SOFTWARE TESTING, VERIFICATION AND RELIABILITY
*Softw. Test. Verif. Reliab.* (2014)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/stvr.1539

# Are concurrency coverage metrics effective for testing: a comprehensive empirical investigation

Shin Hong[1], Matt Staats[2], Jaemin Ahn[3], Moonzoo Kim[1,*,†] and Gregg Rothermel[4]

[1]*Department of Computer Science, KAIST, Daejeon, South Korea*
[2]*SnT Centre, University of Luxembourg, Luxembourg*
[3]*Agency for Defense Development, Daejeon, South Korea*
[4]*Department of Computer Science, University of Nebraska-Lincoln, Lincoln, USA*

SUMMARY

Testing multithreaded programs is inherently challenging, as programs can exhibit numerous thread interactions. To help engineers test these programs cost-effectively, researchers have proposed concurrency coverage metrics. These metrics are intended to be used as predictors for testing effectiveness and provide targets for test generation. The effectiveness of these metrics, however, remains largely unexamined. In this work, we explore the impact of concurrency coverage metrics on testing effectiveness and examine the relationship between coverage, fault detection, and test suite size. We study eight existing concurrency coverage metrics and six new metrics formed by combining complementary metrics. Our results indicate that the metrics are moderate to strong predictors of testing effectiveness and effective at providing test generation targets. Nevertheless, metric effectiveness varies across programs, and even combinations of complementary metrics do not consistently provide effective testing. These results highlight the need for additional work on concurrency coverage metrics. Copyright © 2014 John Wiley & Sons, Ltd.

Received 24 July 2013; Revised 21 April 2014; Accepted 30 April 2014

KEY WORDS: software testing; test coverage metric; concurrent program

## 1. INTRODUCTION

Testing multithreaded programs is challenging, because in most applications, a large number of thread interactions are possible and exploring all potential interactions is infeasible. While several techniques for detecting concurrency faults (e.g., [1–3]) have been developed as alternatives, these techniques have limited accuracy, and thus, more systematic concurrent program testing approaches are desirable.

To address this problem, researchers have developed concurrency coverage metrics for multithreaded programs [4–7]. These metrics, like structural coverage metrics such as branch and statement coverage, define a set of test requirements to be satisfied. In the case of concurrency coverage metrics, the test requirements typically enumerate a set of possible interleavings of synchronization operations or shared variable accesses. Just as structural coverage metrics offer a rough estimate of how well testing has covered a program's structure, concurrency coverage metrics allow engineers to estimate how well they have exercised concurrent program behaviours.

*Correspondence to: Moonzoo Kim, Department of Computer Science, KAIST, Daejeon, South Korea.
†E-mail: moonzoo@cs.kaist.ac.kr

The intuition behind all test coverage metrics is that as more requirements relative to the metric are satisfied, the testing process is more likely to detect faults and thus is more *effective*. Accordingly, to maximize the effectiveness of testing processes, researchers create test adequacy criteria based on these metrics and develop techniques to satisfy them. The development of such techniques has long been an active area of research in the context of structural coverage metrics for non-concurrent programs [8–11], and as multithreaded programs have become more common, the development of techniques centered around concurrency coverage metrics has also become an active area of research [12–15].

Unfortunately, the intuition behind concurrency coverage metrics remains largely unexplored prior to our own recent study [16]. While a large body of evidence exists exploring the impact of structural coverage metrics on testing effectiveness (e.g., [17–19]), we are aware of no study rigorously examining the impact of concurrency coverage metrics. We expect that increasing coverage relative to these metrics will improve testing effectiveness, but we also expect that it will increase test suite size. Thus, we must ask: does improving concurrency coverage directly lead to a more effective testing process, or is it merely a by-product of increasing test suite size? Further, if improving coverage does lead to increased testing effectiveness, what practical gains in testing effectiveness can we expect? Finally, based on the effectiveness of the current state-of-the-art concurrency coverage metrics, what steps should be taken with respect to continuing the development of test case generation techniques for concurrency coverage metrics?

To explore these questions, we studied the application of eight concurrency coverage metrics in testing 12 concurrent programs [16]. For each program and metric pair, we used a randomized test case generation process to generate 90 000 test suites with varying levels of size and coverage, and we measured the relationships between the percentage of test requirements satisfied, the number of test executions, and the fault detection ability of test suites via correlation and linear regression. Additionally, we compared test suites generated to achieve high coverage against random test suites of equal size. Finally, we examined the value of combining complimentary concurrency coverage metrics, and the impact of difficult-to-cover requirements on the testing process. We measured fault detection ability using both mutation analysis (systematically seeding concurrency faults) and real-world faults.

Our results show that each coverage metric explored has value in predicting concurrency testing effectiveness and as a target for test case generation. In sharp contrast to work on sequential coverage metrics [18] and the intent of the concurrency metrics, however, the metrics' results vary across programs. In particular, we found that the correlation between concurrency coverage and fault detection, while often moderate to strong (i.e., 0.4 to 0.8) and stronger than the relationship between test suite size and fault detection, is occasionally low to non-existent.

We also found that while large increases in fault detection effectiveness (up to 25 times) can be found when using concurrency coverage metrics as targets for test case generation relative to random test suites of equal size, in some cases, the results were no better than random testing. Further analysis indicated that it may be possible to develop test case generation approaches that improve fault detection by specifically targeting difficult-to-cover test requirements.

Finally, we found that while combining proposed coverage metrics can alleviate issues involving inconsistency across objects, and test suites reduced with respect to the combined metrics outperform the original metrics in most cases, there still appear to be other factors unaccounted for by the metrics (e.g., configurations of test case generation techniques such as random noise injection probability and length).

Given these results, we believe that while existing concurrency coverage metrics have value, and efforts to develop techniques based on these metrics are justified, additional work on such metrics is required. In particular, the variability in metric effectiveness across programs highlights the need for guidelines to help engineers select from among the many metrics already proposed. Additionally, the impact of the parameters used in random testing, which in some cases are much stronger predictors of testing effectiveness, indicates that the metrics can be improved to better capture the factors that constitute effective concurrency testing.



DOI: 10.1002/stvr

## 2. BACKGROUND AND RELATED WORK

### 2.1. Concurrency coverage metrics

Structural coverage metrics for concurrent programs, like their sequential counterparts such as branch and statement coverage metrics, are used to derive a set of test requirements from the code elements of a program under test. These test requirements typically enumerate a set of thread interleaving cases. Unlike sequential metrics, satisfying a test requirement for a concurrent program requires engineers not only to execute specific code elements (generally synchronization and/or shared data access operations) but also to satisfy constraints on thread interactions. For example, the *Blocked* metric requires every synchronization block/method in a program to be blocked (due to lock contention) at least once during testing [12].

Figure 1 provides an example of how concurrency coverage metrics define test requirements and of ways in which test requirements are covered. The program in Figure 1(a) consists of two threads that execute two synchronized blocks guarded by the same lock `m`. In the example execution shown in Figure 1(b), Thread 1 holds lock `m` first (line 12), defines variable `x` (line 13), and then releases the lock (line 14). While Thread 1 holds lock `m`, Thread 2 attempts to acquire the lock on `m` (line 22), which blocks Thread 2 until Thread 1 releases the lock. After acquiring the lock on `m`, Thread 2 reads variable `x` (line 23) whose value has been defined by Thread 1, assigns the value read to variable `y`, and then releases the lock on `m`.

The concurrency coverage metric *Follows* defines a pair of test requirements that are covered by an execution when two synchronized blocks are executed by two different threads, and these hold the same lock consecutively. In the example, the execution covers *Follows* test requirement $\langle 12, 22 \rangle$ because the lock `m` is first held at line 12 by Thread 1 and then held at line 22 by Thread 2. The second *Follows* test requirement $\langle 22, 12 \rangle$ is not covered in the execution.

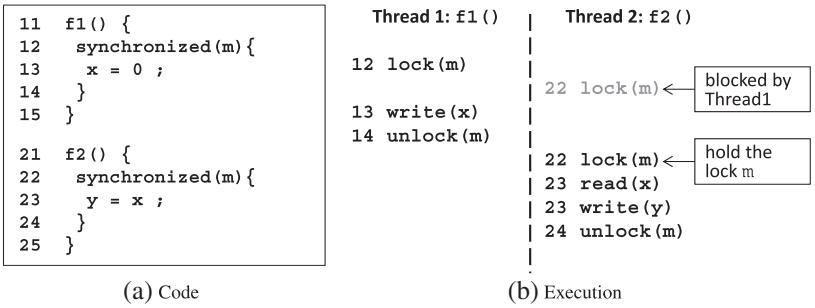
Another metric, *Blocked*, defines one test requirement per synchronized block that is covered by an execution when one thread becomes blocked from acquiring the lock for the synchronized block. In the example, there are two *Blocked* test requirements, 12 and 22; 22 is covered in the execution scenario because Thread 2 is blocked by Thread 1 at line 22.

The coverage metric *PSet* generates test requirements for def-use relations over different threads. In the example, a *PSet* test requirement $\langle 13, 23 \rangle$ is covered as Thread 2 reads the variable `x` at line 23, whose last update is by Thread 1 at line 13.

### 2.2. Assessing the effectiveness of concurrency coverage metrics

In work on concurrency coverage metrics, the effectiveness of achieving high coverage has been argued for primarily through analytical comparisons between coverage definitions and concurrency fault pattern, such as those involving data races and atomicity violations [5, 6, 14].

Trainin *et al.* [6] note that concurrency faults are related to certain test requirements for the *Blocked-Pair* and *Follows* concurrency coverage metrics, which suggests that achieving high levels of coverage should correlate with testing effectiveness. Wang *et al.* [14] highlight how data races




(a) Code (b) Execution

Figure 1. Concurrent execution example.



DOI: 10.1002/stvr

or atomicity violations may be triggered by satisfying *HaPSet* test requirements. Neither analysis empirically demonstrates the benefits of achieving higher coverage.

The study most similar to the one we present in this paper is by Tasiran *et al.* [20], who evaluate the *location-pair* metric empirically, and compare it to two other coverage metrics (*method-pair* and *def-use*) with respect to the correlation between coverage and fault detection. The study uses two case examples and generates faulty versions via concurrency mutation operators and manual fault seeding. The scope of our study is more comprehensive, encompassing 12 case examples and eight concurrency coverage metrics, and we apply a broader set of analyses.

The work presented in this paper is an extension of work previously published [16]. This work differs in three key ways. First, this work explores additional research questions related to the value of combining complementary concurrency coverage metrics and the impact of difficult-to-cover test requirements. These questions are based on hypothesis presented, but not explored in the previous work [16]. These questions seek to address what directions future work in concurrency coverage test generation should take. Second, we have added three more study objects to the study to broaden our base. Finally, we have conducted our analyses for systems using concurrency mutation operators at a per mutant level, rather than averaging behaviour across all mutants. This allows for a more fine grained analysis and highlights how effectiveness can vary within the same system depending on the specific fault present.

## 3. STUDY DESIGN

The purpose of this study is to rigorously investigate the concurrency coverage metrics presented in previous work and to either provide evidence of each metric's usefulness or demonstrate that the metric is of little value. The usefulness of a coverage metric, concurrency or otherwise, invariably relates to many factors, such as the testing budget available, the characteristics of the program under test, and the goals of the testing process. Nevertheless, to show that any coverage metric can be considered useful, it is necessary at minimum demonstrate two things:

1. Increased levels of coverage correspond to increased fault detection effectiveness;
2. These increases are due in part to increasing coverage levels, not merely larger test suite sizes.

Further, to aid practitioners in selecting a coverage metric for use, we should attempt to quantify the relationship between coverage, size, and fault detection effectiveness. In particular, we are interested in the predictive value of each metric and the expected improvements over random testing in terms of fault detection.

Finally, we are interested in how, given the concurrency coverage metrics proposed, we can best approach test case generation for concurrent systems. Specifically, we wish to know whether potential issues with these metrics, already identified in our previous work [16], can be overcome by a combined use of coverage metrics. We also wish to know whether the current state-of-the-art, coverage-guided test generation techniques for concurrent program testing could be improved by the development of techniques targeting difficult-to-cover test requirements. Such techniques would be analogous to existing methods for improving coverage when using sequential coverage metrics, for example, symbolic execution-based and genetic algorithm-based approaches [9, 21].

Our study is thus designed to address four core questions.

- Research question 1 (RQ1): *For each concurrency coverage metric studied, does the coverage achieved positively impact the effectiveness of the testing process for reasons other than increases in test suite size?* In other words, we would like to provide evidence that given two test suites of equal size, the test suite with higher coverage will generally be more effective.
- Research question 2 (RQ2): *For each concurrency coverage metric studied, how does the fault detection effectiveness of test suites achieving maximum coverage compare with that of random test suites of equal size?* While coverage levels may relate to effectiveness, the practical impact of achieving high coverage for some metric over random test suites may be insignificant.



DOI: 10.1002/stvr

- Research question 3 (RQ3): *For the concurrency coverage metrics studied, do combinations of coverage metrics outperform the original coverage metrics?* The effectiveness of coverage metrics can vary, with the most effective metric varying from case example to case example. By combining metrics, we can potentially overcome these inconsistencies.
- Research question 4 (RQ4): *For each concurrency coverage metric studied, does covering difficult-to-cover test requirements result in above average fault detection relative to other coverage requirements?* For a given case example, some coverage metrics contain test requirements that are hard to cover; that is, a small percentage of possible test cases satisfy the requirement, and thus, achieving maximum coverage in such scenarios can require significant effort. We would like to determine whether such effort is potentially justified.

The objects for this study have been drawn from existing work on concurrent software analysis [22–24] and include objects without faults and objects with faults detected in previous studies. Each object is a multithreaded Java program.

We list the objects with the lines of code, numbers of threads, the type of test oracle for the program, and mutants used in Table I. The *LOC* column represents the size of the original source code for each subject. The *number of threads* column shows how many threads are created during test execution, as determined by the test case given for each object. The *test oracle* column describes the test oracle used for the program. 'AS' means that the fault is detected by an assertion that checks application-specific requirement properties, and the number in the parentheses represents the number of assertion statements in the program. 'TO' means that the fault is detected by a timeout (i.e., deadlock). The *incorrect versions* column represents, for the mutation testing objects, the number of generated mutants and the number of mutants used in parentheses (the reason for the differences in these numbers is explained in Section 3.2.1).

### *3.1. Variables and measures*

*3.1.1. Independent variables.* In this study, we manipulate two independent variables: the concurrency coverage metric and the method of test suite construction.

*3.1.1.1. Concurrency coverage metrics.* Numerous concurrency coverage metrics have been proposed, each based on some intuition about how to capture different aspects of concurrent executions. We view these metrics as having two key properties: the *number of code elements* the test requirements consider (either a single element or a pair of elements), and the the *code construct* the metric is defined over (either synchronization operations or data access operations). For example, the *Blocking* and *Blocked* coverage metrics define test requirements based on individual `synchronized` blocks/methods in a Java program [12] and are thus *singular* concurrency coverage metrics, while

Table I. Study objects.

| Type | Program | LOC | Number of threads | Test oracle | Incorrect versions | Number of test executions |
|---|---|---|---|---|---|---|
| Mutation testing | Arraylist | 5866 | 29 | AS(6), TO | 42 (10) | 2000 |
| | Boundedbuffer | 1437 | 31 | AS(6), TO | 34 (6) | 2000 |
| | Vector | 709 | 51 | AS(15), TO | 88 (35) | 2000 |
| Single-fault program | Accountsubtype | 193 | 12 | AS(1) | 1 | 1000 |
| | Alarmclock | 125 | 4 | AS(1) | 1 | 1000 |
| | Clean | 51 | 3 | TO | 1 | 1000 |
| | Groovy | 433 | 3 | TO | 1 | 1000 |
| | Piper | 71 | 9 | TO | 1 | 1000 |
| | Producerconsumer | 87 | 5 | AS(1) | 1 | 1000 |
| | Stringbuffer | 416 | 3 | AS(19) | 1 | 1000 |
| | Twostage | 52 | 3 | AS(1) | 1 | 1000 |
| | Wronglock | 118 | 22 | TO | 1 | 1000 |



DOI: 10.1002/stvr

the *Blocked-Pair* metric is defined over pairs of blocks and is thus a *pairwise* metric. All of these metrics are defined over `synchronized` blocks, and thus, they are all synchronization metrics [6].

We selected eight coverage metrics for use in our study, focusing on well-known metrics while also ensuring that we considered every possible combination of our two key properties. We list the metrics selected in Table II. We concentrated on metrics that generate modest numbers of test requirements, as this makes achieving high levels of coverage feasible in a reasonable time. Thus, coverage metrics that produce very large numbers of test requirements are not included in this study. These include metrics defined over memory addresses or exhaustive sets of interleavings (e.g., *all-du-path* [7], *ALL*, *SVAR* [5]) and the series of extended coverage metrics proposed by Sherman *et al.* [25]. *Access-pair* [25] and *location-pair* [20] are omitted as they are almost equivalent to the *PSet* metric. We interpret the *LR-Def* metric as generating two test requirements for read accesses: one for the use of memory defined by a local thread and the other for the use of memory defined by any remote thread.

In addition to these metrics, we considered six coverage metrics that are combinations of these metrics to investigate the benefits of combining existing metrics (to address *RQ3*). Each combined metric was created by combining the test requirements of one pairwise synchronization-based coverage metric (i.e., *Blocked-Pair*, *Follows*, and *Sync-Pair*) and the test requirements of one pairwise data access-based coverage metric (i.e., *Def-Use* and *PSet*). Hereafter, we refer to the non-combined metrics as *original coverage metrics*, and the six new coverage metrics as *combined coverage metrics*.

We chose these combinations for three reasons: (i) synchronization-based coverage metrics and data access-based coverage metrics represent different paradigms for measuring concurrency coverage and thus seem likely to be complementary; (ii) metrics within a paradigm tend to achieve similar coverage and fault detection effectiveness rates; and (iii) pairwise metrics generally outperform singular metrics (at least as test case generation targets) and thus make a better starting point when attempting to improve concurrency coverage metrics.

*3.1.1.2. Test suite construction.* We used two methods of test suite construction: random selection and greedy test suite reduction. In random selection, test suites are constructed by randomly selecting test executions to construct test suites of specified sizes. In greedy selection, test suites are constructed to achieve maximum achievable coverage using a small number of test executions. These test suite construction methods are used to address *RQ1* and *RQ2*, respectively.

*3.1.2. Dependent variables.* We measure three dependent variables computed over generated test suites: coverage achieved, test suite size, and fault detection effectiveness. Additionally, we measure two dependent variables computed over test requirements: difficulty of covering test requirements, and the fault detection effectiveness achieved when covering test requirements.

*3.1.2.1. Achieved concurrency coverage of test suites.* For a given metric $M$, each test suite $S$'s coverage is computed as the ratio of $M$'s test requirements that are satisfied by $S$ to the total number of test requirements satisfied across *all* executions for a given program *version*. We construct test

Table II. Concurrency coverage metrics used in the study.

| | Synchronization operation | Data access operation |
|---|---|---|
| Singular | Blocking [12], Blocked [12] | LR-Def [5] |
| Pairwise | Blocked-Pair [6], Follows [6], Sync-Pair [15] | PSet [26], Def-Use [20] |
| Combined | Blocked-Pair + Def-Use, Blocked-Pair + PSet<br>Follows + Def-Use, Follows + PSet<br>Sync-Pair + Def-Use, Sync-Pair + PSet | |



DOI: 10.1002/stvr

executions while holding random test case generation parameters constant (see Section 3.2); because different parameters can result in covering different requirements, the coverage of $M$'s requirements is often less than 100%, and our measurements reflect this. However, for the purpose of greedy test suite construction, we define *maximum achievable coverage* as the number of requirements that can be covered for a specific set of test case generation parameters.

*3.1.2.2. Test suite size.* Test suite size is the number of test cases in the test suite and estimates testing cost.

*3.1.2.3. Fault detection effectiveness of generated test suites.* The fault detection effectiveness of a test suite is 'success' when the fault is detected by at least one execution of a test case in the test suite, or 'failure' when the fault is not detected by any test case execution. During analysis, we typically compute the average fault detection effectiveness across many test suites, with results that range from 0.0 to 1.0.

*3.1.2.4. Difficulty of satisfying test requirements.* The difficulty of satisfying each test requirement is computed as the ratio of the number of test executions satisfying the requirement to the total number of test executions.

*3.1.2.5. Fault detection effectiveness of test requirements.* The fault detection effectiveness of a test requirement is the ratio of the number of test executions detecting a fault while covering the test requirement to the number of test executions that cover the test requirement.

### *3.2. Experiment setup*

Conducting our experiment requires us to

1. generate mutants for programs without faults;
2. conduct a large number of random test executions;
3. for each execution, record the requirements covered for all metrics and whether a fault is detected;
4. compute the difficulty and fault detection rate for each requirement generated;
5. perform resampling over executions to construct test suites; and
6. measure the resulting coverage and fault detection effectiveness of each test suite.

*3.2.1. Mutant generation.* We wished to study fault detection in the presence of many diverse fault types, which is not possible when using single-fault programs. Thus, for several of our object programs, we corrected known faults [23] and applied mutation analysis. To choose mutation operators for our study, we drew on concurrency mutation operators used in a recent survey on concurrency mutation testing [27]. These operators are similar to traditional syntactic mutation operators commonly used in other studies [17, 28] but focus on manipulating synchronization constructs, for example, adding and removing synchronization primitives. Table III describes the operators. We applied these operators to generate mutants. We then discarded any mutants that (i) did not fail for any generated test execution, (ii) were malformed, for example, resulted in code that could not be executed, or (iii) were killed by every test execution.

We list the number of mutants generated together with the final number of mutants used within parentheses in Table I. Note that we also use objects containing real faults, thus mitigating the risk present when using concurrency mutation operators, whose usage is less established and studied than structural mutation operators for sequential programs [17]. Hereafter, when referring to 'objects', we are referring to individual faulty programs; for example, 'all objects' refers to all single-fault programs and all mutants.

*3.2.2. Test generation and execution.* We used a randomized test case generation approach to avoid bias that might result from using a directed test generation approach such as those proposed in



DOI: 10.1002/stvr

Table III. Mutation operators.

| Category | Description |
|---|---|
| Change synchronization operations | Exchange synchronized block parameter<br>Remove `wait()`<br>Replace `notifyAll()` with `notify()` |
| Modify synchronized block | Expand synchronized Block<br>Remove synchronized block<br>Remove `synchronized` keyword from method<br>Shift synchronized block<br>Shrink synchronized block<br>Split synchronized block |

[12, 29]. Our approach selects an arbitrary test input and generates a large number of test executions by executing a target program on the test input with varying random delays (i.e., calls to `sleep()`) inserted at shared resource accesses and synchronization operations.

We control two parameters of this approach: the *probability* that a delay will be inserted at each shared resource access or synchronization operation (0.1, 0.2, 0.3, and 0.4), and the maximum length of the *delay* to be inserted (5, 10, and 15 ms). We used these controls because prior work indicates that they can impact the effectiveness of the testing process [13]. The specific values used were selected based on our previous experience in this domain [15] and pilot studies, both of which indicated that larger or finer grained delays and probabilities did not yield significantly different results. In addition to the 12 random scheduling techniques, we ran test executions without inserting any delay noise.

We began by estimating the number of test executions $E$ required to achieve maximum coverage for all eight coverage metrics used, and each of the six combined metrics considered. This was performed by executing the original object for several hours and recording the rate of coverage increase for each metric. For each object, we required either 1000 or 2000 test executions. Following this, for each parameter setting ($13(= 4 \times 3 + 1)$ in total), we conducted $E$ executions for each mutant (for objects with mutants) or each object program (for objects without mutants). During each execution, we recorded (i) the test requirements covered for each coverage metric studied and (ii) whether a fault was detected.

We recorded an execution as detecting a fault if (i) an application-specific assertion statement is not satisfied (i.e., invariant violations), (iii) a crash occurs that throws an uncaught exception (e.g., null pointer dereference, array index out-of-bound, and invalid memory access), or (iii) the program deadlocks, determined by checking whether execution time is exceptionally long.

*3.2.3. Data collection.* After each test execution, we know (i) which test requirements are covered for each coverage metric and (ii) whether the program failed. Based on this information, we can obtain the data for each test requirement — how frequently the test requirement is covered, and how frequently executions that cover the test requirement detect a fault. These data are used for analysis related to *RQ4*.

Using the test execution information, we can, via random resampling, construct test suites of varying sizes and levels of coverage. Ideally, we would like to construct test suites encompassing all possible combinations of size and coverage. Unfortunately, as coverage and size tend to be highly correlated, this is impossible; small test suites with high coverage (or vice-versa) are extremely rare in practice. We instead generated, for each combination of object and coverage metric, 90 000 test suites ranging in size (i.e., number of test executions) from 1 to the maximum size via random sampling of executions. This results in a set of test suites with increasing size and, within each level of size, varying coverage. These test suites are used to help address *RQ1*, *RQ2*, and *RQ3*.

We also generated 100 test suites achieving maximum achievable coverage for each coverage metric. We generated these using a *mostly* greedy test suite reduction approach: from the set of executions, repeatedly select either (i) the test execution satisfying the most unsatisfied requirements (80% chance) or (ii) a random test execution (20% chance) until all requirements are satisfied. This



DOI: 10.1002/stvr

results in a test suite that achieves maximum coverage using fewer test executions that are required by simple random test suite construction. The randomization adds noise, ensuring some variation in the generated suites. These test suites are used to address *RQ2*. To investigate *RQ3*, we apply the same test construction for the six combined coverage metrics as well.

To select a test suite for a single-fault program or mutant, we have one set of executions over the object, and we resample from this set to construct test suites. Each test suite becomes a data point for analysis, having an associated level of coverage, size, and fault detection result (killed/not killed). When constructing each test suite, we held probability and delay constant. This was performed to facilitate later analysis considering the impact of these factors.

Note that the generation process for the original eight metrics and the six combined metrics is the same. We treat a combined metric (e.g., *Follows + PSet*) as a single metric, with its own separate set of coverage requirements, a separate sets of greedy test suites, and so on. This allows for a fair comparison of the original and combined metrics in Section 4.

### *3.3. Threats to validity*

*3.3.1. External.* We conducted our study using only Java programs with standard synchronization operations. These programs are relatively small but have been chosen from existing work in this area, and thus, we believe that our results are at least generalizable to the class of programs that concurrent program testing research focuses on.

For concurrency coverage metrics, it is difficult to accurately determine satisfiable test requirements. For all coverage metrics, however, we appear to have reached saturation during test case generation (see Section 4.1) [25], and thus, a larger number of executions is unlikely to significantly alter our results.

The randomized test generation technique we use was implemented in-house, but we have attempted to match the behaviour of other random testing techniques by constructing a general technique and varying the parameters of probability and delay.

We follow the current practice of concurrency testing research that focuses on analyzing diverse thread interleavings effectively and efficiently by restricting other factors such as test inputs. Thus, our study utilizes various thread schedulings with single test input values, which may not consider the impact of various test input values on concurrent program testing.

*3.3.2. Internal.* Our randomized test case generation technique is implemented on top of Java's internal thread scheduler. When using other algorithmic thread schedulers, such as Probabilistic Concurrency Testing (PCT) [30, 31] or CTrigger [32], results may vary. Additionally, while we have extensively tested our experimentation tools, it is possible that faults in our tools could lead to incorrect conclusions.

*3.3.3. Construct.* Our method of detecting faults may miss faults, for example, errors not captured by an assertion violation or not leading to an exception. In practice, however, much of concurrent testing focuses on detecting faults via imperfect test oracles, and thus, our study uses a realistic approach to fault detection.

We measured the maximum coverage for a metric by tracking all coverage requirements covered in any execution during test generation. This value is likely lower than the actual maximum achievable coverage because there likely exist coverage requirements that are achievable but not covered by any generated execution. Nonetheless, because we generate a large number of executions with different random testing techniques, we expect that missed coverage requirements are few. Furthermore, even if the maximum coverage values are incorrect, only *RQ3* depends on this value, and thus, other conclusions drawn would not change. We did not use a predictive analysis technique for the study because the existing predictive analysis techniques are known to produce false positives (i.e., infeasible test requirements are estimated as feasible).

We used mutation analysis to measure testing effectiveness for some objects. Our seeded faults are designed to mimic actual concurrency faults and, of course, are indeed faults, but the relationship between faults generated by concurrency mutation operators and real concurrency faults has not



DOI: 10.1002/stvr

been thoroughly investigated. Nevertheless, the results for mutation-based objects and objects with real faults are similar.

*3.3.4. Conclusion.* For each object, we constructed from 1 to 88 faults and 100 000 test suites per coverage metric. While more mutants, faults, and test suites could in theory alter our conclusions, in practice, our conclusions remain the same for both single-fault programs, mutation testing-driven programs, and larger numbers of test suites.

## 4. RESULT AND ANALYSIS

Our analyses are designed to study how each coverage metric impacts fault detection effectiveness.

Towards *RQ1*, we visualized the pairwise relationship between variables, measured the correlation between coverage, size, and fault detection effectiveness, and performed linear regression to better understand how both coverage and size contribute to fault detection effectiveness.

Towards *RQ2*, we compared the fault detection effectiveness of test suites satisfying maximum achievable coverage and random test suites of equal size. Towards *RQ3*, we performed the analysis discussed earlier over combinations of pairwise metrics and compared the results with the single-metric versions. Finally, towards *RQ4*, we examined the correlation between the difficulty of covering a test requirement and the average fault detection for test executions covering a test requirement, and we compared the average fault detection for difficult-to-cover with the fault detection for easy-to-cover test requirements.

Ideally, we would like a coverage metric that (i) is highly correlated with fault detection (over 0.7 coverage); (ii) along with size, results in regression models with high fit for fault detection (higher than 0.8); and (iii) allows us to select test suites with significantly higher fault detection than randomly selected test suites of equal size (improvements in fault detection of at least 20%). Any metric fitting such criteria would be useful both as a predictor of fault detection effectiveness and as a test generation target.

### *4.1. Visualization*

To understand the relationship between test suite size, coverage, and fault detection effectiveness, we began by plotting the relationship between each pair of variables. In Figure 2, we show the relationship between size and coverage for each coverage metric, for four single-fault objects (Figure A.1 for all single-fault objects). In Figure 3, we show the same relationship for objects using mutation testing. In Figure 4, we show the relationship between coverage and fault detection for four single-fault objects (Figure A.2 for all single-fault objects). In Figure 5, we show the same relationship for objects using mutation testing. Finally, in Figure 6, we show the relationship between size and fault detection for all objects. Note that expanded versions of Figures 2 and 4 are found in Section 6. To ease readability, we have elected to show only specifically referenced objects here.

Recall from Section 3.2.2 that for each combination of probability and delay (two variables controlled during test generation), 1000 test executions were generated for each single-fault program. Each figure is an average across these traces of the test executions. Additionally, rather than plot a separate figure for each of the dozens of mutants for the *Arraylist*, *Boundedbuffer*, and *Vector* objects, figures for these objects are averages across all mutants. Note that this averaging results in figures that do not necessarily reflect the underlying trends within each mutant, as we discuss later in this section.

In all of the figures, there is typically a fair amount of variation along the y-axis as coverage and size increase. To improve the readability of the figures, we have used two forms of smoothing. In the case of plots of size versus coverage and size versus fault detection, we have used Local regression (LOESS) smoothing with a factor of 0.1. The relationships here are clearly visible with raw plots; the use of mild smoothing allows us to distinguish coverage metrics and objects after plotting. However, plots of coverage versus fault detection are very noisy, as indicated by the correlations shown in Section 4.2. LOESS smoothing is of limited help here, and so to further improve readability,



DOI: 10.1002/stvr

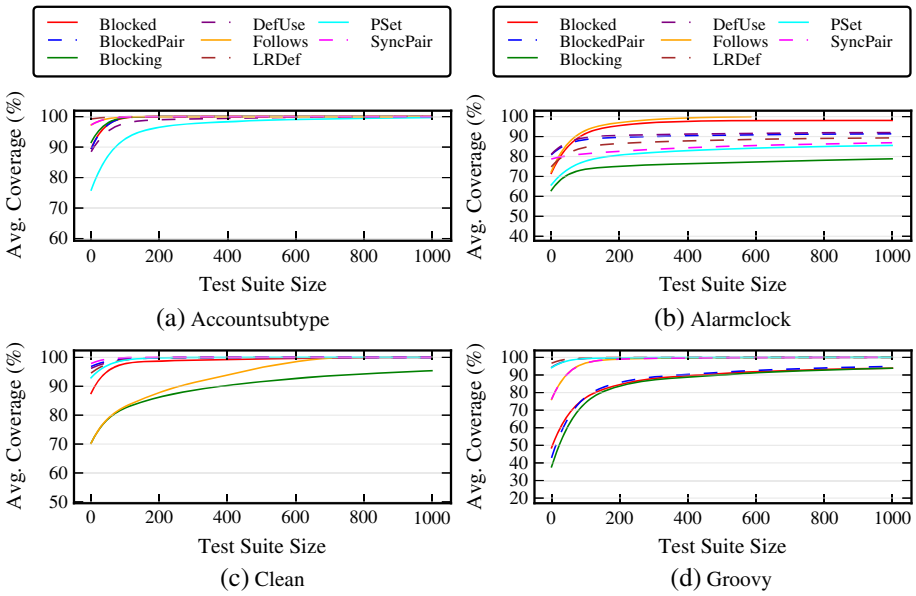



Figure 2. Size versus coverage, four single-fault objects.

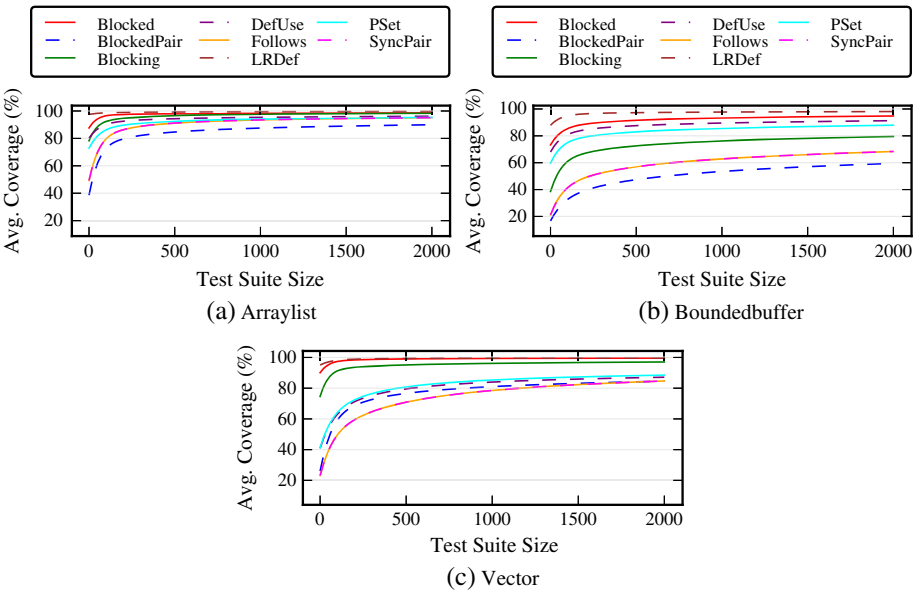



Figure 3. Size versus coverage, mutation objects.

before plotting, we have averaged the fault detection rates for all coverage levels within five percentage points; that is, we have averaged the fault detection rate for test suites achieving 12.5–17.5% coverage, 17.5–22.5% coverage, and so on.

This averaging across mutants and test generation parameters results in graphs that must be carefully interpreted: individual points on the lines can reflect the average of many test suites — particularly for coverage levels above 50% — or few test suites, as very low coverage levels are infrequently achieved in practice. This is unfortunate, but necessary, as the alternative is to plot each combination of coverage metric and object separately, which would require hundreds of figures, or as very dense scatterplots, resulting in unintelligible figures. However, the goal of visualization is just to spot broad trends; rigorous analysis follows in the remaining sections.



DOI: 10.1002/stvr

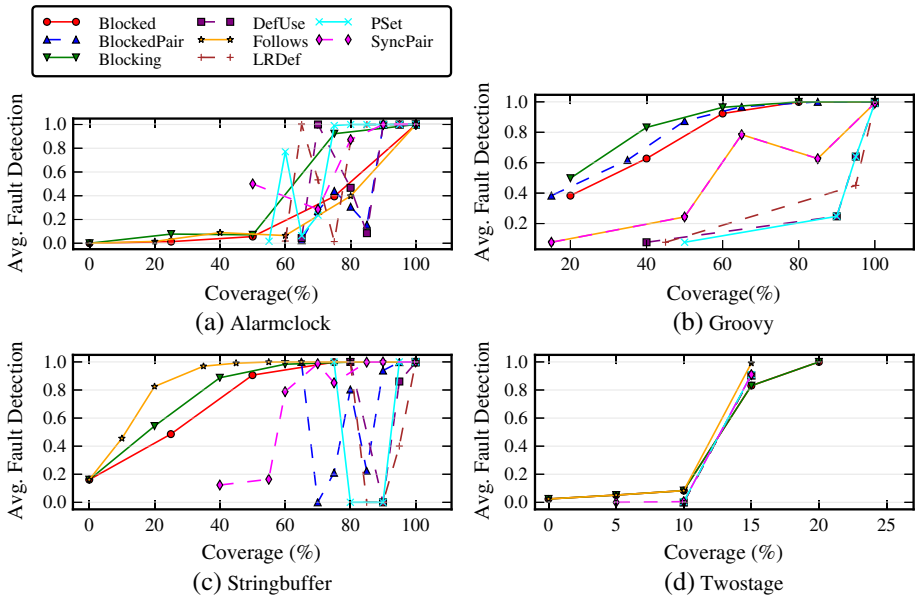



Figure 4. Coverage versus fault detection effectiveness, four single-fault objects.

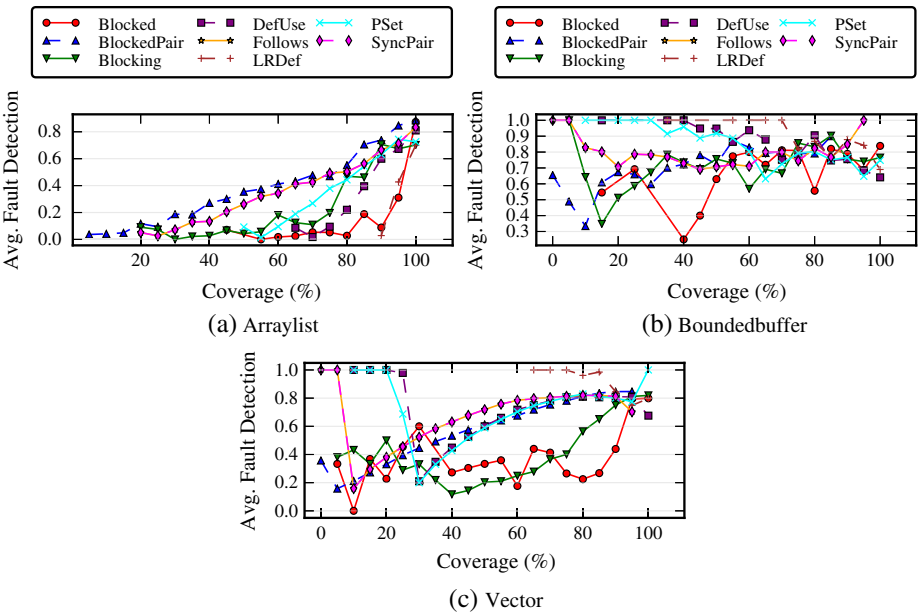



Figure 5. Coverage versus fault detection effectiveness, mutation objects.

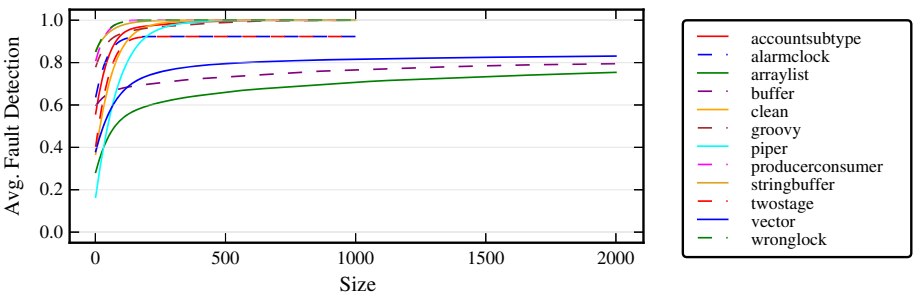



Figure 6. Size versus fault detection effectiveness, all objects.



DOI: 10.1002/stvr

Note that in several cases, coverage achieved is less than 100%. This occurs because each test suite is specific to a single combination of test generation parameters, but the set of test requirements (and thus, the mark for 100% achievable coverage) is computed across all test suites. Thus, it is possible that no single test suite achieves 100% maximum achievable coverage. Similar behaviour is shown in Figure 6, as several test suites of maximum size fail to detect the fault.

We begin by examining the relationship between size and coverage/fault detection, as shown in Figures 2, 3, and 6. We can see that the concurrency coverage metrics often — but clearly not always — exhibit behaviour similar to what we expect from sequential coverage metrics and testing: broadly logarithmic behaviour, with a rapid increase in both fault detection and coverage for small test suite sizes, and smaller increases as test suite size increases. Here, we see small differences in coverage metrics: some coverage metrics begin with very high levels of coverage for even small test suites and thus quickly achieve close to maximum coverage, while others grow in coverage more slowly. For example, *LR-Def* is an extreme case, achieving maximum coverage almost immediately for many programs. In contrast, *Follows*, a more complex metric, often achieves maximum coverage only with larger test suites sizes, that is, those greater than 300. Here, differences are related primarily to the number of 'easy' requirements to satisfy — those metrics that are easier to satisfy have high coverage even for very small test suites, for example, *Blocking*, *Blocked*, and *LR-Def*. Similar variations are also visible in the relationship between size and fault detection (see Figure 6). On the whole, however, the relationship between size and coverage/fault detection is clearly positive.

Less easily inferred from the figures is the relationship between coverage and fault detection (Figures 4 and 5). Clearly, in many cases, the relationship is positive; for example, this is true for all metrics when applied to the *Twostage* and *Arraylist* objects. In other cases, the relationship is noisy, but nevertheless, high coverage appears to result in high fault detection, for example, on the *Alarmclock* object. In some cases, however, the relationship is quite unclear. *Boundedbuffer*, for example, exhibits no clear pattern for any coverage criteria (except when testing one specific mutant, as we discuss later), whereas *Blocked-Pair* coverage varies from seeming clearly related to fault detection (e.g., for the *Groovy* and *Vector* objects) to seeming marginally related to fault detection (e.g., for the *Alarmclock* and *Stringbuffer* objects).

This clear positive relationship between size and fault detection, coupled with the inconsistent, but nevertheless positive relationship between coverage and fault detection, provides informal evidence that both size and coverage impact fault detection effectiveness. We quantify the impact of both factors in the following subsections.

### *4.2. Correlation between variables*

The foregoing visualizations indicate that both test suite size and coverage appear to be positively correlated with fault detection effectiveness and that size is positively correlated with coverage. To measure the strength of these relationships, for each object and coverage metric, we measured the correlation between each variable using Pearson's $r$.[‡] We selected Pearson's $r$ for two reasons. First, we are interested in the application of concurrency coverage metrics as predictors, and thus, measuring the strength of the linear relationship between variables is desirable. Fault detection is guaranteed to increase monotonically with size and coverage, and thus, establishing this using rank correlation (e.g., Spearman or Kendall's tau) yields less new information [33]. Second, single-fault programs can only fail or pass for each test suite; computing correlation over such data is a special case known as point-biserial correlation, for which rank correlation (due to the many ties present) is unsuitable. For every non-zero correlation computed, the $p$-value was (far) less than 0.05 and thus statistically significant at $\alpha = 0.05$.

The computed correlations for single-fault programs are presented in Table IV. For example, for *Accountsubtype*, the correlation between *Blocked* coverage and fault detection/test suite size is 0.39 and 0.11, respectively, while the correlation between size and fault detection (*S-FF*) is 0.22, indicating that coverage is more highly correlated with fault detection than test suite size.

[‡]For small samples, conclusions based on Pearson's can be unsound for non-normal data; in our case, the use of very large number of samples, 30 000–90 000 per correlation computed, mitigates this risk.



DOI: 10.1002/stvr

Table IV. Correlations over coverage metrics.

| | Blocked | Blocked-Pair | Blocking | Def-Use | S-FF |
|---|---|---|---|---|---|
| Accountsubtype | 0.39, 0.11 | 0.39, 0.11 | 0.35, 0.10 | 0.60, 0.28 | 0.22 |
| Alarmclock | 0.77, 0.25 | 0.52, 0.24 | 0.27, 0.23 | 0.56, 0.22 | 0.05 |
| Clean | 0.16, 0.16 | 0.73, 0.23 | 0.19, 0.40 | 0.96, 0.29 | 0.30 |
| Groovy | 0.46, 0.36 | 0.50, 0.37 | 0.45, 0.37 | 0.45, 0.16 | 0.17 |
| Piper | 0.0, 0.0 | 0.62, 0.45 | 0.48, 0.25 | 0.07, 0.03 | 0.38 |
| Producerconsumer | 0.14, 0.03 | 0.17, 0.21 | 0.14, 0.16 | 0.57, 0.15 | 0.12 |
| Stringbuffer | 0.58, 0.18 | 0.67, 0.23 | 0.59, 0.31 | 0.43, 0.12 | 0.13 |
| Twostage | 0.88, 0.23 | 0.94, 0.13 | 0.88, 0.23 | 0.92, 0.13 | 0.10 |
| Wronglock | 0.12, 0.01 | 0.12, 0.01 | 0.12, 0.01 | 0.53, 0.13 | 0.11 |
| | Follows | LR-Def | PSet | Sync-Pair | S-FF |
| Accountsubtype | 0.28, 0.09 | 0.30, 0.12 | 0.57, 0.42 | 0.28, 0.09 | 0.22 |
| Alarmclock | 0.66, 0.29 | 0.59, 0.30 | 0.59, 0.35 | 0.19, 0.26 | 0.05 |
| Clean | 0.17, 0.42 | 0.91, 0.30 | 0.83, 0.28 | 0.09, 0.05 | 0.30 |
| Groovy | 0.52, 0.24 | 0.30, 0.09 | 0.48, 0.18 | 0.52, 0.24 | 0.17 |
| Piper | 0.59, 0.49 | 0.66, 0.27 | 0.67, 0.27 | 0.62, 0.45 | 0.38 |
| Producerconsumer | 0.21, 0.43 | 0.46, 0.26 | 0.30, 0.26 | 0.11, 0.20 | 0.12 |
| Stringbuffer | 0.44, 0.35 | 0.74, 0.14 | 0.87, 0.15 | 0.66, 0.23 | 0.13 |
| Twostage | 0.88, 0.23 | 0.95, 0.13 | 0.96, 0.13 | 0.96, 0.13 | 0.10 |
| Wronglock | 0.0, 0.0 | 0.50, 0.15 | 0.58, 0.21 | 0.0, 0.0 | 0.11 |

Each cell contains coverage and fault detection effectiveness correlation, and size and coverage correlation. S-FF denotes size and fault detection effectiveness correlation.

The correlations for objects with multiple faulty versions are shown as boxplots in Figure 7.[§] The column labeled *X-FF* represents the correlation between the coverage *X* and fault detection, and the column with *X-SZ* represents the correlation between the coverage *X* and the test suite size. The last column labeled *S-FF* is the correlation between test suite size and fault detection.

For example, we can see for *Arraylist* that the correlation between size and fault detection (column labeled 'S-FF') ranges from 0.4 to slightly less than 0.2, with a median slightly under 0.2 and a mean of 0.2. In contrast, the correlation between each coverage metric and fault detection tends to be higher, with means and medians ranging from roughly 0.3 for *Blocking* coverage to roughly 0.7 for *Blocked* coverage. Additionally, several outliers, both above and below the mean, can be seen; for example, in the near perfect correlation of *Blocked* coverage and fault detection for one mutant, and the very low (and sometimes even negative) correlations exhibited for a handful of combinations of coverage and mutant scenarios.

For each metric, there exists at least one single-fault object for which the correlation with fault detection is at or above 0.88. Further, even when coverage weakly correlates with fault detection, this correlation is often higher than the correlation of fault detection and size (*S-FF*). These results provide evidence that each metric is a useful predictor of concurrency testing effectiveness, depending on program.

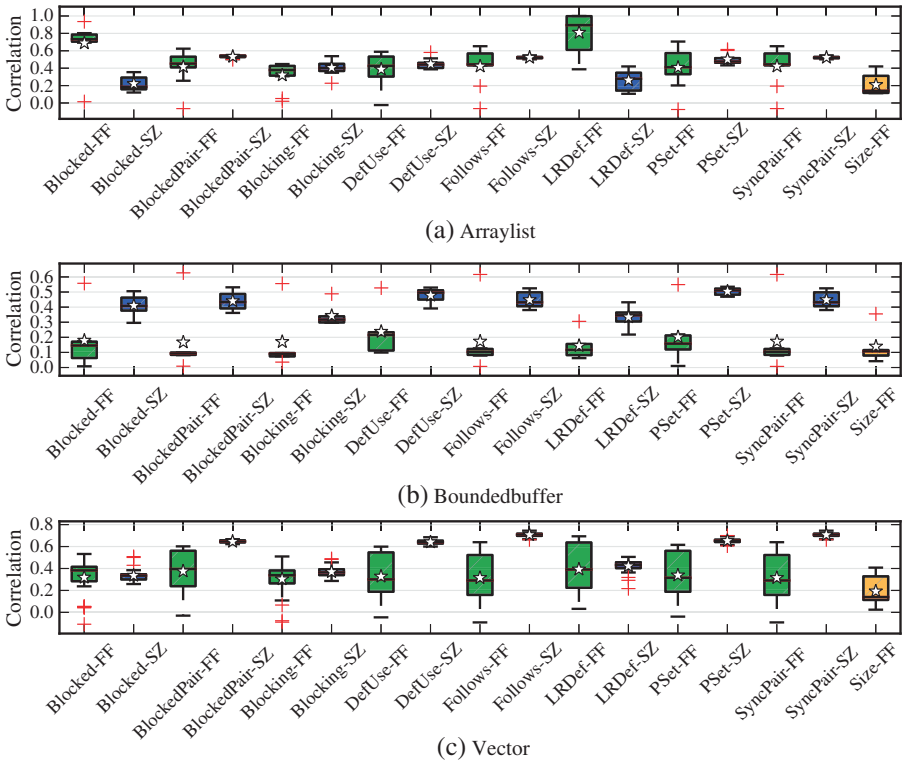
The best metric, however, varies across programs, and no single metric is a consistent predictor of effectiveness, although *PSet* is often quite strong. For the single-fault programs, *PSet* shows the highest correlation for four programs among nine single-fault programs in total, and *PSet* always shows high or moderate correlations except in the case of *Boundedbuffer*. Although *PSet* has a low average/median of 0.2 (*Boundedbuffer*), *PSet* has a better correlation than other coverage metrics.

The reason for this variation is unclear, but we believe that this occurs because the metric's intuition does not always capture the single fault present. This is supported by the results shown in Figure 7, where we see a wide variation even within program depending on the mutant used. For example, for the *Vector* program, the relationship between coverage and fault detection varies

[§]For each boxplot, the mean is shown as a star, the boxplot whiskers represent data within the 1.5 times the interquartile range, and the outliers are shown as red '+' marks. This convention is maintained for boxplots shown in future sections.



DOI: 10.1002/stvr




(a) Arraylist




(b) Boundedbuffer




(c) Vector

Figure 7. Correlations across mutants, mutation objects. FF = fault detection, SZ = test suite size.

strongly for several metrics, for example, *Def-Use*, which varies from exhibiting a negligible relationship to a moderately strong relationship depending on the mutant used. This contrasts strongly with the very consistent relationships between coverage and size for most metrics when applied to all of *Vector*'s mutants.

In any case, the variation in the best metric for a given object indicates that selecting an effective metric may be challenging. Additionally, the occasional low and often moderate correlation between coverage and fault detection (and somewhat surprisingly, size and fault detection) hints that factors other than those captured by the concurrency coverage metrics may relate to fault detection effectiveness. We discuss this further in Section 5.2.

### *4.3. Models of effectiveness*

Based on the previous two analyses, we can see that for every metric, coverage levels do correspond (somewhat) to testing effectiveness. However, we also see that test suite size and coverage are often similarly correlated, and thus, the relationship between size, coverage, and fault detection is unclear. It is possible that, in fact, coverage and size are not very independent of each other in terms of their effect on fault detection; for example, depending on the case example, either coverage or size alone may be a sufficient exploratory variable for fault detection.

Does coverage predict fault detection effectiveness or merely reflect test suite size? And to what extent (if any) does considering coverage improvement increase the ability to predict fault detection? To address these questions, we used linear regression to attempt to model how test suite size and coverage jointly influence the effectiveness of the testing process, with the goal of determining whether coverage has an independent explanatory ability with respect to fault detection.

In linear regression, we model the data as a linear equation $y = \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_p x_p + \varepsilon_i$ where variables $x_i$ correspond to explanatory factors and variable $y$ denotes the dependent variable. After modelling the data, the coefficient of determination $R^2$ is produced. $R^2$ indicates how well the data fit the model, and can be interpreted as the proportion of variability explained by the



DOI: 10.1002/stvr

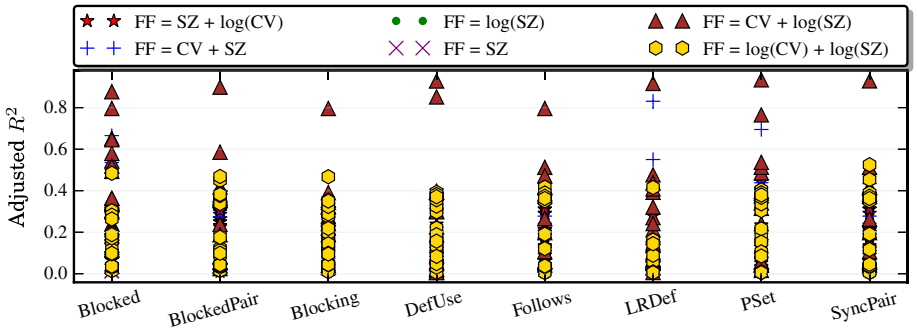



Figure 8. Adjusted $R^2$ for every best fit model, all combinations of objects and coverage metrics. FF = fault detection, SZ = test suite size, CV = percent coverage.

model; for example, a fit of 0.6, which indicates about 60% of the variation, can be explained by the explanatory variables. In many cases, the goal of linear regression is *model selection*: from a set of candidate models, select the model that offers the highest *goodness of fit*, while omitting unneeded explanatory variables.

In our work, we will focus largely upon the *adjusted* $R^2$. Adjusted $R^2$ is a measure of fitness that adjusts for the number of explanatory variables. When comparing two models, a model with more explanatory variables will have a higher adjusted $R^2$ only when additional variables significantly contribute.[¶] Strictly speaking, adjusted $R^2$ cannot be used to indicate the proportion of variance captured, but as adjusted $R^2$ is always less than or equal to $R^2$, we can infer that the proportion of variance captured by a model is equal to or greater than that given by adjusted $R^2$. Thus, if for some model an adjusted $R^2$ of 0.6 is produced, this indicates that the model explains at least 60% of the variation in fault detection.

In this case, we would like to model fault detection effectiveness for each object and coverage metric using test suite size (*SZ*) and/or coverage level (*CV*) as explanatory variables. If the best models always employ coverage levels as an explanatory factor, this indicates that coverage has an independent ability to predict fault detection effectiveness. Accordingly, for every combination of object and coverage metric where coverage varies, we fit all possible linear models employing combinations of *SZ*, *log(SZ)*, *CV*, and *log(CV)* as explanatory variables (with fault detection (*FF*) as the dependent variable). Note that the use of *log* does not necessarily indicate that a factor is less important (in terms of fit) than a factor linearly related, but it indicates that the relationship is logarithmic.

Our fitting process results in over 10 000 regression models, and thus, listing regression models with computed coefficients is infeasible; additionally, we are interested in exploring how well size and coverage levels model fault detection effectiveness, not the specific models. To summarize our data, we began by selecting the best fitting model for each object/coverage metric pair. We plot the associated adjusted $R^2$ in Figure 8 for each coverage metric, across all objects, indicating which set of explanatory variables had the highest fit. For example, we see that for the *Def-Use* metric, for two objects, adjusted $R^2$ was greater than 0.8, indicating high fit with model $FF = \alpha \times CV + \beta \times log(SZ)$, while on all other objects, fit was under 0.4, suggesting a low to moderate fit. Here, we can clearly see the variation in metric effectiveness, with fits ranging from less than 0.2 to over 0.8, indicating a wide variation in predictive power. However, for all coverage metrics, for at least one object, an adjusted $R^2$ of 0.8 or above was observed, indicating high fit, and for many objects, fits above 0.4 were observed, indicating moderate fit.

Following this, we wished to measure the degree to which coverage improves the model fit; that is, how much does adding coverage as a dependent variable improve the fit as compared with

[¶]We also used Mallow's $C_p$ to determine goodness of fit [34]. The results when using Mallow's led to the same conclusions, and we have presented the results using adjusted $R^2$ as we believe that this metric is easier to interpret.



DOI: 10.1002/stvr

models using size alone? To answer this question, we computed minimum and maximum relative improvement in adjusted $R^2$ when using models with two dependent variables over models using size alone as a dependent variable. We list the results in Table V for single-fault objects and plot the results in Figure 9 for mutation objects. In the plots, the columns *MN* and *MX* represent the minimum and the maximum relative increase in adjusted $R^2$ when using two dependent variables for the corresponding object. An *NA* denotes that the improvement cannot be computed, as the linear regression's adjusted $R^2$ is 0.0 (resulting in infinite improvement).

As shown in Table V, in many cases, adjusted $R^2$ greatly improved with the addition of coverage to the regression models. In several instances, for example, when applying nearly every coverage metric to the *Stringbuffer* object, we see improvements over 100%, indicating a more than double increase in adjusted $R^2$. In the case of mutation objects, we see less consistency, with *Arraylist* exhibiting small improvements (less than 10% increases), and *Vector* exhibiting a mix of small to moderates increases ranging from under 5% up to 30% (see Figure 9).

In some cases, however, the improvement found in using coverage as part of the regression model is small, indicating that test suite size is the main component of effective testing. For example, *Blocked* coverage applied to the *Clean* object yields a maximum improvement of only 0.4%, and for the *Boundedbuffer* object (Figure 9), we see several instances where the relative change in adjusted $R^2$ is negative, indicating that the addition of coverage to the model provides no statistically significant improvement to the predictive power of the model.

Based on these analyses, we can see that while no single set of explanatory variables is best, much of the time models based on both coverage and size are preferable to models using only one explanatory variable. Indeed, in several cases, the addition of coverage to the model improves the model fit many times over. This provides evidence that coverage metrics have a predictive ability separate from test suite size. Nevertheless, the adjusted $R^2$ is generally less than 0.8, indicating that while our models do have reasonable predictive power, a significant proportion of variability is not accounted for by the models. Furthermore, in some cases, coverage provides little or no predictive power, leaving test suite size as the sole (and often also weak, per Section 4.2) predictor of testing effectiveness. We discuss this further in Section 5.2.

Table V. Minimum and maximum relative increase in adjusted $R^2$ when using two dependent variables.

| | Blocked | Blocked-Pair | Blocking | Def-Use |
|---|---|---|---|---|
| Accountsubtype | 0.0%, 45.8% | 0.0%, 44.7% | 0.0%, 34.0% | 121.9%, 134.3% |
| Alarmclock | 3293.5%, 3858.0% | 1591.1%, 1767.3% | 351.2%, 483.0% | 1847.4%, 2008.1% |
| Clean | 0.0%, 0.4% | 67.0%, 122.9% | 0.0%, 0.5% | 244.6%, 253.7% |
| Groovy | 198.5%, 313.9% | 241.8%, 355.4% | 182.8%, 280.5% | 131.5%, 209.3% |
| Piper | NA | 16.5%, 30.1% | 0.0%, 13.0% | NA |
| Producerconsumer | 0.0%, 10.4% | 0.0%, 6.9% | 0.0%, 6.5% | NA |
| Stringbuffer | 369.1%, 562.9% | 518.9%, 542.0% | 386.1%, 540.3% | NA |
| Twostage | 1384.0%, 1497.6% | 1624.1%, 1703.9% | 1384.0%, 1497.6% | 1511.5%, 1609.3% |
| Wronglock | 0.0%, 14.3% | 0.0%, 14.3% | 0.0%, 14.3% | 223.5%, 245.1% |
| | Follows | LR-Def | PSet | Sync-Pair |
| Accountsubtype | 0.0%, 20.2% | NA | 104.4%, 116.5% | 0.0%, 20.2% |
| Alarmclock | 2576.8%, 2791.3% | 2170.2%, 2446.5% | 2211.9%, 2621.7% | 138.1%, 179.3% |
| Clean | 0.0%, 1.0% | 199.8%, 216.5% | 142.0%, 164.7% | 0.0%, 0.1% |
| Groovy | 257.7%, 279.2% | 27.0%, 85.7% | 169.2%, 228.0% | 257.7%, 279.2% |
| Piper | 6.3%, 20.5% | NA | 43.6%, 55.3% | 16.5%, 31.1% |
| Producerconsumer | 0.0%, 5.2% | NA | 0.0%, 32.5% | 0.0%, 1.6% |
| Stringbuffer | 166.2%, 296.9% | 624.7%, 653.9% | 927.4%, 948.7% | 514.3%, 619.2% |
| Twostage | 1384.0%, 1497.6% | 1688.0%, 1740.2% | 1724.8%, 1774.8% | 1627.3%, 1764.8% |
| Wronglock | NA | NA | 289.3%, 294.2% | NA |



DOI: 10.1002/stvr

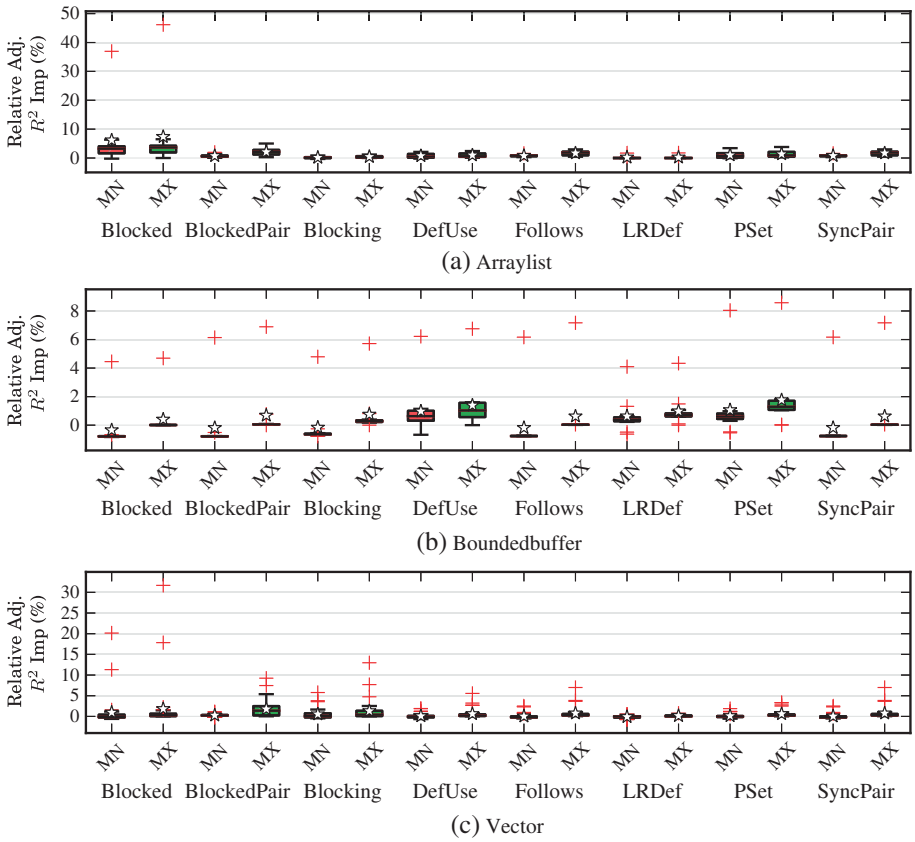



Figure 9. Minimum and maximum relative increase in adjusted $R^2$ when using two dependent variables, mutation objects. MN = minimum, MX = maximum.

### *4.4. Effectiveness of maximum coverage*

Our first three analyses have characterized the relationship between test suite size, coverage, and fault detection effectiveness and statistically established that for each metric, coverage level has a predictive ability for fault detection apart from that of test suite size. From these results, we can see that while not every coverage metric is highly effective for all programs, all coverage metrics do appear to have value. Thus, it is worthwhile to use concurrency coverage metrics (in addition to test suite size) as methods for estimating the concurrency fault detection effectiveness of a testing process.

Per *RQ2*, however, we also would like to quantify the ability of test suites to quickly achieve high levels of concurrency coverage. To do this, for each program and coverage metric, we compared test suites of maximum achievable coverage, generated using a greedy algorithm described in Section 3.2.3, against random test suites of equal size. Our expectation is that if a metric is a reasonable target for test case generation, holding the method of test case generation constant while reducing generated test executions to construct small, high coverage test suites should result in more effective test suites than pure random test case generation.

We began by formulating hypothesis $H$: test suites satisfying maximum achievable coverage will outperform random test suites of equal size in terms of fault detection. We evaluated $H$ for each combination of program and coverage metric using a two-tailed bootstrapped paired permutation test, a non-parametric statistical test that calculates the probability $p$ that two paired sets of data come from the same population [33]. The null hypothesis $H_0$ is that test suites achieving maximum achievable coverage are equally as effective as random test suites of equal size.

For each combination of coverage metric and object (per mutant for mutation objects), there are 100 test suites generated to achieve maximum achievable coverage (hereafter referred to as



DOI: 10.1002/stvr

maximum coverage) (see Section 3.2.3). Each test suite was paired with a randomly selected test suite of equal size. Following this, the permutation test was applied using 250 000 permutations for each $p$-value [33]. Following the test, we computed the average fault detection when using test suites reduced to achieve maximum coverage, the average relative improvement in coverage over random test suites, and the average fault detection for the random test suites.

Table VI lists the results of this analysis for objects with only a single fault. (Note that fault detection is the ratio of test suites detecting the fault to the total number of test suites.) Figure 10 plots the fault detection for greedily reduced test suites and random test suites of equal size across mutants as a boxplot. The column *MFF* represents the fault detection for the reduced test suites for each object and coverage metric studied, and the column *RFF* represents fault detection for random test suites of equal size. Figure 11 plots the relative increase in coverage when using greedy reduced tests suites over randomly generated test suites of equal size.

Our analysis results imply that achieving high coverage generally yields not only statistically significant but also practically significant increases in fault detection: large, often twofold or more

Table VI. Maximum achievable coverage test suite statistics.

| | Blocked | | | | Blocked-Pair | | | | Blocking | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | MFF | RFF | Cv | Sz | MFF | RFF | Cv | Sz | MFF | RFF | Cv | Sz |
| Accountsubtype | 0.19 | 0.06 | 31.9% | 2.06 | 0.14 | 0.04 | 35.0% | 2.16 | 0.09* | 0.05* | 29.5% | 2.00 |
| Alarmclock | 0.92 | 0.34 | 54.0% | 1.99 | 0.92 | 0.32 | 13.3% | 2.20 | 0.29* | 0.20* | 81.4% | 2.06 |
| Clean | 0.0 | 0.07 | 34.7% | 1.93 | 0.0 | 0.10 | 0.0% | 2.71 | 0.0 | 0.08 | 46.9% | 2.3 |
| Groovy | 0.67* | 0.64* | 151.0% | 3.72 | 0.63* | 0.59* | 182.4% | 3.86 | 0.63 | 0.51 | 206.5% | 3.4 |
| Piper | 0.00* | 0.02* | 0.0%* | 1.0 | 0.39 | 0.03 | 13.9% | 2.07 | 0.25 | 0.02 | 30.0% | 1.96 |
| Producerconsumer | 0.21* | 0.23* | 5.4% | 1.17 | 0.63 | 0.50 | 0.0%* | 4.31 | 0.52 | 0.29 | 38.0% | 2.13 |
| Stringbuffer | 0.78 | 0.53 | 168.4% | 2.36 | 1.0 | 0.87 | 6.1% | 6.50 | 0.97 | 0.62 | 209.5% | 3.06 |
| Twostage | 0.92 | 0.16 | 431.9% | 3.14 | 0.92 | 0.1 | 15.3% | 3.2 | 0.92 | 0.1 | 405.0% | 3.1 |
| Wronglock | 0.24* | 0.26* | 7.4% | 1.0 | 0.21 | 0.35 | 3.1%* | 1.0 | 0.26* | 0.33* | 2.3%* | 1.0 |
| | Def-Use | | | | Follows | | | | LR-Def | | | |
| | MFF | RFF | Cv | Sz | MFF | RFF | Cv | Sz | MFF | RFF | Cv | Sz |
| Accountsubtype | 0.13 | 0.3 | 22.0% | 2.99 | 0.24 | 0.06 | 7.1% | 1.92 | 0.23 | 0.03 | 1.9% | 1.87 |
| Alarmclock | 0.92 | 0.30 | 23.4% | 3.51 | 0.52 | 0.26 | 62.3% | 2.03 | 0.2* | 0.27* | 49.6% | 2.01 |
| Clean | 1.0 | 0.04 | 5.2% | 2.0 | 0.03* | 0.08* | 111.7% | 1.28 | 0.03* | 0.07* | 14.3% | 1.03 |
| Groovy | 0.35* | 0.43* | 5.3% | 3.0 | 0.26 | 0.45 | 59.1% | 3.02 | 0.30* | 0.38* | 6.3% | 2.09 |
| Piper | 0.0* | 0.02* | 0.5% | 1.13 | 0.70 | 0.09 | 13.0% | 3.54 | 0.01* | 0.03* | 2.8% | 1.78 |
| Producerconsumer | 1.0 | 0.36 | 4.1% | 2.0 | 0.5* | 0.5* | 24.7% | 3.71 | 1.0 | 0.31 | 5.9% | 2.30 |
| Stringbuffer | 0.33 | 0.56 | 6.2% | 2.33 | 1.0 | 0.83 | 238.1% | 4.46 | 0.4* | 0.30* | 14.3% | 1.4 |
| Twostage | 0.92 | 0.13 | 8.3% | 2.92 | 0.92 | 0.07 | 374.5% | 2.92 | 0.03* | 0.03* | 72.3% | 1.19 |
| Wronglock | 0.34* | 0.46* | 19.5% | 2.14 | 0.34* | 0.35* | 0.0%* | 1.0 | 0.28* | 0.33* | 5.9% | 2.0 |
| | PSet | | | | Sync-Pair | | | | | | | |
| | MFF | RFF | Cv | Sz | MFF | RFF | Cv | Sz | | | | |
| Accountsubtype | 0.36* | 0.44* | 29.4% | 6.6 | 0.21 | 0.0 | 8.1% | 1.87 | | | | |
| Alarmclock | 0.92 | 0.4 | 35.0% | 5.20 | 0.53 | 0.26 | 14.9% | 2.04 | | | | |
| Clean | 1.0 | 0.11 | 11.4% | 2.93 | 0.06* | 0.06* | 8.7% | 1.30 | | | | |
| Groovy | 0.33* | 0.4* | 6.8% | 3.0 | 0.41* | 0.46* | 52.0% | 3.02 | | | | |
| Piper | 0.43 | 0.06 | 5.1% | 1.94 | 0.64 | 0.03 | 53.6% | 3.49 | | | | |
| Producerconsumer | 1.0 | 0.4 | 6.3% | 2.34 | 0.5* | 0.38* | 30.4% | 3.72 | | | | |
| Stringbuffer | 1.0 | 0.76 | 7.3% | 3.0 | 1.0 | 0.74 | 38.7% | 4.35 | | | | |
| Twostage | 0.92 | 0.06 | 26.8% | 2.92 | 0.92 | 0.07 | 66.6% | 2.92 | | | | |
| Wronglock | 0.46 | 0.60 | 47.3% | 2.96 | 0.31* | 0.30* | 0.0%* | 1.0 | | | | |

*= not statistically significant difference at $\alpha = 0.05$.
MFF, maximum coverage fault detection; RFF, random fault detection; Cv, percent increase in coverage over random; Sz, test suite size.



DOI: 10.1002/stvr

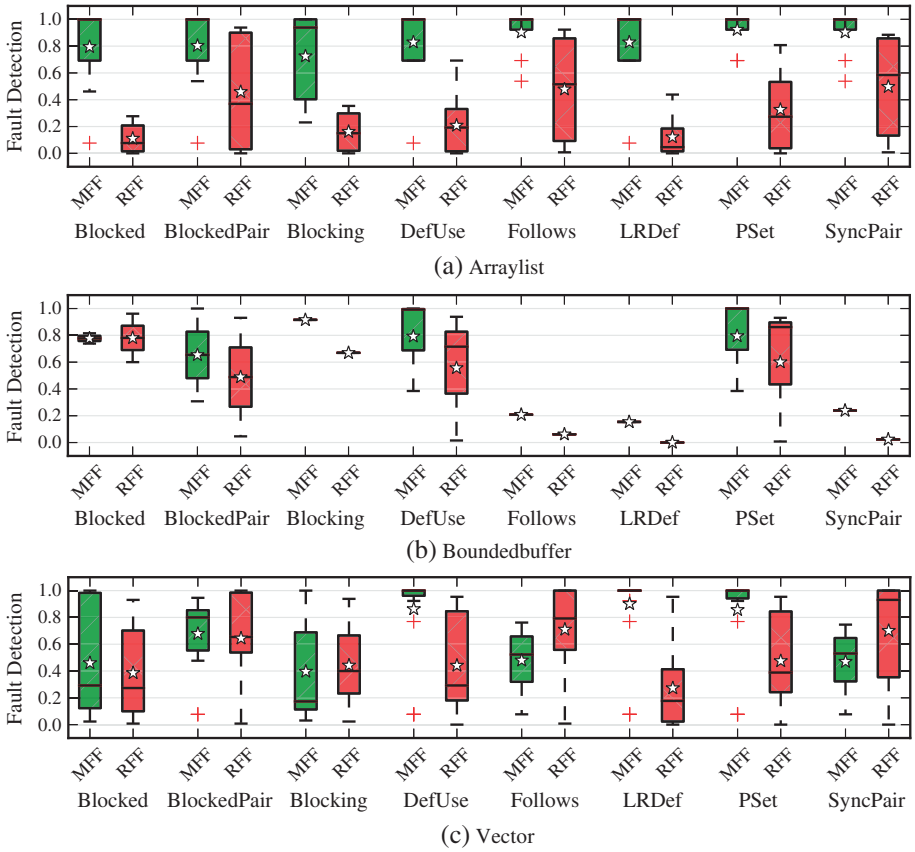



Figure 10. Maximum fault detection, greedy versus random, across mutants. MFF = maximum fault detection, RFF = random fault detection.

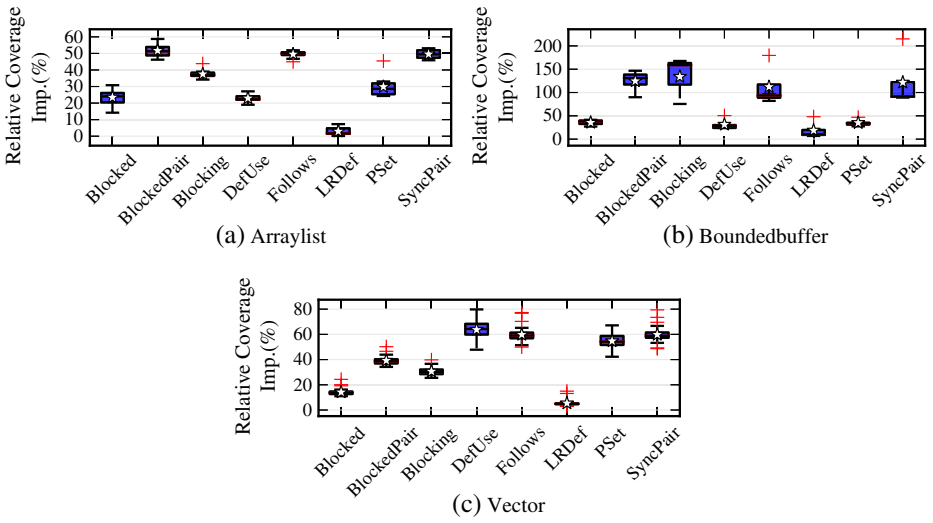



Figure 11. Relative improvement in coverage, greedy versus random, across mutants.

increases can be observed. For example, for the *Clean* object with the *Def-Use* coverage metric, the average fault detection of test suites achieving maximum coverage is generally higher (up to 25 times higher) than that of randomly generated test suites.



DOI: 10.1002/stvr

We can see a similar tendency for mutation object *Arraylist*. For the *Arraylist* object, the mean fault detection of maximum achievable test suites of every coverage metric is higher than or equal to the highest fault detection of corresponding randomly generated test suites.

Note that, for the *Boundedbuffer* object, the reduced test suites with respect to a coverage metric provide useful results although their correlations with fault detection are low. In contrast, *LR-Def* displays moderate to high correlations in fault detection as shown in Table IV, but the reduced test suites with respect to *LR-Def* do not have higher fault detection than randomly generated test suites in most cases.

We were surprised, however, that there were object/coverage metric pairs for which reduction to maximize coverage had a *negative* impact on the fault detection effectiveness of the testing process. For example, for *Wronglock*, test suites reduced to satisfy *Blocked-Pair* found the fault 21% of the time, as compared with 35% when using random test suites of equal size.

The case in which *Def-Use* was applied to *Stringbuffer* was more surprising. Here, we see greedily reduced test suites detecting the fault 33% of the time on average, relative to the 56% detection rate for randomly reduced test suites of equal size. As we demonstrate in Section 4.6, however, when achieving maximum coverage for complex coverage metrics, there exist several difficult-to-cover test requirements that are satisfied only by specific test executions that do not necessarily detect a fault (see Table X). During greedy test suite reduction, these executions must be selected to achieve maximum coverage and are thus useless with respect to fault detection but always present. We hypothesize that this is the cause of this unusual behaviour.

### *4.5. Effect of combining concurrency coverage metrics*

In the previous subsections, we demonstrated that while every coverage metric has a meaningful value as a predictor of fault detection effectiveness and also as a target for test generation, there is strong variation in the relative usefulness of the coverage metrics for both purposes across target programs. This implies that identifying a single proposed concurrency coverage metric to use for testing an arbitrary target program may be unrealistic.

One possible solution for addressing this variability is to combine complimentary concurrency coverage metrics, mitigating the shortfalls of each [14, 16]. To determine whether this solution is effective, we created and studied the effectiveness of six combined coverage metrics. The rationale for selecting these metrics was detailed in Section 3.1.1, but in short, these combinations were viewed as most likely to yield improvements over the original metrics.

*4.5.1. Combined coverage metrics as predictors.* We begin by examining the effectiveness of our combined metrics as predictors of testing effectiveness. In Table VII and Figure 12, we present the correlation of coverage and fault detection effectiveness of the combined coverage metrics as compared with the original metrics they are derived from. Based on these results, we see that the combined metrics are a mixed bag in terms of improvements. Across the single-fault objects, in 26 of the 54 combinations of combined metrics and objects, the combined metric achieves a correlation equal to or higher than the highest correlation observed from its composite original metrics. Typically, in these cases, the gains over the highest correlation observed from an original metric are small, but in some cases, the gains over the lowest performing metric are quite high. For example, in the case of the *Arraylist* object, the lowest correlation in the combined coverage metric is upgraded from the original coverage metrics, whereas the highest correlation still remains.

In the case of the *Wronglock* object, only data access metrics are effective predictors of fault detection, with all pairwise synchronization-based metrics achieving no higher than 0.12 correlation. Similar behaviour also occurs for the *Accountsubtype* object. In these scenarios, the failure of synchronization-based metrics is masked by the inclusion of data access metrics (notably, *PSet*, which per Section 4.2 we found to be the single most effective original metric overall). For the *Wronglock* and *Accountsubtype* objects, the all combined coverage metrics show the moderate correlations ($0.53 \sim 0.58$ for *Wronglock*, and $0.58 \sim 0.61$ for *Accountsubtype*).

In the opposite scenario, however, where synchronization-based metrics outperform data access metrics in terms of correlation, results are more mixed. For example, the combination of *Def-Use* to

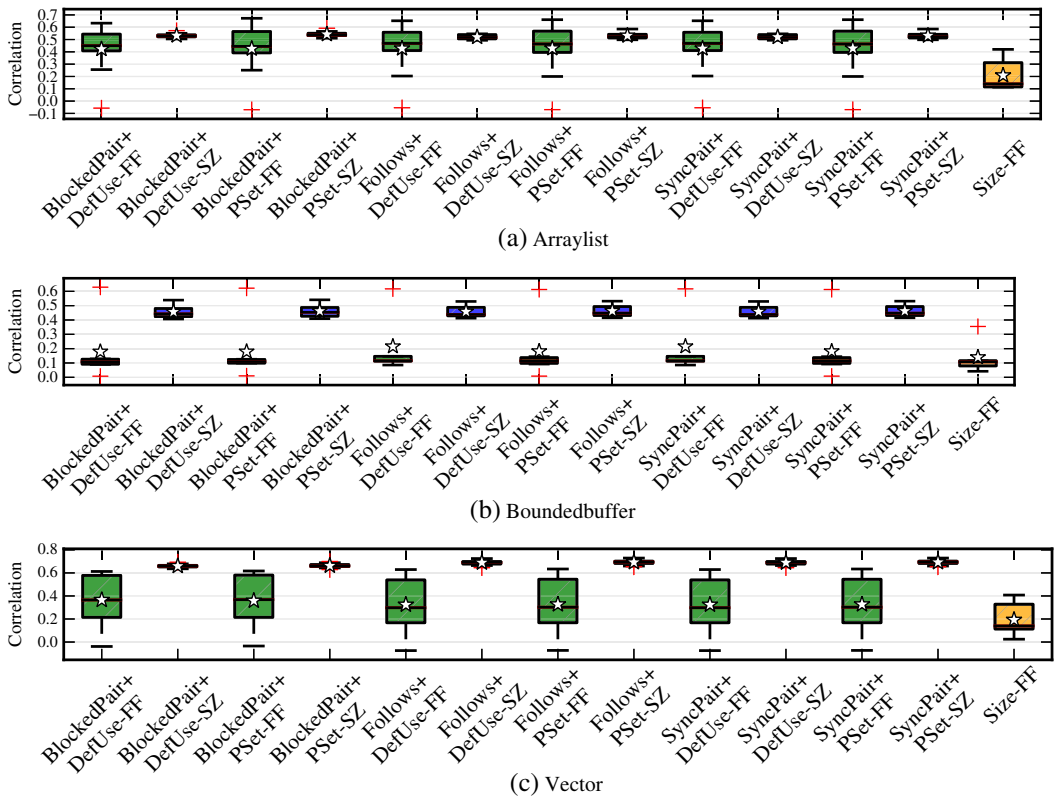


DOI: 10.1002/stvr

Table VII. Correlations over combined metrics.

| | Blocked-Pair + Def-Use | | | Blocked-Pair + PSet | | | Follows + Def-Use | | |
|---|---|---|---|---|---|---|---|---|---|
| | CM | Blocked-Pair | Def-Use | CM | Blocked-Pair | PSet | CM | Follows | Def-Use |
| Accountsubtype | 0.61 | 0.39 | 0.60 | 0.59 | 0.39 | 0.57 | 0.60 | 0.28 | 0.60 |
| Alarmclock | 0.60 | 0.52 | 0.56 | 0.65 | 0.52 | 0.59 | 0.52 | 0.66 | 0.56 |
| Clean | 0.38 | 0.73 | 0.96 | 0.21 | 0.73 | 0.83 | 0.73 | 0.17 | 0.96 |
| Groovy | 0.56 | 0.50 | 0.45 | 0.55 | 0.50 | 0.48 | 0.51 | 0.52 | 0.45 |
| Piper | 0.59 | 0.62 | 0.07 | 0.48 | 0.62 | 0.67 | 0.62 | 0.59 | 0.07 |
| Producerconsumer | 0.31 | 0.17 | 0.57 | 0.15 | 0.17 | 0.30 | 0.17 | 0.21 | 0.57 |
| Stringbuffer | 0.46 | 0.67 | 0.43 | 0.61 | 0.67 | 0.87 | 0.67 | 0.44 | 0.43 |
| Twostage | 0.92 | 0.94 | 0.92 | 0.88 | 0.94 | 0.96 | 0.94 | 0.88 | 0.92 |
| Wronglock | 0.53 | 0.12 | 0.53 | 0.58 | 0.12 | 0.58 | 0.53 | 0.0 | 0.53 |
| | Follows + PSet | | | Sync-Pair + Def-Use | | | Sync-Pair + PSet | | |
| | CM | Follows | PSet | CM | Sync-Pair | Def-Use | CM | Sync-Pair | PSet |
| Accountsubtype | 0.58 | 0.28 | 0.57 | 0.60 | 0.28 | 0.60 | 0.58 | 0.28 | 0.57 |
| Alarmclock | 0.25 | 0.66 | 0.59 | 0.27 | 0.19 | 0.56 | 0.55 | 0.19 | 0.59 |
| Clean | 0.20 | 0.17 | 0.83 | 0.07 | 0.09 | 0.96 | 0.66 | 0.09 | 0.83 |
| Groovy | 0.52 | 0.52 | 0.48 | 0.51 | 0.52 | 0.45 | 0.52 | 0.52 | 0.48 |
| Piper | 0.63 | 0.59 | 0.67 | 0.61 | 0.62 | 0.07 | 0.67 | 0.62 | 0.67 |
| Producerconsumer | 0.11 | 0.21 | 0.30 | 0.26 | 0.11 | 0.57 | 0.14 | 0.11 | 0.30 |
| Stringbuffer | 0.66 | 0.44 | 0.87 | 0.66 | 0.66 | 0.43 | 0.74 | 0.66 | 0.87 |
| Twostage | 0.92 | 0.88 | 0.96 | 0.90 | 0.96 | 0.92 | 0.96 | 0.96 | 0.96 |
| Wronglock | 0.58 | 0.0 | 0.58 | 0.53 | 0.0 | 0.53 | 0.58 | 0.0 | 0.58 |

Each cell contains coverage and fault detection correlation.
CM, combined metric correlation.




(a) Arraylist

(b) Boundedbuffer

(c) Vector

Figure 12. Correlations across mutants, combined metrics. FF = fault detection, SZ = test suite size.




DOI: 10.1002/stvr

*Follows* results in a moderate correlation of 0.52, but this is a small drop from the original metrics' respective correlations of 0.56 and 0.66. In fact, examining our original suggestion of *PSet*, we find that for 23 of the 24 combinations of combined metrics including *PSet* and single-fault objects, *PSet*'s correlation is within 0.05 of the combined correlation, and for 17 combinations, it is equal to or greater than *PSet*'s correlation.

More concerning are scenarios where combinations of metrics significantly reduce the correlation. For example, in the case of *Follows* + *PSet*, the combined metric often performs far worse than either metric alone (e.g., *Alarmclock*, *Clean*, and *Producerconsumer* all show the correlation dropping by 50%). Similar scenarios can be seen when using other combinations as well. Thus, while it is true that in some cases, a combination of metrics can be a better predictor than single metrics alone, we cannot offer a general recommendation, as there are also many cases where combinations are less effective predictors.

*4.5.2. Combined metrics as test case generation targets.* While having more effective predictors of testing effectiveness is useful, we are also interested in having more effective test case generation targets. In Table VIII and Figure 13, we present the fault detection results for test suites achieving the maximum achievable coverage for the single-fault objects and for the mutation testing objects, respectively. In Table IX, we present the relative improvement in fault detection when using combined coverage metrics over the original coverage metrics for the single-fault objects.

The results show that for every object and for every combined coverage metric, the fault detection effectiveness of the reduced test suite with respect to a combined coverage metric is higher than or equal to that of an original coverage metric. Naturally, the fault detection for a given coverage metric can only remain the same or increase by combining it with another metric (the concurrency coverage metrics studied, like typical sequential coverage metrics, are monotonic). Therefore, the existence of improvements is not especially interesting.

Table VIII. Maximum achievable coverage test suite statistics, combined metrics.

| | Blocked-Pair + Def-Use | | | | Blocked-Pair + PSet | | | | Follows + Def-Use | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | MFF | RFF | Cv | Sz | MFF | RFF | Cv | Sz | MFF | RFF | Cv | Sz |
| Accountsubtype | 0.15 | 0.40 | 18.8% | 3.28 | 0.36* | 0.46* | 23.7% | 6.85 | 0.17* | 0.28* | 13.6% | 3.12 |
| Alarmclock | 0.92 | 0.30 | 22.4% | 3.88 | 0.92 | 0.45 | 32.0% | 5.56 | 0.92 | 0.35 | 19.7% | 4.32 |
| Clean | 1.0 | 0.18 | 11.6% | 3.72 | 1.0 | 0.23 | 26.8% | 4.16 | 1.0 | 0.11 | 4.1% | 2.22 |
| Groovy | 0.65* | 0.60* | 17.3% | 3.99 | 0.69* | 0.58* | 23.8% | 4.00 | 0.30 | 0.41 | 10.0% | 3.00 |
| Piper | 0.4 | 0.01 | 4.3% | 2.06 | 0.7 | 0.02 | 18.7% | 2.10 | 0.68 | 0.06 | 12.7% | 3.59 |
| Producerconsumer | 1.0 | 0.60 | 8.2% | 4.61 | 1.0 | 0.69 | 20.7% | 4.83 | 1.0 | 0.55 | 7.4% | 4.01 |
| Stringbuffer | 1.0 | 0.87 | 26.7% | 6.89 | 1.0 | 0.89 | 69.0% | 6.9 | 1.0 | 0.79 | 12.2% | 4.38 |
| Twostage | 0.92 | 0.13 | 22.5% | 3.76 | 0.92 | 0.16 | 228.0% | 3.73 | 0.92 | 0.11 | 15.1% | 2.92 |
| Wronglock | 0.34* | 0.43* | 17.7% | 2.17 | 0.54* | 0.56* | 40.6% | 2.97 | 0.41* | 0.42* | 17.1% | 2.16 |
| | Follows + PSet | | | | Sync-Pair + Def-Use | | | | Sync-Pair + PSet | | | |
| | MFF | RFF | Cv | Sz | MFF | RFF | Cv | Sz | MFF | RFF | Cv | Sz |
| Accountsubtype | 0.36* | 0.47* | 19.3% | 6.64 | 0.21 | 0.35 | 13.4% | 3.14 | 0.4* | 0.42* | 18.5% | 6.68 |
| Alarmclock | 0.92 | 0.46 | 13.6% | 5.93 | 0.92 | 0.41 | 0.9%* | 4.38 | 0.92 | 0.53 | 27.1% | 6.00 |
| Clean | 1.0 | 0.08 | 9.5% | 2.97 | 1.0 | 0.11 | 0.9% | 2.17 | 1.0 | 0.07 | 7.6% | 2.93 |
| Groovy | 0.46* | 0.42* | 15.7% | 3.0 | 0.38* | 0.46* | 10.1% | 3.01 | 0.33* | 0.39* | 14.6% | 3.02 |
| Piper | 0.68 | 0.03 | 48.3% | 3.57 | 0.70 | 0.08 | 18.9% | 3.56 | 0.68 | 0.1 | 16.0% | 3.53 |
| Producerconsumer | 1.0 | 0.55 | 52.5% | 4.13 | 1.0 | 0.43 | 15.8% | 3.99 | 1.0 | 0.55 | 10.5% | 4.22 |
| Stringbuffer | 1.0 | 0.80 | 54.8% | 4.56 | 1.0 | 0.72 | 12.8% | 4.52 | 1.0 | 0.82 | 15.8% | 4.61 |
| Twostage | 0.92 | 0.13 | 111.2% | 2.92 | 0.92 | 0.09 | 0.0% | 2.92 | 0.92 | 0.10 | 29.9% | 2.92 |
| Wronglock | 0.52* | 0.61* | 41.6% | 2.97 | 0.32 | 0.46 | 17.6% | 2.14 | 0.48 | 0.6 | 43.0% | 3.01 |

*= not statistically significant difference at $\alpha = 0.05$.
MFF, maximum coverage fault detection; RFF, random fault detection; Cv, percent increase in coverage over random; Sz, test suite size.



DOI: 10.1002/stvr

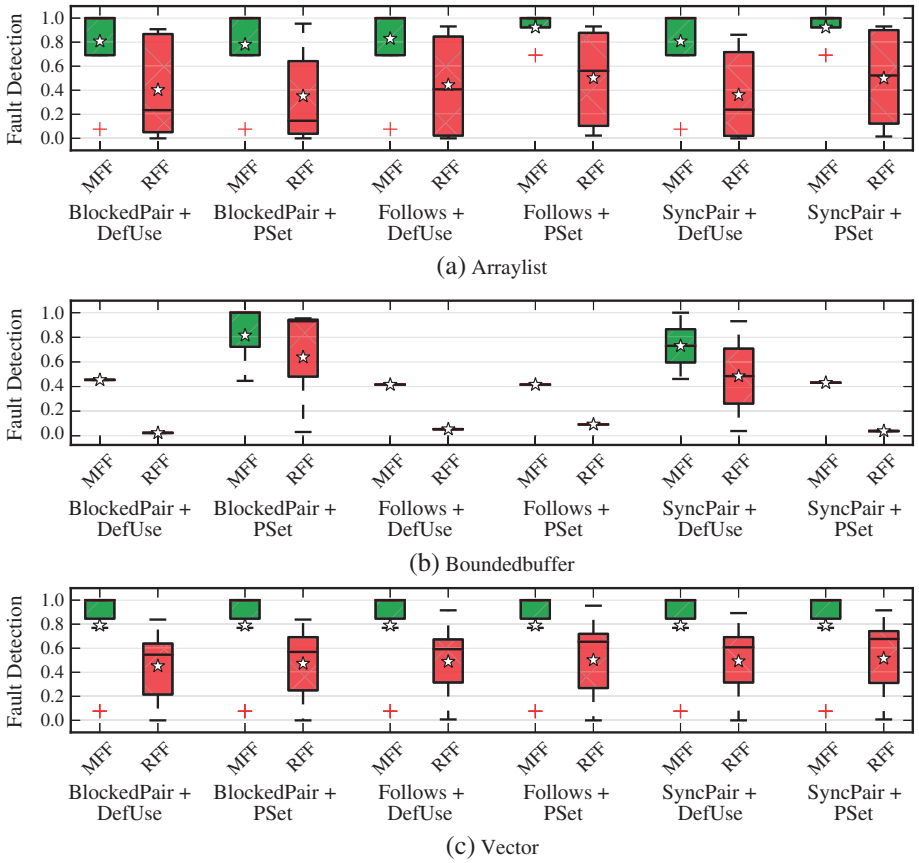



Figure 13. Maximum fault detection, greedy versus random, across mutants, combined metrics. MFF = maximum fault detection, RFF = random fault detection.

Instead, we wish to determine whether combinations either offer improvements over both metrics simultaneously, indicating a clear improvement in fault detection for some objects and indicating less variability in the effectiveness of the metric as a test generation target; or alternatively, whether combinations offer improvements over each metric in different scenarios. In other words, we wish to determine whether, for some combined metric $A+B$, improvements are found over only $A$ for one object, while improvements are found over only $B$ for some other object.

Based on Table IX, we can see that statistically significant examples of both types of improvements exist. For example, when applying the *Blocked-Pair + PSet* coverage metric over the *Piper* object, improvements over *PSet* and *Blocked-Pair* of 62.5% and 78.4% exist.

Additionally, for the *Follows + Def-Use* combination, we can see that for both *Alarmclock* and *Clean*, the combined metric is an improvement over *Follows* by 76.4% and 3150.0%, while for the *Piper* and *Stringbuffer* objects, it is a comparable improvement over *Def-Use*. Similar patterns can be seen for all other combinations of metrics, indicating that the combined metrics do frequently reduce variability as compared with the use of individual metrics.

This reduction in variability is further illustrated by examining the fault detection rates for original test suites (Section 4.4). While the fault detection effectiveness across combined metrics is consistent within each object, the fault detection effectiveness for original metrics sometimes varies strongly across metrics. For example, within pairwise metrics (i.e., those used to create combined metrics), test suites generated for the *Clean* object vary in average fault detection from 0.0 to 1.0 as shown in Table VI, while the average fault detection for combined metrics is always 1.0. Other objects exhibit similar behaviour.

As noted in Section 4.4, there is no best original metric to use as a test case generation target. However, several combined metrics when used as test case generation targets always produce, on



DOI: 10.1002/stvr

Table IX. Relative improvement in fault detection using combined metrics.

| | Blocked-Pair + Def-Use | | Blocked-Pair + PSet | | Follows + Def-Use | |
|---|---|---|---|---|---|---|
| | Blocked-Pair | Def-Use | Blocked-Pair | PSet | Follows | Def-Use |
| Accountsubtype | 5.2%* | 11.1%* | 147.3% | 0.0%* | 0.0%* | 27.7%* |
| Alarmclock | 0.0%* | 0.0%* | 0.0%* | 0.0%* | 76.4% | 0.0%* |
| Clean | inf% | 0.0%* | inf% | 0.0%* | 3150.0% | 0.0%* |
| Groovy | 2.4%* | 84.7% | 8.4%* | 104.5% | 14.2%* | 0.0%* |
| Piper | 1.9%* | inf% | 78.4% | 62.5% | 0.0%* | inf% |
| Producerconsumer | 56.6% | 0.0%* | 56.6% | 0.0%* | 100.0% | 0.0%* |
| Stringbuffer | 0.0%* | 195.4% | 0.0%* | 0.0%* | 0.0%* | 195.4% |
| Twostage | 0.0%* | 0.0%* | 0.0%* | 0.0%* | 0.0%* | 0.0%* |
| Wronglock | 60.7% | 0.0%* | 153.5% | 18.3%* | 20.0%* | 20.0%* |
| | Follows + PSet | | Sync-Pair + Def-Use | | Sync-Pair + PSet | |
| | Follows | PSet | Sync-Pair | Def-Use | Sync-Pair | PSet |
| Accountsubtype | 50.0% | 0.0%* | 0.0%* | 55.5%* | 85.7% | 8.3%* |
| Alarmclock | 76.4% | 0.0%* | 71.4% | 0.0%* | 71.4% | 0.0%* |
| Clean | 3150.0% | 0.0%* | 1344.4% | 0.0%* | 1344.4% | 0.0%* |
| Groovy | 71.4% | 36.3%* | 0.0%* | 8.6%* | 0.0%* | 0.0%* |
| Piper | 0.0%* | 58.9% | 9.5%* | inf% | 5.9%* | 58.9% |
| Producerconsumer | 100.0% | 0.0%* | 100.0% | 0.0%* | 100.0% | 0.0%* |
| Stringbuffer | 0.0%* | 0.0%* | 0.0%* | 195.4% | 0.0%* | 0.0%* |
| Twostage | 0.0%* | 0.0%* | 0.0%* | 0.0%* | 0.0%* | 0.0%* |
| Wronglock | 51.1% | 13.3%* | 2.4%* | 0.0%* | 53.6% | 5.0%* |

* = not statistically significant difference at $\alpha = 0.05$.

average, higher fault detection than any single original metric (excluding fault detection values that are not statistically significant). In fact, every combined metric containing *PSet* exhibits this behaviour. Note that these test suites are typically larger than those generated solely from original metrics, but given the small size of all test suites (less than seven tests on average), this seems acceptable.

This result also supports our conjecture that there are other factors that influence testing effectiveness beyond those that the concurrency coverage metrics studied capture (see Section 5.2).

In summation, while the predictive value of combined metrics differs from that of original metrics in ways that is not necessarily positive or negative, combined metrics as test case generation targets — in particular those metrics based on a combination of *PSet* with a pairwise, synchronization metric — are clearly superior to any original metric studied.

### *4.6. Effectiveness of difficult-to-cover test requirements*

Our analysis has clearly demonstrated that increasing coverage levels of the presented concurrency coverage metrics tends to result in practically significant increases in fault detection effectiveness. Nevertheless, this does not necessarily imply that all test requirements are worth the effort required to cover them. Per *RQ4*, we would like to determine whether difficult-to-cover test requirements — those that are satisfied by only a small percentage of tests — yield fault detection gains beyond those found in the other, easier to cover test requirements. This is key to establishing if specialized techniques that target hard to cover test requirements are likely to yield improvements in fault detection (akin to techniques for covering branches in structural coverage metrics).

First, we begin by establishing that difficult-to-cover test requirements exist. In Figure 14, we plot, for each covered test requirement, the percentage of test executions covering the requirement, that is, difficulty of covering the test requirement (Figure A.3 for all objects, in Section 6). Requirements have been ordered from least likely to be covered to mostly likely to be covered. (The x-axis represents the difficulty percentile; i.e., at 40%, the requirement plotted is easier than 40% of all requirements and more difficult than 60%.) For each object and coverage criteria, there exists significant variation in the difficulty of covering test requirements —



DOI: 10.1002/stvr

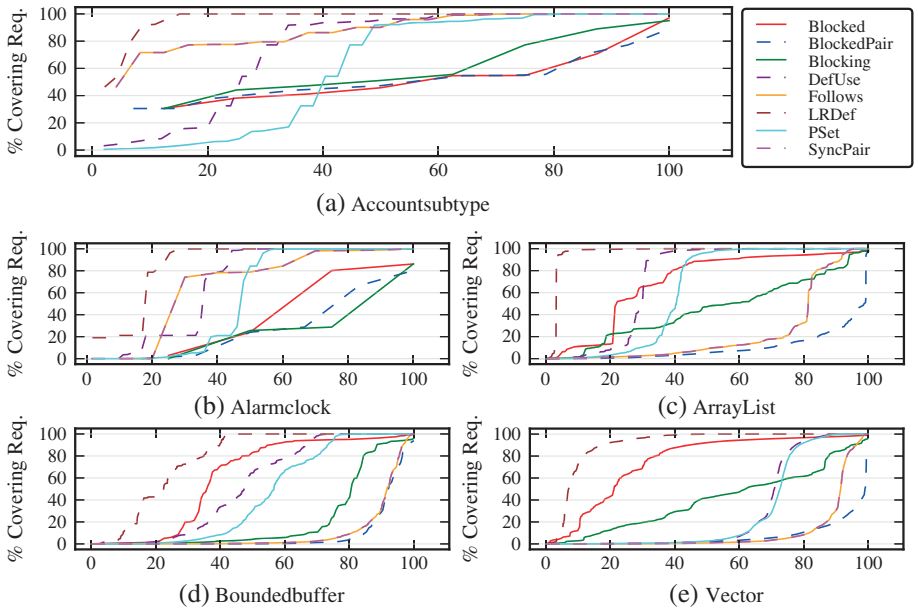



Figure 14. Relative difficulty of covering individual coverage requirements for four single-fault objects and all mutation objects. The x-axis represents the difficulty percentile. Requirements covered by the fewest number of executions are leftmost on the x-axis; requirements covered by the largest number of executions are rightmost. The y-axis indicates the percentage of executions that cover the requirement.

most objects contain several requirements that are covered by few executions (less than 1%), with most test executions being relatively easily covered (with greater than 10% covering the test executions).

Having established that difficult-to-cover test requirements exist, we would like to determine whether these test requirements are, on average, particularly effective at detecting faults. Towards this, in Table X, we present the average fault detection of test executions covering difficult-to-cover requirements (defined as the 10% most difficult requirements to cover) as compared with other test requirements. We selected the 10% threshold as it frequently resulted in one or fewer test executions being selected, while larger thresholds were too easy to cover to be considered 'difficult'. Note that 'NA' indicates that the number of requirements was less than 10; that is, there was no bottom 10%.

Here, we see that in some instances, there does appear to be a practically and statistically significant difference in the fault detection rate of test executions satisfying difficult-to-cover test requirements relative to other test requirements. For example, for the *Arraylist* object, difficult-to-cover test requirements of all coverage metrics are better than other test requirements, with the relative differences in fault detection effectiveness ranging from 91.5% to 942.4%. Clearly, for many objects, the effort needed to satisfy difficult-to-cover requirements is potentially worthwhile.

In other cases, however, the relative difference between difficult-to-cover and easy-to-cover test requirements is either practically marginal (e.g., for the *Vector* where differences are small and often close to zero) or not statistically significant (e.g., the *Groovy* and *Stringbuffer* object). Given these results, it is difficult to draw any conclusions concerning the value of difficult-to-cover requirements in testing a particular program.

In some cases, the extra effort is clearly unlikely to be rewarded as the relative differences are minor.

On the other hand, in many cases, the relative difference is quite large, but (due to the small number of test requirements) not statistically significant. Thus, it appears that studies with objects that produce larger numbers of test requirements are required to better address this question. We



DOI: 10.1002/stvr

Table X. Fault detection effectiveness for difficult and easy to cover test requirements.

| | Blocked | | | Blocked-Pair | | | Blocking | | |
|---|---|---|---|---|---|---|---|---|---|
| | DFF | EFF | % | DFF | EFF | % | DFF | EFF | % |
| Arraylist | 0.75 | 0.05 | 1259.3% | 0.15 | 0.08 | 91.5% | 0.46 | 0.04 | 942.4% |
| Boundedbuffer | 0.19* | 0.30* | 0.0%* | 0.23* | 0.26* | 0.0%* | 0.17* | 0.26* | 0.0%* |
| Vector | 0.13* | 0.10* | 31.8%* | 0.06 | 0.08 | 0.0% | 0.10* | 0.09* | 2.9%* |
| Accountsubtype | 0.07* | 0.07* | 0.0%* | 0.07* | 0.07* | 0.0%* | 0.07* | 0.07* | 0.0%* |
| Alarmclock | NA | NA | NA | 1.0* | 0.28* | 254.9%* | NA | NA | NA |
| Clean | NA | NA | NA | 0.0* | 0.0* | 0.0%* | 0.0* | 0.0* | 0.0%* |
| Groovy | 0.34* | 0.23* | 46.5%* | 0.34* | 0.23* | 47.6%* | 0.34* | 0.23* | 50.2%* |
| Piper | NA | NA | NA | 0.47* | 0.04* | 978.3%* | NA | NA | NA |
| Producerconsumer | NA | NA | NA | 0.31 | 0.17 | 79.8% | 0.21* | 0.18* | 18.8%* |
| Stringbuffer | NA | NA | NA | 0.79* | 0.50* | 58.7%* | 0.88* | 0.49* | 77.5%* |
| Twostage | 1.0 | 0.10 | 854.7% | 1.0* | 0.21* | 356.1%* | 1.0 | 0.25 | 294.7% |
| Wronglock | NA | NA | NA | NA | NA | NA | NA | NA | NA |

| | Def-Use | | | Follows | | | LR-Def | | |
|---|---|---|---|---|---|---|---|---|---|
| | DFF | EFF | % | DFF | EFF | % | DFF | EFF | % |
| Arraylist | 0.10 | 0.05 | 106.1% | 0.19 | 0.06 | 181.5% | 0.18 | 0.04 | 345.5% |
| Boundedbuffer | 0.39 | 0.30 | 30.0% | 0.28* | 0.25* | 11.3%* | 0.21 | 0.29 | 0.0% |
| Vector | 0.04 | 0.07 | 0.0% | 0.04 | 0.06 | 0.0% | 0.11* | 0.10* | 19.5%* |
| Accountsubtype | 0.07* | 0.07* | 5.3%* | 0.07* | 0.07* | 0.0%* | 0.07* | 0.07* | 0.0%* |
| Alarmclock | 0.14* | 0.16* | 0.0%* | 1.0* | 0.17* | 464.2%* | 0.21 | 0.11 | 90.9% |
| Clean | 0.5* | 0.03* | 1490.9%* | 0.0* | 0.00* | 0.0%* | 0.0* | 0.03* | 0.0%* |
| Groovy | 0.23 | 0.21 | 8.1% | 0.19* | 0.22* | 0.0%* | 0.22* | 0.21* | 2.9%* |
| Piper | 0.01 | 0.01 | 0.0% | 0.19* | 0.08* | 123.5%* | 0.01 | 0.02 | 0.0% |
| Producerconsumer | 0.32* | 0.18* | 69.9%* | 0.35 | 0.18 | 93.8% | 0.67 | 0.19 | 248.3% |
| Stringbuffer | 0.0 | 0.30 | 0.0% | 0.0* | 0.40* | 0.0%* | 0.16* | 0.32* | 0.0%* |
| Twostage | 0.75 | 0.04 | 1611.9% | 1.0* | 0.19* | 407.4%* | 0.03 | 0.04 | 0.0% |
| Wronglock | 0.28* | 0.26* | 4.7%* | NA | NA | NA | 0.29* | 0.26* | 10.3%* |

| | PSet | | | Sync-Pair | | |
|---|---|---|---|---|---|---|
| | DFF | EFF | % | DFF | EFF | % |
| Arraylist | 0.16 | 0.06 | 176.5% | 0.19 | 0.06 | 181.5% |
| Boundedbuffer | 0.35* | 0.32* | 10.8%* | 0.28* | 0.25* | 11.3%* |
| Vector | 0.07 | 0.08 | 0.0% | 0.04 | 0.06 | 0.0% |
| Accountsubtype | 0.09 | 0.07 | 19.9% | 0.07* | 0.07* | 0.0%* |
| Alarmclock | 0.48* | 0.26* | 82.7%* | 1.0* | 0.17* | 464.2%* |
| Clean | 0.5* | 0.02* | 2289.7%* | 0.0* | 0.00* | 0.0%* |
| Groovy | 0.23* | 0.21* | 7.5%* | 0.19* | 0.22* | 0.0%* |
| Piper | 0.18 | 0.01 | 867.2% | 0.19* | 0.08* | 123.5%* |
| Producerconsumer | 0.39 | 0.19 | 107.2% | 0.35 | 0.18 | 93.8% |
| Stringbuffer | 0.5* | 0.30* | 65.5%* | 0.0* | 0.40* | 0.0%* |
| Twostage | 1.0 | 0.10 | 839.7% | 1.0* | 0.19* | 407.4%* |
| Wronglock | 0.31 | 0.27 | 13.0% | NA | NA | NA |

* = not statistically significant at $p = 0.05$.
DFF, difficult-to-cover fault detection; EFF, easy-to-cover fault detection; %, percent increase in average fault detection for DFF over EFF coverage requirements.

discuss the implications of this for concurrent test case generation approaches in the next section (see Section 5.4).

## 5. DISCUSSION

Our results have addressed our original research questions as follows. Per *RQ1* and *RQ2*, we have shown that for every coverage metric, for some programs, (i) the metric is a moderate, independent



DOI: 10.1002/stvr

predictor of fault detection, and (ii) the testing process can be made more effective by using test suites that achieve maximum coverage instead of random test suites of equal size.

In short, we have provided evidence that existing concurrency coverage metrics can be useful. Consequently, testers can use concurrency coverage metrics as part of their testing process with confidence, either to estimate testing effectiveness or as a goal for the testing process. Furthermore, testing researchers can justify as worthwhile the effort spent developing tools and techniques based on concurrency coverage metrics.

Nevertheless, the variation in the relative effectiveness of coverage metrics raises issues concerning how to apply these metrics in practice. Additionally, the generally moderate levels of correlation and fit observed hint that while these metrics appear effective, improvements to these metrics are both possible and desirable.

Towards addressing this variability and to better understand how test generation should be approached to improve fault detection, we proposed and addressed research questions *RQ3* and *RQ4*. Per *RQ3*, we have seen that using two coverage metrics combined can, in some cases, improve the reliability of coverage metrics as estimators of testing effectiveness and particularly as test generation targets. Per *RQ4*, we have shown that at least in some cases, satisfying difficult-to-cover test requirements often returns meaningful improvements in fault detection. These results provide some guidance how test generation for concurrency testing can be improved with respect to the resulting fault detection rates.

In the remainder of this section, we discuss the practical implications of the study and highlight additional areas of research that we believe should be explored.

### *5.1. Practical implications for testers*

Following a study of several coverage metrics, the question every tester naturally asks is as follows: *which metric should I use?*

Examining the correlation with fault detection (Table IV and Figure 7) and the fault detection effectiveness of maximum test suite result (Table VI and Figure 10), we see that if a tester must select a single 'best' metric, *PSet* seems to be the only possible choice. For seven objects among nine single-fault objects, *PSet* coverage's correlation with fault detection is over 0.57. *PSet* always achieved a greater correlation with fault detection than size (*S-FF*). Additionally, the reduced test suites with respect to *PSet* achieve higher fault detection than random test suites of equal size for six objects and achieve lower fault detection than random test suite for only one object (*Wronglock*). *PSet* is clearly not ideal in many scenarios — *Def-Use* was similarly effective as a generation target for *Boundedbuffer* while requiring fewer test executions and *Blocking* was more effective as a generation target for *Groovy* — but on the whole, it was consistently effective as a predictor and for test case generation.

With respect to the other metrics, our results suggest basic guidelines. Recall from Table II the coverage metric properties of *singular/pairwise*. Comparing the results for *singular* and *pairwise* metrics while holding the other metric property (*synchronization/data access*) constant reveals two patterns.

First, the fault detection for maximum coverage test suites for pairwise metrics tends to be equal to or higher than when using singular metrics. Thus, as test case generation target, it is preferable to select pairwise metrics. Second, pairwise metrics generally have higher correlation with fault detection and more reliable overall tendency across programs than singular metrics. For every single-fault object, the correlation of *Blocked-Pair* is higher than or equal to the correlations of its singular versions *Blocked* and *Blocking*. In contrast, *LR-Def* often shows as high correlations as *Def-Use* or *PSet* do. But, the maximum test suites of *LR-Def* show significantly less fault detection than *Def-Use* and *PSet*, which indicates its practical limitation.

Of course, as noted previously, pairwise metrics have more requirements and thus require more test executions to achieve maximum coverage. Nevertheless, the stronger correlation between pairwise coverage metrics and fault detection indicates that investing the effort needed to satisfy a pairwise coverage metric is preferable to investing the same amount of effort satisfying a singular



DOI: 10.1002/stvr

metric. When a test reaches a likely saturation point in a singular coverage metric, we recommend achieving as many pairwise coverage requirements as possible rather than targeting a few remaining singular requirements.

The previous advice relates to the previously proposed individual metrics. Based on the results given in Section 4.5 related to *RQ3*, if we are primarily interested in selecting a test generation target, we would do well to use combined metrics. While the correlations for combined metrics, shown in Table VII, are not always improvements over those for the original metrics, fault detection rates for test suites achieving maximum coverage are typically improved. In particular, we recommend a metric combining *PSet* and a pairwise synchronization coverage metric (e.g., *Follows*), as this provides a somewhat reliable testing estimator and more effective test generation target than any of the original metrics used. As with the move from singular to pairwise metrics, this increases the number of requirements (being a combination of two pairwise metrics), but as shown in Table VIII, for the systems studied, the size of the resulting test suites is not significantly larger than the size of suites defined over the original metrics.

As a final note, for some objects, there was a large difference in fault detection depending on the code constructs (*synchronization/data access*) used to define the metrics. For example, when using data access-based coverage metrics with *Wronglock*, the correlation with fault detection was roughly four times that of synchronization-based metrics. However, for *Piper*, the opposite was true; data access-based metrics show poor fault detection in the reduced test suites. Even among combined metrics, which are intended to reduce these variations by combining metrics based on different constructs, this behaviour was still observed, for example, *Follows* + *PSet* as compared with *Blocked-Pair* + *PSet* for the *Arraylist* and *Boundedbuffer* systems.

We found this surprising: while in theory, such behaviour can also exist between foundationally different sequential coverage metrics (e.g., metrics defined over def-use pairs versus those defined over branch constructs), in our experience, such dramatic differences do not occur in practice.

### 5.2. Limitations of existing concurrency metrics

As noted, in some cases, the concurrency coverage metrics explored exhibited low correlation with fault detection and/or poor fit during linear regression. These results stand in sharp contrast to results related to sequential coverage criteria, where, for example, much better linear regression fit has been achieved using only test suite size and coverage levels, with adjusted $R^2$ values over 0.90 being typical [17, 18]. In contrast, we observed few adjusted $R^2$ values greater than 0.8, indicating that a great deal of effectiveness is unaccounted for by test suite size and coverage. By uncovering additional factors that contribute to fault detection effectiveness, we may be able to improve our concurrency coverage metrics and testing techniques.

As an initial step towards this, we extended our linear regression analysis to consider two additional factors: the probability of a delay being inserted (*PB*), and the length of the delay inserted (*DL*) (see Section 3.2.2). These factors were controlled for during test execution and have been observed to impact the effectiveness of concurrent testing in previous work [13, 15]. We then repeated our regression analysis, selecting the model with the highest fit for each combination of coverage metric and program.

Following this, we compared each selected model's fit against the same model with *PB* and *DL* omitted as explanatory variables. We found that while sometimes the improvement when using *PB* and *DL* as explanatory variables was small ($< 0.01$), often the improvement was significant: the average relative increase in adjusted $R^2$ was 50.5% (maximum 814%), and the average improvement in adjusted $R^2$ was 0.05 (maximum 0.37). In some cases, *PB* and *DL* account for the bulk of the predictive power; for example, for *Alarmclock*, the best adjusted $R^2$ for the (usually effective) *PSet* metric increased from 0.45 to 0.78, an improvement of 75.1%.

We believe that these results highlight the need to further improve concurrency coverage metrics to provide better guidance to testers and testing techniques. Ideally, a coverage metric should perfectly capture the effectiveness of the testing process, providing a highly accurate estimate of



DOI: 10.1002/stvr

Table XI. Relation between fault types and concurrency coverage metrics.

| Fault type | Study object | Coverage metrics of highest correlation with fault detection | Coverage metrics of highest fault detection with maximum test suites |
|---|---|---|---|
| Atomicity violation | Stringbuffer | PSet (LR-Def) | Blocked-pair, Follows, PSet, Sync-pair |
| | Twostage | PSet, Sync-Pair, (Blocked, Blocked-Pair, Blocking, Def-Use, Follows, LR-Def, Sync-Pair) | Blocked, Blocked-Pair, Blocking, Def-Use, Follows, PSet, Sync-Pair |
| Data race | Accountsubtype | Def-Use | Follows |
| | Alarmclock | Blocked | Blocked, Blocked-pair, Def-Use, PSet |
| | Wronglock | PSet | NA |
| Deadlock (with wait) | Clean | Def-Use (Blocked-Pair, LR-Def, PSet) | Def-Use, PSet |
| | Groovy | Follows, Sync-Pair | Blocking |
| | Piper | PSet | Follows |
| Order violation | Producerconsumer | Def-Use | Def-Use, LR-Def, PSet |

testing effectiveness, upon which techniques for improving coverage can be built. At a minimum, we would like concurrency coverage metrics to be better predictors than *PB* and *DL*, as the most effective set of parameters — much like the metrics explored — varies unpredictably depending on program.

### *5.3. Relationship between metric effectiveness and fault type*

One potential factor that may account for the variability in testing is the types of faults present. Concurrency faults, in contrast to sequential faults — which can take nearly any form — are errors in specific constructs: for example, data races, for example, unsynchronized accesses to a shared variable with at least one write operation; and deadlocks, for example, incorrect synchronization orders such as `wait(m)` after `notify(m)`. Thus, detecting these faults can be easier or more difficult depending on the metric used, as different metrics focus on different code constructs.

To investigate this, in Table XI, we again present the best metrics, as measured by correlation and the effectiveness of maximum coverage test suites, for each object grouped by the type of fault present. The best metrics with respect to correlation are presented in the third column, while the best metrics with respect to fault detection rate for maximum achievable coverage test suites are presented in the fourth column ('NA" indicates that no metric was better than random with statistical significance).[‖] In the case of ties for best, all metrics are presented. Furthermore, in the case of correlation, all metrics achieving high correlation ($> 0.7$) are listed in parentheses. Note that we present only the single-fault objects as the type of faults present is already known from previous work [22–24]; when using mutation operators, we cannot be certain of the type of fault without a large amount of effort, an infeasible task for each mutant. Additionally, note that this (like the previous subsection) is an exploratory ad-hoc analysis; additional work will be required to verify the observations made.

Our expectation was that if the test requirements of a coverage metric $M$ are formulated over constructs matching those involved with fault type $T$, metric $M$ should perform well over objects of exhibiting fault type $T$. For example, we expected that *Def-Use* and *PSet* should perform well

[‖]To select the best metric with respect to fault detection, we exclude coverage metrics whose fault detection is not statistically significantly different than randomly generated test suites of equal size.



DOI: 10.1002/stvr

over objects exhibiting data race and atomicity violations, as the test requirements generated by these coverage metrics are based upon data access operations. We also expected that *Blocked-Pair*, *Follows*, and *Sync-Pair* metrics should perform well on objects exhibiting deadlock faults, as the test requirements of these coverage metrics are based on lock operations.

As shown in Table XI, there is no clear relationship between the fault type and the most effective coverage with respect to correlation. For example, for data race faults, *Def-Use*, *Blocked*, and *PSet* have the highest correlations on *Accountsubtype*, *Alarmclock*, and *Wronglock*, respectively. Indeed, even the best type of metric (synchronization/data access) varies depending on the program. Clearly, there is no best coverage metric for any fault type.

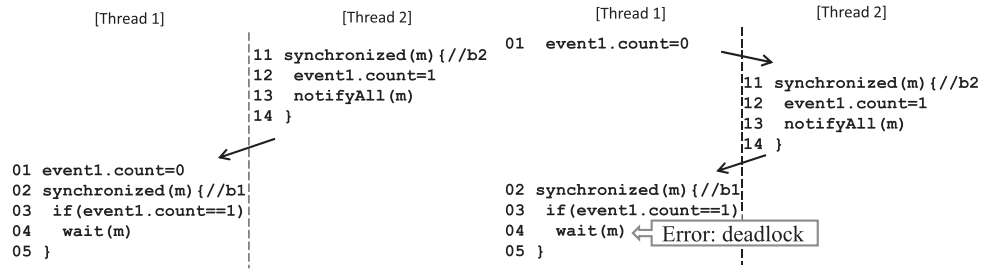
We see similar results with respect to fault detection effectiveness for maximum coverage test suites. For example, for deadlock faults, *Def-Use* and *PSet* have the highest fault detection with maximum test suites for *Clean*. However, for *Groovy* and *Piper*, *Blocking* and *Follows* have the highest fault detection with maximum achievable coverage test suites, respectively. Again, not only is there no best metric, but also there is no best type.

One possible reason why we observed no relationship between fault type and concurrency coverage metrics is because test requirements for concurrency coverage metrics do *not* capture concurrency faults *precisely*. To better understand why, consider Figure 15. In the figure, (a) and (b) show two executions that cover *Sync-Pair* requirement $\langle b2, b1 \rangle$ (i.e., a synchronization block $b2$ happens before a $b1$) where $b2$ is a synchronized block of Thread 2 (lines 11 to 14) containing `notifyAll(m)` and $b1$ is a synchronized block of Thread 1 (lines 2 to 5) containing `wait(m)`. Because `wait(m)` and `notifyAll(m)` should be used inside a synchronized block on $m$, we expect to detect the deadlock caused by calling `wait(m)` after `notify(m)` by covering the test requirements for *Sync-Pair* coverage, including $\langle b2, b1 \rangle$. However, no test requirement for *Sync-Pair* coverage is guaranteed to capture the deadlock situation precisely, as shown in Figure 15. In this case, both Figure 15(a) and Figure 15(b) cover $\langle b2, b1 \rangle$, but only Figure 15(b) raises a deadlock.

In contrast, to detect this specific deadlock fault, the sequence of data accesses on the variable `event1.count` is more important than the sequence of lock operations. Figure 15 shows that the fault appears when Thread 1 executes a waiting operation on the lock `m` (line 04) after Thread 2 executes a notification on the same lock (line 13). The fault detection depends on the sequence of data accesses on `event1.count` (i.e., line 01 $\rightarrow$ line 12 $\rightarrow$ line 03). We suspect that this is the reason that the data access coverage metrics PSet and Def-Use show high correlation with the fault detection for *Clean*.

This case implies that not only the coverage metric that captures a faulty thread interaction is important for fault detection but also the coverage metric that captures execution paths up to the faulty thread interaction is important.

Such issues on concurrency coverage metrics again highlight the need to better understand how to capture what represents effective testing. Additionally, they help explain why using multiple concurrency coverage criteria, per Section 4.5, can be an effective strategy to improve fault detection.




(a) a correct execution that covers a *Sync-Pair* test requirement $\langle b2, b1 \rangle$

(b) an execution that covers a *Sync-Pair* test requirement $\langle b2, b1 \rangle$, which raises a deadlock

Figure 15. Two execution scenarios of *Clean*.



DOI: 10.1002/stvr

### *5.4. Implications for concurrent test case generation research*

Work on test case generation methods for concurrency testing is an active — but relative to work on sequential testing — young area of research. In sequential test case generation, several techniques focus on methods for satisfying difficult-to-cover test requirements (e.g., symbolic execution, genetic approaches), and many, if not most approaches, center around a single metric, branch coverage. In contrast, current approaches to concurrent test generation have little ability to target specific difficult-to-cover requirements, and the coverage metric used to evaluate these approaches has not been standardized.

Given this, it seems reasonable to consider whether, as in sequential testing, effort to develop new techniques for covering difficult-to-cover requirements is warranted, and if so, what coverage metric(s) should be targeted. We have already largely addressed the latter question earlier in Section 5.1: *PSet*, combined with any of the three pairwise, synchronized metrics already proposed, offers the most consistently high levels of fault detection. As noted previously in Sections 4.3 and 5.3, however, there exist additional factors that current concurrency coverage metrics fail to capture. Thus, future work on concurrent test generation could be greatly improved by first considering how we can better (or perhaps more consistently) capture effective concurrent testing as a metric.

The answer to the former question — whether to target difficult test requirements — is similarly ambiguous. Given our results for *RQ4*, it seems that while in some cases, difficult requirements do offer improved fault detection relative to other requirements (e.g., for the *Arraylist* object), in most cases, no statistically significant improvements were found. Nevertheless, no statistically significant decreases in fault detection were observed, and thus, if a test generation method that increases the likelihood of satisfying difficult requirements could be found, it would certainly improve testing effectiveness. Of course, the details of any new technique — specifically, whether the technique would slow the overall rate of test case generation — would determine whether it represents an improvement over existing approaches; there is little doubt that the potential to improve fault detection via targeting of difficult requirements exists.

## 6. CONCLUSION

In this work, we have evaluated the relationship between eight previously proposed concurrency coverage metrics and fault detection effectiveness using 12 concurrent programs drawn from previous work in concurrency testing. We observed moderate correlations between coverage and fault detection effectiveness, established via linear regression that each coverage metric has a predictive value separate from test suite size, and found statistically and practically significant increases in fault detection effectiveness when using test suites reduced to achieve maximum coverage relative to random test suites of equal size. In addition, we confirmed that combinations of these coverage metric provide more reliable performance across different programs, particularly with respect to test generation, and that difficult-to-cover test requirements may be particularly effective with respect to fault detection. These results demonstrate that existing concurrency coverage metrics — in particular combinations of *PSet* and a pairwise synchronization-based coverage metrics — can be effective metrics for evaluating concurrency testing effectiveness and thus provide key evidence supporting the construction of techniques based on these metrics.

Nonetheless, while each metric explored was useful in some contexts, the predictive and test case generation value of each metric, even combined metrics that were proposed specifically to avoid this variation, often varied considerably from program to program, indicating that more work in this area is required. We hope to explore methods for improving these metrics in the future and encourage others to do the same.

## APPENDIX:

In this appendix, we present the results (discussed in Section 4) for all study objects.



DOI: 10.1002/stvr

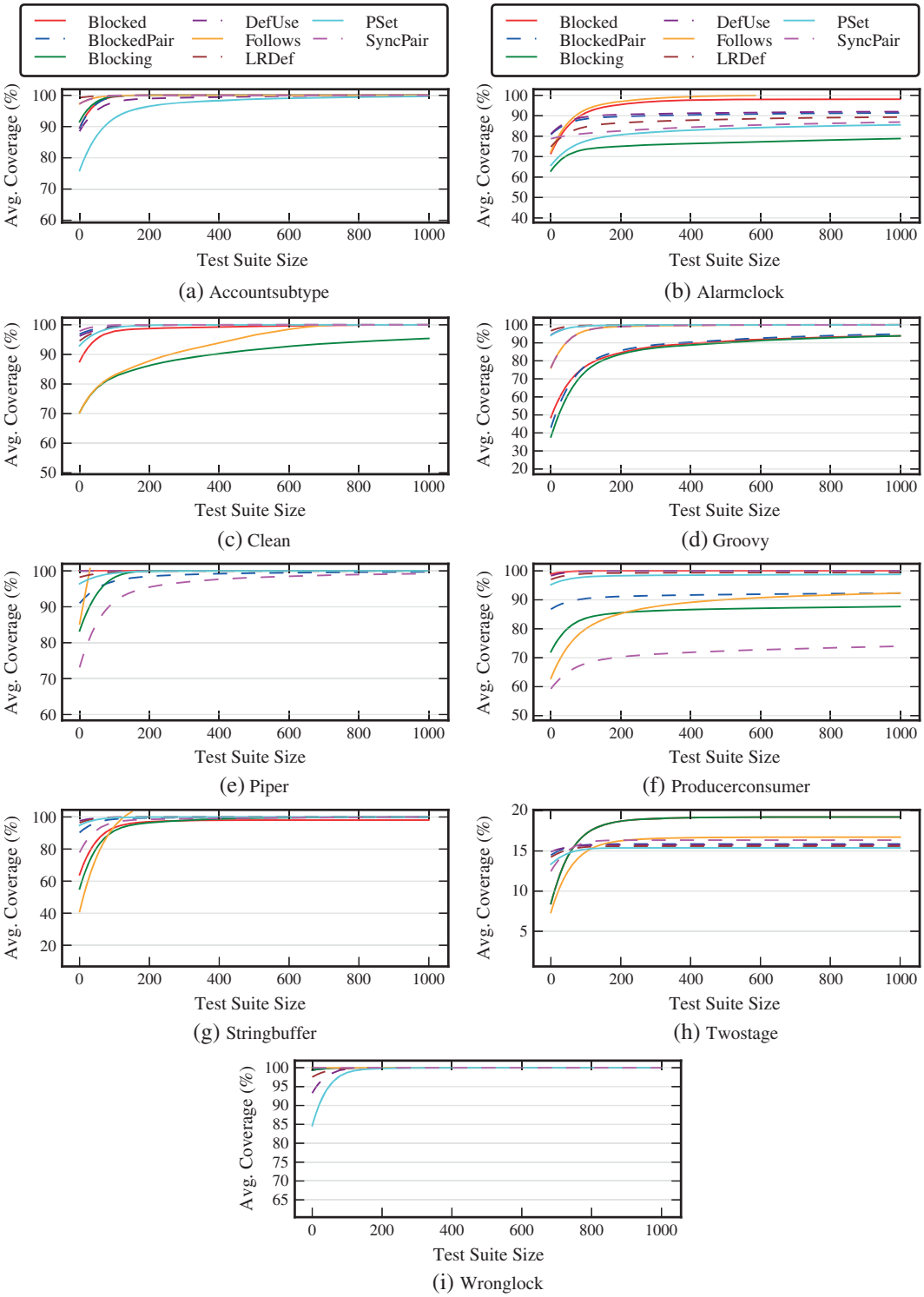



Figure A.1. Size versus coverage, all single-fault objects.




DOI: 10.1002/stvr

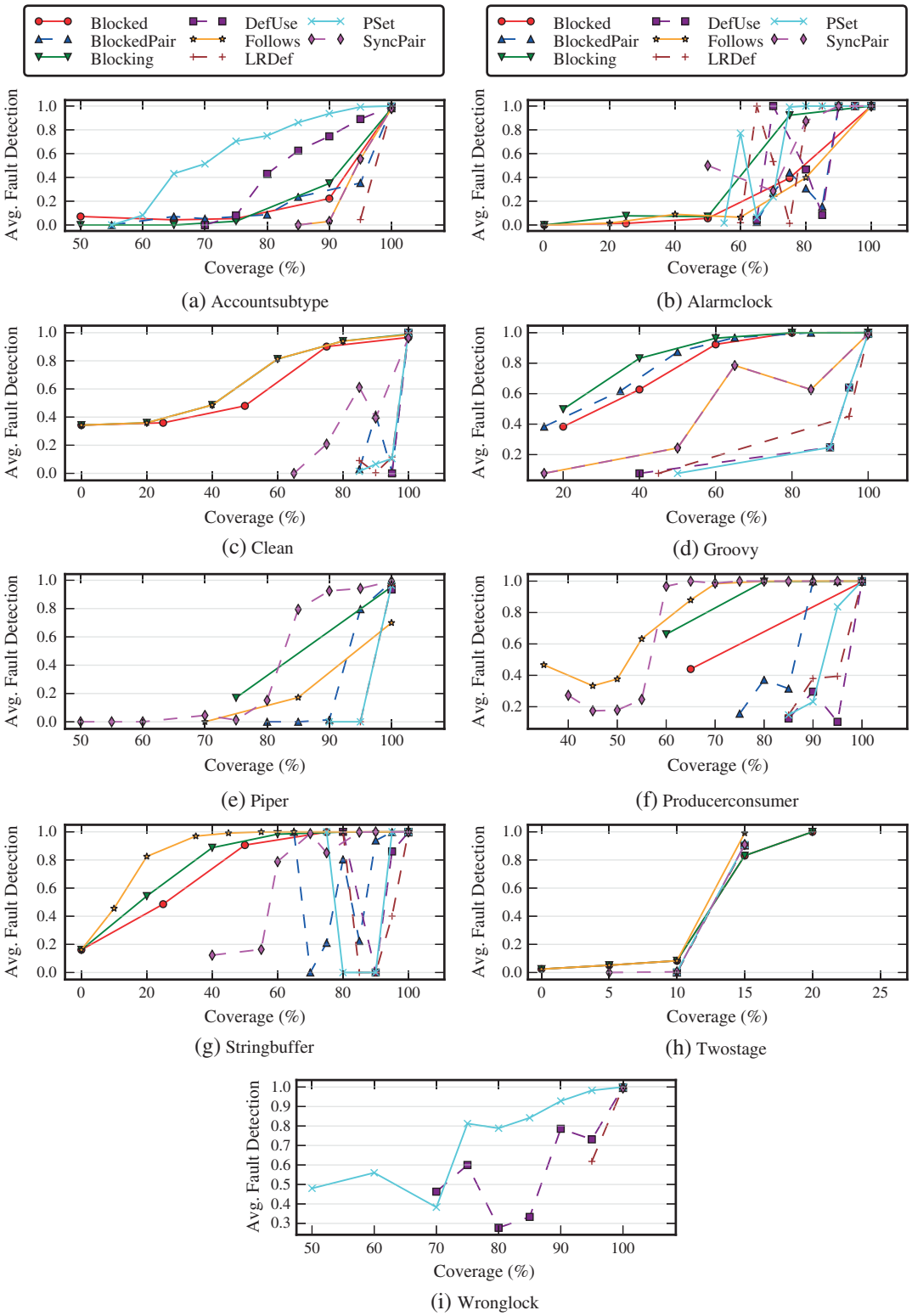



Figure A.2. Coverage versus fault detection effectiveness, all single-fault objects.




DOI: 10.1002/stvr

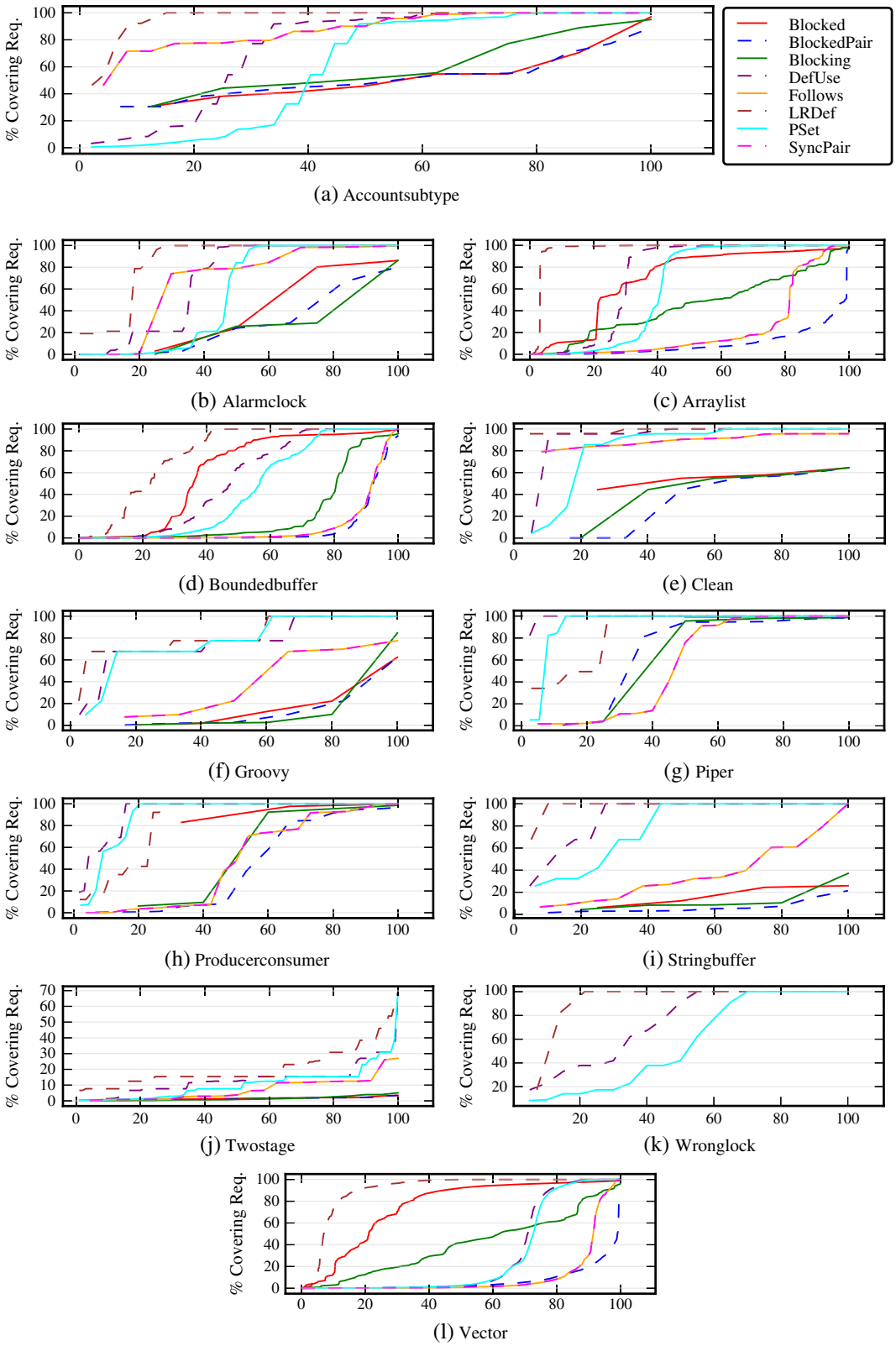



Figure A.3. Percentage of test executions covering test requirements, sorted, all single-fault and mutation objects.



DOI: 10.1002/stvr

ACKNOWLEDGEMENTS

This work is supported in part by the National Research Foundation of Korea (NRF) Mid-career Research Program funded by the Ministry of Science, ICT and Future Planning (MSIP), Korea (NRF-2012R1A2A2A01046172), the IT R&D Program of Ministry of Knowledge Economy (MKE)/Korea Evaluation Institute of Industrial Technology(KEIT), Korea (10041752), the Information Technology Research Center (ITRC) support program funded by MSIP and supervised by the National IT Industry Promotion Agency (NIPA), Korea (NIPA-2014-H0301-14-1023), the World Class University program through the NRF funded by the Korean Ministry of Education, Science and Technology (MEST), Korea (R31-30007), the National Science Foundation through award CNS-0720757, the Air Force Office of Scientific Research through award FA9550-10-1-0406, and the Fonds National de la Recherche, Luxembourg (FNR/P10/03).

REFERENCES

1. Savage S, Burrows M, Nelson G, Sobalvarro P, Anderson T. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)* 1997; **15**(4):391–411.
2. Engler D, Ashcraft K. RacerX: effective, static detection of race conditions and deadlocks. *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, New York, NY, USA, 2003; 237–252.
3. Hong S, Kim M. Effective pattern-driven concurrency bug detection for operating systems. *Journal of Systems and Software (JSS)* 2013; **86**(2):377–388.
4. Bron A, Farchi E, Magid Y, Nir Y, Ur S. Applications of synchronization coverage. *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Chicago, Illinois, USA, 2005; 206–212.
5. Lu S, Jiang W, Zhou Y. A study of interleaving coverage criteria. *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering(ESEC/FSE)*, Dubrovnik, Croatia, 2007; 533–536.
6. Trainin E, Nir-Buchbinder Y, Tzoref-Brill R, Zlotnick A, Ur S, Farchi E. Forcing small models of conditions on program interleaving for detection of concurrent bugs. *Proceedings of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD)*, Chicago, Illinois, USA, 2009.
7. Yang CD, Souter AL, Pollock LL. All-du-path coverage for parallel programs. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, Clearwater Beach, Florida, USA, 1998; 153–162.
8. Tracey N, Clark J, Mander K, McDermid J. An automated framework for structural test-data generation. *Proceedings of the IEEE International Conference on Automated Software Engineering (ASE)*, Honolulu, HI, 1998; 285–288.
9. Godefroid P, Klarlund N, Sen K. DART: directed automated random testing. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Chicago, IL, USA, 2005; 213–223.
10. Pacheco C, Lahiri SK, Ernst MD, Ball T. Feedback-directed random test generation. *Proceedings of the International Conference on Software Engineering (ICSE)*, Minneapolis, USA, 2007; 75–84.
11. Cadar C, Dunbar D, Engler D. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, USA, 2008; 209–224.
12. Edelstein O, Farchi E, Nir Y, Ratsaby G, Ur S. Multithreaded Java program test generation. *IBM Systems Journal* 2002; **41**(1):111–125.
13. Křena B, Letko Z, Vojnar T, Ur S. A platform for search-based testing of concurrent software. *Proceedings of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD)*, Trento, Italy, 2010; 48–58.
14. Wang C, Said M, Gupta A. Coverage guided systematic concurrency testing. *Proceedings of the International Conference on Software Engineering (ICSE)*, Waikiki, Honolulu, Hawaii, USA, 2011; 221–230.
15. Hong S, Ahn J, Park S, Kim M, Harrold M J. Testing concurrent program to achieve high synchronization coverage. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, Minneapolis, MN, USA, 2012; 210–220.
16. Hong S, Staats M, Ahn J, Kim M, Rothermel G. The impact of concurrent coverage metrics on testing effectiveness. *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST)*, Luxembourg, Luxembourg, 2013; 232–241.
17. Andrews JH, Briand LC, Labiche Y, Namin AS. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering (TSE)* 2006; **32**(8):608–624.
18. Namin AS, Andrews JH. The influence of size and coverage on test suite effectiveness. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, Chicago, IL, USA, 2009; 57–68.
19. Zhu H, Hall PAV, May JHR. Software unit test coverage and adequacy. *ACM Computing Surveys (CSUR)* 1997; **29**(4):366–427.
20. Tasiran S, Keremoglu ME, Muslu K. Location pairs: a test coverage metric for shared-memory concurrent programs. *Empirical Software Engineering (ESE)* 2012; **17**(3):129–165.
21. Xu Z, Kim Y, Kim M, Rothermel G, Cohen M. Directed test suite augmentation: techniques and tradeoffs. *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, Santa Fe, NM, USA, 2010; 257–266.



DOI: 10.1002/stvr

22. Dwyer MB, Person S, Elbaum SG. Controlling factors in evaluating path-sensitive error detection techniques. *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, Portland, Oregon, USA, 2006; 92–104.
23. Park CS, Sen K. Randomized active atomicity violation detection in concurrent programs. *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, Atlanta, Georgia, USA, 2008; 135–145.
24. Nistor A, Luo Q, Pradel M, Gross TR, Marinov D. BALLERINA: automatic generation and clustering of efficient random unit tests for multithreaded code. *Proceedings of the International Conference on Software Engineering (ICSE)*, Zurich, Switzerland, 2012; 727–737.
25. Sherman E, Dwyer MB, Elbaum S. Saturation-based testing of concurrent programs. *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering(ESEC/FSE)*, Amsterdam, The Netherlands, 2009; 53–62.
26. Yu J, Narayanasamy S. A case for an interleaving constrained shared-memory multi-processor. *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, Austin, Texas, USA, 2009; 325–336.
27. Bradbury JS, Cordy JR, Dingel J. Mutation operators for concurrency Java (J2SE 5.0). *Workshop on Mutation Analysis (MUTATION)*, Raleigh, North Carolina, USA, 2006; 11–20.
28. Do H, Rothermel G. A controlled experiment assessing test case prioritization techniques via mutation faults. *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, Budapest, Hungary, 2005; 411–420.
29. Stoller SD. Testing concurrent Java programs using randomized scheduling. *Proceedings of the International Workshop on Runtime Verification (RV)*, Copenhagen, Denmark, 2002; 142–157.
30. Burckhardt S, Kothari P, Musuvathi M, Nagarakatte S. A randomized scheduler with probabilistic guarantees of finding bugs. *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Pittsburg, PA, USA, 2010; 167–178.
31. Nagarakatte S, Burckhardt S, Martin MMK, Musuvathi M. Multicore acceleration of priority-based schedulers for concurrency bug detection. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Beijing, China, 2012; 543–554.
32. Park S, Lu S, Zhou Y. CTrigger: exposing atomicity violation bugs from their hiding places. *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Washington, DC, USA, 2009; 25–36.
33. Kvam PH, Vidakovic B. *Nonparametric Statistics with Applications to Science and Engineering*. Wiley: Hoboken, New Jersey, 2007.
34. Mallows CL. Some comments on *Cp*. *Technometrics* 1973; **15**(4):661–675.



DOI: 10.1002/stvr