	463	0	8	463	8	463	8	463
TicTacToe	0	3	33	6	36	60	90	0	3	33	6	36	60	90
Tree	0	3	23	14	34	14	34	0	3	23	14	34	14	34
Triangle	0	5	42	6	51	9	51	0	5	42	6	51	6	51
Vending Machine	0	6	17	9	17	9	18	0	6	19	9	20	9	21
Total	**0**	**136**	**3,409**	**272**	**3,518**	**398**	**3,657**	**0**	**136**	**3,413**	**274**	**3,523**	**400**	**3,662**

TABLE 11
Cost of Test Oracle Strategies–Total Number of All Assertions, Part 1 (OS1-OS5)

Programs	Cost of Test Oracle Strategies													
	Edge							Edge-Pair						
	NOS	SIOS	OS1	OS2	OS3	OS4	OS5	NOS	SIOS	OS1	OS2	OS3	OS4	OS5
ATM	0	23	156	51	156	83	188	0	40	250	103	250	175	322
BlackJack	0	27	1,313	27	1,313	27	1,313	0	52	2,473	52	2,585	52	2,585
Calculator	0	168	1,908	168	1,908	168	1,908	0	894	10,047	1,313	10,047	1,323	10,047
CorssLexic	0	210	1,304	410	1,304	410	1,304	0	757	4,852	1,261	4,852	1,261	4,852
DFGraph Coverage	0	79	1,091	167	1,091	167	1,091	0	644	9,016	1,139	9,016	1,139	9,016
Dynamic Parser	0	409	456	409	456	409	456	0	1,234	1,417	1,234	1,417	1,234	1,417
Graph Coverage	0	208	1,796	448	1,796	448	1,796	0	606	4,904	951	4,904	951	4,904
JMines	0	42	6,428	805	6,428	805	6,428	0	152	21,553	1,911	21,553	1,911	21,553
Logic Coverage	0	95	1,471	155	1,471	155	1,471	0	484	7,766	674	7,766	674	7,766
MM Coverage	0	229	4,911	541	5,208	541	5,208	0	571	12,746	1,139	13,226	1,139	13,226
Poly	0	58	81	91	91	91	91	0	238	333	357	357	357	357
Roc	0	133	331	494	694	761	973	0	313	829	1194	1710	1867	2410
Snake	0	121	3,229	450	3,229	450	3,229	0	342	9,128	718	9,128	718	9,128
TicTacToe	0	8	160	16	167	267	437	0	17	236	33	252	609	828
Tree	0	49	209	87	247	87	247	0	147	587	329	769	329	769
Triangle	0	37	310	90	315	90	315	0	272	2,290	538	2,340	538	2,340
Vending Machine	0	89	156	96	163	96	170	0	211	380	229	398	229	421
Total	**0**	**1,985**	**25,310**	**4,505**	**26,037**	**5,055**	**26,625**	**0**	**6,974**	**88,807**	**13,185**	**90,570**	**14,506**	**91,941**

TABLE 12
Cost of Test Oracle Strategies–Total Number of All Assertions, Part 2 (OT1-OT5)

Programs	**Cost of Test Oracle Strategies**													
	Edge							**Edge-Pair**						
	NOS	**SIOS**	**OT1**	**OT2**	**OT3**	**OT4**	**OT5**	**NOS**	**SIOS**	**OT1**	**OT2**	**OT3**	**OT4**	**OT5**
ATM	0	23	68	42	69	80	92	0	40	85	56	87	99	119
BlackJack	0	27	475	27	475	27	475	0	52	556	52	556	52	556
Calculator	0	168	322	322	322	322	322	0	894	1,323	1,313	1,323	1,323	1,323
CorssLexic	0	210	420	210	420	210	420	0	757	1,288	757	1,288	757	1,288
DFGraph Coverage	0	79	262	79	262	79	262	0	644	1,675	644	1,675	644	1,675
Dynamic Parser	0	409	428	409	428	409	428	0	1,234	1,260	1,234	1,260	1,234	1,260
Graph Coverage	0	208	423	208	423	208	423	0	606	920	606	920	606	920
JMines	0	42	792	42	792	42	792	0	152	792	42	792	42	1,802
Logic Coverage	0	95	801	95	801	95	801	0	484	1,870	484	1,870	484	1,870
MM Cov-erage	0	229	3,188	229	3,188	229	3,188	0	571	4,832	571	4,832	571	4,832
Poly	0	58	63	68	68	68	68	0	238	250	250	250	250	250
Roc	0	133	185	424	476	624	676	0	313	417	969	1073	1478	1578
Snake	0	121	359	121	359	121	359	0	342	595	342	595	342	595
TicTacToe	0	8	80	12	80	80	85	0	17	111	23	112	118	118
Tree	0	49	111	62	111	62	111	0	147	211	194	211	194	211
Triangle	0	37	85	42	85	42	90	0	272	488	295	488	295	511
Vending Machine	0	89	90	94	94	94	94	0	211	212	219	219	219	219
Total	**0**	**1,985**	**8,152**	**2,486**	**8,453**	**2,792**	**8,686**	**0**	**6,974**	**16,885**	**8,061**	**17,551**	**8,708**	**19,137**

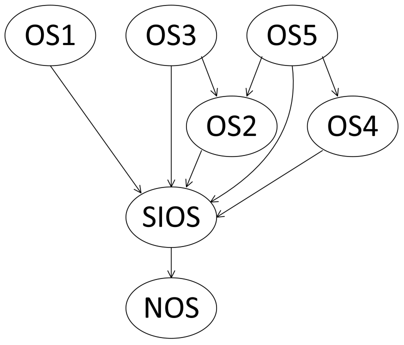

Fig. 11. More effective relationships among the high frequency oracle strategies (OSes). Edges are drawn from more effective strategies to less effective strategies.

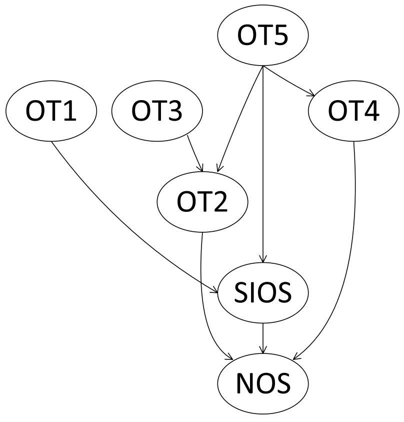

Fig. 12. More effective relationships among the low frequency oracle strategies (OTs). Edges are drawn from more effective strategies to less effective strategies.

5.5.2 Discussion of the Effectiveness of OSes

In Table 5, only 0.63 of the failure were revealed by the EP-adequate tests for the most precise OS (OS5). This is a low score considering that 90% mutation is considered a good test set [1]. 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 revealing failures than EC and EPC on the model. A previous paper [25] 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.

Tables 5 and 6 show that NOS was not very good at revealing failures. If we assume the edge coverage tests were able to reveal a maximum of 5952 failures (the number revealed by OS5), then NOS only revealed 3342 out of a possible 5952 failures, or only 56%. That is, 44% of the effort of designing and building the tests was wasted! In our interpretation, this is like buying a dozen eggs at the grocery but only eating six or seven because we're too lazy

TABLE 13
Cost-effectiveness of Test Oracle Strategies, Part 1

Programs	Cost-effectiveness of Test Oracle Strategies											
	Edge						Edge-Pair					
	SIOS	OS1	OS2	OS3	OS4	OS5	SIOS	OS1	OS2	OS3	OS4	OS5
ATM	0.0493	0.0145	0.0197	0.0145	0.0156	0.0108	0.0525	0.0146	0.0200	0.0146	0.0156	0.0108
BlackJack	0.1131	0.0007	0.1131	0.0007	0.1131	0.0007	0.1131	0.0007	0.1131	0.0007	0.1131	0.0007
Calculator	0.0461	0.0038	0.0461	0.0038	0.0461	0.0038	0.0502	0.0040	0.0502	0.0040	0.0502	0.0040
CrossLexic	0.0626	0.0035	0.0635	0.0035	0.0635	0.0035	0.0641	0.0036	0.0650	0.0036	0.0650	0.0036
DFGraph Coverage	0.0573	0.0037	0.0573	0.0037	0.0573	0.0037	0.0573	0.0037	0.0573	0.0037	0.0573	0.0037
Dynamic Parser	0.0454	0.0296	0.0454	0.0296	0.0454	0.0296	0.0454	0.0296	0.0454	0.0296	0.0454	0.0296
Graph Coverage	0.0496	0.0050	0.0496	0.0050	0.0496	0.0050	0.0496	0.0050	0.0496	0.0050	0.0496	0.0050
JMines	0.2510	0.0004	0.2510	0.0004	0.2510	0.0004	0.7681	0.0010	0.7681	0.0010	0.7681	0.0010
Logic Coverage	0.1075	0.0048	0.1075	0.0048	0.1075	0.0048	0.1075	0.0048	0.1075	0.0048	0.1075	0.0048
MM Coverage	0.0186	0.0005	0.0186	0.0005	0.0186	0.0005	0.0186	0.0005	0.0186	0.0005	0.0186	0.0005
Poly	0.1602	0.0804	0.0965	0.0804	0.0965	0.0804	0.1609	0.0689	0.0804	0.0689	0.0804	0.0689
Roc	0.0124	0.0041	0.0029	0.0021	0.0026	0.0020	0.0124	0.0041	0.0029	0.0021	0.0026	0.0020
Snake	0.0492	0.0016	0.0874	0.0016	0.0874	0.0016	0.0494	0.0016	0.0876	0.0016	0.0876	0.0016
TicTacToe	0.1480	0.0135	0.0775	0.0129	0.0081	0.0054	0.1617	0.0147	0.0809	0.0135	0.0083	0.0056
Tree	0.1770	0.0269	0.0405	0.0182	0.0405	0.0182	0.1976	0.0269	0.0442	0.0182	0.0442	0.0182
Triangle	0.0973	0.0127	0.1052	0.0125	0.1052	0.0125	0.0973	0.0127	0.1084	0.0128	0.1084	0.0128
Vending Machine	0.1003	0.0408	0.0926	0.0490	0.0926	0.0463	0.1019	0.0375	0.0936	0.0421	0.0936	0.0401
Average	**0.0909**	**0.0145**	**0.0750**	**0.0143**	**0.0706**	**0.0135**	**0.1240**	**0.0138**	**0.1055**	**0.0133**	**0.1009**	**0.0125**

to cook the others. Thus, checking runtime exceptions is not enough.

We also noticed that SIOS can reveal more than 80% of the failures 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 failures. In contrast, checking more program states (as OS5 does) that are not affected by the test inputs is not likely to reveal more failures. If more program states have to be checked, checking return values or parameter members (OS2 and OS4) is more cost-effective, while checking object members (OS1, OS3, OS5, OT1, OT3, and OT5) is much more costly but adds little in terms of effectiveness.

5.5.3 Discussion of RQ2

The results show that checking outputs and internal state variables after each transition was not significantly more effective than checking the same outputs and internal state variables 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 as failures if all outputs and internal state variables are checked only once, as the OT strategies do. Checking the same outputs and internal state variables multiple times did not help much in our study. Therefore, if testers use a tool such as STALE to generate test oracles automatically, they should generate test oracles after each transition to achieve higher effectiveness. However, if testers write test oracles by hand, they should check outputs and internal state variables at the end of the automated tests. This lets them avoid writing redundant test oracles for transitions that appear multiple times in tests.

5.5.4 Discussion of RQ3

We found statistical evidence that EP-adequate tests were not significantly more effective than edge-adequate tests (RQ3). This may be because EPC did not require many more mappings on the state machine diagrams, even though EPC had more tests. The models could affect the results. If a model has lots of nodes that have multiple incoming and outgoing edges, the edge-pairs could be very different from edges. Then EPC could result in stronger tests than EC. Furthermore, if a state machine diagram is designed with multiple variables, then each state represents multiple variable states (constraints). EPC abstract tests are usually more complex than EC abstract tests, and tour the same states multiple times. Satisfying the constraints (finding appropriate test values) in complex tests is harder because it causes more program states, when compared with less complex tests. In this case, EPC may need more mappings than EC. It should be noted that this result cannot be assumed to apply to EPC and EC on source code, since this study was restricted to deriving tests from state machine diagrams, which tend to be simpler than control flow graphs.

TABLE 14
Cost-effectiveness of Test Oracle Strategies, Part 2

Programs	Cost-effectiveness of Test Oracle Strategies											
	Edge						Edge-Pair					
	SIOS	OT1	OT2	OT3	OT4	OT5	SIOS	OT1	OT2	OT3	OT4	OT5
ATM	0.0493	0.0139	0.0196	0.0141	0.0154	0.0108	0.0525	0.0141	0.0198	0.0143	0.0156	0.0108
BlackJack	0.1131	0.0007	0.1131	0.0007	0.1131	0.0007	0.1131	0.0007	0.1131	0.0007	0.1131	0.0007
Calculator	0.0461	0.0037	0.0461	0.0037	0.0461	0.0037	0.0502	0.0039	0.0502	0.0039	0.0502	0.0039
CrossLexic	0.0626	0.0035	0.0626	0.0035	0.0626	0.0035	0.0641	0.0036	0.0641	0.0036	0.0641	0.0036
DFGraph Coverage	0.0573	0.0037	0.0573	0.0037	0.0573	0.0037	0.0573	0.0037	0.0573	0.0037	0.0573	0.0037
Dynamic Parser	0.0454	0.0296	0.0454	0.0296	0.0454	0.0296	0.0454	0.0296	0.0454	0.0296	0.0454	0.0296
Graph Coverage	0.0496	0.0046	0.0496	0.0046	0.0496	0.0046	0.0496	0.0046	0.0496	0.0046	0.0496	0.0046
JMines	0.2510	0.0003	0.2510	0.0003	0.2510	0.0003	0.7681	0.0010	0.7681	0.0010	0.7681	0.0010
Logic Coverage	0.1075	0.0048	0.1075	0.0048	0.1075	0.0048	0.1075	0.0048	0.1075	0.0048	0.1075	0.0048
MM Coverage	0.0186	0.0005	0.0186	0.0005	0.0186	0.0005	0.0186	0.0005	0.0186	0.0005	0.0186	0.0005
Poly	0.1602	0.0801	0.0961	0.0801	0.0961	0.0801	0.1609	0.0689	0.0804	0.0689	0.0804	0.0689
Roc	0.0124	0.0041	0.0029	0.0021	0.0026	0.0020	0.0124	0.0041	0.0029	0.0021	0.0026	0.0020
Snake	0.0492	0.0011	0.0492	0.0011	0.0492	0.0011	0.0494	0.0011	0.0494	0.0011	0.0494	0.0011
TicTacToe	0.1480	0.0142	0.0775	0.0130	0.0082	0.0054	0.1617	0.0148	0.0810	0.0136	0.0084	0.0056
Tree	0.1770	0.0269	0.0405	0.0182	0.0405	0.0182	0.1976	0.0269	0.0442	0.0182	0.0442	0.0182
Triangle	0.0973	0.0136	0.0938	0.0118	0.0938	0.0118	0.0973	0.0143	0.0976	0.0124	0.0976	0.0124
Vending Machine	0.1003	0.0354	0.0854	0.0452	0.0854	0.0427	0.1019	0.0322	0.0864	0.0389	0.0864	0.0370
Average	**0.0909**	**0.0142**	**0.0715**	**0.0139**	**0.0672**	**0.0132**	**0.1240**	**0.0135**	**0.1021**	**0.0131**	**0.0976**	**0.0123**

Briand et al. [8] found similar results for RQ3 when they compared *round-trip path coverage* (RT) [7] to *disjunct coverage* (DC) [8] with the very precise test oracle strategy. They used statecharts that have guard constraints while we used state invariant constraints. The guard constraints can be transformed into *disjunctive normal form* (conjunctive expressions combined using the *or* operator). For RT, testers need to generate tests to satisfy all guard constraints (each guard constraint is treated as a whole and only one conjunctive expression needs to be satisfied). For DC, testers need to generate additional tests to satisfy all conjuncts in the guard constraints. Thus, DC is expected to be more effective than RT when the statecharts have disjunctive guard constraints. One experiment showed that DC reveals the same failures as RT for one of the four classes because that class has no disjunctive guard constraints. In another experiment, they used the *category partition* (CP) input-domain based coverage criterion [37] to generate additional tests. They found the tests generated by (DC + CP) revealed more failures than DC and RT for two classes (CP was applied only to two classes) because using CP generated more test inputs.

Our experimental results for RQ3 indicate that a stronger coverage criterion (EPC) may not reveal more failures than a weaker coverage criterion (EC) for model-based testing (we used state machine diagrams). This is because applying a stronger coverage criterion to a model will not always generate more test inputs than a weaker coverage criterion would. This result is in general agreement with Briand et al.'s [8] findings. For example, if a statechart has no disjunctive guard constraints, DC is as effective as RT in terms of detecting faults even if DC subsumes RT. Therefore, whether a stronger coverage criterion can reveal more faults than a weaker coverage criterion depends on the model and coverage criteria.

5.5.5 General Guidance

Checking program states frequently could require too many assertions in tests. OT5 needed 8,686 assertions for EC (19,137 for EPC) and OS5 needed 26,625 (91,941 for EPC), as shown in Tables 11 and 12. 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 adds an additional hidden cost to achieve the additional precision.

Figure 9 shows that SIOS was the most cost-effective for both EC and EPC. Given a time budget, testers can choose an OS that maximizes the effectiveness. When the time budget is tight, SIOS may be the only choice. Otherwise, testers should choose OS2 or OS4 because they are almost as effective as OS5 but require fewer assertions than OS5.

5.6 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. We mitigated that threat by choosing programs that varied by size, application, and type. Another threat to external validity is that we created UML state machine diagrams by hand. If the original programmers created the state machines, they might create different models. Nevertheless, STALE provides a mechanism to reduce errors from the sate machine diagrams when generating tests. We used STALE to create mappings for transitions and constraints in state machine diagrams as well as the associated test oracles. When concrete tests were generated, the test oracle assertions for transitions and constraints were evaluated automatically. If the expected values were not equal to the actual values, we checked the cause of the error. If the error came from the UML diagram, we corrected the diagram to ensure every constraint specified in the diagram is satisfied in the concrete tests.

Another threat to external validity is that we generated tests by using the tool STALE to create mappings by hand. The results may have been different if we used different mappings or coverage criteria or different automated tools (none are available for this test scenario). We also experimented only with model-based testing, although the ideas about test oracles apply to any type of testing.

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. If generating the assertions was partly or completely automated (currently not possible), that could change this analysis and our conclusions dramatically. Another internal threat is that the first author identified equivalent mutants by hand. Mistakes could have affected the results in small ways.

6 Conclusions and Future Work

This paper makes five contributions to software test oracle research. First, this paper extends the traditional and fundamental RIP model to the RIPR model. This fundamental result changes a basic concept in testing and is already being reflected in the second edition of the Introduction to Software Testing textbook [1] (under production). Second, the paper introduces a formal definition for test oracle strategy. Third, we define the concept of *subsumption* for OSes. Fourth, it defines ten new test oracle strategies (OSes) to reveal failures caused by model-based tests. Finally, we empirically compared these ten new OSes with each other and with two baseline OSes: the null test oracle strategy (NOS) and the state invariant strategy (SIOS). Some of the ten test oracle strategies and the empirical results apply strictly to model-based testing. The other contributions are more general and apply to any situation where test automation and test oracles are used.

The traditional RIP model has been around since the 1980s and has subtly guided testing research and practice. However, once we started automating our tests in test automation frameworks, we needed to include automated checks to verify whether the result of the test was correct (a test oracle). Since we cannot check the entire output state of the program, a failure is only *revealed* to the tester if the test oracle checks part of the output state that is incorrect. Thus the RIPR model emphasizes that not only do the tests need to cause faults to result in failures, the tester must also have a way to observe those failures.

The ten novel test oracles vary in terms of precision (how much of the state they observe) and frequency (how often they observe the state). An important question is how much of the state, and how frequently, do we need to observe to reveal as many failures as possible with a reasonable cost.

In our experiment, we generated test inputs to satisfy two graph coverage test criteria on 17 programs. The criteria were edge coverage and edge-pair coverage as defined on UML state machine diagrams. Then the twelve OSes were applied to the edge-adequate and EP-adequate tests, resulting in 24 sets of tests for each program. These tests were run against 9,722 faults for a total of 96,714,456 executions. We recorded the effectiveness in terms of faults found and the cost in terms of creating the test oracles.

The experiment yielded several findings. First, we found that the practice of simply looking for runtime exceptions (NOS) is **not very effective**, thus testers need to check program states. This is significant because this fairly common practice is hugely wasteful. So using NOS is not recommended for most cases. This is not fully reflected in our cost-effectiveness model, because we did not include the cost of generating the test itself. A poor OS such as NOS causes testers to miss failures, and in such an extreme case, it means that much of the effort of generating tests is wasted. However, under an extremely tight time budget, testers can still use NOS since it does not require testers to check extra program states. Even though it wastes the effort in generating a third of the tests, the cost of generating the test oracle is zero. Second, we found that a more precise OS was **not always significantly more effective** than a less precise OS with the same test inputs and frequency. This could happen if one OS is not much more precise than the other.

Third, we found that the test oracle strategies with more frequency (OSi) were **not more effective** than strategies that checked with less frequency (OTi). This is good news to practitioners because checking with more frequency is more costly. Fourth, we found that the edge-pair coverage (EPC) tests were **not significantly stronger** than the edge coverage (EC) tests because EPC did not have much more mappings (test inputs) than EC. Because Briand et al. [8] had similar results, we conclude that whether a stronger coverage criterion can reveal more faults than a weaker coverage criterion depends on the model and coverage criteria, as discussed in Section 5.5.

Finally, we found that the approach of checking the state invariants from the state machine diagrams (SIOS) was reasonably effective and relatively inexpensive, thus **cost-effective**. If testers need more "bang for the buck," we found that checking return values and parameter members (OT2 and OT4) was almost as effective as checking everything (OT5), but less expensive.

In the future, we hope to seek ways to improve SIOS, as well as to develop new test oracle strategies. Since answering RQ3 depends on the model and coverage criteria, we hope to define guidance about which coverage criterion should be applied if the model satisfies certain conditions in model-based testing. Using mutation analysis to select which program states to check also seems promising [32],

[42], but could be costly because testers have to run mutation analysis before providing test oracle data.

References

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

[2] Paul Ammann, Jeff Offutt, Wuzhi Xu, and Nan Li. Graph coverage web applications. Online, 2008. http://cs.gmu.edu:8080/offutt/coverage/GraphCoverage, last access Jan 2016.

[3] 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.

[4] Anonymous. Class of tree. Online, 2008. http://homepage.cs.uiowa.edu/∼sriram/21/fall08/code/tree.java, last access May 2013.

[5] Earl Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, May 2015.

[6] Earl Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. Repository of publications on the test oracle problem. Online, 2015. http://crestweb.cs.ucl.ac.uk/resources/oracle_repository/, last access July 2015.

[7] Robert V. Binder. *Testing Object-Oriented Systems-Models, Patterns, and Tools*. Addison-Wesley Object Technology, 1999.

[8] 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.

[9] 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.

[10] Al Danial. CLOC. Online, 2006. https://github.com/AlDanial/cloc, last access Dec 2015.

[11] Harvey Deitel and Paul Deitel. *Java: How to program*. Pearson Education, Inc., 6th edition, 2005.

[12] Richard A. DeMillo and Jeff Offutt. Constraint-based automatic test data generation. *IEEE Transaction on Software Engineering*, 17(9):900–910, September 1991.

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

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

[15] Object Management Group. Object constraint language. Online, 2006. http://www.omg.org/spec/OCL, last access July 2014.

[16] 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.

[17] Hierons, Bogdanov, Bowen, Cleaveland, Derrick, Dick, Gheorghe, Harman, Kapoor, Krause, Luttgen, Simons, Vilkomir, Woodward, and Zedan. Using formal specifications to support testing. *ACM Computing Surveys*, 41(2), January 2009.

[18] W. E. Howden. Theoretical and empirical studies of program testing. *IEEE Transactions on Software Engineering*, 4(4):293–298, July 1978.

[19] Lewis, Chase, and Coleman. Class of blackjack. Online, 2004. http://faculty.washington.edu/moishe/javademos/blackjack/, last access May 2013.

[20] 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.

[21] Nan Li. The structured test automation language framework. Online, 2013. http://cs.gmu.edu/∼nli1/stale/, last access August 2014.

[22] 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.

[23] Nan Li and Jeff Offutt. An empirical analysis of test oracle strategies for model-based testing. In *Proceedings of the 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, Cleveland, Ohio, USA, April 2014.

[24] Nan Li and Jeff Offutt. A test automation language framework for behavioral models. In *The 11th Workshop on Advances in Model Based Testing*, Graz, Austria, April 2015.

[25] 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.

[26] David Lilja. *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, New York, NY, USA, 2005.

[27] Barbara Liskov and John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley Professional, 1st edition, 2000.

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

[29] 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 2014.

[30] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. MuJava : An automated class mutation system. *Wiley's journal of Software Testing, Verificaton, and Reliability*, 15(2):97–133, June 2005.

[31] 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 2014.

[32] 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.

[33] Larry J. Morell. *A Theory of Error-based Testing*. PhD thesis, University of Maryland, College Park, MD, USA, 1984. Technical report TR-1395.

[34] Larry J. Morell. A theory of error-based testing. *IEEE Transactions on Software Engineering*, 16(8):844–857, August 1990.

[35] Jeff Offutt. *Automatic Test Data Generation*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA, 1988. Technical report GIT-ICS 88/28.

[36] Jeff Offutt and Aynur Abdurazik. Generating tests from UML specifications. In *Proceedings of the Second IEEE International Conference on the Unified Modeling Language (UML99)*, pages 416–429, Fort Collins, CO, October 1999. Springer-Verlag Lecture Notes in Computer Science Volume 1723.

[37] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating fuctional tests. *Communications of the ACM*, 31(6):676–686, June 1988.

[38] Mauro Pezzè and Michal Young. *Software Testing and Analysis: Process, Principles, and Techniques*. Wiley, Hoboken, NJ, 2008.

[39] Mikko Rusma. Class of triangle. Online, 2004. http://www.cs.du.edu/∼snarayan/sada/teaching/COMP3705/-FilesFromCD/ Exercises/Lab4_WhiteBox/Triangle.java, last access May 2013.

[40] 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.

[41] 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.

[42] 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.

[43] Matt Staats, Michael W. Whalen, and Mats P. E. Heimdahl. Better testing through oracle selection. In *Proceedings of the 33rd Inter-*

national Conference on Software Engineering (NIER Track), ICSE '11, pages 892–895, Waikiki, Honolulu, HI, USA, May 2011. ACM.

[44] Matt Staats, Michael W. Whalen, and Mats P.E. Heimdahl. Programs, tests, and oracles: The foundations of testing revisited. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 391–400, New York, NY, USA, 2011. ACM.

[45] 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.

[46] 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.