Guidelines for Coverage-Based Comparisons of Non-Adequate Test Suites

MILOS GLIGORIC, University of Illinois at Urbana-Champaign
ALEX GROCE, Oregon State University
CHAOQIANG ZHANG, Oregon State University
ROHAN SHARMA, University of Illinois at Urbana-Champaign
MOHAMMAD AMIN ALIPOUR, Oregon State University
DARKO MARINOV, University of Illinois at Urbana-Champaign

A fundamental question in software testing research is how to compare test suites, often as a means for comparing test-generation techniques that produce those test suites. Researchers frequently compare test suites by measuring their *coverage*. A coverage criterion C provides a set of test requirements and measures how many requirements a given suite satisfies. A suite that satisfies 100% of the (feasible) requirements is called C-*adequate*. Previous rigorous evaluations of coverage criteria mostly focused on such *adequate* test suites: given two criteria C and C', are C-adequate suites (on average) more effective than C'-adequate suites? However, in many realistic cases, producing adequate suites is impractical or even impossible.

This paper presents the first extensive study that evaluates coverage criteria for the common case of *non-adequate* test suites: given two criteria C and C', which one is better to use to compare test suites? Namely, if suites $T_1, T_2, \ldots, T_n$ have coverage values $c_1, c_2, \ldots, c_n$ for C and $c'_1, c'_2, \ldots, c'_n$ for C', is it better to compare suites based on $c_1, c_2, \ldots, c_n$ or based on $c'_1, c'_2, \ldots, c'_n$? We evaluate a large set of plausible criteria, including basic criteria such as statement and branch coverage, as well as stronger criteria used in recent studies, including criteria based on program paths, equivalence classes of covered statements, and predicate states. The criteria are evaluated on a set of Java and C programs with both manually written and automatically generated test suites. The evaluation uses three correlation measures. Based on these experiments, two criteria perform best: branch coverage and an intra-procedural acyclic path coverage. We provide guidelines for testing researchers aiming to evaluate test suites using coverage criteria as well as for other researchers evaluating coverage criteria for research use.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms: Experimentation

Additional Key Words and Phrases: Coverage criteria, non-adequate test suites

ACM Reference Format:
Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov, 2014. Guidelines for Coverage-Based Comparisons of Non-Adequate Test Suites. *ACM Trans. Softw. Eng. Methodol.* V, N, Article A (January YYYY), 33 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

This material is based upon work partially supported by the National Science Foundation under Grant Nos. CCF-0746856, CNS-0958199, CCF-1012759, and CCF-1054876.

Author's addresses: M. Gligoric, R. Sharma, and Darko Marinov, Department of Computer Science, University of Illinois at Urbana-Champaign; A. Groce, C. Zhang, and M. A. Alipour, School of Electrical Engineering & Computer Science, Oregon State University.

DOI:http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Software testing helps developers to improve the quality of their code. Developers or test engineers run test suites and inspect failures to identify faults in the code. A fundamental task in software testing research is evaluating (and improving) test suites. For example, evaluating suites is central to the development of automated test-generation techniques whose goal is to generate high-quality suites.

To compare suites, researchers typically use real faults, seeded faults, and/or coverage criteria. For real faults, researchers measure how many faults (previously known or newly found) the suites find. However, collecting code with real faults and analyzing failures takes substantial effort. Thus, experiments often use a relatively small set of real faults, preventing rigorous statistical analysis of the results [Arcuri and Briand 2011].

Researchers also use mutation testing [Hamlet 1977; DeMillo et al. 1978; Jia and Harman 2011] to seed a large number of artificial faults and measure the mutation score, i.e., how many mutants a suite kills. Several studies [Andrews et al. 2005; Andrews et al. 2006] show that the results obtained on mutants predict detection of real faults, i.e., suites that kill more mutants are *likely, on average,* to find more real faults. While mutation testing can provide a good basis for statistical analysis [Arcuri and Briand 2011], it can also be prohibitively expensive to perform. Even a small program with only a few hundred lines of code may have thousands of mutants, and determining killed mutants may require running a test suite on each mutant.

Researchers therefore most often use *coverage* to compare suites. A traditional coverage criterion provides a finite set of test requirements for the code under test, and one measures how many requirements a given suite satisfies. For example, statement and branch coverage are well-known structural criteria [Ammann and Offutt 2008]. A suite that satisfies 100% of the (feasible) requirements for a criterion C is called C-adequate. Measuring test coverage is almost always much cheaper than performing mutation testing; even if the criterion has a high runtime overhead, it only requires running tests once per program, as opposed to once per mutant. Coverage criteria are widely used in testing research and practice, e.g., papers on automated testing techniques often report that one technique is better than another because it generates, say, "suites with 10% more branch coverage on average."

This paper addresses the following question: What coverage criteria should researchers use to evaluate suites? Research comparing[1] coverage criteria dates back at least 20 years [Frankl and Weiss 1993; Hutchins et al. 1994; Frankl and Iakounenko 1998] but has largely *focused on adequate test suites*: given two criteria C and C', do C-adequate suites (on average) find more faults than C'-adequate suites? However, testing practice and research widely use non-adequate test suites because determining which test requirements are feasible is hard, generating suites for all feasible requirements is often impractical, and some recently used criteria [Chaki et al. 2004; Ball 2004; Wang and Roychoudhury 2005; Visser et al. 2006; Pacheco et al. 2007; Chilimbi et al. 2009; Sharma et al. 2011; Groce 2011; Groce et al. 2012] even have an infinite (or astronomically large) set of requirements.

To the best of our knowledge, there has been no *extensive* study comparing coverage criteria over multiple *non-adequate suites* for the same program, except for the recent ISSTA conference paper [Gligoric et al. 2013] by the authors. This paper focuses on two critical questions:

[1] Note that we use the term "comparison" to refer to both comparisons of suites and comparisons of coverage criteria, but the intended use should be clear from the context.

(1) Are *any* coverage criteria able to predict mutation scores for non-adequate suites, and thus suitable for use in evaluations?
(2) Given two criteria C and C', is it better to use C or C' to compare test suites? Namely, if suites $T_1, T_2, \ldots, T_n$ have coverage values $c_1, c_2, \ldots, c_n$ for C and $c'_1, c'_2, \ldots, c'_n$ for C', is it better to compare suites based on $c_1, c_2, \ldots, c_n$ or based on $c'_1, c'_2, \ldots, c'_n$?

To illustrate the key difference in comparisons with adequate and non-adequate suites, consider a comparison of statement coverage (SC) with branch coverage (BC). For adequate suites, it is well known that BC subsumes SC: a suite with 100% BC would have 100% SC and should, on average, be likely to find more faults than another suite with 100% SC but less than 100% BC. For non-adequate suites, however, the situation is less clear. For instance, suppose a suite T_1 has 50% BC and 75% SC, and a suite T_2 has 60% BC and 65% SC. (Our experiments show that up to 11% of test-suite pairs have such discordant values for BC and SC; more details are provided in Section 4.) Should we use BC and declare T_2 better (60%>50%), or should we use SC and declare T_1 better (75%>65%); is T_1 or T_2 more likely to kill more mutants? Substituting a variety of criteria for branch and statement coverage, this scenario describes a common occurrence in evaluation of testing techniques.

The major contribution of this paper is an evaluation of multiple criteria, both traditional (statement and branch) and recently used (based on program paths, equivalence classes of covered statements, and predicate states). We evaluated criteria on a large set of Java and C programs with both manually written and automatically generated tests. We measured the effectiveness of criteria (using three statistical correlation coefficients) in terms of how well they predicted the mutation scores of suites (and thus, arguably, the real-fault detection of suites [Andrews et al. 2005; Andrews et al. 2006]). We designed our experiments to have a direct application to the evaluation of suites (and thus testing techniques) in testing research, and propose that our experimental approach would easily extend to other criteria, programs, and subjects. A minor contribution of this paper is the first implementation and evaluation of Ball's predicate-complete test coverage criterion (PCT) [Ball 2004; Ball 2005]. In Section 3, we describe all implementation challenges we faced in both Java and C.

Our results show that a variety of criteria are able to effectively predict mutation scores. This provides support for previous research studies that used these criteria to compare test suites. Moreover, for future studies, we propose two guidelines for researchers using coverage criteria to evaluate suites. First, our results show that branch coverage performs as well as or better than all other criteria studied, in terms of ability to predict mutation scores, and has a very low measurement overhead and implementation complexity. However, in some settings, branch coverage provides values that do not distinguish between test suites. Second, if researchers want a stronger criterion that can distinguish more test suites, but comes at the price of increased measurement overhead and implementation complexity, our results show that an acyclic intra-procedural variation of path coverage is about as effective as branch coverage. Our results also demonstrate that for *non-adequate suites*, criteria that are stronger (in terms of subsumption for *adequate suites*) do *not necessarily* have better ability to predict mutation scores. Additionally, as a guideline for future studies evaluating the effectiveness of criteria themselves, we suggest that results be based on a large set of suites generated by as many techniques as feasible for as many subjects as feasible, and that multiple correlations be measured to ensure that the results do not depend on a particular choice of correlation. Our tools, source code, and experimental subjects, along with more results, are publicly available at: http://mir.cs.illinois.edu/coco/.

The contributions of this work include:

- The first extensive study comparing how coverage criteria predict mutation scores for non-adequate suites.
- The first implementation of the Predicate-Complete Test (PCT) coverage criterion.
- The first evaluation of the Dynamic Basic Block (DBB) measurement as a coverage criterion. DBB was previously proposed for fault localization [Baudry et al. 2006].
- Some guidelines for using coverage criteria to compare suites in testing research.
- A guideline for performing future studies on comparing coverage criteria.

2. COVERAGE CRITERIA

Our comparison uses several criteria: SC and BC (as they are most common in practice), DBB (as this criterion showed promising results for fault localization [Baudry et al. 2006] and has never been used for comparison of test suites or testing techniques), PCT [Ball 2004; Ball 2005] (as a criterion with a strong theoretical foundation that has not been implemented and evaluated previously), and AIMP and IMP (as representatives of path-coverage criteria). We do not use data-flow criteria in our comparison for two reasons: it has been shown recently [Hassan and Andrews 2013] that data-flow coverage correlates well with BC, and we were not aware of any tool for data-flow coverage that scales out-of-the-box to the larger programs we used in our evaluation.

In this section, we define the criteria that we used in the study and illustrate them using a simple Java data structure. (Note that our implementations support larger programs in both Java and C.) Figure 1.a shows the relevant part of a class implementing the binomial heap data structure [Visser et al. 2006; Cormen et al. 2009] that supports fast union operation. The figure shows only the part of the `BinomialHeap` class relevant for our discussion. Each `BinomialHeap` object has a pointer to the root of the heap (`nodes`). Every node keeps a value (`key`) and pointers to its parent, sibling, and child. The `decreaseKey` method decreases the value of a node, which may affect the heap invariant that each parent should not have a higher value than its children, so the value is propagated to ancestors until the appropriate position is found.

2.1. Dynamic Basic Block Coverage (DBB)

We first describe Dynamic Basic Block (DBB) coverage, which may be unfamiliar to most readers outside the fault-localization community. Baudry et al. [Baudry et al. 2006] proposed the notion of a dynamic basic block[2] to measure a test suite's effectiveness for fault localization. Suppose we are given a program and execute a number of tests on the program. Consider a partition of the program statements into equivalence classes, where two statements belong to the same equivalence class if and only if they are covered by the same set of tests. Each equivalence class is called a *dynamic basic block* (DBB). The Baudry et al. study [Baudry et al. 2006] showed that the larger the number of DBBs a test suite has, the more effective the test suite is for spectrum-based fault localization. The underlying rationale is that having few DBBs equates to a suite having little ability to distinguish statements with respect to their causal impact on fault behavior. We use the number of DBBs as a test coverage metric instead, on the grounds that these equivalence classes show distinct program behaviors that could be explored. To illustrate DBB, consider the instance of `BinomialHeap` shown in Figure 1.d.

Assuming that there are two tests available for the `decreaseKey` method — `(9, 8)` and `(9, 2)` — the total number of DBBs is two. The first DBB includes all the statements before the while loop, i.e., lines between 9 and 16 (Figure 1.a); these lines are

[2] Not to be confused with dynamic basic blocks as used in computer architecture or compilers [Patel et al. 2000].

```
 1 // public class BinomialHeap { ...
 2 static class Node {
 3   int key;
 4   Node parent;
 5   // ...
 6 }
 7 Node nodes;
 8
 9 void decreaseKey(int oldKey, int newKey) {
10   Node tmp = nodes.findNodeWithKey(oldKey);
11   if (tmp == null)
12     return;
13   tmp.key = newKey;
14   Node tmpParent = tmp.parent;
15   while ((tmpParent != null)
16          && (tmp.key < tmpParent.key)) {
17     int z = tmp.key;
18     tmp.key = tmpParent.key;
19     tmpParent.key = z;
20     tmp = tmpParent;
21     tmpParent = tmpParent.parent;
22   }
23 }
```

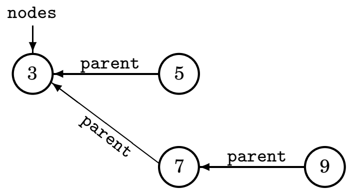
(a)

```
 1void decreaseKey(int oldKey, int newKey) {
 2  try {
 3    Coverage.beginMethod(0);
 4    Node tmp = nodes.findNodeWithKey(oldKey);
 5    if (tmp == null) {
 6      Coverage.cover(1,p$10(nodes),p$20(tmp));
 7      return;
 8    }
 9    Coverage.cover(2,p$10(nodes),p$20(tmp));
10
11    tmp.key = newKey;
12    Node tmpParent = tmp.parent;
13    while ((tmpParent != null)
14           && (tmp.key < tmpParent.key)) {
15      Coverage.cover(3,p$10(nodes),p$20(tmp),
16        p$21(tmpParent),p$49(tmp,tmpParent));
17      int z = tmp.key;
18      tmp.key = tmpParent.key;
19      tmpParent.key = z;
20      tmp = tmpParent;
21      tmpParent = tmpParent.parent;
22    }
23    Coverage.cover(4,p$10(nodes),p$20(tmp),
24      p$21(tmpParent),p$49(tmp,tmpParent));
25  } catch (Exception e) {
26    Coverage.endMethod();
27  }
28}
```

(b)

```
 1 // tmp.key < tmpParent.key
 2 boolean p$49(Node tmp, Node tmpParent) {
 3   try {
 4     if (Coverage.testAndSetInPredicate())
 5       return false;
 6     if (tmpParent == null)
 7       return false;
 8     if (tmp == null)
 9       return false;
10     return tmp.key < tmpParent.key;
11   } catch (Exception _) {
12     return false;
13   } finally {
14     Coverage.resetInPredicate();
15   }
16 }
```

(c)



(d)

Fig. 1: `BinomialHeap` as running example

covered by both tests. The second DBB includes all the statements in the body of the while loop, i.e., lines between 17 and 22; these lines are covered only by the second test. We say that this test suite has DBB coverage of 2. In general, a program with s statements having a test suite of t tests can partition the program into up to $\min(s, 2^t)$ DBBs. DBB is obviously not useful for suites that consist of only a single very large test, and has a limited value to distinguish suites that have a small number of tests.

2.2. Intra-Method Path Coverages (IMP and AIMP)

We next describe two forms of path-based coverage used in our evaluation. Whole-program path coverage was proposed over 20 years ago [Larus 1999] to measure how many different paths tests execute from the beginning to the end of a program. Even for loop-free programs, whole program paths result in a number of test requirements exponential in the number of branches in a program, so more recent work [Chilimbi et al. 2009; Wang and Roychoudhury 2005; Groce 2009;

Groce et al. 2012] used more scalable *intra-method paths (IMP)*, where each path is for a single method execution only (similar to Godefroid's notion of *compositional* path coverage [Godefroid 2007]). An intra-method path starts at the beginning of a method, includes the IDs of the executed basic blocks[3], does not include nested method invocations, and ends when the execution returns from the method. IMP subsumes BC (and thus SC) but faces the problem that loops introduce an unbounded number of test requirements.

Our second variant of path coverage, *acyclic intra-method paths (AIMP)*, retains subsumption of BC but bounds the total number of requirements by considering only acyclic paths in intra-method control-flow graphs [Ball and Larus 1996]. The number of AIMP paths is therefore bounded by $m \cdot 2^k$ where m is the number of methods in a program and k is the maximum number of branches in a single method. The paths to be covered have no repeated IDs, i.e., AIMP modifies IMP such that a repeated basic block ID ends the current path and starts a new path[4]. Ball and Larus present an efficient approach to compute AIMP coverage [Ball and Larus 1996].

Figure 1.b shows an instrumented version of `decreaseKey` that can be used to collect IMP and AIMP coverages. (The `p$` methods will be discussed in the next section.) `Coverage.beginMethod` and `Coverage.endMethod` are invoked at the beginning and end of the method, respectively, and they are used to begin and end a path. `Coverage.cover` is invoked at each basic block and is used to collect the block IDs in a path. In addition, for AIMP, the `Coverage.cover` method may end the current path and start a new path if the block ID is repeated on the current path. For example, consider the instance of `BinomialHeap` shown in Figure 1.d.

Invoking `decreaseKey` on that heap with arguments `(9, 8)` executes the IMP $0 \rightarrow 2 \rightarrow 4$ and covers the same path for AIMP. (Note that 0, 2, and 4 refer to IDs of basic blocks). Invoking `decreaseKey` on that heap with `(9, 2)` instead executes the IMP $0 \rightarrow 2 \rightarrow 3 \rightarrow 3 \rightarrow 4$ but covers two paths for AIMP: $0 \rightarrow 2 \rightarrow 3$ and $3 \rightarrow 4$. Note that IMP and AIMP collect paths for *every method* run, e.g., each invocation of `decreaseKey` calls `findNodeWithKey` (which may invoke other methods), so for each invocation, IMP has one path (and AIMP at least one path) for both methods.

2.3. Predicate-Complete Test Coverage (PCT)

Predicate-complete test (PCT) coverage [Ball 2004; Ball 2005] was introduced by Ball as a finite-state alternative to path coverage, inspired by predicate abstraction in model checking [Ball and Rajamani 2001]. Like path coverage, PCT subsumes both BC and SC, but unlike some versions of path coverage, PCT does not face the problem that loops introduce an unbounded number of test requirements. PCT is incomparable to (i.e., neither subsumes nor is subsumed by) path coverages such as IMP and AIMP, even for loop-free programs. Several research studies [Visser et al. 2006; Pacheco et al. 2007; Sharma et al. 2011; Groce 2011; Groce et al. 2012] compared test suites using PCT, but with manually selected predicates for measuring PCT; we refer to this version as $\mathrm{PCT}_{\mathrm{MS}}$.

PCT defines coverage using Boolean predicates extracted from the program source, in particular from branch conditions, implicit run-time checks, and program assertions. These predicates are evaluated at many program points, e.g., at all statements or all starts of basic blocks, potentially far from where the predicates appear in the program source. In fact, evaluating predicates both *near and far* from where they appear

[3] These are the standard basic blocks, not dynamic basic blocks from DBB. When we want to refer to DBBs, we explicitly use "dynamic".

[4] Our AIMP uses the notion of simple path common in graph theory, where no vertex is repeated, rather than the definition of prime path found in some testing literature [Ammann and Offutt 2008].

is what makes PCT even stronger than MC/DC or other related criteria sometimes called "predicate coverage" [Ammann and Offutt 2008] that evaluate predicates only near where they appear. The test requirements for PCT are to cover all (feasible) combinations of predicate values at all the points. In the limit, for n predicates at p points, there are $p \cdot 2^n$ combinations (many often infeasible and not every point has all n predicates). The PCT coverage for a test suite is measured as the number of combinations of predicate values obtained during the execution of the test suite.

We next illustrate PCT using the `BinomialHeap` example. The first step is to extract a set of Boolean predicates from the code under test. Our example code has two conditional statements at lines 11 and 15 (Figure 1.a), which lead to three predicates: `tmp == null`, `tmpParent != null`, and `tmp.key < tmpParent.key`. Note that we take as a predicate each atomic condition rather than the complex expression. The implicit runtime checks in our example guard against dereferencing null: `nodes != null`, `tmp != null`, and `tmpParent != null`. Note that the same predicate may be extracted several times, so syntactically identical duplicates are removed (Section 3). A key goal for PCT is to extract *all* predicates, as otherwise PCT may not subsume BC or MC/DC.

The second step is to insert evaluation of predicates at *all* appropriate program points. Our tool first generates a method for evaluating each predicate and then inserts calls to these methods. Note that one cannot simply evaluate the predicate as it could lead to problems, e.g., raise an exception if certain variables are `null`. The method for each predicate performs the necessary checks. Figure 1.c shows the method for the predicate `tmp.key < tmpParent.key`. The methods `Coverage.testAndSetInPredicate` and `Coverage.resetInPredicate` guard against infinite recursion. The `catch` clause handles exceptions in predicate evaluations.

For program points, our PCT tools for Java and C allow instrumenting all statements, PCT_{ST}, or all beginnings of basic blocks, PCT_{BB}. Figure 1.b shows an example instrumentation at the basic-block level. Each `Coverage.cover` call informs the tool that a certain program point (identified with an integer ID) is being executed with a specific combination of predicate values. Note that predicates cannot be evaluated at points where their variables are not in scope, e.g., the predicates for `tmpParent` cannot be evaluated before line 12. Our tools insert evaluation for *all* extracted predicates that can be evaluated. Some predicates can be evaluated far from where they are extracted, e.g., `nodes != null` is evaluated on line 15 (Figure 1.b), although it is extracted based on line 4 (Figure 1.b). Some predicates (on instance fields, rather than on method local variables) can even be extracted in one method and evaluated in another method.

While $\mathrm{PCT_{BB}}$ maintains the key subsumption properties of PCT over BC, it is only an approximation of $\mathrm{PCT_{ST}}$ because statements within a block can change predicate values. The example shows that this is not unusual: `tmp.key`, `tmpParent.key`, and `tmp` are all modified inside the block beginning at line 14 (Figure 1.b) in ways that may introduce combinations of predicate values that will never be seen at basic block entries.

3. PCT IMPLEMENTATION

As stated previously, one of our contributions is the first implementation of Ball's PCT [Ball 2004], and therefore we are the first to encounter a number of unique challenges related to this coverage criterion. We find it important to document our experiences related to these unique challenges. Specifically, we find that the design of a programming language may impose fundamental problems for correct and efficient implementation of PCT, by making certain information unavailable at runtime. In the following sections, we first discuss the implementation for both Java and C, challenges that are both unique and shared by these languages, and how we addressed these challenges.

3.1. Java Implementation

We implemented our tool for measuring PCT for Java as an Eclipse headless plugin [Eclipse 2013] that performs source-to-source instrumentation. The tool can instrument the code under test for measuring PCT at each statement or each basic block.

3.1.1. Extracted Predicates. According to the original source on PCT [Ball 2004], all atomic predicates should be extracted from conditional statements, implicit run-time checks, and assertions. For complex conditionals or assertions, e.g., `A || (B && C)`, each of `A`, `B`, and `C` must be treated as a separate predicate (otherwise, PCT could not subsume multiple condition coverage). However, the original source [Ball 2004] gives no specific instructions on which run-time checks to consider. To limit the cost of instrumentation, our tool considers only two types of run-time checks for creating predicates: null dereference and index out of bounds. It creates one predicate for each field access (e.g., predicate `obj != null` for `obj.f`) and method invocation (e.g., predicate `obj != null` for `obj.m()`), and three predicates for each array element access (e.g., predicates `arr != null`, `0 <= i`, and `i < arr.length` for `arr[i]`).

3.1.2. Minimizing the Set of Predicates. We maintain predicates as a set and do not instrument multiple occurrences of the same predicate multiple times, for efficiency reasons. We are limited in our ability to detect semantically, rather than syntactically, equivalent predicates (the problem is undecidable in general); even when semantically duplicate predicates appear in instrumentation, they do not change the total number of covered location-predicate values (redundant bits in a bit vector do not change bit vector equality).

3.2. C Implementation

We implemented our tool for measuring PCT for C as a source-to-source transformation using the CIL framework [Necula et al. 2002]. Like the Java version, the C version allows us to choose instrumenting each basic block or each statement.

The challenges in extracting predicates in C are somewhat different than in Java. C is arguably a simpler language than Java, e.g., lacking inheritance or exceptions and having simpler scoping rules. Unfortunately, attempting to instrument real-world C programs for PCT faces challenges rooted in the C language itself.

The fundamental problem is that C is an unsafe language. In Java, it is easy to perform runtime checks to avoid invalid memory accesses: the length of an array can be queried, and if a reference is not NULL, it is valid. In C, however, arrays do not carry length information and pointers can be non-NULL yet point to deallocated or remote memory—a C pointer is simply an arbitrary memory address. The only way to safely capture values for most predicates involving pointers or arrays in C would be to further instrument the program to track array lengths and check pointers for validity. However, the overhead of such instrumentation is unfortunately high for many C programs, e.g., even an efficient tool such as Purify [Purify 2013] can have 2–5X slowdown in runtime and 2–10X overhead in memory usage, and we estimate that the *additional* slowdown and overhead over our predicate instrumentation would be even higher. Therefore, in our tool we have chosen not to extract predicates using pointers or array referencing.

The core instrumentation is quite simple: after transforming the input code to CIL's canonical form, a CIL visitor first traverses the program collecting predicates (and their scopes), and then another visitor inserts function calls to capture values at each block or statement.

3.3. Challenges

During the implementation of our tools we encountered several technical challenges *unique to measuring PCT coverage*. We discuss these challenges, marking each with the language—^{J} for Java, ^{C} for C, and ^{JC} for both—in which the challenge is identified.

3.3.1. Side Effects^{JC}. Simply extracting all expressions that appear in conditional statements and evaluating these exact syntactic expressions at certain program points can lead to incorrect instrumentation because a conditional expression may contain side effects, such as assignments, prefix/postfix operators, or invocations of methods/-functions that modify the program state. Because of side effects, the state of the instrumented program at some point in the execution may not match the state of the original program at the corresponding point in the original execution. To identify side effects, we implemented a (simple) purity analysis [Rountev 2004; Sălcianu and Rinard 2005] using WALA [WALA 2013] for Java. The analysis checks each extracted predicate and does not instrument elsewhere for those that are not side-effect free. In C, CIL removes the problem of side effects by using temporary variables to make all conditionals side-effect free. This means that in C, we often instrument a predicate for a temporary variable only assigned once. This is not clearly worse than simply not instrumenting the predicate: it captures some additional states, without adding any spurious states since the temporary value is local in scope.

3.3.2. Recursive Predicate Invocation^{JC}. Each predicate can contain an arbitrary expression, as long as the expression does not have side effects. Therefore, a predicate may contain an invocation of a method that invoked the predicate, which would lead to infinite recursion. To prevent this, the instrumentation inserts special method calls at the beginning and end of each predicate. Recall Figure 1.c. The method `Coverage.testAndSetInPredicate` checks a Boolean flag that indicates whether a predicate evaluation has started. If no evaluation has started, it sets the Boolean flag and starts the predicate evaluation. If the flag was already set, the predicate would not be evaluated. The method `Coverage.resetInPredicate` simply resets the Boolean flag to mark the end of the evaluation.

3.3.3. Field/Element Access or Method Invocation^{JC}. A predicate can contain arbitrary (side-effect free) expressions including field accesses, method invocations, or array-element accesses. Since a predicate can be evaluated at any program point where the variables used in the predicate are visible, some of these expressions could lead to null pointer dereference or index out of bounds exceptions (in Java) or other problems (in C). For C, our tool does not use such predicates. For Java, our tool adds checks to the predicates, specifically a null check for each field access and method invocation and both a null check and bound check for each array element access. If all checks are satisfied, the predicate is evaluated, otherwise the evaluation of the predicate is ignored (i.e., we return a default value). In the example in Figure 1.c, there are checks for `tmp != null` and `tmpParent != null`.

3.3.4. Checked Exceptions^{J}. A Java predicate can in general contain an invocation of a method that declares some checked exceptions (e.g., `IOException`). Such exceptions have to be either propagated to a caller (by specifying the types of the exceptions in the `throws` clause) or caught. We did not want to simply ignore such predicates (especially since they can be important for bugs related to exceptional control flow [Fu and Ryder 2005]). Instead, our implementation adds code to catch the exception(s) and ignores the evaluation of the predicate if an exception is caught. Figure 1.b shows such a `catch` block, although it is not strictly required for that example predicate. In practice, only a small percentage of predicates requires catching exceptions, because our purity analysis already filters out most of the methods that may throw an exception.

3.3.5. Inner/Anonymous Classes and Class Hierarchy^J. Our current implementation does not instantiate certain predicates that could be in theory instantiated across class boundaries but do not occur often in practice. First, inner/anonymous classes in Java can access predicates from the outer classes. However, an additional check would be needed to ensure that a predicate from an outer class can be instantiated in an inner class: all local variables needed as the predicate arguments must be declared `final`. Similarly, some predicates extracted from inner classes could be instantiated in the outer classes if all the variables used in the predicate are declared in the outer classes. Second, predicates that are extracted from a class and reference its instance fields are in principle visible in all subclasses (that do not shadow these fields). The reason to ignore these predicates is additional implementation challenges required to track the relations and to keep predicates across instrumenting multiple classes.

3.3.6. Method Size Limit^J. In a few cases, our instrumentation produced code that was so large that it was rejected by Java compilers. Namely, there were many predicates and points in the instrumented code, and some of the instrumented methods exceeded the 64KB limit set by the Java classfile specification [VMSpec 2013]. One approach to reduce the size would be to (randomly) select only some predicates and/or program points for PCT where the predicates should be instantiated. However, a good way to select predicates and/or points is not known as of now. Thus, we decided to ignore all predicates and points that lead to methods that exceed the limit.

4. EXPERIMENTAL METHODOLOGY

To compare coverage criteria, we examine first and foremost how well the coverage values predict test suite quality in terms of mutation scores. We additionally consider the cost of measuring coverage. We compare 8 criteria: two traditional criteria (SC and BC) and three sets of recently used criteria based on equivalence classes of covered statements (DBB), program paths (IMP and AIMP), and predicate states (PCT_{MS}, PCT_{BB}, and PCT_{ST}).

The testing literature does not have one agreed upon methodology for comparing test coverage criteria, so we motivate and describe the methodology we use. Coverage values are used to evaluate suites, typically as predictors for finding real faults. Intuitively, if a good criterion deems one suite better than another suite, then we expect the better suite to find, *on average*, more faults. However, performing *large* controlled experiments with real faults is hard due to the difficulty of collecting many suitable faulty programs, and statistical validity is difficult to attain with the typically small number of faults in each program. For these reasons, while older studies on comparing coverage criteria used (a small number of) real faults [Frankl and Weiss 1993; Hutchins et al. 1994; Frankl and Iakounenko 1998], more recent studies use (a large number of) systematically seeded mutants [Cai and Lyu 2005; Andrews et al. 2006; Namin and Andrews 2009].

Specifically we examine the ability of coverage values to *predict (the relative ordering or absolute values of) mutation scores*. To visualize this concept, Figure 2 shows eight plots (for eight coverage criteria) that relate coverage values and number of killed mutants for `BinomialHeap`. Each point represents one of 300 suites (selected as explained in Section 4.2). The X-axis shows coverage, normalized between 0.0 and 1.0, and the Y-axis shows number of killed mutants[5]. It is clear in all eight plots that if a suite A

[5] The number of killed mutants is not normalized, but dividing by a constant never changes values for our three correlations.

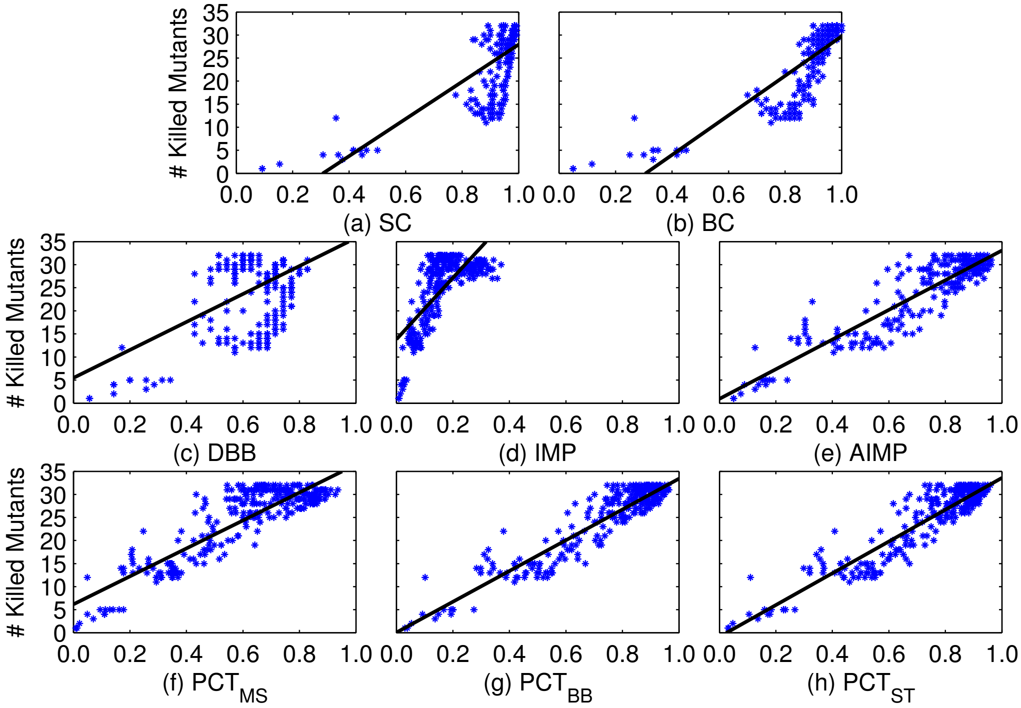


Fig. 2: Correlation of (normalized) coverage criteria and the number of killed mutants for BinomialHeap

has a higher coverage than a suite B, then the suite A also *likely* has a higher mutation score than the suite B. The purpose of our statistical evaluation is to quantify the degree to which this relationship holds for each criterion, and thus to compare criteria. We apply three different standard statistical tools: Kendall's τ_b rank correlation, Spearman's ρ rank correlation, and the R^2 coefficient of determination for linear regression, discussed in detail in Section 4.4. Intuitively, Kendall's τ_b and Spearman's ρ measure how well coverage values predict the relative ordering of mutation scores, and R^2 correlates coverage values with mutation scores using a linear regression model.

Table I: Subject programs used in the evaluation (basic statistics)

Subject	NBNC	Size of test pool	Total mutants	Killed mutants
language: Java				
JFreeChart	72,490	2,217	45,409	14,932
JodaTime	27,472	3,828	24,956	16,478
AvlTree	344	11,041	335	51
BinomialHeap	264	8,423	205	37
BinTree	100	13,825	55	16
FibHeap	264	12,842	186	38
FibonacciHeap	397	4,478	295	74
HeapArray	98	4,064	122	61
IntAVLTreeMap	213	17,072	199	38
IntRedBlackTree	296	20,419	279	210
LinkedList	245	1,307	167	5
NodeCachLList	234	1,776	159	16
SinglyLList	98	1,762	57	10
TreeMap	449	14,076	463	106
TreeSet	323	17,400	360	82
language: C				
Space	6,200	1,350	1,142	753
SQLite	81,934	117,240	52,367	19,294
YAFFS2	11,760	5,000	10,674	4,186
Printtokens	479	4,130	536	442
Printtokens2	401	4,115	343	343
Replace	512	5,542	613	530
Schedule	292	2,650	140	125
Schedule2	297	2,710	300	251
SglibRbtree	1,564	5,000	443	193
Tcas	135	1,608	311	311
Totinfo	340	917	511	511

Table II: Subject programs used in the evaluation (coverage statistics)

	SC	**BC**		**DBB**	**Intra-method paths**		**PCT**						
Subject	stmts	branches		states	states		predicates		points			states	
		static	exe		IMP	AIMP	MS	BB	MS	BB	ST	BB	ST
					language: Java								
JFreeChart	23,132	17,866	12,083	3,409	11,182	7,281	-	13,536	-	32,907	42,372	34,899	45,406
JodaTime	9,480	7,357	6,364	3,498	5,141	4,826	-	2,913	-	9,476	10,570	16,673	18,723
AvlTree	47	104	20	18	110	30	4	87	104	189	167	153	156
BinomialHeap	130	60	60	35	442	79	9	49	60	109	150	335	419
BinTree	51	32	32	25	2,388	51	7	26	32	51	54	224	228
FibHeap	113	60	44	19	43,795	60	14	67	56	98	160	132	228
FibonacciHeap	120	66	45	18	5,658	46	14	58	62	100	156	139	252
HeapArray	59	32	30	17	1,319	37	3	19	32	50	60	205	235
IntAVLTreeMap	100	56	47	41	99	45	4	52	56	100	112	242	277
IntRedBlackTree	171	90	83	60	275	106	6	76	90	149	177	479	534
LinkedList	34	36	8	8	531	26	4	40	36	78	107	34	53
NodeCachLList	59	34	14	12	550	32	4	34	34	68	103	82	129
SinglyLList	43	26	20	12	478	20	3	22	26	39	55	67	95
TreeMap	207	147	101	72	317	119	6	102	119	239	280	749	837
TreeSet	175	93	83	67	313	106	6	69	94	150	183	462	521
					language: C								
Space	3,366	1,190	1,014	584	5,384	654	-	1,552	-	884	3,927	5,708	25,100
SQLite	23,565	17,304	15,676	-	749,052	16,514	-	21,285	-	13,786	37,313	529,272	1,432,590
YAFFS2	3,236	4,274	1,852	229	180,770	1,361	-	4,149	-	3,520	8,273	27,501	755,42
Printtokens	185	66	63	50	1,568	115	-	70	-	73	265	292	1,050
Printtokens2	200	162	159	64	2,346	131	-	108	-	133	282	908	2,339
Replace	234	180	169	93	3,803	164	-	190	-	177	345	1,041	1,968
Schedule	150	58	55	39	1,838	75	-	52	-	64	176	545	1,554
Schedule2	128	88	83	33	2,455	82	-	54	-	75	190	705	1,751
SglibRbtree	502	378	238	114	2,175	106	-	426	-	350	720	3,794	9,841
Tcas	64	66	61	14	50	50	-	45	-	72	133	1,311	2,603
Totinfo	117	88	79	34	1,039	80	-	55	-	76	238	977	3,109

4.1. Experimental Subjects

Programs: Table I summarizes the programs used in our experiments, showing the name and number of NBNC (non-blank, non-comment) lines of code (measured by CLOC [Cloc 2013]) for each program. We used a total of 26 programs, 15 Java programs and 11 C programs. All Java programs but two are implementations of data structures that have been used in numerous previous studies, primarily on comparing different testing techniques [Visser et al. 2006; Galeotti et al. 2010; Sharma et al. 2010; Sharma et al. 2011; Groce 2011; Groce et al. 2012]. `JFreeChart` [JFreeChart 2013] is an open-source library for both interactive and non-interactive manipulation of charts. `JodaTime` [JodaTime 2013] is an open-source library for manipulating date and time. For C, seven programs are from the Siemens suite from the SIR repository [Hutchins et al. 1994; Do et al. 2005], `Space` [Vokolos and Frankl 1998; Do et al. 2005] is a bigger program from the same repository, `SglibRbtree` [Vittek et al. 2006] is the red-black tree implementation from the Sglib library, `YAFFS2` [YAFFS2 2013] is a widely used open-source flash file system for embedded devices (the default image format for older versions of Android), and `SQLite` [SQLite 2013] is a widely deployed database engine.

Tests: Table I also shows the total number of tests in the test pools from which various test suites are selected. For Java data structures, we use test pools *automatically generated* in previous studies [Sharma et al. 2011; Groce 2011; Groce et al. 2012] using three test-generation techniques: random (*Random*), shape abstraction (*ShapeAbs*) [Visser et al. 2006], and adaptation-based programming (*ABP*) [Groce 2011; Groce et al. 2012]. Table I shows the total number of tests generated by all three techniques. For `JFreeChart` and `JodaTime`, we use the large, publicly available pool of *manually written* JUnit tests. For C programs, we use the Siemens/SIR test pools for the programs from SIR. For `SglibRbtree` and `YAFFS2`, we generated random tests (feedback-directed [Groce et al. 2007] for `YAFFS2`). For `SQLite` we use manually written tests available from the SQLite repository [SQLite 2013].

Mutants: Table I also tabulates for each program the number of mutants created and the total number of mutants killed by the entire test pool (while different suites selected from the pool kill different number of mutants). In all cases, we consider a mutant to be killed if it either results in an assertion violation, an uncaught exception or other abnormal program termination, or leads to a timeout. In the case of YAFFS2, we additionally check the return value of each API call (since a test is a sequence of API calls with known correct returns). The percentage of killed mutants is low because we mutated *all* the methods in the code but automatically generated tests execute only *some* core methods for the smaller subjects [Sharma et al. 2011]. Low absolute mutation scores are suitable for our purpose of examining non-adequate suites, the typical case for suites for large programs. Non-adequate suites will seldom attain extremely high mutation scores [Just et al. 2012]. Additionally, we did not investigate which mutants are equivalent, as this does not affect our analysis (because compensating for equivalent mutants is equivalent to dividing mutation score by a constant, which does not affect τ_b, ρ, or R^2).

For Java programs, we used Javalanche [Schuler and Zeller 2009] to create mutants. Because the number of mutants may be lower than one would expect, it should be noted that Javalanche uses selective mutation [Offutt et al. 1993] to reduce the cost of mutation testing. Selective mutation applies only a subset of mutation operators that are empirically shown to approximate the results that would be achieved if all operators were used. In particular, Javalanche uses only the following operators: replace numerical constants, negate jump condition, replace arithmetic operator, replace method calls, and remove method calls. Still, Javalanche created over 45K and 24K mutants for JFreeChart and JodaTime, respectively.

For C programs, we created mutants using the tool implemented by Andrews et al. [Andrews et al. 2005], which produces mutants based on a set of operators selected through an empirical study on selective mutation [Namin et al. 2008]. Specifically, the tool uses the following operators: replace constants; delete statements; negate decisions in conditional statements; and replace a relational, arithmetic, logical, bit-wise, increment/decrement, or arithmetic-assignment operator by another operator from the same class. For all programs but Space we use all the mutants.

For Space, the exact numbers reported in this section are based on a random sample of 10% of the mutants. We initially sampled 10% of the mutants because running these mutants takes considerable time, and we relied on recent studies [Zhang et al. 2010; Zhang et al. 2013] that showed that results obtained on a random sample of mutants can provide a good approximation of the results obtained on the entire set of mutants. More recently we used multiple machines to repeat the experiment for all the mutants for Space and got almost identical results as for the sampled 10% (e.g., for all three correlation coefficients we use, the difference between the values on 100% of mutants and 10% of the mutants are below 0.007, and there was no statistically significant difference between the coefficients). This additionally confirms the recent studies [Zhang et al. 2010; Zhang et al. 2013] that sampling mutants can often be done to speed up the experiments without affecting the overall conclusion.

Statement and Branch Coverage Information: The SC and BC columns in Table II provide information for statement and branch coverage, respectively; "static" shows the number of branches in the code, and "exe" shows the number of branches executed by at least one test.

DBB Information: Table II also provides DBB-specific information, i.e., the total number of DBBs obtained by using a single test suite consisting of the entire test pool summarized in Table I. Note that the total number of DBBs differs when we select different test suites from the test pool. For SQLite, DBBs are not meaningful as "suites" consist of a single lengthy execution sequence with no breakdown into separate tests.

IMP and AIMP Information: Table II also provides the total number of paths executed by the entire test pool.

PCT Information: Table II finally provides PCT-specific information, i.e., the total number of predicates used in the instrumentation, the number of program points at which these predicates are inserted, and the number of executed states (i.e., encountered states during the execution) by the entire test pool. *MS* ("Manually Selected") denotes a set of predicates and points that were first manually selected for four data structures by Visser et al. [Visser et al. 2006] and then similarly selected for the remaining structures by Sharma et al. [Sharma et al. 2011]. These programs, with manually selected predicates for PCT coverage, are publicly available [Coverage 2013]. *BB* ("Basic Blocks") and *ST* ("Statements") denote the results of automatic instrumentation by our PCT coverage tools. Recall that our tools select (almost) all predicates from the code and insert each predicate at (almost) all program points where the variables from the predicate are in scope.

4.2. Test Suites

We used two approaches for selecting test suites, to see if results are robust in the face of different suite compositions. The bounds in our approaches (e.g., 100 test suites) were chosen before experimentation, to limit computation time while providing sufficiently many samples for statistical analysis, or were chosen to match previous papers.

Coverage-varied Selection: For each program, to ensure that the selected test suites are of varying coverage and size, we created suites by first uniformly selecting a coverage level between 1% and 100% and then randomly selecting tests from the test pool until they reached the selected level of *PCT_{BB}* coverage. We picked $\mathrm{PCT}_{\mathrm{BB}}$ as one strong criterion but could have used any other criterion. We follow these steps for both Java data structures and large subjects. The difference is that we select different number of test suites. For the Java data structures we selected 100 suites from the pool for each of the three test-generation techniques (Random, ShapeAbs, and ABP), giving a total of 300 test suites. For `JFreeChart` and `JodaTime` we used 100 test suites from the entire pool of available tests. Similarly, for all C programs except `SQLite` we used 300 suites, again from the pool of available tests. For `SQLite` each "test" in the pool is essentially a large suite of tests that must run together, so we treated each of the 592 "tests" as a suite.

Size-varied Selection: We also followed another suite selection approach, used in previous studies of coverage criteria [Namin and Andrews 2009; Hassan and Andrews 2013]. For each program, we created 100 random suites for each size (number of tests) between 1 and 50, which gives 5,000 suites per program. Also, this approach creates many suites that are near adequate in at least one criterion and does not include suites based on different test generation techniques, which most closely reflect the intended purposes of our evaluation. `SQLite` was handled as for the Coverage-varied Selection.

4.3. Metrics

We collected several metrics for the selected test suites.

Coverage Criteria: For each suite, we measured several coverage values (for both Java and C): SC, BC, DBB, IMP, AIMP, $\mathrm{PCT}_{\mathrm{MS}}$ (except for `JFreeChart`, `JodaTime`, and all C programs), $\mathrm{PCT}_{\mathrm{BB}}$, $\mathrm{PCT}_{\mathrm{ST}}$, and mutation score.

Runtime Overhead: We separately ran each coverage measurement so that we could measure the runtime overhead. We performed all Java experiments on a machine with a 4-core Intel Core i7 2.70GHz processor and 4GB RAM, running Linux version 3.2.0 and Java OpenJDK 64-Bit Server VM, version 1.7.0_04. We performed all C experiments on a machine with a 4-core Intel Xeon E5400 2.83GHz processor and 4GB RAM, running Linux version 2.6.32.

4.4. Correlation Analysis

To evaluate the relationship between coverages and mutation scores, we computed three correlation measures.

Kendall's τ_b: One core question of this paper is whether (and which) coverage criteria can be used to effectively predict the *rank order* of suites' mutation scores. This is the primary use of coverage in recent studies; authors have tended to focus on claiming that some testing technique is "better", and relatively small differences in coverage values have been used to justify a claim of "better" [Visser et al. 2006; Groce et al. 2012]. The most robust and usefully interpreted statistical measure for this question is *Kendall's τ rank correlation coefficient* [Kendall 1938; Cliff 1996].

Consider the coverage and mutation score data as a set of pairs (C, M), where C is the coverage value for a suite and M is the mutation score for that suite. Two pairs (C_1, M_1) and (C_2, M_2) are called *concordant* if the ordering of C_1 and C_2 matches the ordering of M_1 and M_2, i.e., $C_1 < C_2$ and $M_1 < M_2$ or $C_1 > C_2$ and $M_1 > M_2$. The pairs are called *discordant* if $C_1 < C_2$ and $M_1 > M_2$ or $C_1 > C_2$ and $M_1 < M_2$. Kendall's τ is the ratio of the difference between the number of concordant and discordant pairs and the total number of pairs. Kendall's original τ does not handle ties well, and thus was not suitable for our study, where several criteria can have many ties among suites for some subjects; extended discussion is in Section 5.7.

To illustrate concordant and discordant pairs, consider three test suites T_1, T_2, T_3 that have respective coverage values $0.3, 0.4, 0.5$ and mutation scores $0.7, 0.6, 0.8$. There are 2 concordant pairs — (T_1, T_3), (T_2, T_3) — where higher/lower coverage values have higher/lower mutation scores, and one discordant pair — (T_1, T_2).

Kendall's τ_b, used in our study, is a standard adaptation that adjusts for ties [Costner 1965]. Using a non-parametric rank correlation allows us to avoid the difficult question of whether the relationship between any criterion and mutation score is linear; τ_b does *not* make any assumption about the underlying functional relationships. A final attractive feature of τ_b is that in the absence of ties, the value can be intuitively interpreted: $0.5 + |\frac{\tau}{2}|$ is the probability of correctly predicting the ordering of mutation scores using the ordering of coverage values [Costner 1965]. Despite these desirable features of τ_b, our study is among the first to use τ_b in comparison of multiple coverage criteria. (A few studies [Wong et al. 1994; Namin and Andrews 2009; Wong et al. 1995] only mention τ or use it for other purposes.) Values for τ_b range from -1.0 (which would indicate that the coverage values are always opposite of the mutation score) to 1.0 (which would indicate a perfect predictive power for a criterion); a τ_b of 0.0 indicates there is no relationship between the rank ordering by the criterion and rank ordering by mutation score.

Spearman's ρ: A statistic similar to τ or τ_b is Spearman's ρ [Spearman 1904]; it is also a rank correlation coefficient. The primary arguments for ρ are tradition and ease of calculation. Also, Spearman's ρ handles ties by averaging the ranks. In many cases, ρ and τ/τ_b are very similar in value. Intuitively, we use ρ to measure the degree to which the coverage values and mutation scores are monotonic. When ρ is positive, it implies that coverage value tends to increase when mutation score increases, and when ρ is negative, it implies that coverage value tends to decrease when mutation score decreases. A ρ correlation coefficient of 1.0 indicates a perfect increasing monotone fit, and a coefficient of -1.0 indicates a perfect decreasing monotone fit. (Only a few previous studies of coverage criteria [Wong et al. 1994; Namin et al. 2008] briefly mention Spearman's ρ.)

$\mathbf{R^2}$: We also formed linear regression models for each criterion and obtained the R^2 *coefficient of determination* for the fits of those models to our data. It is well known that mutation scores do *not* depend linearly on coverage values [Cai and Lyu 2005; Andrews et al. 2006; Namin and Andrews 2009; Hassan and Andrews 2013], but R^2 still gives an indication of correlation. Intuitively, it attempts to answer the question: if one suite has $X\%$ higher coverage value than another suite, does it have a $c \cdot X\%$ higher mutation score? More precisely, it shows how well a linear model fits the actual data points, with 1.0 indicating a perfect fit and 0.0 indicating there is no relationship between the coverage and mutation score. Figure 2 shows lines that best fit the observed data.

5. EXPERIMENTAL RESULTS

5.1. Kendall's τ_b Rank Correlation

Tables III and IV show Kendall's τ_b correlation values for all subjects and all criteria we examined, for Coverage-varied Selection and Size-varied Selection, respectively. Each row highlights the best (darker/green) and worst (lighter/red) values. Note that we ignore the second column when highlighting. Values for PCT_{MS} are missing where manual selection of predicates was not used, and values for `SQLite` are repeated for both approaches. The first key observation is that most criteria had τ_b values over 0.5, often over 0.7, for most subjects. Using any of the criteria studied would correctly predict mutation score rankings for a large fraction of all test suites. Based on the standard Guilford scale [Guilford 1956], we would say that the mean values often showed

Table III: τ_b values for Coverage-varied Selection

Subject	Size	SC	BC	DBB	IMP	AIMP	PCT$_{MS}$	PCT$_{BB}$	PCT$_{ST}$
language: Java									
JFreeChart	0.958	0.962	0.966	0.961	0.845	0.964	-	0.951	0.936
JodaTime	0.937	0.966	0.972	0.958	0.965	0.964	-	0.959	0.961
AvlTree	0.012	0.773	0.774	0.665	0.783	0.785	0.756	0.789	0.816
BinomialHeap	-0.152	0.617	0.775	0.069	0.487	0.585	0.527	0.637	0.631
BinTree	0.389	0.132	0.220	0.340	0.341	0.351	0.491	0.417	0.510
FibHeap	0.058	0.759	0.807	0.692	0.278	0.395	0.509	0.634	0.515
FibonacciHeap	0.202	0.494	0.512	0.259	0.539	0.527	0.497	0.480	0.478
HeapArray	-0.017	0.803	0.801	-0.377	0.761	0.726	0.638	0.771	0.703
IntAVLTreeMap	0.239	0.777	0.770	0.612	0.788	0.815	0.786	0.728	0.762
IntRedBlackTree	0.111	0.710	0.741	-0.020	0.712	0.751	0.697	0.748	0.737
LinkedList	-0.048	0.756	0.746	0.603	0.713	0.716	0.746	0.705	0.701
NodeCachLList	-0.142	0.737	0.724	0.020	0.527	0.670	0.693	0.531	0.495
SinglyLList	0.243	0.577	0.586	0.174	0.451	0.495	0.492	0.571	0.634
TreeMap	0.242	0.747	0.772	0.578	0.690	0.748	0.721	0.743	0.755
TreeSet	0.063	0.755	0.784	0.346	0.696	0.770	0.737	0.752	0.772
language: C									
Space	0.876	0.926	0.929	0.881	0.913	0.929	-	0.917	0.911
SQLite	0.585	0.908	0.904	-	0.837	0.909	-	0.906	0.904
YAFFS2	0.347	0.688	0.702	0.347	0.501	0.690	-	0.667	0.680
Printtokens	0.552	0.894	0.781	0.548	0.901	0.916	-	0.794	0.855
Printtokens2	0.561	0.851	0.845	0.564	0.826	0.831	-	0.839	0.844
Replace	0.541	0.717	0.699	0.533	0.691	0.697	-	0.677	0.681
Schedule	0.437	0.773	0.776	0.408	0.747	0.766	-	0.716	0.711
Schedule2	0.339	0.766	0.767	0.338	0.683	0.749	-	0.691	0.751
SglibRbtree	0.693	0.763	0.793	0.691	0.680	0.698	-	0.765	0.762
Tcas	0.639	0.732	0.773	0.710	0.739	0.739	-	0.766	0.749
Totinfo	0.380	0.673	0.758	0.389	0.743	0.748	-	0.671	0.711
Standard deviation	ignored	0.166	0.147	0.318	0.172	0.158	0.116	0.134	0.133
Geometric mean	ignored	0.707	0.735	-	0.660	0.709	0.627	0.711	0.717
Arithmetic mean	ignored	0.741	0.757	0.452	0.686	0.728	0.638	0.724	0.729
The best results	ignored	5	13	0	1	5	0	0	3
The worst results	ignored	1	0	21	4	0	0	0	0

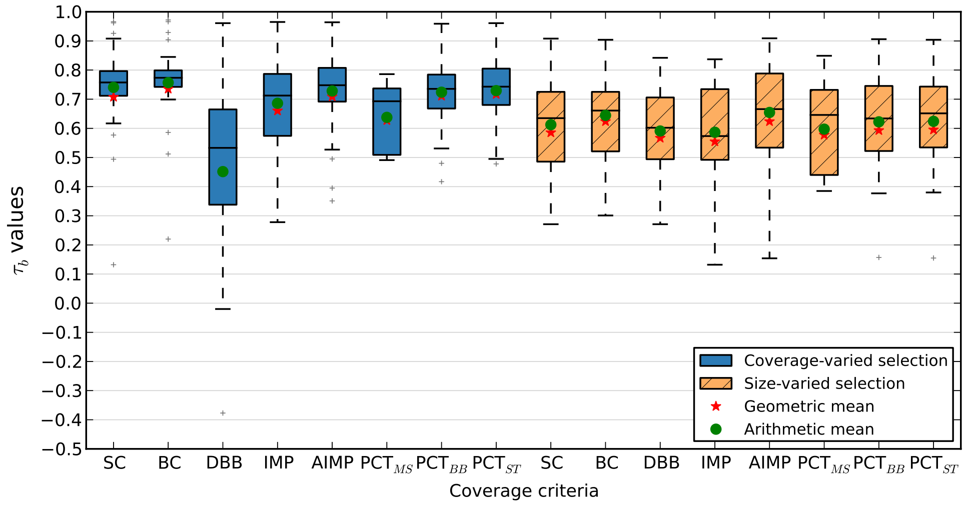


Fig. 3: Visualization of tables III and IV

Table IV: τ_b values for Size-varied Selection

Subject	Size	SC	BC	DBB	IMP	AIMP	PCT_{MS}	PCT_{BB}	PCT_{ST}
language: Java									
JFreeChart	0.703	0.777	0.818	0.813	0.768	0.792	-	0.818	0.776
JodaTime	0.748	0.808	0.835	0.842	0.836	0.840	-	0.826	0.815
AvlTree	0.560	0.301	0.301	0.301	0.556	0.492	0.494	0.520	0.530
BinomialHeap	0.428	0.624	0.629	0.629	0.367	0.521	0.409	0.467	0.450
BinTree	0.594	0.271	0.510	0.271	0.587	0.696	0.564	0.658	0.656
FibHeap	0.495	0.566	0.637	0.584	0.475	0.641	0.676	0.622	0.617
FibonacciHeap	0.479	0.409	0.419	0.411	0.492	0.487	0.440	0.389	0.395
HeapArray	0.507	0.728	0.723	0.728	0.519	0.742	0.646	0.592	0.583
IntAVLTreeMap	0.584	0.684	0.682	0.706	0.633	0.677	0.665	0.621	0.617
IntRedBlackTree	0.489	0.671	0.726	0.717	0.757	0.803	0.755	0.778	0.758
LinkedList	0.130	0.353	0.849	0.353	0.132	0.154	0.849	0.157	0.155
NodeCachLList	0.358	0.404	0.355	0.403	0.343	0.393	0.404	0.377	0.380
SinglyLList	0.466	0.494	0.494	0.494	0.419	0.824	0.385	0.667	0.699
TreeMap	0.492	0.680	0.700	0.696	0.759	0.777	0.746	0.741	0.738
TreeSet	0.511	0.703	0.739	0.733	0.736	0.774	0.732	0.764	0.754
language: C									
Space	0.793	0.853	0.858	0.836	0.815	0.881	-	0.769	0.759
SQLite	0.585	0.908	0.904	-	0.837	0.909	-	0.906	0.904
YAFFS2	0.583	0.614	0.640	0.591	0.466	0.655	-	0.640	0.632
Printtokens	0.642	0.815	0.627	0.670	0.730	0.829	-	0.617	0.688
Printtokens2	0.533	0.717	0.695	0.587	0.548	0.605	-	0.655	0.679
Replace	0.541	0.483	0.504	0.520	0.566	0.539	-	0.485	0.493
Schedule	0.551	0.776	0.720	0.630	0.546	0.653	-	0.731	0.745
Schedule2	0.562	0.474	0.493	0.512	0.588	0.532	-	0.529	0.548
SglibRbtree	0.567	0.646	0.627	0.602	0.581	0.583	-	0.628	0.647
Tcas	0.677	0.589	0.720	0.689	0.703	0.703	-	0.747	0.729
Totinfo	0.448	0.576	0.554	0.455	0.492	0.517	-	0.478	0.478
Standard deviation	ignored	0.173	0.156	0.161	0.170	0.172	0.157	0.166	0.163
Geometric mean	ignored	0.585	0.624	0.567	0.555	0.624	0.577	0.593	0.595
Arithmetic mean	ignored	0.612	0.645	0.591	0.587	0.655	0.597	0.622	0.624
The best results	ignored	4	3	3	4	10	3	2	1
The worst results	ignored	9	1	3	11	0	1	2	2

high (> 0.7) or nearly high (> 0.6) correlation, and almost all correlations were at least moderate (> 0.4). All values below 0.4, for criteria other than DBB, IMP and PCT_{MS}, came from just 4 simple Java data-structure classes. Given DBB's occasionally negative correlations, it is not clear that DBB is a useful criterion for suite evaluation for any purpose but fault localization, although even DBB often correlated very well.

The second key observation is that the absolute values and relative effectiveness of criteria vary with subject and test-suite selection approach, in a few cases by a wide range. However, considering all subjects and both approaches, it is clear that BC performs very well, and AIMP seems to perform best of the non-branch criteria (although PCT_{BB} and PCT_{ST} have slightly higher means for Coverage-varied Selection). For large subjects, coverage and mutation score ties were rare enough (more details in Section 5.7) that the values in the tables can be reasonably interpreted as indicating these criteria predict mutation score rank successfully 80% or more of the time. We additionally note that our results support, to a considerable extent, previous studies that used newer path and predicate criteria to evaluate test suites/techniques [Chaki et al. 2004; Ball 2004; Wang and Roychoudhury 2005; Visser et al. 2006; Pacheco et al. 2007; Chilimbi et al. 2009; Sharma et al. 2011; Groce 2011; Groce et al. 2012]: while PCT criteria were not our best, the PCT_{MS} with manually selected predicates performed well, and PCT performed better than IMP, which was used in fewer studies. Our results also indicate the benefit of using multiple criteria to evaluate suites, as is common practice in studies: while the worst correlation for some subjects is below 0.5, the best is over 0.5 in all but two subjects. Agreement between multiple criteria should increase confidence in a ranking.

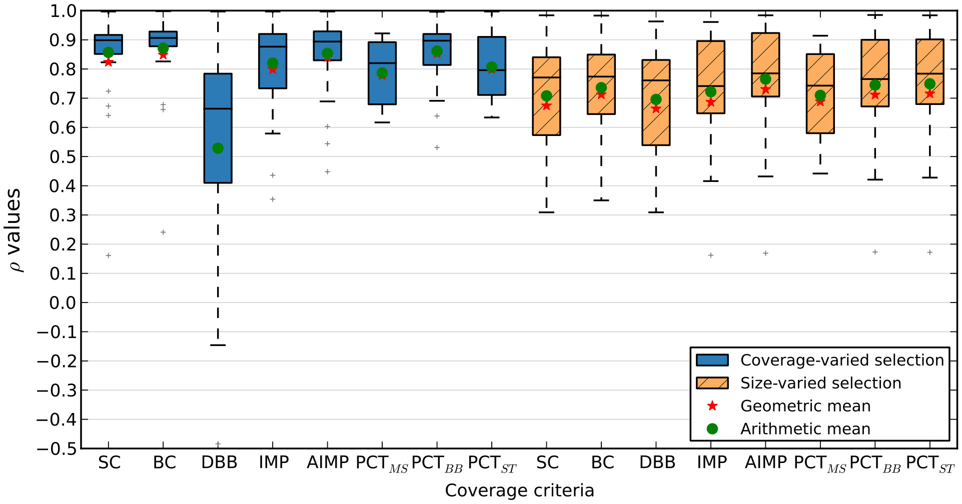


Fig. 4: ρ values for Size-varied and Coverage-varied Selection

To summarize the collected statistics (tables III and IV), we created a plot (Figure 3) that shows the distribution of τ_b (across all subjects) for all coverage criteria. In addition to the values commonly shown on a boxplot (e.g., median, outliers, upper and lower hinge, etc.), we show the geometric mean as (red) star and the arithmetic mean as (green) circle. In the following sections, we summarize the distributions (for Spearman's ρ and R^2) using plots; the exact values are available on the project's page [CoCo 2014].

5.2. Spearman's ρ Rank Correlation

Figure 4 shows Spearman's ρ correlation distribution and summary statistics across all subjects and all criteria. The first key observation is that most criteria had positive ρ values for most subjects. Negative values occurred only with DBB for Coverage-varied Selection. The second key observation is similar to that for Kendall's τ: it is clear that BC performs very well, and AIMP seems to perform best of the non-branch criteria (though PCT_{BB} has slightly higher means for Coverage-varied Selection).

5.3. Linear Regression

Figure 5 shows R^2 values across our subjects and all criteria. For the primary research question of this paper (the validity of using criteria to predict ranking of mutation scores), R^2 is less relevant than τ_b and ρ, and the validity of relative R^2 values may be compromised by non-linear relationships. However, the overall picture of the correlation between criteria and mutation scores changes from τ_b and ρ to R^2 only in that R^2 suggests that AIMP is often *better* than BC coverage for quantitative prediction. This confirms the claim that AIMP is the most useful non-BC criterion. We also note that in some cases R^2 for a coverage criterion is too low to suggest it as a valid predictor of mutation score, but Kendall's τ_b and Spearman's ρ show that the criterion nonetheless manages to have a high probability to correctly predict rank order of mutation scores.

Test Suite Size: We also examined the importance of suite size as a criterion, because previous work has considered the possibility that coverage criteria are primarily valuable because they force the production of large suites. This is not a major concern for us, because we minimize size as a confounding factor by using a wide

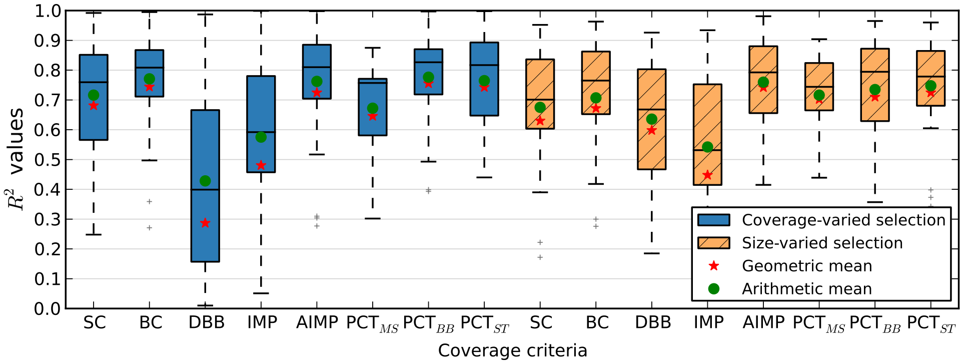


Fig. 5: R^2 values (mutants∼coverage) for Size-varied and Coverage-varied Selection

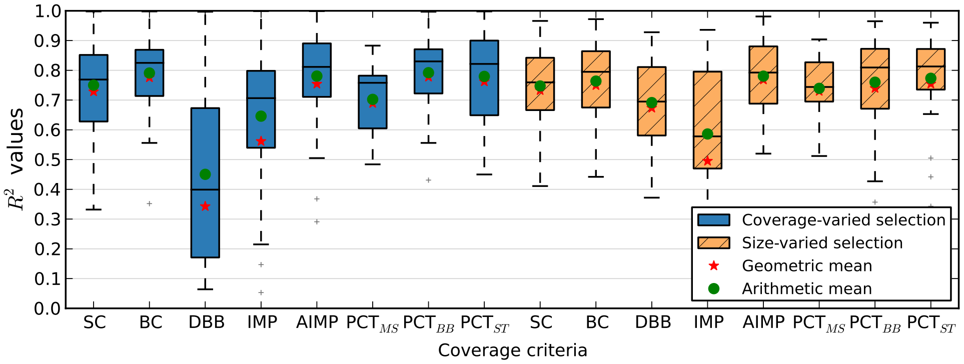


Fig. 6: R^2 values (mutants∼coverage+size) for Size-varied and Coverage-varied Selection

range of sizes with numerous suites of each size, and computing τ_b and ρ over *all* pairs (including many tied in size). We also note that a trend towards comparing only suites that require the same *computational effort* further reduces the importance of size [Harder et al. 2003; Groce et al. 2012; Groce et al. 2012]. For our subjects, using size alone to predict mutation score is an extremely ineffective predictor, with values of τ_b, ρ, and R^2 much worse than for other criteria (often < 0.25); we were surprised to even see small *negative* values for τ_b for some subjects (Figure 3). Further, as we show in figure 6 and 7, using size as an additional variable in regressions [Namin and Andrews 2009] did not change our general results: adding either $size$ or $\log(size)$ to coverage values improved R^2 for PCT criteria the most, but BC and AIMP still had higher correlations overall.

5.4. Combining Criteria

After observing the high effectiveness of BC, we attempted to exploit it by using BC as a base criterion and breaking ties with stronger criteria. Specifically, we lexicographically compared pairs, e.g., $\langle BC, AIMP \rangle$, for each suite such that BC is the primary criterion to compare suites, and iff two suites have the same BC, then the second criterion (AIMP in the example) is used to predict the mutation score ranking. However, the

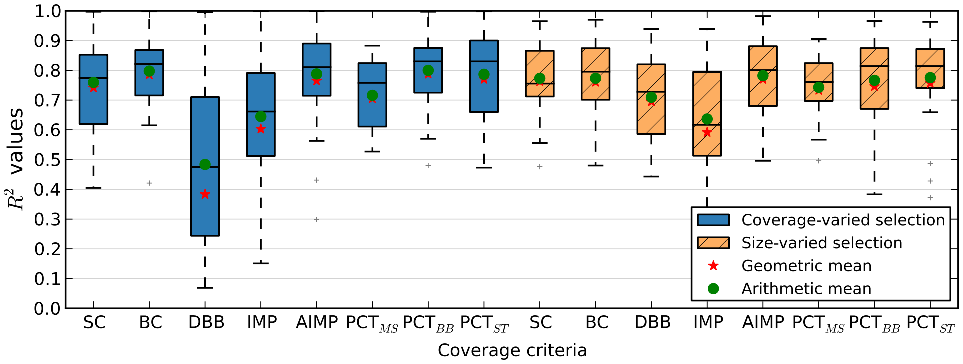


Fig. 7: R^2 values (mutants∼coverage+log(size)) for Size-varied and Coverage-varied Selection

Table V: Overhead measured as ratio of execution time the entire test pool on instrumented to original code

			Overhead/Slowdown			
Subject	**SC**	**BC**	**IMP/AIMP**	**PCT_{MS}**	**PCT_{BB}**	**PCT_{ST}**
			language: Java			
JFreeChart	4.21	3.71	3.84	-	4.30	4.79
JodaTime	55.38	63.50	92.31	-	67.50	61.88
AvlTree	3.73	2.07	39.87	4.14	22.59	21.92
BinomialHeap	2.48	2.14	13.01	4.96	11.58	12.27
BinTree	2.13	1.63	4.91	2.22	3.65	3.74
FibHeap	2.38	1.86	7.65	3.13	5.63	7.54
FibonacciHeap	2.05	1.31	5.95	3.00	4.17	5.48
HeapArray	1.79	2.00	6.41	2.34	6.62	6.70
IntAVLTreeMap	2.29	1.59	15.75	2.48	7.56	7.70
IntRedBlackTree	2.13	1.41	10.88	2.65	5.10	6.19
LinkedList	1.63	0.94	4.28	1.64	3.15	3.57
NodeCachLList	1.56	1.09	6.01	1.74	5.07	5.68
SinglyLList	1.97	1.86	5.85	3.22	4.80	5.14
TreeMap	2.25	1.62	15.33	3.45	11.41	10.19
TreeSet	2.02	1.66	14.11	4.59	10.98	9.24
			language: C			
Space	0.87	0.87	1.33	-	0.86	1.02
SQLite	1.40	1.40	31.83	-	15.87	58.43
YAFFS2	1.96	1.96	108.25	-	9.82	28.58
Printtokens	1.88	1.88	1.85	-	1.75	1.81
Printtokens2	2.29	2.29	2.85	-	2.35	2.86
Replace	2.30	2.30	2.68	-	2.17	2.59
Schedule	1.33	1.33	1.63	-	1.42	1.57
Schedule2	1.82	1.82	2.62	-	1.85	1.99
SglibRbtree	0.99	0.99	4.71	-	1.98	2.69
Tcas	1.99	1.99	2.01	-	2.27	2.65
Totinfo	1.66	1.66	2.13	-	1.77	1.90
Geometric mean	2.20	1.90	6.96	2.88	4.75	5.66

correlations were almost uniformly worse than for either criterion alone. It is possible that some other weighting of multiple criteria would perform better than any of the studied approaches; however, the complexity of devising such a scheme and measuring multiple criteria does not make this an immediately attractive approach, given that studied criteria are already effective.

Table VI: Statistics about percentage of tests that kill a mutant and execute a branch

	Tests killing mutants [%]				Tests executing branch [%]			
Subject	**Min**	**Max**	**Mean**	**SD**	**Min**	**Max**	**Mean**	**SD**
language: Java								
JFreeChart	0.05	26.79	0.34	1.00	0.05	29.72	0.44	1.41
JodaTime	0.03	75.10	0.61	2.65	0.03	82.42	1.35	5.29
AvlTree	0.01	100.00	41.94	38.69	45.39	100.00	77.05	17.12
BinomialHeap	0.07	98.72	41.86	28.09	2.48	98.72	67.52	24.19
BinTree	1.40	99.23	33.31	32.53	9.77	99.23	74.16	19.13
FibHeap	0.02	100.00	38.45	42.80	2.16	100.00	64.05	39.45
FibonacciHeap	0.02	99.98	32.91	37.54	4.89	99.98	69.60	27.84
HeapArray	1.33	100.00	49.87	37.24	1.48	100.00	59.33	33.26
IntAVLTreeMap	0.04	100.00	61.74	31.46	5.73	100.00	58.89	30.25
IntRedBlackTree	0.00	99.51	17.97	29.87	4.77	99.51	51.75	27.80
LinkedList	69.01	100.00	91.80	13.15	63.43	92.35	76.63	10.49
NodeCachLList	22.52	100.00	69.31	25.38	3.21	94.37	63.14	25.26
SinglyLList	7.15	94.32	41.90	29.95	24.80	94.32	47.70	22.85
TreeMap	0.04	99.29	20.11	26.44	2.29	99.29	40.67	26.34
TreeSet	0.03	99.42	26.96	29.95	3.33	99.42	49.57	27.15
language: C								
Space	0.07	100.00	17.22	27.41	0.07	100.00	24.67	33.16
SQLite	0.17	100.00	26.85	38.77	0.21	100.00	26.73	37.33
YAFFS2	0.02	100.00	32.83	42.23	0.02	100.00	77.61	33.90
Printtokens	0.17	100.00	38.86	34.60	0.29	99.27	57.95	39.36
Printtokens2	0.73	99.27	39.17	36.89	0.73	98.54	52.55	36.29
Replace	0.02	89.32	24.09	24.57	0.40	99.60	39.02	31.53
Schedule	0.04	100.00	45.61	29.06	0.45	98.87	64.28	30.86
Schedule2	0.04	85.28	60.40	28.82	0.33	98.86	69.92	36.61
SglibRbtree	0.70	100.00	81.24	32.05	0.02	100.00	62.60	37.91
Tcas	0.06	100.00	19.35	32.37	1.87	98.13	24.54	20.49
Totinfo	9.16	100.00	44.04	30.50	8.29	99.89	61.26	29.63

5.5. Cost of Measurement

While our key questions are about the predictive power of coverage criteria, we are also interested in the cost of measuring coverage. Table V shows the average overhead of measuring various criteria using our prototype tools. Our implementation of IMP/AIMP is simple; Ball and Larus [Ball and Larus 1996] provide a much faster precise approach, and the hash-based imprecise approach of Hassan and Andrews would also apply [Hassan and Andrews 2013]. Our results generally show feasibility for experimental evaluation of test suites, even with a very simple implementation. The key point is that our *worst* slowdown was slightly over 108X, and computing mutation score can take over 1000X. In some cases, the instrumented code is faster and takes even less time than the original code due to lightweight instrumentation and usual noise in experiments. Note that the table does not include numbers for DBB, as the values for this criterion are computed from statement coverage.

5.6. Quality of Mutants

Our results depend on the quality of the mutants, i.e., the difficulty of killing them. If all the mutants are easy to kill, a simple coverage criterion may perform unrealistically well. We therefore compare the percentage of tests that kill specific mutants to execution rates for branches. Table VI shows the results; we can see that some mutants, especially for large programs, can be killed by only a small fraction of tests, e.g., only 0.05% of all tests kill the least killed mutant for `JFreeChart`. It is clear that on average mutants are “harder” than branches for most subjects, with a lower minimum and mean kill/execute rate as well as a higher standard deviation.

Table VII: Percentage of tied pairs of test suites created using Coverage-varied Selection

Subject	**SC**	**BC**	**DBB**	**IMP**	**AIMP**	**PCT$_{MS}$**	**PCT$_{BB}$**	**PCT$_{ST}$**	**Mutants**
language: Java									
JFreeChart	0.00	0.06	0.08	0.02	0.06	-	0.00	0.00	0.02
JodaTime	0.02	0.04	0.10	0.04	0.06	-	0.06	0.00	0.04
AvlTree	7.96	9.53	18.12	5.79	6.18	1.74	1.11	1.34	4.93
BinomialHeap	10.80	10.36	10.66	0.84	3.25	0.48	0.86	0.67	7.84
BinTree	18.19	10.18	24.03	1.38	3.05	0.96	0.67	0.74	21.00
FibHeap	11.25	11.69	15.09	1.75	2.89	5.04	2.96	1.82	11.26
FibonacciHeap	9.08	9.16	12.78	1.96	4.19	1.35	1.59	0.91	5.32
HeapArray	24.14	14.79	17.85	1.18	5.28	3.27	0.93	1.00	5.03
IntAVLTreeMap	4.50	5.28	5.24	2.33	5.16	0.74	0.88	0.74	6.84
IntRedBlackTree	2.34	4.63	4.09	0.93	1.61	0.43	0.33	0.34	0.96
LinkedList	25.46	24.18	29.64	15.79	14.72	24.18	15.95	14.91	44.83
NodeCachLList	17.93	16.83	20.48	4.56	9.14	12.15	5.67	7.09	21.09
SinglyLList	16.71	16.85	17.65	3.26	7.23	3.71	5.31	4.87	16.46
TreeMap	2.21	4.71	4.24	0.79	1.57	0.43	0.26	0.21	1.75
TreeSet	1.99	3.96	4.97	0.87	1.83	0.60	0.45	0.32	1.89
language: C									
Space	0.09	0.19	13.52	0.31	0.34	-	0.07	0.03	0.27
SQLite	4.10	2.53	4.10	3.38	3.39	-	2.31	2.51	2.19
YAFFS2	0.25	0.32	83.58	0.21	0.54	-	0.05	0.02	0.23
Printtokens	1.41	4.03	45.47	2.01	2.14	-	1.23	0.36	0.45
Printtokens2	1.38	1.34	34.71	1.79	1.35	-	0.29	0.16	0.57
Replace	1.03	1.05	22.67	1.43	0.99	-	0.18	0.18	1.53
Schedule	11.20	6.04	47.48	2.89	3.69	-	0.52	0.22	1.23
Schedule2	4.88	6.31	64.39	3.05	3.88	-	1.03	0.27	1.12
SglibRbtree	0.50	1.14	26.63	1.15	2.73	-	0.06	0.02	2.71
Tcas	10.63	2.53	18.10	5.91	5.91	-	0.86	0.81	1.51
Totinfo	4.97	4.90	52.52	10.42	3.73	-	1.12	0.62	3.09
Arithmetic mean	7.42	6.64	23.01	2.85	3.65	4.24	1.72	1.54	6.31

Table VIII: Percentage of tied pairs of test suites created using Size-varied Selection

Subject	**SC**	**BC**	**DBB**	**IMP**	**AIMP**	**PCT$_{MS}$**	**PCT$_{BB}$**	**PCT$_{ST}$**	**Mutants**
language: Java									
JFreeChart	0.04	0.05	0.12	0.09	0.08	-	0.08	0.07	0.04
JodaTime	0.04	0.07	0.12	0.12	0.11	-	0.06	0.06	0.03
AvlTree	91.39	91.42	91.42	3.04	26.30	6.08	20.02	38.77	10.20
BinomialHeap	38.67	38.61	38.96	0.59	19.66	2.30	5.63	3.79	36.71
BinTree	92.60	67.52	92.61	0.43	10.59	2.68	4.25	2.18	15.89
FibHeap	21.60	16.63	25.18	0.51	13.69	1.90	6.49	6.24	9.19
FibonacciHeap	39.37	40.83	41.74	0.51	25.18	9.17	6.92	5.76	2.97
HeapArray	43.99	44.26	44.06	0.45	13.22	14.44	2.82	2.22	20.87
IntAVLTreeMap	34.41	34.51	36.31	6.42	21.23	1.57	3.23	2.51	34.68
IntRedBlackTree	18.31	20.58	21.82	1.03	2.56	0.76	0.54	0.50	0.81
LinkedList	86.00	97.58	86.01	0.54	26.21	97.58	28.82	27.36	98.25
NodeCachLList	45.96	47.72	45.98	0.60	24.77	26.66	21.02	19.01	85.07
SinglyLList	86.30	86.30	86.31	0.94	51.34	17.76	21.96	20.98	55.79
TreeMap	9.31	12.49	12.98	0.71	1.95	0.75	0.26	0.22	2.33
TreeSet	14.65	18.06	18.56	0.89	2.80	1.19	0.97	0.77	3.50
language: C									
Space	0.12	0.32	0.56	0.21	0.49	-	0.05	0.01	0.37
SQLite	4.10	2.53	4.10	3.38	3.39	-	2.31	2.51	2.19
YAFFS2	0.98	0.81	1.13	0.01	0.52	-	0.08	0.04	0.23
Printtokens	10.16	9.94	4.77	0.55	3.56	-	2.80	0.80	2.13
Printtokens2	9.94	9.04	3.24	0.53	4.51	-	1.47	0.85	4.68
Replace	4.35	3.15	2.06	0.48	1.77	-	0.41	0.24	2.70
Schedule	30.96	16.56	4.39	1.30	8.48	-	1.98	0.82	5.45
Schedule2	43.20	7.80	7.36	3.06	5.73	-	0.89	0.60	4.61
SglibRbtree	1.19	2.90	2.16	0.49	6.95	-	0.28	0.10	9.66
Tcas	24.81	9.19	19.16	3.91	3.91	-	0.28	0.17	0.94
Totinfo	42.99	37.72	7.32	0.93	5.26	-	1.56	0.62	63.61
Arithmetic mean	30.59	27.56	26.86	1.22	10.93	14.06	5.20	5.28	18.19

Table IX: Percentage of discordant/concordant pairs of test suites created using Coverage-varied Selection (averaged over all subject programs)

						Discordant Pairs				
		SC	**BC**	**DBB**	**IMP**	**AIMP**	**PCT_{MS}**	**PCT_{BB}**	**PCT_{ST}**	**Mutants**
						language: Java				
Concordant Pairs	**SC**		1.97	20.59	9.44	5.97	8.55	7.08	7.70	9.94
	BC	85.61		21.46	9.19	5.26	7.61	5.86	7.40	8.91
	DBB	64.10	63.24		28.31	24.79	28.46	25.63	25.25	23.12
	IMP	79.30	80.21	58.14		6.13	8.18	8.41	9.29	14.27
	AIMP	82.06	83.47	60.95	88.19		5.28	5.83	6.29	11.85
	PCT_{MS}	78.76	80.61	56.29	85.82	87.15		6.76	7.30	14.10
	PCT_{BB}	82.29	84.22	61.52	87.23	88.42	88.07		3.61	11.79
	PCT_{ST}	81.76	82.73	62.00	86.49	88.08	87.65	93.27		11.86
	Mutants	73.94	75.39	58.58	74.36	75.73	72.75	77.30	77.32	
						language: C				
	SC		9.32	5.31	13.23	10.98	-	11.45	10.46	9.17
	BC	84.92		6.87	7.62	4.45	-	3.91	3.72	14.06
	DBB	55.58	54.53		6.62	6.44	-	7.37	7.29	7.56
	IMP	80.94	87.41	54.91		4.93	-	9.02	8.76	15.92
	AIMP	83.52	90.87	55.15	90.75		-	6.36	5.98	14.45
	PCT_{MS}	-	-	-	-	-		-	-	-
	PCT_{BB}	84.53	93.09	54.93	87.70	90.68	-		3.14	15.52
	PCT_{ST}	85.72	93.45	55.08	88.14	91.26	-	96.03		15.19
	Mutants	86.23	82.06	54.31	80.10	81.89	-	82.59	83.15	

5.7. Ties for Criteria

A final concern about using criteria to compare test suites in research is the problem of ties — cases when test suites achieve the same coverage. For small subjects and large test pools, researchers often report that branch and statement coverage are highly similar (if not exactly the same) for test techniques that actually have different effectiveness for larger subjects. We investigated the likelihood of criteria with smaller number of requirements having larger number of ties. Tables VII and VIII show that there are indeed often more than 10% of tied suite pairs for simple subjects with some criteria, but with the exception of `LinkedList`, very seldom more than 5% with the other, stronger criteria.

6. DISCUSSION

The most surprising result in our study is that BC performs so well. A second somewhat surprising result is that, of non-BC criteria, AIMP performs best and performs *much* better than the more frequently used IMP, despite the fact that IMP subsumes AIMP. We believe that these two results are related. The ranking of criteria (to predict mutation scores) does *not* follow the subsumption hierarchy, although one might expect stronger criteria to predict mutation scores better than weaker criteria do. In fact, in many cases, exactly the opposite is true. Our belief is that there is a fundamental tension between strength and predictive power. Consider a criterion C that is weaker than another criterion C'; C' is most likely a better predictor than C for C-*adequate* suites (e.g., if we have many suites with 100% BC, then we cannot predict varying mutation scores among those suites using BC itself, but we could still use AIMP), but C' is less likely a better predictor than C for C-*non-adequate* suites (e.g., IMP is a worse predictor than AIMP, but BC is a better predictor than SC).

Viewed differently, we can consider the question: how much *information* does the coverage value for one criterion provide about the coverage value for another criterion? We realize that a subsumed criterion often (but not always) provides *more* information about the criterion that subsumes it than the reverse. For example, if a suite has an absolute BC value of k (with each test contributing at least one unique branch), we know that the suite has absolute AIMP, IMP, and PCT values of at least k. However,

Table X: Percentage of discordant/concordant pairs of test suites created using Size-varied Selection (averaged over all subject programs)

		Discordant Pairs								
		SC	**BC**	**DBB**	**IMP**	**AIMP**	**PCT$_{MS}$**	**PCT$_{BB}$**	**PCT$_{ST}$**	**Mutants**
	language: Java									
Concordant Pairs	**SC**		0.94	0.99	7.92	3.65	4.95	4.64	4.59	6.01
	BC	55.17		0.66	7.27	2.78	4.55	3.70	4.09	5.22
	DBB	56.03	55.01		7.05	2.86	4.03	3.84	4.00	5.02
	IMP	50.16	51.23	49.76		9.81	12.52	13.55	14.06	12.17
	AIMP	52.42	53.81	52.09	73.47		7.69	5.94	6.11	7.70
	PCT$_{MS}$	44.94	46.70	44.41	72.27	64.81		9.95	10.31	11.14
	PCT$_{BB}$	53.35	54.73	52.91	77.26	74.23	70.45		2.57	10.07
	PCT$_{ST}$	53.51	54.42	52.83	76.30	73.86	69.37	87.16		10.12
	Mutants	45.90	47.75	45.70	62.06	60.50	57.15	61.94	61.23	
	language: C									
	SC		10.64	9.14	15.52	13.47	-	13.16	12.30	10.07
	BC	68.98		13.60	13.33	7.36	-	3.90	4.72	15.95
	DBB	73.24	73.66		13.18	13.38	-	16.67	16.22	13.79
	IMP	68.12	76.63	80.85		10.11	-	15.72	16.07	17.30
	AIMP	68.24	80.80	78.15	85.22		-	10.09	10.35	16.05
	PCT$_{MS}$	-	-	-	-	-		-	-	-
	PCT$_{BB}$	70.54	86.57	77.39	82.01	85.03	-		4.28	18.09
	PCT$_{ST}$	71.73	86.00	78.31	82.14	85.21	-	94.28		18.12
	Mutants	69.99	69.55	73.27	72.86	71.95	-	72.33	72.69	

Table XI: Percentage of discordant pairs (c, m) and (c', m') such that $|c - c'| > N$ and $|m - m'| > N$ for Coverage-varied Selection

		N [%]										$max(\|c - c'\|*$
		5	10	15	20	25	30	35	40	45	50	$\|m - m'\|)$
Criterion	SC	24.99	9.93	4.28	2.86	0.91	0.75	0.06	0.00	0.00	0.00	1,946.14
	BC	36.28	16.89	8.20	5.34	2.25	1.41	0.56	0.34	0.17	0.09	5,617.69
	DBB	66.78	39.69	24.04	14.35	9.03	2.73	0.55	0.13	0.00	0.00	2,339.91
	IMP	8.50	2.85	1.57	1.50	0.43	0.24	0.01	0.00	0.00	0.00	1,422.40
	AIMP	41.68	19.96	10.90	6.65	2.67	1.29	0.33	0.16	0.07	0.03	4,703.60
	PCT$_{MS}$	47.87	16.74	6.19	4.37	1.05	0.70	0.00	0.00	0.00	0.00	1,770.60
	PCT$_{BB}$	34.44	14.20	7.09	4.65	1.68	0.87	0.31	0.18	0.09	0.04	5,039.00
	PCT$_{ST}$	35.18	15.20	7.53	4.37	1.45	0.79	0.46	0.29	0.15	0.08	5,820.53

Table XII: Percentage of discordant pairs (c, m) and (c', m') such that $|c - c'| > N$ and $|m - m'| > N$ for Size-varied Selection

		N [%]										$max(\|c - c'\|*$
		5	10	15	20	25	30	35	40	45	50	$\|m - m'\|)$
Criterion	SC	4.57	0.59	0.28	0.16	0.02	0.01	0.00	0.00	0.00	0.00	3,972.51
	BC	5.10	1.11	0.46	0.21	0.04	0.01	0.00	0.00	0.00	0.00	6,060.72
	DBB	15.93	3.61	1.89	0.55	0.18	0.02	0.01	0.00	0.00	0.00	2,138.50
	IMP	4.64	0.67	0.17	0.11	0.02	0.00	0.00	0.00	0.00	0.00	1,527.22
	AIMP	8.93	2.08	0.96	0.38	0.11	0.03	0.00	0.00	0.00	0.00	5,732.49
	PCT$_{MS}$	17.57	2.65	0.54	0.31	0.06	0.04	0.00	0.00	0.00	0.00	2,557.20
	PCT$_{BB}$	7.53	1.17	0.39	0.18	0.04	0.01	0.00	0.00	0.00	0.00	6,304.83
	PCT$_{ST}$	6.56	0.93	0.31	0.13	0.02	0.01	0.00	0.00	0.00	0.00	7,022.32

if we know that a suite has absolute AIMP, IMP, or PCT coverage of k, with each test contributing at least one path or PCT state, the absolute BC may be arbitrarily lower than k. In a sense, the weaker criteria in these cases provide “more” information about a suite, so we can expect them to better predict mutation scores. For example, a suite may obtain very high AIMP coverage without executing most code in the program, if the suite takes a huge number of paths through a single loop with many internal branches; similarly, absolute PCT coverage cannot distinguish between a suite that covers many (irrelevant) states of a small portion of a program and a suite that covers fewer states but executes most of the program. Given the nearly uniform distribution of mutants across a program, suites that do not execute most of the code are likely to have poor mutation scores. In contrast, a high BC value indicates that many easy-to-kill mutants are almost certainly killed. BC thus "warns" if a suite misses many "easy" faults; IMP/AIMP and PCT may not "warn".

The predictive power of BC weakens, however, as suites approach adequacy, when more ties are seen in BC, but mutation scores continue to diverge. The best predictive coverage may be the criterion that minimizes potentially meaningless information without converging too rapidly on 100% coverage. Among our evaluated criteria, AIMP seems to balance information content and avoidance of ties best: it always has a percentage of tied values for suites that is between the very high percentage of ties for BC and the very low percentages for PCT and IMP criteria. IMP had the lowest percentage of ties of all criteria but also proved nearly the least useful for predicting mutation score (Tables VII and VIII).

The usefulness of AIMP is encouraging. Hassan and Andrews have suggested that one reason def-use and other dataflow coverages have been little used in practice, despite encouraging results in some studies, is the difficulty of implementing the required static analyses [Hassan and Andrews 2013]. AIMP is usually trivial to add to instrumentation for collecting BC, if a fairly high overhead is acceptable (as done in this paper), and can be much more efficient if needed [Ball and Larus 1996; Ohmann and Liblit 2013]. Moreover, loop-free paths within a single function are intuitively easy to interpret, and Godefroid's compositional approach to dynamic symbolic execution essentially maximizes AIMP [Godefroid 2007]. In future studies evaluating test suites, our results suggest that IMP should be replaced with AIMP.

We believe PCT coverage may be less effective than AIMP because it uses *too many* predicates. PCT is inspired by abstraction in software model checking, which does not use all in-scope predicates at all points (which leads to a state-space explosion) but instead only uses those relevant to a specification [Henzinger et al. 2002; Chaki et al. 2003]. Investigating whether the superior performance of AIMP truly indicates that path-sensitivity is more important than logical-state-space coverage would require a similar selectivity. Unfortunately, the approaches used in model checking are impractical for testing large programs.

Unlike other coverage criteria, based on our experiments, DBB poorly predicts mutation score. However, note that this holds primarily for data structures for which the number of tests is large as the tests were automatically generated. More precisely, many tests may increase the number of DBBs, but the number of killed mutants may remain constant because most of the mutants for data structures are not hard to kill (Table VI). As seen in Table III, this may not be the case with larger programs. Although DBB may not be a good coverage criterion for predicting mutation score, it is effective for fault localization, according to a previous study [Baudry et al. 2006].

We note that, while BC and AIMP perform the best among the criteria we evaluated, the differences between the effective criteria are often not statistically significant. It is not possible, from our data, to draw a general conclusion that BC and AIMP always perform better for predicting the mutation score. The poor behavior of DBB, however, is statistically significant when compared to most other criteria, with `p-value < 0.05`, and in some cases IMP also has a significantly poorer performance. Given the small and non-random set of projects, however, this significance should be viewed as heuristic at best.

To quantitatively support our initial example from the introduction (Section 1)—using discordant pairs to illustrate the difficulty in choosing coverage for evaluating test suites—we measured average percentage of concordant and discordant pairs for all pairs of criteria. We observe that there are substantial percentages of discordant pairs (tables IX and X). Also, we observe that the percentage of discordant pairs is similar for Java and C, and does not vary substantially among the compared coverage criteria or between Size-varied Selection and Coverage-varied Selection.

Finally, tables XI and XII show some statistics about the discordant pairs. Specifically, we show the percentage of the discordant pairs (i.e., (c, m) and (c', m')) that have *both* difference in coverage ($|c - c'|$) and difference in mutation score ($|m - m'|$) above some threshold ($N \in \{5\%, 10\%, \ldots, 50\%\}$). The results show that one may frequently encounter discordant pairs with large differences. The last column (in tables XI and XII) characterizes the max combined differences in mutation score and coverage, i.e., $max(|c - c'| * |m - m'|)$. For BC, for example, the max of 5,617.69 for Coverage-varied Selection comes from two test suites where the difference in coverage is 68.35% while the difference in mutation score is 82.19%, and the max of 6,060.72 for Size-varied Selection comes from two test suites where the difference in coverage is 73.49% while the difference in mutation score is 82.47%.

6.1. Threats to Validity

The primary threat is to external validity: our set of programs and suites, while fairly large by the standards of previous literature on comparing coverage criteria, may not be representative of general results. In particular, we examined a larger number of data structures and a smaller number of real-world programs, and our examples were chosen in a partly opportunistic, rather than random, way: we needed subjects with many tests available or easily produced. Our selection of Java data structures, however, at minimum sheds light on the validity of several previous evaluations of testing techniques over these subjects. Construct validity is primarily threatened by ignoring some predicates for PCT because of technical constraints (e.g., we were not able to generate predicates in a class where instrumented methods would exceed the 64KB limit set by the Java classfile specification).

7. RELATED WORK

Many previous studies have investigated the effectiveness of coverage criteria. The contribution of this paper is to perform a large study to address the specific needs of researchers now investigating automated testing techniques: given two test suites, likely non-adequate, what criteria are best for predicting the ability of those suites to kill mutants (and thus, arguably, detect faults)? Are criteria recently adopted by researchers effective for this purpose?

Frankl and Weiss [Frankl and Weiss 1993] performed an experimental comparison of branch coverage (BC) and def-use coverage, showing that def-use is more effective than BC and that there is stronger correlation between def-use and fault detection than BC and fault detection; their primary conclusions concerned *adequate* suites, but some experiments included *non-adequate* suites. Our work targets similar questions but differs in that we compare SC, BC, DBB, IMP, AIMP, and PCT coverages, use larger applications, use a much larger set of tests produced by various testing techniques, use (many) mutants as opposed to (few) real bugs, and extensively explore non-adequate test suites.

Cai and Lyu [Cai and Lyu 2005] also investigated the correlation between different coverage criteria—BC, decision coverage, P-use, and C-use—and fault detection, using a linear regression model. Their conclusions are drawn based on experiments on one example, with 426 mutants and 1,200 tests. Different test suites were formed: all tests, tests from a specification, randomly generated tests, tests that cause exceptions, and tests that do not cause exceptions. Their results showed that coverage criteria were only a moderate indicator for fault detection, with large variance for different test suites. Some other studies [Hutchins et al. 1994; Frankl and Iakounenko 1998] also showed small or inconsistent correlation between coverage criteria and fault detection. Namin and Andrews [Namin and Andrews 2009] investigated the correlation between coverage criteria, effectiveness, and size of a test suite. The study showed that both coverage and size are *non-linearly* correlated with effectiveness. An additional conclusion was that the best result is achieved if both size and coverage are taken into account. Gupta and Jalote [Gupta and Jalote 2008] examined the *efficiency* of coverage criteria using minimal *adequate* test suites for SC, BC, and predicate coverage (the latter simply being coverage of all atomic predicates from conditionals measured only at the conditionals, not to be confused with PCT). In their results, while predicate coverage was the most effective (correlated to mutation score), BC was the most efficient when suite size was considered. Other studies [Li et al. 2009; Adolfsen 2011] used smaller programs and suites than the listed studies, and/or only examined small sets of (seeded) faults. A different kind of study by Wei et al. examined the correlation of BC to fault detection in 14 Eiffel classes, over a period of 2,520 hours of random testing (divided into 6 hour runs) [Wei et al. 2012]. They found that the correlation between BC and fault detection was very high during the first 10 minutes of testing, when new branches were frequently being covered, but once BC was close to saturated, the correlation became weak, and over 50% of faults were detected during the period between 30 minutes and 6 hours, when BC seldom increased. Their conclusion was that BC is a poor stopping criterion for random testing, and in this setting was not by itself a good measure of suite quality.

Studies investigating related questions (e.g., which criteria are best for prioritizing/minimizing regression suites) are numerous, with results that also vary, though BC has arguably performed fairly well [Rothermel et al. 2001]. Harder et al. examined the power of various adequacy criteria, noting the possibility of size as a confounding factor [Harder et al. 2003]. Some recent related work is that of Hassan and Andrews [Hassan and Andrews 2013], which extends previous work [Namin and Andrews 2009] to a comparison of BC, def-use coverage, and a novel coverage, called Multi-point Stride Coverage (MPSC), that has resemblances to a generalized version of AIMP. Their results showed that def-use coverage was highly correlated with BC in practice, BC was more correlated with fault detection than other criteria, and MPSC was fairly well correlated with fault detection. Since some MPSC coverages subsume AIMP, we would like to compare the two approaches using rank correlation to see if our findings with respect to strength and predictive power hold here as well. Of all previous studies, we find that only a few [Wong et al. 1994; Wong et al. 1995; Namin et al. 2008; Namin and Andrews 2009; Inozemtseva 2012] mention Kendall's τ or Spearman's ρ correlations, and those do not provide a comparison of multiple criteria as candidates for use in evaluating suites. For example, Inozemtseva [Inozemtseva 2012] only measures block coverage, and uses machine learning to find a regression involving this measure combined with suite size, but proposes no guidance as to whether block is the best coverage to measure. In contrast, we use τ_b and ρ to compare criteria, across a variety of suite selection and generation approaches. Recently, Papadakis et al. [Papadakis et al. 2014] used Kendall τ to compare correlation between mutation score and fault detection with correlation between t-wise coverage (for several Combinatorial Interaction Testing input models) and fault detection; the results showed that mutation score correlates better with fault detection than t-wise coverage correlates with fault detection.

Most recently, similar to our prior work [Gligoric et al. 2013], Inozemtseva and Holmes [Inozemtseva and Holmes 2014] studied correlation between test suite coverage (statement, branch, and modified condition coverage), size, and mutation score. The study was performed on five Java programs, including `JFreeChart` and `JodaTime`. Test suites were created by random sampling from the pool of existing (manually written) tests; 1,000 test suites were created for each size between three test methods and the available number of test methods. This approach for creating suites is similar to our Size-varied Selection. Their paper shared our conclusion that correlation does not follow the subsumption hierarchy (though in their work, this is reported over fewer criteria). The correlation results, when size is not controlled for, are also similar to our results for Size-varied Selection. Finally, their study reports low correlation when size is controlled for, in contrast to some studies. We also showed that size alone is a worse predictor of mutation score than any coverage criteria.

Another recent study [Gopinath et al. 2014] explicitly adapts the evaluation measures for coverage criteria used in this paper and applies them to a different, but related problem. Rather than comparing multiple suites for a single program (the typical research problem), the study addresses the problems of software developers attempting to determine whether a single, existing suite (be it manually written or automatically generated) for a program is effective. The goal (prediction of mutation scores) is the same, but the purpose is to determine if a single suite would have good mutation score, not to compare suites. Based on data from hundreds of open-source Java programs on GitHub, that study finds that *statement* coverage (vs. block, BC, and a variation of AIMP) best predicts mutation score for both manually written suites in the repository and tests automatically generated by Randoop [Pacheco et al. 2007]. We speculate that the difference in problem statement (correlation across multiple programs with a single suite vs. across multiple suites for each program) drives the difference in results, especially as it presumably results in many fewer ties. In general the results are not radically different than our own—all correlations (τ_b and R^2) are above 0.65 (and some above 0.9) for all criteria for manually written suites, though τ_b for suites generated by Randoop is relatively low for all criteria (0.48–0.54). The results also confirm our claim that the subsumption hierarchy does not match correlation with mutation scores; in fact, the ranking of criteria in their study is precisely the opposite of the subsumption hierarchy.

Shuler and Zeller [Schuler and Zeller 2013] propose the idea of *checked coverage* as a measure of oracle, rather than suite, effectiveness. Checked coverage measures coverage over the dynamic slice of statements influencing oracle statements only. They show that for seven open-source projects this approach is better able to detect degradation of oracle quality than even mutation testing. We focus only on traditional suite quality measurement, where the test inputs rather than the oracle alone are the primary target for evaluation.

Baudry et al. [Baudry et al. 2006] introduced the concept of dynamic basic blocks (DBBs) for measuring a test suite's fault-localization capability. Our work evaluates the use of DBBs as a coverage metric rather than for fault localization.

Ball [Ball 2004] introduced the theory behind PCT coverage and showed that PCT subsumes BC and various decision coverages, and is incomparable to path coverage. Although PCT was introduced in 2004 and was used to compare test-generation techniques, it was not extensively evaluated empirically. Our study is the first that implements PCT and empirically investigates the PCT criterion.

Another category of related work includes studies that used some of the criteria we used but for measuring the quality of test suites, which inspired our efforts. Visser et al. [Visser et al. 2006] were the first to instrument code for measuring an approximation of PCT coverage and compared a number of advanced test-generation techniques against random testing using PCT. Because of the lack of tools that can perform instrumentation for PCT, predicates were selected manually. Specifically, not all predicates were selected, the constructed predicates were not instantiated consistently at all points (either blocks or statements), and some predicates were instantiated when they were not in scope. Pacheco et al. [Pacheco et al. 2007] used the same approach to PCT to demonstrate the effectiveness of feedback in random test generation. Later, Sharma et al. [Sharma et al. 2011] compared random testing and shape abstraction on the same set of predicates as previous studies, but predicates were instantiated systematically at all basic blocks. An extended version of that instrumentation was used [Groce 2011; Groce et al. 2012] to evaluate the effectiveness of a new test-generation technique based on reinforcement learning.

The last category of related work includes tools for measuring code coverage. There are many tools available for both Java [Emma 2013; Cobertura 2013] and C [gcov 2013] that can measure method, statement, branch, and path coverage. Additionally, tools for mutation testing [Schuler and Zeller 2009] can be placed in this category. Ours is the first tool for systematically measuring Ball's PCT coverage. Because detailed empirical evaluation requires such a tool, we implemented tools, both for Java and C, that can instrument code for measuring PCT. Using our tools, we were able to automatically and systematically instrument reasonably large code bases. The only previous attempt (to our knowledge) to address PCT in practical automated terms was in the FShell system [Holzer et al. 2009], which can perform model checking queries to find paths to satisfy PCT coverage goals in C programs, but relies on being provided a list of relevant predicates, does not distinguish between variables with the same name in different scopes, and does not instrument for runtime collection of coverage data.

8. CONCLUSIONS

This paper considers these questions: (1) for researchers wishing to compare test suites but lacking a statistically significant number of real faults and lacking the computational resources to perform mutation testing, is it useful to compare suites using coverage criteria; if so, (2) which criteria are best at predicting mutation scores? Recent literature has shown that these are critical questions to answer, because publications are increasingly using coverage criteria to compare test suites and techniques. Our results suggest that due to high effectiveness and low overhead, researchers should use *branch coverage* to compare suites whenever possible, but most evaluated criteria performed well in terms of predicting mutation score for most of our subjects, with only dynamic basic blocks arguably ineffective for many small subjects. A variation of intra-procedural acyclic path coverage performed best of all non-branch coverage criteria, and has desirable simplicity, ease of implementation, and reasonable overhead. Future work should evaluate these and other criteria on a larger set of subject programs and test suites.

ACKNOWLEDGMENTS

We would like to thank Yu Lin, Qingzhou Luo, and Shalini Shamasunder for discussions about this work, Mladen Laudanovic and Douglas Simpson for help with statistical analysis, Lingming Zhang for help with Javalanche, Jamie Andrews for extremely valuable comments and for providing the C mutation tool, and Fredrik Kjolstad for help with WALA.

This material is based upon work partially supported by the National Science Foundation under Grant Nos. CCF-0746856, CNS-0958199, CCF-1012759, and CCF-1054876.

REFERENCES

Martijn Adolfsen. 2011. *Industrial validation of test coverage quality*. Master's thesis. University of Twente.

Paul Ammann and Jeff Offutt. 2008. *Introduction to Software Testing*. Cambridge University Press.

James H. Andrews, Lionel C. Briand, and Yvan Labiche. 2005. Is mutation an appropriate tool for testing experiments?. In *International Conference on Software Engineering*. 402–411.

James H. Andrews, Lionel C. Briand, Yvan Labiche, and Akbar Siami Namin. 2006. Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. *Trans. Softw. Eng.* 32 (2006), 608–624.

Andrea Arcuri and Lionel C. Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *International Conference on Software Engineering*. 1–10.

Thomas Ball. 2004. *A Theory of Predicate-Complete Test Coverage and Generation*. Technical Report MSR-TR-2004-28. Microsoft Research.

Thomas Ball. 2005. A Theory of Predicate-Complete Test Coverage and Generation. In *Formal Methods for Components and Objects*. 1–22.

Thomas Ball and James R. Larus. 1996. Efficient Path Profiling. In *International Symposium on Microarchitecture*. 46–57.

Thomas Ball and Sriram K Rajamani. 2001. Automatically Validating Temporal Safety Properties of Interfaces. In *Workshop on Model Checking of Software*. 103–122.

Benoit Baudry, Franck Fleurey, and Yves Le Traon. 2006. Improving Test Suites for Efficient Fault Localization. In *International Conference on Software Engineering*. 82–91.

Xia Cai and Michael R. Lyu. 2005. The effect of code coverage on fault detection under different testing profiles. In *International Workshop on Advances in Model-Based Testing*. 1–7.

Sagar Chaki, Edmund M. Clarke, Alex Groce, and Ofer Strichman. 2003. Predicate Abstraction with Minimum Predicates. In *Correct Hardware Design and Verification Methods*. 19–34.

Sagar Chaki, Alex Groce, and Ofer Strichman. 2004. Explaining Abstract Counterexamples. In *Symposium on the Foundations of Software Engineering*. 73–82.

Trishul M. Chilimbi, Ben Liblit, Krishna Mehra, Aditya V. Nori, and Kapil Vaswani. 2009. HOLMES: Effective statistical debugging via efficient path profiling. In *International Conference on Software Engineering*. 34–44.

Norman Cliff. 1996. *Ordinal Methods for Behavioral Data Analysis*. Pyschology Press.

Cloc 2013. Count lines of code. http://cloc.sourceforge.net/.

Cobertura 2013. Cobertura. http://cobertura.sourceforge.net/.

CoCo 2014. CoCo. http://mir.cs.illinois.edu/coco/.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. 2009. *Introduction to Algorithms, Third Edition*. The MIT Press.

Herbert L. Costner. 1965. Criteria for Measures of Association. *American Sociological Review* 3 (1965).

Coverage 2013. Instrumented Container Classes - Predicate Coverage. http://mir.cs.illinois.edu/coverage/.

Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11 (1978), 34–41.

Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. 2005. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Softw. Engg.* 10 (2005), 405–435.

Eclipse 2013. Eclipse. http://http://www.eclipse.org/.

Emma 2013. EMMA. http://emma.sourceforge.net/.

Phyllis G. Frankl and Oleg Iakounenko. 1998. Further empirical studies of test effectiveness. In *Symposium on the Foundations of Software Engineering*. 153–162.

Phyllis G. Frankl and Stewart N. Weiss. 1993. An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow Testing. *Trans. Software Eng.* 19 (1993), 774–787.

Chen Fu and Barbara G. Ryder. 2005. Navigating error recovery code in Java applications. In *Workshop on Eclipse Technology eXchange*. 40–44.

Juan Pablo Galeotti, Nicolás Rosner, Carlos Gustavo López Pombo, and Marcelo Fabian Frias. 2010. Analysis of invariants for efficient bounded verification. In *International Symposium on Software Testing and Analysis*. 25–36.

gcov 2013. gcov–a Test Coverage Program. http://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. 2013. Comparing Non-adequate Test Suites Using Coverage Criteria. In *International Symposium on Software Testing and Analysis*. 302–313.

Patrice Godefroid. 2007. Compositional dynamic test generation. In *Symposium on Principles of Programming Languages*. 47–54.

Rahul Gopinath, Carlos Jensen, and Alex Groce. 2014. Code Coverage for Suite Evaluation by Developers. In *International Conference on Software Engineering*. 72–82.

Alex Groce. 2009. (Quickly) Testing the Tester via Path Coverage. In *Workshop on Dynamic Analysis*. 22–28.

Alex Groce. 2011. Coverage rewarded: Test input generation via adaptation-based programming. In *International Conference on Automated Software Engineering*. 380–383.

Alex Groce, Alan Fern, Jervis Pinto, Tim Bauer, Mohammad Amin Alipour, Martin Erwig, and Camden Lopez. 2012. Lightweight Automated Testing with Adaptation-Based Programming. In *International Symposium on Software Reliability Engineering*. 161–170.

Alex Groce, Gerard Holzmann, and Rajeev Joshi. 2007. Randomized Differential Testing as a Prelude to Formal Verification. In *International Conference on Software Engineering*. 621–631.

Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. 2012. Swarm Testing. In *International Symposium on Software Testing and Analysis*. 78–88.

Joy Paul Guilford. 1956. *Fundamental Statistics in Pyschology and Education*. McGraw-Hill.

Atul Gupta and Pankaj Jalote. 2008. An approach for experimentally evaluating effectiveness and efficiency of coverage criteria for software testing. *Softw. Tools Technol. Transf.* 10 (2008), 145–160.

Richard G. Hamlet. 1977. Testing Programs with the Aid of a Compiler. *Trans. Softw. Eng.* 3 (1977), 279–290.

Michael Harder, Jeff Mellen, and Michael D. Ernst. 2003. Improving test suites via operational abstraction. In *International Conference on Software Engineering*. 60–71.

Mohammad Mahdi Hassan and James H. Andrews. 2013. Comparing Multi-point Stride Coverage and Dataflow Coverage. In *International Conference on Software Engineering*. 172–181.

Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. 2002. Lazy abstraction. In *Symposium on Principles of Programming Languages*. 58–70.

Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. 2009. Query-Driven Program Testing. In *International Conference on Verification, Model Checking, and Abstract Interpretation*. 151–166.

Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. 1994. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *International Conference on Software Engineering*. 191–200.

Laura Inozemtseva and Reid Holmes. 2014. Coverage is not strongly correlated with test suite effectiveness. In *International Conference on Software Engineering*. 435–445.

Laura Michelle McLean Inozemtseva. 2012. *Predicting Test Suite Effectiveness for Java Programs*. Master's thesis. University of Waterloo.

JFreeChart 2013. JFreeChart. http://www.jfree.org/jfreechart/.

Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *Trans. Soft. Eng.* 37 (2011), 649–678.

JodaTime 2013. JodaTime. http://joda-time.sourceforge.net/.

René Just, Gregory M. Kapfhammer, and Franz Schweiggert. 2012. Using Non-redundant Mutation Operators and Test Suite Prioritization to Achieve Efficient and Scalable Mutation Analysis. In *International Symposium on Software Reliability Engineering*. 11–20.

Maurice Kendall. 1938. A New Measure of Rank Correlation. *Biometrika* 1-2 (1938), 81–89.

James R. Larus. 1999. Whole Program Paths. In *Programming Language Design and Implementation*. 259–269.

Nan Li, Upsorn Praphamontripong, and Jeff Offutt. 2009. An Experimental Comparison of Four Unit Test Criteria: Mutation, Edge-Pair, All-Uses and Prime Path Coverage. In *International Workshop on Mutation Analysis*. 220–229.

Akbar Siami Namin and James H. Andrews. 2009. The influence of size and coverage on test suite effectiveness. In *International Symposium on Software Testing and Analysis*. 57–68.

Akbar Siami Namin, James H. Andrews, and Duncan J. Murdoch. 2008. Sufficient mutation operators for measuring test effectiveness. In *International Conference on Software Engineering*. 351–360.

George Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. 2002. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *International Conference on Compiler Construction*. 213–228.

A. Jefferson Offutt, Gregg Rothermel, and Christian Zapf. 1993. An experimental evaluation of selective mutation. In *International Conference on Software Engineering*. 100–107.

Peter Ohmann and Ben Liblit. 2013. Lightweight Control-Flow Instrumentation and Postmortem Analysis in Support of Debugging. In *International Conference on Automated Software Engineering*. 378–388.

Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *International Conference on Software Engineering*. 75–84.

Mike Papadakis, Christopher Henard, and Yves Le Traon. 2014. Sampling Program Inputs with Mutation Analysis: Going Beyond Combinatorial Interaction Testing. In *International Conference on Software Testing, Verification and Validation*. 1–10.

Sanjay J Patel, Tony Tung, Satarupa Bose, and Matthew M Crum. 2000. Increasing the size of atomic instruction blocks using control flow assertions. In *International Symposium on Microarchitecture*. IEEE, 303–313.

Purify 2013. IBM Rational Purify Documentation. ftp://ftp.software.ibm.com/software/rational/docs/documentation/manuals/unixsuites/pdf/purify/purify.pdf.

Gregg Rothermel, Roland Untch, Chengyun Chu, and Mary Jean Harrold. 2001. Test Case Prioritization. *Trans. Softw. Eng.* 27 (2001), 929–948.

Atanas Rountev. 2004. Precise Identification of Side-Effect-Free Methods in Java. In *International Conference on Software Maintenance*. 82–91.

David Schuler and Andreas Zeller. 2009. Javalanche: efficient mutation testing for Java. In *Symposium on the Foundations of Software Engineering*. 297–298.

David Schuler and Andreas Zeller. 2013. Checked coverage: an indicator for oracle quality. *Software Testing, Verification and Reliability* 23, 7 (2013), 531–551.

Rohan Sharma, Milos Gligoric, Andrea Arcuri, Gordon Fraser, and Darko Marinov. 2011. Testing Container Classes: Random or Systematic?. In *Fundamental Approaches to Software Engineering*. 262–277.

Rohan Sharma, Milos Gligoric, Vilas Jagannath, and Darko Marinov. 2010. A Comparison of Constraint-Based and Sequence-Based Generation of Complex Input Data Structures. In *Software Testing, Verification, and Validation Workshops*. 337–342.

Charles Spearman. 1904. The proof and measurement of association between two things. *The American journal of psychology* 15, 1 (1904), 72–101.

SQLite 2013. SQLite. http://www.sqlite.org/.

Alexandru Sălcianu and Martin Rinard. 2005. Purity and Side Effect Analysis for Java Programs. In *Verification, Model Checking, and Abstract Interpretation*. 199–215.

Willem Visser, Corina S. Pasareanu, and Radek Pelánek. 2006. Test input generation for Java containers using state matching. In *International Symposium on Software Testing and Analysis*. 37–48.

Marian Vittek, Peter Borovansky, and Pierre-Etienne Moreau. 2006. A Simple Generic Library for C. In *International Conference on Software Reuse*. 423–426.

VMSpec 2013. Java class file format. http://docs.oracle.com/javase/specs/jvms/se5.0/html/ClassFile.doc.html.

Filipos I. Vokolos and Phyllis G. Frankl. 1998. Empirical Evaluation of the Textual Differencing Regression Testing Technique. In *International Conference on Software Maintenance*. 44–53.

WALA 2013. WALA: T. J. Watson Libraries for Analysis. http://wala.sf.net.

Tao Wang and Abhik Roychoudhury. 2005. Automated path generation for software fault localization. In *International Conference on Automated Software Engineering*. 347–351.

Yi Wei, Bertrand Meyer, and Manuel Oriol. 2012. Is Branch Coverage a Good Measure of Testing Effectiveness? In *Empirical Software Engineering and Verification*, Bertrand Meyer and Martin Nordio (Eds.). Vol. 7007. Springer Berlin Heidelberg, 194–212.

W. Eric Wong, Joseph R. Horgan, Saul London, and Aditya P. Mathur. 1994. Effect of test set size and block coverage on the fault detection effectiveness. In *International Symposium on Software Reliability*. 230–238.

W. Eric Wong, Joseph R. Horgan, Saul London, and Aditya P. Mathur. 1995. Effect of Test Set Minimization on Fault Detection Effectiveness. In *International Conference on Software Engineering*. 41–50.

YAFFS2 2013. YAFFS: A flash file system for embedded use. http://www.yaffs.net.

Lingming Zhang, Milos Gligoric, Darko Marinov, and Sarfraz Khurshid. 2013. Operator-based and random mutant selection: Better together. In *International Conference on Automated Software Engineering*. 92–102.

Lu Zhang, Shan-Shan Hou, Jun-Jue Hu, Tao Xie, and Hong Mei. 2010. Is operator-based mutant selection superior to random mutant selection?. In *International Conference on Software Engineering*. 435–444.

Received January 2014; revised July 2014; accepted August 2014