2013 IEEE Sixth International Conference on Software Testing, Verification and Validation

The Impact of Concurrent Coverage Metrics on Testing Effectiveness

Shin Hong*, Matt Staats†, Jaemin Ahn*, Moonzoo Kim*, Gregg Rothermel†‡

*Department of Computer Science
KAIST
Daejeon, South Korea
{hongshin|jaemin|moonzoo}@cs.kaist.ac.kr

†Division of Web Science and Technology
KAIST
Daejeon, South Korea
staatsm@kaist.ac.kr

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

***Abstract*—When testing multithreaded programs, the number of possible thread interactions makes exploring all interactions infeasible in practice. In response, researchers have developed concurrent coverage metrics for multithreaded programs. These metrics allow them to estimate how well they have exercised concurrent program behavior, just as branch and statement coverage metrics do for sequential program testing. However, unlike sequential coverage metrics, the effectiveness of concurrent coverage metrics in testing remains largely unexamined. In this paper, we explore the relationship between concurrent coverage and fault detection effectiveness by studying the application of eight concurrent coverage metrics in testing nine concurrent programs. Our results show that existing concurrent coverage metrics are often moderate to strong predictors of concurrent testing effectiveness, and are generally reasonable targets for test suite generation. Nevertheless, their relative effectiveness as predictors and test generation targets varies across programs, and thus additional work is needed in this area.**

***Keywords*-Software testing; software metrics; concurrent programs.**

I. 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 concurrent fault detection techniques [1], [2], [3] have been developed as an alternative, these testing techniques have limited accuracy, and thus more systematic concurrent program testing approaches are desirable.

In response, researchers have developed concurrent coverage metrics for multithreaded programs [4], [5], [6], [7]. These metrics, like existing test coverage metrics defined over structural program elements such as branches or statements, define a set of requirements to be satisfied. In the case of concurrent coverage metrics, the coverage 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 the program's structure, concurrent coverage metrics allow engineers to estimate how well they have exercised concurrent program behaviors.

The intuition behind all test coverage metrics is that as more requirements for the metric are satisfied, the testing process becomes more likely to detect faults. Thus, to maximize the effectiveness of testing processes, researchers create test adequacy criteria based on these metrics, and develop techniques to quickly 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], [9], [10], [11], and as multithreaded programs have become more common the development of techniques centered around concurrent coverage metrics has also become an active area of research [12], [13], [14], [15].

Unfortunately, the intuition behind concurrent coverage metrics remains largely unexplored. While a large body of evidence exists exploring the impact of structural coverage metrics on testing effectiveness (e.g., [16], [17], [18]), we are aware of no study rigorously examining the impact of proposed concurrent coverage metrics. We expect that increasing concurrent coverage will improve testing effectiveness, but we also expect that it will increase test suite size. Thus we must ask: does improving concurrent coverage directly lead to a more effective testing process, or is it merely a byproduct of increasing test suite size? Furthermore, if improving coverage does lead to improvements, what practical gains in testing effectiveness can we expect?

To explore these questions, we studied the application of eight concurrent coverage metrics in testing nine concurrent programs. For each program and metric pairing, we used a randomized test case generation process to generate 100,000 test suites with varying levels of size and coverage, and measured the relationships between the percentage of coverage 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. 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 test generation adequacy criterion. However, in sharp contrast to work on sequential coverage metrics [17] and the intent of the concurrent metrics, the metrics' results are inconsistent,

978-0-7695-4968-2/13 $26.00 © 2013 IEEE
DOI 10.1109/ICST.2013.32

and vary across programs. In particular, we found that the correlation between concurrent 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 9 times more) can be found when using concurrent coverage metrics as test case generation criteria relative to random test suites of equal size, in some cases the results were no better than random testing. Finally, we found that we could improve our ability to model concurrency testing effectiveness by considering factors other than test suite size and coverage (e.g., test generation parameters), with improvements in linear regression fit up to 814% as measured by adjusted R^2.

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.

II. Background and Related Work

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 concurrent programs requires us 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].

In work on concurrent coverage metrics, the effectiveness of achieving high coverage levels has been argued for primarily through analytical comparisons between coverage definitions and bug patterns, such as those involving data races and atomicity violations [5], [6], [14]. Little empirical evidence exists demonstrating that high levels of coverage correlate with better fault detection ability.

Trainin et al. [6] note that concurrency faults are related to certain test requirements for the *blocked-pair* and *follows* concurrent coverage metrics, which suggests that achieving high levels of coverage should correlate with testing effectiveness. Wang et al. [14] highlight how data races 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. [19], 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 finding ability. The study uses two case examples and generates faulty versions via concurrent mutation operators and manual fault seeding. Our study's scope is more comprehensive, encompassing nine case examples and eight concurrent coverage metrics, and we apply a broader set of analyses.

III. Study Design

The purpose of this study is to rigorously investigate existing concurrent coverage metrics, 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, concurrent 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, we should 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.

Furthermore, to aide 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, the cost of achieving high levels of coverage, and the expected improvements over random testing.

Our study is thus designed to address two core questions.

Research Question 1 (RQ1): *For each concurrent 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 concurrent coverage metric studied, how does the fault detection effectiveness of test suites achieving maximum coverage compare to 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.

The objects for this study have been drawn from existing work on testing concurrent software [20], [21], and include objects without faults, and objects with faults detected in previous studies. We list the objects with the lines of code, numbers of threads, and mutants used in Table I.

A. Variables and Measures

1) Independent variables: In this study, we manipulate two independent variables: the concurrent coverage metric, and the method of test suite construction.

Concurrent Coverage Metrics. Numerous concurrent coverage metrics have been proposed, each based on some unique

TABLE I
STUDY OBJECTS

Type	Program	LOC	Num. threads	Incorrect versions	Test exec.
Mutation testing	ArrayList	5866	29	42	2000
	BoundedBuffer	1437	31	34	2000
	Vector	709	51	88	2000
Single fault program	Alarmclock	125	4	1	1000
	Clean	51	3	1	1000
	Piper	71	9	1	1000
	Producerconsumer	87	5	1	1000
	Stringbuffer	416	3	1	1000
	Twostage	52	3	1	1000

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 elements the metric is defined over (either synchronization elements or shared data access operations). For example, the *blocking* and *blocked* coverage metrics define requirements based on individual `synchronized` blocks/methods in a Java program [12], and are thus *singular* concurrent coverage metrics, while 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 are synchronization metrics [6].

We selected eight coverage metrics for use in our study, focusing on well-known coverage 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. [22]. *Access-pair* [22] and *location-pair* [19] 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.

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.

2) Dependent Variables: We measure three dependent variables computed over generated test suites: coverage achieved, test suite size, and fault detection effectiveness.

Achieved concurrent coverage. For a give 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 executions while holding random test generation parameters constant (see Section III-B); 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 than can be covered for a specific set of test generation parameters.

TABLE II
CONCURRENT COVERAGE METRICS USED IN STUDY

	Synchronization operation	Data access operation
Singular	*blocking* [12], *blocked* [12]	*LR-Def* [5]
Pairwise	*blocked-pair* [6], *follows* [6], *sync-pair* [15]	*PSet* [23], *Def-Use* [19]

Test suite size. Test suite size is the number of test executions in the test suite, and estimates testing cost.

Fault detection effectiveness. When computing the fault detection effectiveness of the testing process, we use concurrent mutation operators with correct objects (see Section III-B1). We then compute the fault detection effectiveness of a testing approach as the number of mutants killed/detected. When computing fault detection effectiveness for objects that contain known faults, detection of the fault is treated as success, and failure to detect the fault is treated as failure.

B. 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) perform resampling over executions to construct test suites and finally (5) measure the resulting coverage and fault detection effectiveness of each test suite.

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. Accordingly, for several of our object programs we corrected known faults [21] and applied mutation analysis. To choose mutation operators for our study, we drew on concurrent mutation operators used in a recent survey on concurrent mutation testing [24]. These operators are similar to traditional syntactic mutation operators commonly used in other studies [16], [25], but focus on manipulating concurrency constructs, e.g., adding/removing synchronization primitives. Table III describes the operators. We applied these operators to generate mutants. We then discarded any mutants that (1) did not fail for any generated test execution, (2) were malformed, e.g., resulted in code that could not be executed, or (3) were killed by every test execution.

We list the final number of mutants used in Table I. Note that we also use objects containing real faults, thus mitigating the risk present when using concurrent mutation operators, whose

TABLE III
MUTATION OPERATORS [24]

Category	Description
Change Synchronization Operations	Exchange Synchronized Block Parameter
	Remove `wait()`
	Replace `notifyAll()` with `notify()`
Modify Synchronized Block	Expand Synchronized Block
	Remove Synchronized Block
	Remove `synchronized` Keyword from Method
	Shift Synchronized Block
	Shrink Synchronized Block
	Split Synchronized Block

usage is less established and studied than structural mutation operators [16].

2) Test Generation and Execution: We used a randomized test case generation approach to avoid bias that might result from using a directed test case generation approach such as those proposed in [12], [26]. 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 msec, 10 msec, and 15 msec). We used these controls because previous 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 twelve 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. This was done 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 (=4×3+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 (1) the test requirements covered for each coverage metric studied, and (2) whether a fault was detected. We recorded an execution as detecting a fault if (1) an uncaught exception is thrown by the program (i.e., a crash fault), (2) the program deadlocks, determined by checking whether execution time is exceptionally long, or (3) a program-specific assertion is violated.

3) Data Collection: After each test execution we know (1) which test requirements are covered for each coverage metric and (2) whether the program failed. Using this 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, 100,000 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 address *RQ1*.

To address *RQ2*, 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 (1) the test execution satisfying the most unsatisfied requirements (80% chance) or (2) a random test execution (20% chance) until all requirements are satisfied. This results in a test suite that achieves maximum coverage using fewer test executions than are required by simple random test suite construction. The randomization adds noise, ensuring some variation in the generated suites.

Selecting a test suite for a single-fault program is straightforward: we have one set of executions over the program, and we resample from this set to construct test suites.[1] Each test suite becomes a data point for analysis, having an associated level of coverage, size, and fault detection result (killed/not killed).

The construction of test suites for objects using mutation generation is more complex. Each mutant differs in the synchronization primitives present, and thus we cannot replicate a sequence of interleavings (i.e., run the same test execution) across all mutants. Therefore, when constructing test suites for objects with mutants, we began by generating 100,000 separate test suites for each mutant. To compute the fault detection effectiveness of combinations of coverage levels and sizes across mutants, we randomly selected a mutant and a test suite associated with that mutant. Following this, for all remaining mutants we randomly selected test suites with the same (or as similar as possible) level of coverage and size, and computed the average coverage and size (which may vary slightly across mutants) and the number of mutants detected. These aggregated values become a data point for analysis. We repeated this cross-mutant selection 100,000 times.

C. Threats to Validity

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 concurrent testing research focuses on.

For concurrent coverage metrics, it is difficult to accurately determine satisfiable requirements. For all coverage metrics, however, we appear to have reached saturation during test case generation (see Sec. IV-A) [22], and thus a larger number of executions is unlikely to significantly alter our results.

[1]When constructing each test suite, we held probability and delay constant. We did this to facilitate later analysis considering the impact of these factors.

The random testing technique we use is implemented in-house, but we have attempted to match the behavior of other random testing techniques by constructing a general technique and varying the parameters of probability and delay. Our study employs only a single test input value, varying scheduling, and thus we do not consider the impact of test input value on concurrent testing. However, most concurrent testing techniques assume that intensive testing is required for each test input value, and thus our study reflects the current approach to concurrent testing.

Internal: Our random testing technique is implemented upon Java's internal thread scheduler, and when using other thread schedulers 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.

Construct: Our method of detecting faults may miss faults, e.g., 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 used mutation analysis to measure testing effectiveness for some objects. Our seeded faults are designed to mimic actual concurrent faults, and of course are indeed faults, but the relationship between faults generated by concurrent mutation operators and real concurrency faults has not been thoroughly investigated. Nevertheless, the results for mutation-based objects and objects with real faults are similar.

Conclusion: For each object, we constructed from 1 to 88 faults and 100,000 test suites per coverage metric. While more mutants/faults/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.

IV. Results 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.[2]

A. 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 Figures 1 and 2 we show how each pair of factors interacts for the *Vector* and *Stringbuffer* objects (elements on the y-axis are average coverage and fault detection levels for elements on the x-axis). Note that the figures for each object vary, in particular those figures relating to single-fault objects (which have significant levels of noise). However these figures capture the core behavior typical across all objects: an initial rapid increase in fault detection (coverage) as coverage (size) increases, followed by a continued, but subdued increase for higher coverages or larger sizes.

The concurrent coverage metrics exhibit behavior similar to what we expect from sequential coverage metrics and testing: broadly logarithmic behavior, with a rapid increase in both fault detection and coverage for small test suite sizes, and smaller increases as test suite size grows. Here we see strong 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 both programs. In constrast, the relationship between coverage and fault detection is positive and essentially linear. Here differences are mostly related to the number of "easy" requirements to satisfy – those metrics that are easier to satisfy have high coverage even for very small test suites, e.g., *blocking*, *blocked*, *LR-Def* when used with *Vector*. For single-fault programs (like *Stringbuffer*), there are fewer requirements and only a single fault, and thus the increase observed is less consistent (though the relationship is positive overall).

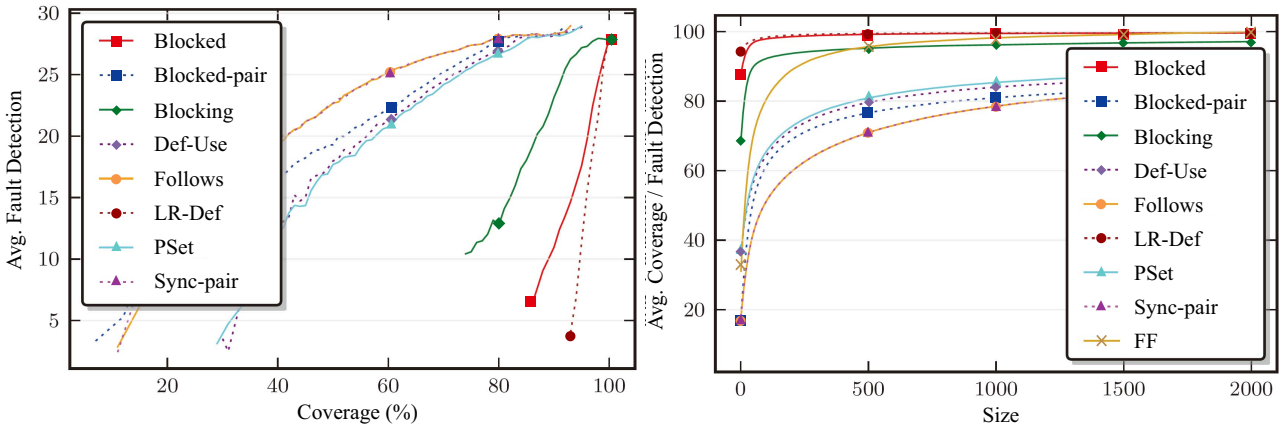
Perhaps most striking is the tendency for metrics to cluster, exhibiting similar levels of coverage for all test suite sizes. These clusters correspond roughly to the metric properties of *singular/pairwise* and *synchronization/data access* first discussed in Section II. This grouping is natural: *pairwise* metrics, being defined over pairs of elements rather than single elements, naturally have more requirements and thus require more tests. Metrics defined over a specific type of primitive are likely to have similar fault detection effectiveness and coverage behavior, but their fault detection effectiveness varies depending on program behavior. For example, for *Vector*, the *sync-pair* and *follows* metrics — both pairwise, synchronization-based metrics — achieve coverage the most slowly, with a rate of increase similar to that of fault detection.

B. Correlation Between Variables

Figures 1 and 2 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.[3] We selected Pearson's r for two reasons. First, we are interested in the application of concurrent 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

[2]Applying these analyses to nine objects and eight coverage metrics produces more results than we can include; we have summarized the data and released the full data at http://pswlab.kaist.ac.kr/data/conc-cov-impact

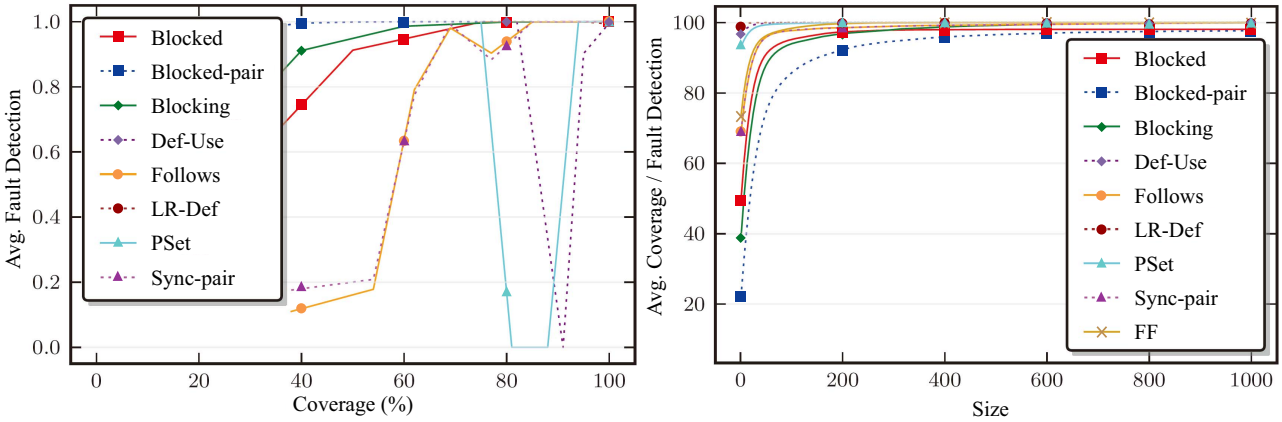
[3]Note that while for small samples conclusions based on Pearson's can be unsound for non-normal data, in our case the use of very large number of samples, 100,000 per correlation computed, mitigates this risk.



(a) Coverage vs Fault Detection Effectiveness

(b) Size vs Coverage + Fault Detection Effectiveness

Fig. 1. Relationship Between Variables, Vector



(a) Coverage vs Fault Detection Effectiveness

(b) Size vs Coverage + Fault Detection Effectiveness

Fig. 2. Relationship Between Variables, Stringbuffer

new information [27]. 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 are presented in Table IV. For example, for *ArrayList* the correlation between *blocked* coverage and fault detection/test suite size is (0.77,0.33), while the correlation between size and fault detection (*S-FF*) is 0.48. Across all programs using mutation faults, the correlation between each pair of variables in mutation testing is moderate to high (above 0.52) for all programs and coverage metrics. However, for every single-fault program, several metrics have low correlation with fault detection. We believe this occurs because the metric's intuition does not capture the single fault present. In contrast, the diversity of mutation-derived faults leads to more consistent correlations for other objects.

For each metric there exists at least one program for which the correlation with fault detection is at or above 0.87. Furthermore, coverage is often more strongly correlated with fault detection than 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, though *PSet* with a low of 0.27 for *producerconsumer* is often quite strong. This indicates that selecting an effective metric for a given program may be challenging. Furthermore, 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 concurrent coverage metrics may relate to fault detection effectiveness.

C. 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 can also see that test suite size and coverage are often similarly correlated, and thus the relationship between size, coverage and fault detection is unclear; does coverage predict fault detection effectiveness, or merely reflect test suite size? To address this we used linear regression to attempt to model how test suite size and coverage

TABLE IV
CORRELATIONS: EACH CELL = (COVERAGE & FAULT DETECTION CORR., SIZE & COVERAGE CORR.), S-FF = SIZE & FAULT DETECTION CORR.

	blocked	blocked-pair	blocking	Def-Use	follows	LR-Def	PSet	sync-pair	S-FF
ArrayList	0.77 , 0.33	0.81 , 0.55	0.71 , 0.51	0.75 , 0.49	0.79 , 0.53	0.84 , 0.39	0.81 , 0.54	0.79 , 0.53	0.48
BoundedBuffer	0.55 , 0.44	0.60 , 0.45	0.54 , 0.35	0.60 , 0.50	0.60 , 0.46	0.52 , 0.39	0.61 , 0.53	0.60 , 0.46	0.38
Vector	0.82 , 0.40	0.91 , 0.64	0.76 , 0.39	0.88 , 0.65	0.86 , 0.70	0.91 , 0.51	0.89 , 0.66	0.87 , 0.70	0.54
Alarmclock	0.77 , 0.25	0.67 , 0.29	0.28 , 0.22	0.55 , 0.22	0.19 , 0.26	0.19 , 0.12	0.59 , 0.35	0.19 , 0.26	0.05
Clean	0.15 , 0.16	0.17 , 0.41	0.19 , 0.40	0.97 , 0.29	0.10 , 0.05	0.0 , 0.00	0.85 , 0.29	0.11 , 0.05	0.30
Piper	0.01 , 0.00	0.59 , 0.49	0.48 , 0.25	0.06 , 0.02	0.62 , 0.45	0.20 , 0.09	0.67 , 0.27	0.62 , 0.45	0.38
Producerconsumer	0.14 , 0.02	0.19 , 0.43	0.13 , 0.16	0.55 , 0.15	0.10 , 0.20	0.63 , 0.29	0.27 , 0.25	0.09 , 0.19	0.11
Stringbuffert	0.57 , 0.18	0.42 , 0.35	0.56 , 0.30	0.42 , 0.11	0.65 , 0.23	0.0 , 0.04	0.87 , 0.15	0.63 , 0.22	0.12
Twostage	0.87 , 0.23	0.87 , 0.23	0.87 , 0.23	0.95 , 0.13	0.96 , 0.13	0.04 , 0.02	0.96 , 0.13	0.96 , 0.13	0.10

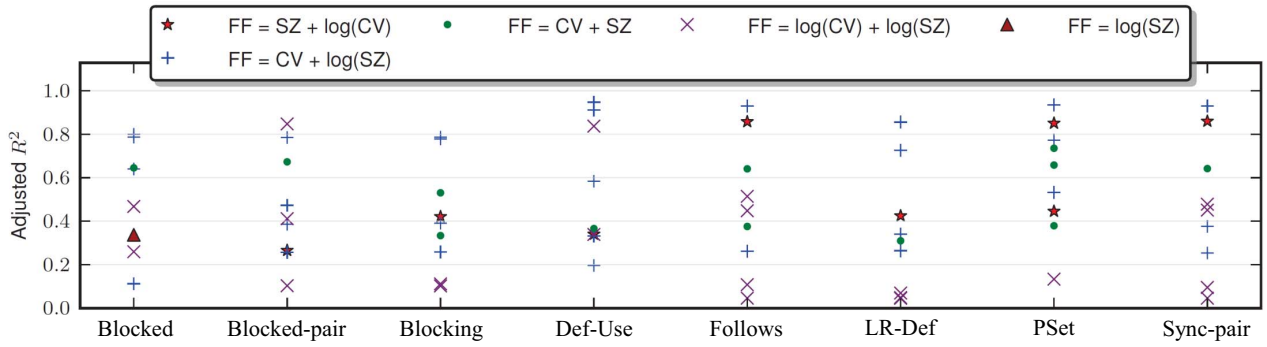


Fig. 3. Adjusted R^2 for Every Best Fit Model, All Combinations of Objects & Coverage Metrics

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. 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 this case we would like to model fault detection effectiveness for each object and coverage metric using test suite size and/or coverage 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 *TS*, *log(TS)*, *CV*, and *log(CV)* as explanatory variables (with fault detection (*FF*) as the dependent variable).

We next computed the adjusted R^2 to determine the goodness of fit for each model; this 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.[4] 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.

Our fitting process results in a large number of 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 3 for each coverage metric, across all objects, indicating which set of explanatory variables had the highest fit. For example, we see that for the *sync-pair* metric, for two objects adjusted R^2 was less than 0.2, indicating very low fit, while for one object fit was roughly 0.9, suggesting a high fit with model $FF = \alpha \times CV + \beta \times log(SZ)$. Here we can clearly see the variation in metric effectiveness, with fit 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.

Our results indicate that while no single set of explanatory variables is best, in all instances but one, models based on both coverage and size are preferable to models using only one explanatory variable. This provides evidence that coverage metrics have a predictive ability separate from test suite size. However, 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

[4]We also used Mallow's C_p to determine goodness of fit [28]. The results when using Mallow's led to the same conclusions, and we have presented results using adjusted R^2 as we believe this metric is easier to interpret.

TABLE V
MAXIMUM ACHIEVABLE COVERAGE TEST SUITE STATISTICS.

(* = Not statistically significant difference at $\alpha = 0.05$)
(MFF = MAX CV FF, RFF = Random FF, Cv = % Increase in Coverage Over Random, Sz = Test Suite Size)

	blocked				**blocked-pair**				**blocking**				**Def-Use**			
	MFF	**RFF**	**Cv**	**Sz**	**MFF**	**RFF**	**Cv**	**Sz**	**MFF**	**RFF**	**Cv**	**Sz**	**MFF**	**RFF**	**Cv**	**Sz**
ArrayList	7.15	0.75	24.2%	2.02	7.23	4.21	50.5%	26.2	5.76	1.19	37.6%	3.26	7.46	1.73	24.0%	5.18
BoundedBuffer	3.38	3.17	32.5%	4.35	4.26	3.96	105%	43.9	3.69	3.49	122%	5.64	4.38	3.61	26.3%	10.9
Vector	7.78	8.33	14.8%	3.25	23.6	22.3	38.5%	97.4	9.06	10.1	31.5%	5.40	27.8	18.2	64.4%	33.1
Alarmclock	0.92	0.06	94.0%	1.17	0.92	0.20	100%	1.67	0.29	0.12	103%	1.43	0.92	0.25	21.2%	3.23
Clean	0.0*	0.02*	58.9%	1.25	0.0	0.13	93.8%	2.13	0.0*	0.02*	65.0%	1.76	1.0	0.09	6.13%	2.01
Piper	0.0*	0.0*	2.24%	1.0	0.38	0.0	43.8%	1.24	0.15	0.03	31.8%	1.17	0.0	0.03	0.80%	1.02
Producerconsumer	0.08	0.19	5.81%	1.03	0.59*	0.56*	57.8%	3.81	0.46	0.16	39.5%	1.48	1.0	0.21	6.79%	1.14
Stringbuffer	0.83	0.57	230%	2.06	1.0	0.87	229%	6.27	0.98	0.63	306%	2.37	0.04	0.59	7.28%	2.04
Twostage	0.92	0.14	366%	2.93	0.92	0.12	407%	2.95	0.92	0.07	379%	2.93	0.92	0.21	6.95%	2.92

	follows				**LR-Def**				**PSet**				**sync-pair**			
	MFF	**RFF**	**Cv**	**Sz**	**MFF**	**RFF**	**Cv**	**Sz**	**MFF**	**RFF**	**Cv**	**Sz**	**MFF**	**RFF**	**Cv**	**Sz**
ArrayList	7.23	4.06	47.0%	20.2	7.46	0.78	2.68%	1.41	7.38	2.72	28.1%	8.56	7.23	3.94	46.8%	20.3
BoundedBuffer	4.23	3.95	87.0%	42.7	2.72	2.93	13.3%	2.96	4.38	3.80	32.4%	17.7	4.17	4.04	85.0%	42.4
Vector	21.0	23.1	56.2%	121	27.8	7.85	5.15%	2.93	27.8	19.7	53.8%	45.5	20.7	22.9	62.3%	121
Alarmclock	0.49	0.16	16.8%	1.79	0.12	0.17	12.1%	2.01	0.92	0.26	38.8%	4.51	0.47	0.17	13.4%	1.75
Clean	0.0	0.04	16.1%	1.09	0.0*	0.0*	0.0%	1.0	1.0	0.08	13.9%	2.18	0.01	0.04	13.5%	1.02
Piper	0.69	0.05	53.4%	3.0	0.0	0.12	14.6%	1.21	0.46	0.01	6.01%	1.25	0.68	0.04	48.7%	3.0
Producerconsumer	0.50	0.57	29.3%	3.65	1.0	0.29	16.9%	1.60	1.0	0.32	6.60%	1.68	0.47	0.53	28.0%	3.65
Stringbuffer	1.0	0.75	41.5%	4.68	0.08	0.34	1.56%	1.08	1.0	0.59	10.3%	3.0	1.0	0.86	37.0%	4.82
Twostage	0.92	0.15	55.8%	2.92	0.0	0.06	2.29%	1.06	0.92	0.12	22.8%	2.92	0.92	0.12	47.5%	2.92

models.[5] We discuss this further in Section V-B.

D. Effectiveness of Maximum Coverage

Our first three analyses have characterized the relationship between test suite size, coverage and fault detection 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 case examples, all coverage metrics do appear to have value. Thus, it is worthwhile to use concurrent coverage metrics (in addition to test suite size) as methods for estimating the concurrent fault detection effectiveness of a testing process.

However, per *RQ2*, we also would like to quantify the ability of test suites generated to quickly achieve high levels of concurrent coverage. To do this, for each case example and coverage metric, we compared test suites of maximum achievable coverage, generated using a greedy algorithm described in Section III-B3, against random test suites of equal size. The 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 cases 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 case example and coverage criteria 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 [27].[6]

Each test suite generated to achieve maximum achievable coverage (hereafter referred to as maximum coverage) 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 [27]. 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 V lists the results of this analysis.[7] Initially we were surprised that there were object / coverage metric pairs for which the reduction to maximize coverage had a *negative* impact on the fault detection effectiveness of the testing process. For example, we see for *Stringbuffer* that test suites reduced to satisfy *LR-Def* found the fault 8% of the time, as compared to 34% when using random test suites of equal size. However, these scenarios correspond to very low correlations with fault detection ($0.3 <$) and are thus reasonable. The case of *Vector* was more surprising. We hypothesize that when achieving maximum coverage for complex coverage metrics (e.g. *sync-pair*), there exist several hard-to-cover test requirements which are satisfied only by specific test executions that do not detect mutants. 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.

Nevertheless, achieving high coverage generally yields not only statistically significant, but also practically significant increases in fault detection: large, often twofold or more increases can be observed. For example, for the *ArrayList* object, we observed increases in average fault detection of 1.7 to 9.5 times at maximum coverage. Similar patterns can be seen for many combinations of objects and coverage metrics,

[5]Computed R^2 values, omitted for space reasons, support this statement.

[6]The null hypothesis H_0 is that test suites achieving maximum achievable coverage are equally effective as random test suites of equal size.

[7]For single fault programs fault detection is the ratio of test suites detecting the fault to the total number of test suites.

providing evidence that achieving high concurrent coverage has merit.

V. Discussion

Our results have addressed our original research questions *RQ1* and *RQ2*: for every coverage metric, we have shown that for some programs (1) the metric is a moderate, independent predictor of fault detection, and (2) 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 concurrent coverage metrics can be useful. Consequently, testers can use concurrent 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 concurrent 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. In the remainder of this section, we discuss the practical implications of the study and highlight additional areas of research that we believe must be explored.

A. Practical Implications for Testers

Following a study of several coverage metrics, the question every tester naturally asks is: *which metric should* I *use?* Examining Tables IV and V, we see that if a tester must select a "best" metric, *PSet* seems to be the only possible choice. For eight objects, *PSet* coverage's correlation with fault detection is over 0.59. Additionally, *PSet* always achieves a greater correlation with fault detection than size (*S-FF*), and is the only metric whose greedy reduced test suites always achieve better fault detection than random test suites of equal size. *PSet* is clearly not ideal in many scenarios – *Def-Use* is similarly effective as a generation target for *Vector* while requiring fewer test executions and *sync-pair* is more effective as a generation target for *Alarmclock* – but on the whole it is consistently effective as both 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 (*synchronized/data access*) constant reveals two patterns hinted at in Section IV-A. Generally speaking, the correlation with fault detection for pairwise metrics tends to be approximately equal or higher (sometimes much higher, e.g., *Def-Use* and *PSet* versus *LR-Def*) than when using singular metrics. Thus as predictors of testing effectiveness, it is preferable to select pairwise metrics.

Second, pairwise metrics also excel as targets for test case generation. For example, for *BoundedBuffer*, singular metrics detect on average no more than 3.69 faults, while all pairwise metrics detect on average at least 4.17 faults. For *Clean*, only two metrics, both pairwise, detect the fault with probability over 0.01 (*PSet* and *Def-Use*). Of course, as noted previously, pairwise metrics have more requirements, and thus require more test executions to achieve maximum coverage. Nevertheless, even if for budget reasons we must settle for test suites with less than maximum coverage, given the relatively strong relationship between pairwise metric coverage and fault detection, we see no reason to select singular metrics. Instead we recommend achieving as much coverage as possible with a pairwise metric, rather than, for example, achieving maximum coverage for a singular metric and then stopping testing or switching to simple random testing.

Finally, for some objects, there is a large difference in fault detection depending on the primitive (*synchronization/data access*) used to define the metrics. For example, when using data access-based metrics (*PSet*, *Def-Use* and *LR-Def*) with *producerconsumer*, the correlation with fault detection is roughly three times that of synchronization-based metrics, with similarly dramatic increases in fault detection over random testing at maximum coverage. However, for *Piper* the opposite is true; data-access based metrics perform poorly.

Thus, while *PSet* is perhaps the most consistent metric, our results offer no universal recommendation — in addition to the usual caveats (e.g., the choice of metric depends on testing budget, testing goals, etc.) metrics that appear excellent in some circumstances perform poorly in others, with corresponding variance in relative effectiveness. We found this surprising: while in theory such behavior 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. This provides evidence that calls to use concurrency metrics in tandem [14] — specifically *PSet* and *follows* — should be heeded, and that additional work to understand how to select metrics is required.

B. Limits of Existing Concurrency Metrics

As noted, in some cases the concurrent coverage metrics explored exhibit 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 [16], [17]. 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 improve our concurrent 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 III-B2). 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 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 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.

VI. Conclusion and Future Work

In this work, we have evaluated the relationship between eight concurrency coverage metrics and fault detection effectiveness using nine concurrent programs drawn from previous work in concurrency testing. We observed moderate correlations between coverage and fault detection effectiveness (up to 0.96), 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. These results thus demonstrate that existing concurrent coverage metrics — in particular *PSet* — 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 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.

VII. Acknowledgements

This work is supported in part by the NRF (2012046172), the World Class University program under the National Research Foundation of Korea and funded by the Ministry of Education, Science and Technology of Korea (Project No: R31-30007), the National Science Foundation through award CNS-0720757, and the Air Force Office of Scientific Research through award FA9550-10-1-0406.

References

[1] S. Savage, M. Burrows, , G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Trans. Comput. Syst.*, no. 4, 1997.

[2] D. Engler and K. Ashcraft, "RacerX: effective, static detection of race conditions and deadlocks," in *SOSP*, 2003.

[3] S. Hong and M. Kim, "Effective pattern-driven concurrency bug detection for operating systems," *Jour. Sys. Softw.*, vol. 86, no. 2, 2013.

[4] A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur, "Applications of synchronization coverage," in *Symp. Princ. Prac. of Paral. Prog. (PPoPP)*, 2005.

[5] S. Lu, W. Jiang, and Y. Zhou, "A study of interleaving coverage criteria," in *Euro. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, 2007.

[6] E. Trainin, Y. Nir-Buchbinder, R. Tzoref-Brill, A. Zlotnick, S. Ur, and E. Farchi, "Forcing small models of conditions on program interleaving for detection of concurrent bugs," in *Works. Paral. Dist. Sys. Test. Analy. Debug. (PADTAD)*, 2009.

[7] C. D. Yang, A. L. Souter, and L. L. Pollock, "All-du-path coverage for parallel programs," in *Int'l Symp. Softw. Test. Analy. (ISSTA)*, 1998.

[8] N. Tracey, J. Clark, K. Mander, and J. McDermid, "An automated framework for structural test-data generation," in *Int'l Conf. on Autom. Softw. Eng. (ASE)*, 1998.

[9] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," *Conf. on Prog. Lang. Desig. Impl. (PLDI)*, 2005.

[10] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Int'l Conf. on Softw. Eng. (ICSE)*, 2007.

[11] C. Cadar, D. Dunbar, and D. Engler, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs," in *Conf. on Oper. Sys. Desig. Impl. (OSDI)*, 2008.

[12] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur, "Multithreaded java program test generation," in *Conf. Java Grande*, 2001.

[13] B. Křena, Z. Letko, T. Vojnar, and S. Ur, "A platform for search-based testing of concurrent software," in *Works. Paral. Dist. Sys. Test. Analy. Debug. (PADTAD)*, 2010.

[14] C. Wang, M. Said, and A. Gupta, "Coverage guided systematic concurrency testing," in *Int'l Conf. Softw. Eng. (ICSE)*, 2011.

[15] S. Hong, J. Ahn, S. Park, M. Kim, and M. J. Harrold, "Testing concurrent program to achieve high synchronization coverage," in *Int'l Symp. on Software Testing and Analysis (ISSTA)*, 2012.

[16] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Trans. Softw. Eng. (TSE)*, vol. 32, no. 8, Aug. 2006.

[17] A. S. Namin and J. H. Andrews, "The influence of size and coverage on test suite effectiveness," in *Int'l Symp. Softw. Test. Analy. (ISSTA)*, 2009.

[18] H. Zhu, P. Hall, and J. May, "Software unit test coverage and adequacy," *ACM Computing Surveys (CSUR)*, vol. 29, no. 4, 1997.

[19] S. Tasiran, M. E. Keremoglu, and K. Muslu, "Location pairs: a test coverage metric for shared-memory concurrent programs," *Empirical Software Engineering (ESE)*, vol. 17, no. 3, 2012.

[20] M. B. Dwyer, S. Person, and S. G. Elbaum, "Controlling factors in evaluating path-sensitive error detection techniques," in *Symp. Found. Softw. Eng. (FSE)*, 2006.

[21] C. S. Park and K. Sen, "Randomized active atomicity violation detection in concurrent programs," in *Symp. Found. Softw. Eng. (FSE)*, 2008.

[22] E. Sherman, M. B. Dwyer, and S. Elbaum, "Saturation-based testing of concurrent programs," in *Euro. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, 2009.

[23] J. Yu and S. Narayanasamy, "A case for an interleaving constrained shared-memory multi-processor," in *Int'l Symp. Comp. Arch. (ISCA)*, 2009.

[24] J. S. Bradbury, J. R. Cordy, and J. Dingel, "Mutation operators for concurrency java (J2SE 5.0)," in *Works. on Mutation Analysis*, 2006.

[25] H. Do and G. Rothermel, "A controlled experiment assessing test case prioritization techniques via mutation faults," in *Int'l Conf. Softw. Maint. (ICSM)*, 2005.

[26] S. D. Stoller, "Testing concurrent java programs using randomized scheduling," in *Works. Runt. Veri. (RV)*, 2002.

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

[28] C. Mallows, "Some comments on *Cp*," *Technometrics*, 1973.