Predicting Testability of Concurrent Programs

Tingting Yu, Wei Wen, Xue Han, Jane Huffman Hayes
Department of Computer Science
University of Kentucky
Lexington, Kentucky, 40506, USA
tyu@cs.uky.edu, wei.wen0@uky.edu, xha225@g.uky.edu, hayes@cs.uky.edu

***Abstract*—Concurrent programs are difficult to test due to their inherent non-determinism. To address the nondeterminism problem, testing often requires the exploration of thread schedules of a program; this can be time-consuming for testing real-world programs. We believe that testing resources can be distributed more effectively if testability of concurrent programs can be estimated, so that developers can focus on exploring the low testable code. Voas introduces a notion of testability as the probability that a test case will fail if the program has a fault, in which testability can be measured based on fault-based testing and mutation analysis. Much research has been proposed to analyze testability and predict defects for sequential programs, but to date, no work has considered testability prediction for concurrent programs, with program characteristics distinguished from sequential programs. In this paper, we present an approach to predict testability of concurrent programs at the function level. We propose a set of novel static code metrics based on the unique properties of concurrent programs. To evaluate the performance of our approach, we build a family of testability prediction models combining both static metrics and a test suite metric and apply it to real projects. Our empirical study reveals that our approach is more accurate than existing sequential program metrics.**

I. Introduction

The advent of multicore processors has greatly increased the prevalence of concurrent programs in order to achieve higher performance. Unfortunately, failures due to concurrency faults still occur prolifically in deployed concurrent systems [60]. To tackle this, engineers use software testing as the primary method to detect concurrency faults [51].

It is challenging to test real world concurrent programs primarily because concurrency faults are sensitive to execution interleavings that are imposed by various concurrency constructs (e.g., synchronization operations). Unless a specific interleaving that can cause faults to occur and cause their effects to be visible is exercised during testing, they will remain undetected. To address this problem, numerous research efforts have been focused on applying dynamic analysis techniques to testing for concurrency faults by monitoring program execution [5], [14], [16] and controlling thread interleavings.

While software testing is an expensive process in general, concurrent software testing can be particularly expensive. Existing dynamic techniques typically monitor every shared memory access and synchronization operation so they incur significant runtime overhead. Recent work [13] reports that testing concurrent programs can introduce a 10x-100x slowdown for each test run. Such overhead increases as test suite size increases. Therefore, it is desirable to determine which code regions are more likely to contain concurrency faults as this can guide developers to focus the testing efforts on the identified code.

Software testability of a program or code region is a prediction of the amount of effort required for software testing as well as of the likelihood for revealing faults. Voas et al. define software testability as the likelihood of a program failing on the next test input from a predefined input distribution, given that there is a fault in the program [54]. Mutation testing has been used to evaluate testability of a given program for a given testing criterion [55]. The mutation scores obtained from running a number of mutants are used to measure testability. Since testability is a dynamic attribute of software, it is very computationally-intensive to measure directly. One approach to address this is to apply a testability prediction model to software at various levels of granularity (e.g., functions, classes, files). In practice, software development organizations often use test cases generated by the same testing criterion to perform unit testing across multiple releases of the software [3]. Thus, a testability prediction model based on inexpensive measurements from the current software release could be a cost-effective way to measure testability throughout the life of a software product.

Software testability prediction is related to software defect prediction. There has been much research on software defect prediction by combining static code metrics to identify defect-prone source code artifacts [59]. A variety of statistical and machine learning techniques have been used to build defect prediction models [11]. Similar approaches have also been applied to predict testability based on mutation scores. For example, Jalbert et al. [25] propose a machine learning approach to predict mutation scores based on a combination of source code and test suite metrics. Their results show that the combination approach can be effective.

However, all existing research has focused on sequential software. To date, no work has considered concurrent software systems for which testing is extremely expensive due to the large number of possible thread interleavings and instrumentation overhead. Unlike testability prediction for sequential programs, that often relies on a set of well-defined and traditional code metrics (e.g., lines of code, cyclomatic complexity), testability prediction for concurrent programs must consider the unique concurrency properties in its fault models: threads, shared variable accesses between threads, and synchronization operations. A software component with a low testability score

suggests that the current testing approach is not adequate for exposing concurrency faults and, therefore, effort should be made to test this component (e.g., adding instrumentation).

In this paper, we address the problem of testability prediction for concurrent programs. We propose five novel code metrics specific to concurrent programs (concurrent code metrics or CPM) by taking unique features related to concurrency properties into account. Specifically, we adapt the concurrency control flow graph (CCFG) to generate code metrics involving (i) concurrent cyclomatic complexity, (ii) number of shared variables, (iii) number of conditional basic blocks that contain concurrency constructs, (iv) number of synchronization operations, and (v) access distance between shared variables in a local thread. We conjecture that these metrics and their combination can predict the testability of a program. The testability score (mutation score) is computed by applying a variety of mutation operators specific to concurrent programs. Then, we create accurate testability predictors by combining both code metrics and test suite metrics (i.e., code coverage), as the testability score is affected by the quality of the test suite. We use different statistical and machine learning techniques to determine the best combination of metrics for predicting testability.

To evaluate our approach, we apply it to seven real word applications. Our results show that, given test cases generated from specifications (widely used test case generation approach in industry), concurrent program metrics (static code metrics and test suite metrics) outperform traditional metrics for sequential programs in terms of predicting testability. To further demonstrate the practicality of our approach, we study the bug repository of the applications and show that modules with low testability tend to produce post-release concurrency faults. The ultimate benefit of our approach is that developers can predict whether a testing technique is likely to reveal concurrency faults in specific code modules. Our paper makes the following contributions:

1) The first approach to effectively predict testability of concurrent programs,
2) A set of novel source code metrics specific to concurrent programs, and
3) An empirical study showing the effectiveness of our approach.

II. Background and Definitions

This section provides background information on concurrency and testability analysis. Related work is discussed further in Section VII.

A. Mutation Analysis for Concurrent Programs

Mutation testing is an approach for evaluating test suites and testing techniques using a large number of systematically seeded program changes, allowing for statistical analysis of results [3], [27]. Mutation testing typically involves three stages: (1) Mutant generation – in this stage, a predefined set of mutation operators are used to generate mutants from program source code or byte code. A mutation operator is a rule that is applied to a program to create mutants, such as AOR (arithmetic operator replacement) [58]. (2) Mutant execution – in this stage, the goal is execution of test cases against both the original program and the mutants. (3) Result analysis – in this stage, the goal is to check the mutation score obtained by the test suite, where mutation score is defined as the ratio of the number of killed mutants to the number of all (non-equivalent) generated mutants.

TABLE I: **List of Concurrency Mutation Operators**

Operator	*Description*	*Consequence*
rmlock	Remove call to lock/unlock	order/atomicity violations
msem	Modify permit count in semaphore	order/atomicity violations
mwait	Modify parameter/time in cond_timedwait	performance fault
rmwait	Remove call to cond_wait/cond_timedwait	order/atomicity violations
swptw	Swap cond_wait with cond_timedwait	deadlock, performance fault
rmsig	Remove call to cond_signal/cond_broadcast	order/atomicity violations
swptw	Swap cond_signal with cond_broadcast	order/atomicity violations
rmjoinyld	Remove call to join/yield	starvation, crash
repjn	Replace join with sleep	crash
rmvol	Remove volatile keyword	order/atomicity violations
swplck	Swap lock-unlock pairs	order/atomicity violations, deadlock
shfecs	Shift critical section	order/atomicity violations
shkecs	Shrink critical section	order/atomicity violations
epdecs	Expand critical section	deadlock, performance fault
spltecs	Split critical section	order/atomicity violations
rmbarrier	Remove call to barrier_wait	order/atomicity violations
mbarrier	Modify parameter/time in barrier_wait	performance fault

Mutants that contain a single fault are called first-order mutants. First-order mutants have been widely used for mutation analysis of sequential programs. There has been some work to generate first-order mutants for concurrent programs [6], [18]. For example, Ghosh generates concurrency-related mutants by removing single synchronization keywords [18]. Bradbury et al. proposed a set of first-order mutation operators for Java [6]. However, more recent work has shown that first-order mutants are not sufficient to simulate subtle concurrency faults due to the complexity of thread synchronizations [21], [31], [36]. Therefore, some research has investigated higher-order mutants [23], [26] for concurrent mutation operators [31] by inserting two or more faults. Higher-order mutants subsume first-order mutants, as killing the former is a sufficient but not necessary condition for killing the latter.

To generate higher order mutants for concurrent programs, Kaiser et al. [31] propose a set of mutation operators for multi-threaded Java programs based on concurrency bug patterns that include subtle concurrency faults (e.g., data races). Kusano et al. [36] implemented *CCmutator* based on the Clang/LLVM compiler framework to inject concurrency faults for multi-threaded C/C++ applications Their work considers both first-order and higher-order mutants. In this work, we extended *CCmutator* by synthesizing existing work to include various concurrency-related mutation operators. Table I summarizes the mutation operators used in this paper, including operator names, descriptions, and their possible consequences [40].

These operators include mutex locks, condition variables, atomic objects, semaphores, barriers, and thread creation and join. For example, removing a lock-unlock pair can create potential data races and atomicity violations, swapping lock pairs can create potential deadlock, shifting and splitting critical sections can introduce potential data races and order violations

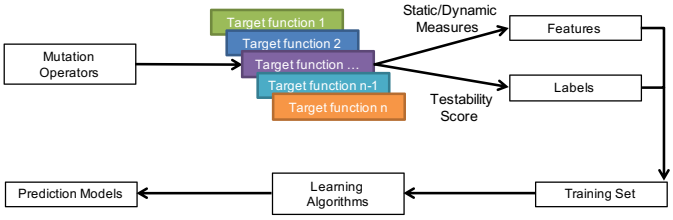

Fig. 1: **Process of Predicting Testability**

as some variables are no longer synchronized. Replacing a call to join with a call to sleep can cause nondeterministic behavior. If the sleep time is sufficiently long, the program may appear to be correct. Otherwise, the program may crash due to the improper join.

We use the term location to indicate a place in a program where a fault can occur. Although the techniques we propose can be used at different granularities (e.g., statements), this paper concentrates on locations that correspond to single program instructions.

B. Software Testability

In this work, we use the PIE model proposed by Voas [54] to measure testability. PIE applies mutation analysis to predict a program location's ability to cause the program to produce observable failures if the location were to contain a fault. Specifically, software testability is a function of <*program*, *test_selection_criteria*>, where inputs can be generated using a certain criterion. A highly testable program is more likely to reveal faults than a low testable program given the same test selection criterion.

Based on the PIE model, we define testability as the probability that existing faults will be revealed by existing test cases. To measure the testability of a single faulty location l, we use the product of execution probability E(l) and propagation probability P(l), denoted by TB(l) = E(l) * P(l), where E(l) indicates the probability l is executed and P(l) indicates the probability l propagates to the output. In this work, testability is measured at the function level, so the testability is calculated by averaging the TB(l) of all mutants in a function f, denoted as:

$$\text{TB}(f) = \overline{\sum_{i=1}^{|M|} TB(l)_i}$$

Above, TB(f) is the testability of a function f, $|M|$ is the number of mutants, and TB$(l)_i$ is the testability of a single location i in f.

Static code metrics have also been considered as an *indirect* measurement to predict the effort needed for testing [56]. For example, if a module has high complexity, more effort may be needed to effectively test the module. Compared to dynamic testability analysis, the direct measurement of testability [33], static metrics require much less computation and are thus more efficient. It should be noted, though, that a static metric can be too coarse-grained to predict testability [56]. We strive for a cost-effective testability predictor, and posit that we can leverage various static metrics to learn a precise model to predict dynamic testability.

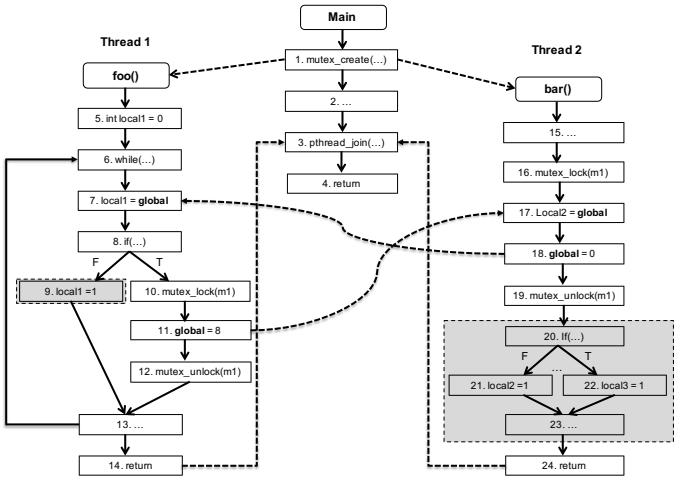

Fig. 2: **Concurrency Control Flow Graph**

C. Machine Learning and Testability Prediction

There has been a great deal of research on applying machine learning techniques to predict faults [11], [54] and testability [32] based on various code metrics and/or test suite metrics. In our research, the testability score is the dependent variable while code metrics and test suite metrics form the independent variables. We aim to explore the effect of metrics on the testability of each function using machine learning methods.

Figure 1 shows the process of testability prediction in this work. First, we define instances as units of programs, these can be files, classes or functions. The instances that we consider are at the function level. We then apply concurrency mutation operators to the program to generate a set of mutants. Next, the mutants are executed by existing test cases and a testability score is obtained for each function. The testability scores are labels in machine learning. For building a regression model, we predict continuous values (e.g., testability scores). For building a classification model, we label functions as LOW, MEDIUM, or HIGH in terms of their testability scores. The code metrics and test suite metrics are used as features. Having both features and labels, the next step is to train prediction models. Finally, the prediction models classify instances as LOW, MEDIUM, or HIGH or predict the concrete testability score.

III. Concurrency-related Code Metrics

In this section, we define a family of static code metrics specific to concurrent programs. We use these metrics combined with test suite metrics (i.e., code coverage) to predict concurrent program testability.

A. Concurrency Control Flow Graph

A concurrent program P consists of threads that communicate with each other through shared variables and synchronization operations. Given the program source code, we can construct a concurrent control flow graph (CCFG) for a

procedure $p \in P$ based on a flow and context-sensitive points-to analysis, where p can be accessed by multiple threads. The idea of building CCFGs is not new and there has been research on using CCFGs to achieve different objectives [17], [30]. For example, Kahlon et al. [30] build a context-sensitive CCFG to perform staged data race detection. Our CCFG is similar to those used in existing work but is implemented to satisfy our goal of predicting concurrent program testability. First, p is constructed into a control flow graph (CFG), denoted as ($N(p)$, $E(p)$). A node $N(p)$ is an instruction I and an edge $I_i \rightarrow I_j \in E(p)$ describes the control flow and data flow between nodes in this CFG. In the CCFG, we add additional edges to represent communications between procedures potentially running on two different threads. Figure 2 illustrates an example CCFG, where the solid lines reflect the local edges and the dotted lines reflect the cross-threads' edges, including `fork`, `join`, and `communication` edges. The `Main` function creates two threads, on which functions `foo` and `bar` are running, respectively. The variables marked as bold are shared between threads. For readability purposes, we use statements rather than instructions to represent each node.

Specifically, a `fork` edge is added from the program location where `thread_create` instruction is called to the entry node of the procedure to be executed. In Figure 2, edges $< 1, foo >$ and $< 1, bar >$ form two fork edges. If the thread on which the procedure is to be executed is specified as in the thread pool model, the procedure is duplicated on the other thread. A `join` edge is added from the return of a procedure that is executed by the fork to the node representing `thread_join` instruction. In Figure 2, edges $< 14, 3 >$ and $< 24, 3 >$ are considered to be join edges. A `communicate` edge is added from a write of one shared variable (SV) on one thread to the read of the same SV on the other thread. For example, Figure 2 contains two communication edges involving two shared variable pairs $< 11, 17 >$ and $< 18, 7 >$.

B. Static Complexity Metrics

We introduce five code metrics specific to concurrent programs. These metrics are generated from the CCFG.

Synchronization Point Count (SPC). We define the *synchronization point count*, $SPC(f)$, as the number of nodes involving synchronization operations (SOs) in a function f. The use of the SPC metric is based on the intuition that the number of SOs contributes to the complexity of concurrent programs. As the number of SOs increase, the program is more likely to contain more faults related to synchronization usage, such as deadlock and atomicity violations. The SPC metric for a function is defined as:

$$SPC(f) = num\_of\_syncs(f)$$

Here, we consider mutex, semaphore, conditional variables, and barriers as synchronization operations. In the Figure 2 example, $SPC(main)$ = 2, $SPC(foo)$ = 2, and $SPC(bar)$ = 2, because each function contains two synchronization operations (e.g., `mutex_lock`).

Shared Variable Count (SVC). In this metric, we count the nodes in the CCFG involving shared variable (SV) read/write in the procedure p. The shared variables can affect data communication between threads. The increasing complexity of SV usage is likely to cause incorrect data state to propagate across threads. As such, we define SVC for p as:

$$SVC(f) = num\_of\_SVs(f)$$

In Figure 2, $SVC(foo)$ = 2 and $SVC(bar)$ = 2. The variables marked with bold are SVs. Note that we do not count SVs passed as lock objects. $SVC(main)$ = 0, because there are no global variables in the `main` function.

Conditional Synchronization Count (CSC). We define *conditional synchronization count*, $CSC(f)$, as the number of conditional basic blocks (e.g., branches) within function f that contain at least one shared variable or synchronization operation. The CSC metric takes into account the local control flow of a procedure in one thread that can potentially complicate the communication with procedures running on other threads. The intuition is that the conditional block containing concurrency elements increases the complexity of a function in terms of multithreading communication, affecting the sensitivity of inputs that reach such synchronization points. The CSC metric for f is defined as:

$$CSC(f) = num\_of\_cond\_syncs(f)$$

As Figure 2 shows, $CSC(foo)$ = 2, because there are two conditional blocks in `foo` that contain SVs (i.e., `while`, `if`). The shaded areas are irrelevant basic blocks that CSC does not count . Note that $CSC(bar)$ = 0 and $CSC(main)$ = 0, because the two functions contain no conditional basic blocks containing SVs or synchronization points (the two shaded conditional basic blocks in `bar` are irrelevant basic blocks).

Concurrency Cyclomatic Complexity (CCC). We extend the traditional McCabe's cyclomatic complexity [44] to measure complexity of concurrent programs, denoted as *concurrency cyclomatic complexity* (CCC). To compute CCC, we first prune the CCFG to transform it into CCFG', which involves two steps: 1) remove each basic block b that is irrelevant to concurrency properties (i.e., SVs and synchronizations) computed by the CSC metric, as well as remove their incoming and outgoing edges; and 2) add an auxiliary edge from each predecessor of b to each successor of b. Thus, the $CCC(p)$ of a function is defined with reference to its CCFG':

$$CCC(f) = E' - N' + 2$$

Here, E' is the number of edges and N' is the number of nodes in the f of CCFG, where E' includes both ingoing and outgoing edges for f. In Figure 2, the shaded nodes are irrelevant and thus removed. Auxiliary edges are added from node 8 to node 13, and from node 19 to node 24. Thus, $CCC(main)$ = 8 - 5 + 2 = 5, $CCC(foo)$ = 15 - 10 + 2 = 7, and $CCC(bar)$ = 10 - 7 + 2 = 5. Note that the sequential Cyclomatic Complexity of the three functions are 5 - 4 + 1 = 2, 12 - 11 + 2 = 3, and 11 - 11 + 2 = 2.

Shared Variable Access Distance (SVD). The distance between two shared variable (SV) accesses in one thread is also an important factor for concurrency fault exposure [39]. For example, if `x` is written at l_1 and later read at l_2 by the same thread $T1$, and there exists a different thread $T2$ that updates the value of `x`, the distance between l_1 and l_2 can impact the chances of $T2$ interleaving between them. We consider access distance as the instruction gap between two SV accesses (reads or writes) in the same procedure p. To compute $SVD(f)$, we first identify all SV pairs (SVP) in p. For each SV pair $< sv1, sv2 >$, we calculate all instruction gaps by traversing all path segments[1] between $sv1$ and $sv2$. Note that one SVP can associate with multiple distance values due to possible control flow edges. To consider all path segements, we calculate the mean distance over all path segments. Specifically, we average all distance values if they are normally distributed, otherwise we use their trimmed mean. Thus, $SVD(f)$ is defined as:

$$SVD(f) = \overline{\sum_{i=1}^{N} \sum_{i=j}^{M} Dis(SVP_{i,j})}$$

Here, N is the number of shared variable pairs and M is the number of path segments for an SVP. In the function `foo` of Figure 2, there are two path segments: (7,8,10,11) and (11,12, 13, 6,7) for SVPs <7, 11> and <11, 7>, respectively. Suppose each node counts for 2 instructions, then $SVD(foo)$ = (8 + 10) / 2 = 9.

C. Test Suite Metrics

Since dynamic testability analysis requires a test suite, the quality of the test suite is also a considerable factor in building effective predictors. While a test suite provides various observable attributes (e.g., the number of test cases) for software metrics, coverage metrics are the most commonly used metrics as they directly measure the relationship between the test suite and source code. Thus, our predictors are based on a combination of static code and coverage metrics of the program under test.

The *interleaving coverage criteria* have been widely used to measure test suite quality for concurrent programs [7], [24], [39]. An interleaving criterion is a pattern of inter-thread dependencies through SV accesses that helps select representative interleavings to effectively expose concurrency faults. An interleaving criterion is satisfied if all feasible interleavings of SV defined in the criteria are covered. In this work, we employ a Def-Use criterion, which is satisfied if and only if a write `w` in one thread happens before a read `r` in another thread and there is no other write to the variable read by `r` between them. In fact, the Def-Use criterion is equivalent to communication edge coverage in the context of CCFG.

To combine with static metrics, test suite metrics are also measured at the function level. However, it is quite possible that an interleaving def-use pair involves two functions, such as the two shared variable pairs (i.e., $< 18, 7 >$ and $< 11, 17 >$) in Figure 2. In this case, the function includes both variables in a pair rather than a single variable. For example, the coverage of the function `foo` is 100% if both communication edges are covered; covering only single variables (e.g., node 7 or node 11) is not considered valid.

[1]A path segment is a path slice for which every node is visited at most once.

TABLE II: **Object Program Characteristics**

Program	*NLOC*	*instances*	*mutants*	*tests*	$mutants_e$	$mutants_p$	$stmt_{cov}$	Int_{cov}
MYSQL1	861k	2478	2,267	1,813	81%	44%	70%	79%
MYSQL2	1,192k	1083	2,644	2,972	85%	39%	71%	82%
MEMCACHED	380k	153	234	1,028	82%	35%	77%	81%
AGET	1,850	5	24	572	95%	68%	54%	95%
PFSCAN	752	6	73	488	90%	82%	92%	88%
PBZIP2	4373	24	41	520	100%	98%	78%	100%
BZIP2SMP	4236	19	226	509	100%	92%	82%	100%

D. Implementation

Our metrics are implemented using the popular Clang/LLVM compiler platform [37] using LLVM opt pass. Clang's CFG provides a directed graph for each function, where the nodes are the basic blocks and the directed edges represent how the control flows. As noted above, our CCFG extends the basic CFG by adding edges describing inter-thread communications. We apply shared variable analysis to identify variables shared by two threads, such as heap objects and data objects that are passed to a function (e.g., thread starter function) called by another thread. Since our metrics implemented by LLVM intermediate representation (IR) are based on single static assignment (SSA) form, we can potentially leverage compiler front-ends to handle other languages.

IV. Empirical Study

Our goal is to evaluate the effectiveness of concurrent program metrics (CPM). We consider the following research questions:

RQ1: Does each single concurrent program metric on f correlate with the testability score of f?

RQ2: Can the combination of concurrent program metrics be used to predict testability of new functions in the same project, and does it outperform sequential program metrics?

RQ3: Can the combination of concurrent program metrics be used to predict testability of new functions in a new project?

RQ4: Can the testability predictor be used to predict real concurrency faults?

A. Objects of Analysis

As objects of analysis we chose eight concurrent programs. Table II lists these programs and the numbers of lines of non-comment code they contain (other columns are described later). MySQL [50] is a free distribution of one of the most widely used open source database applications. We chose version 5.0.11 (MYSQL1) and version 5.5.3 (MYSQL2). While the two program versions share similar functionality, the implementation varies considerably from MYSQL1 to MYSQL2. MEMCACHED [45] is a high performance distributed memory object caching system to ease database loading for dynamic

web applications. We use version 1.4.4. AGET [1] is a multithreaded HTTP/FTP download accelerator aimed to provide similar functionality as FlashGet on Windows. We use version 0.57. PFSCAN is a file scanner that offers a combined functionality of find, xargs, and fgrep in parallelism. We use version 1.0. BZIP2SMP [9] is a parallel implementation of BZIP2 targeting improvements on symmetric multiprocessing (SMP) machines. We use version 1.0. PBZIP2 [8] is a file compressor that implements parallel BZIP2 to speed up compression on SMP machines. We use version 1.1.12. Since the number of functions in the last four programs were too small (fewer than 20 functions) to effectively apply machine learning, we merged them into one dataset, SMALL. We selected these programs because they are representative of real-world code and have been widely used in academic research. In addition, they are each applicable to one or more of the classes of mutation operators described in Section II. Column 3 of Table II shows the number of functions (or instances) that contain concurrency constructs; only these functions are considered in the study.

To address our research questions we also required test cases. A test case includes input data and command options. For the three large programs that had been released with tests, we used them as input data (e.g., .test files in MySQL programs). The input data is combined with various command options regulated by the constraints. The four small object programs (i.e., AGET, PFSCAN, BZIP2SMP and PBZIP2), however, were not equipped with tests. For these programs, we created black-box suites. Engineers often use such test suites designed based on system parameters and knowledge of functionality [10]. We followed this approach, using the category-partition method [52], which employs a Test Specification Language (TSL) to encode choices of parameters and environmental conditions that affect system operations and combine them into test inputs. Column 6 of Table II lists the numbers of tests ultimately utilized for each object program.

To address our research questions, we also required faulty versions of our object programs. We extended CCMUTATOR [36] to create seeded concurrency faults of the classes described in Section II. This process left us with the numbers of mutants reported in Column 4 of Table II.

Testing also requires test oracles. For programs released with existing test suites and with built-in oracles provided, we used those. Otherwise we checked program outputs, including messages printed on the console and files generated and written by the programs.

B. Data collection

We executed our test cases on all of the mutants of each object program. To control for variance due to randomization of thread interleavings, we ran each mutant 100 times. A mutant is marked as being executed or propagated if it does this at least once. We used a Linux cluster to perform the executions, distributing each job on a distinct node. The testability score was computed by following the process described in Section II-A. Columns 6-7 of Table II report the percentage of mutants executed and killed.

TABLE III: **List of Code Metrics**

CPM Metrics	*Description*
ConcurrencyComplexity(CCC)	Concurrent program complexity
CountSharedVaraible (CSV)	# of shared variable
CountConditionalBasicBlock (CBB)	# of conditional basic blocks
CountSynchronizations(CSO)	# of synchronization operations
CountDistance(CCD)	average distance between shared variables
InterleavingCoverage(CCV)	interleaving coverage by running tests
SPM Metrics	*Description*
CountInstruction (CI)	# of instructions
CountBasicBlock (CB)	# of basic blocks
CountParameter(CP)	# of parameters for a function
CyclomaticComplexity(CC)	Mcabe's cyclomatic complexity
StatementCoverage(SCV)	statement coverage by running tests

To gather metric data, for each function we computed five concurrency-related code metrics and four sequential code metrics. The method of computing concurrency metrics is described in Section III. The four sequential metrics include number of instructions (CI), number of basic blocks (CB), the number of parameters (CP), and McCabe's complexity (CC). All metrics are collected using LLVM pass.

As for the test suite metric, we used the open source test coverage tool GCov to measure statement coverage (SCV). To measure interleaving coverage (CCV), we followed the approach described in Section III-C. To record thread information, we used PIN [41] to instrument the entry point of each function and recorded thread IDs that exercised the function. Column 8-9 of Table II report the statement coverage and interleaving coverage, respectively. Table III summarizes a set of concurrent program metrics (CPM) and a set of sequential program metrics (SPM), which includes both static code metrics and test suite metrics.

C. Prediction Models

We build four prediction models using statistical and machine learning techniques from Weka [22]. We employ both linear regression and classification models. For linear regression, the model was trained to predict a specific testability score percentage. For the classification techniques, we categorized all testability scores as *LOW*, *MEDIUM*, or *HIGH* to reduce the value prediction to a three-group classification problem. The ranges of values in each category are determined based on the distribution of the testability scores in our training data. Since the testability scores are unlikely to follow a balanced distribution (i.e., LOW: 0%– 33%, MEDIUM: 34%0 – 66%, and HIGH: 67%0 – 100%), we adjusted the group ranges to accommodate the distribution with *freq3bin* [25], [48] that depends on equal frequency count (equal number of instances in each bin). Specifically, all testability scores are sorted in descending order. Then the first ceiling $(\#of(all\_Instances)/3)$ instances are assigned *HIGH*, the second ceiling $(\#of(all\_Instances)/3)$ instances are assigned *MEDIUM*, and the third ceiling $(\#of(all\_Instances)/3)$ instances are assigned *LOW*.

We used Linear Regression, Bayesian Network, J48 Decision Tree, and Logistic Regression in Weka [22]. We chose them because they are popular and have been shown to be

effective at predicting defects in a recent study [19]. To examine our research questions, we applied the four models to CPM and SPM metrics.

D. Performance Measures

To evaluate and compare the performance of CPM and the baseline metric SPM, we measure their prediction performance on linear regression and classification models, respectively.

1) Linear Regression: We calculated the correlation coefficient (CC), the mean absolute error (MAE), and the root mean squared error (RMSE) [57]. Correlation coefficient measures the correlation between predicted and actual testability scores. If the correlation coefficient is closer to 1, the metrics are more correlated to testability. Both MAE and root MSE represent the difference between predicted and actual testability scores. If both error values of a prediction model are less than others, it means the model has higher prediction accuracy. To compare the prediction model of CPM with SPM, we measured all three of these values.[2]

2) Classification: We used F-measure to evaluate performance of CPM and SPM across the three classification models. F-measure usually represents the harmonic mean of precision and recall. The computed F-measure values are between 0 and 1 and a larger F-Measure value indicates a higher classification quality. The following outcomes are used to define precision, recall, and F-measure: (1) Category A is correctly classified as A ($a \rightarrow a$); (2) Category A is incorrectly classified as B ($a \rightarrow b$); and (3) Category B is incorrectly classified as A($b \rightarrow a$). We use the above outcomes to evaluate the prediction accuracy of our models with the following measures:

- **Precision:** the number of instances correctly classified as A ($N_{a \rightarrow a}$) over the number of all instances classified as A.
$$PrecisionP(a) = \frac{N_{a \rightarrow a}}{N_{a \rightarrow a} + N_{b \rightarrow a}}$$
- **Recall:** the number of instances correctly classified as A ($N_{b \rightarrow b}$) over the total number of instances in A.
$$RecallP(b) = \frac{N_{a \rightarrow a}}{N_{a \rightarrow a} + N_{a \rightarrow b}}$$
- **F-measure:** a composite measure of precision P(b) and recall R(b) for buggy instances.
$$F - measureF(a) = \frac{2 * P(a) * R(a)}{P(a) + R(a)}$$

To evaluate our prediction models, we used 10-fold cross validation, widely used to evaluate prediction models [38], [49]. Since 10-fold cross validation randomly samples instances and puts them in ten folds [2], we repeated this process 100 times for each prediction model to avoid sampling bias [38].

To assess whether prediction performance of different metric sets were statistically significant, we applied the t-test to the data sets, comparing each pair of metric sets within each model. We checked if the mean of F-measure values of CPM was greater than the mean of F-measures of CPM at the 95% confidence level ($p - value < 0.05$).

[2]Here we use the Guildford scale [28], in which correlations with absolute value less than 0.4 are described as "low," 0.4 to 0.7 as "moderate," 0.7 to 0.9 as "strong," and over 0.9 as "very stong."

TABLE IV: **Correlation Coefficient**

Prog.	CPM						SPM				
	CCC	CSV	CBB	CSO	CCD	CCV	CI	CB	CP	CC	SCV
MYSQL1	**-0.526**	**-0.483**	**-0.537**	**-0.540**	**-0.600**	**0.676**	-0.339	**-0.412**	-0.275	-0.289	**-0.451**
MYSQL2	**-0.556**	**-0.558**	**-0.555**	-0.182	**-0.402**	0.611	-0.138	-0.150	0.000	-0.130	**-0.567**
MEMCACHED	**-0.444**	**-0.478**	**-0.430**	-0.353	**-0.661**	**0.676**	**-0.455**	**-0.418**	-0.199	-0.381	**0.507**
SMALL	**-0.536**	**-0.527**	**-0.545**	**-0.633**	**-0.682**	**0.632**	**-0.503**	**-0.489**	-0.132	**-0.425**	**-0.445**

E. Threats to Validity

The primary threat to external validity for this study involves the representativeness of our programs, mutants, coverage criteria, and test cases. Other systems may exhibit different behaviors, as may other forms of test cases. However, the programs we investigated are popular open source programs. Furthermore, the test cases are either those provided with the programs or created via a commonly used process (TSL in this case): they are representative of test cases that could be used in practice to test these programs. Most of the test subjects we used had relatively good test suites (i.e., of the covered mutants, the mutation scores were above 80%). Mutants can be influenced by external factors such as mutation operators. We used only concurrency mutation operators. However, concurrency faults can also be introduced by sequential glitches. In addition, other interleaving criteria (e.g., synchronization coverage) may lead to different coverage results. We controlled for these threats by using well studied concurrency mutation operators and popular interleaving criteria.

The primary threats to internal validity for this study are possible faults in the implementation of our approach and in the tools that we used to perform evaluation. We controlled for this threat by extensively testing our tools and verifying their results against smaller programs for which we could manually determine the correct results. We also chose to use popular and established tools (e.g., LLVM, Weka) for implementing the various stages of our approach.

Where construct validity is concerned, our measurements involve using metrics extracted from source code and test coverage (independent variables) to predict testability based on mutation score (dependent variable). Other code metrics and test suite metrics are also of interest. Furthermore, other machine learning performance measures can be used to measure effectiveness and accuracy. To control for this threat, we chose commonly used F-measures.

V. Results and Analysis

Results for each research question are presented below.[3]

A. RQ1: Correlation Analysis

To investigate effectiveness of linear regression, we determined the correlation between each metric for each function F with the testability score.

Table IV shows the correlation coefficient between each single metric and testability score across all four datasets. All correlation coefficients are significant at the 0.05 level. The numbers rendered in bold font indicate strong or moderate

[3]All data we used in our experiments are publicly available at http://cs.uky.edu/~tyu/research/cpm

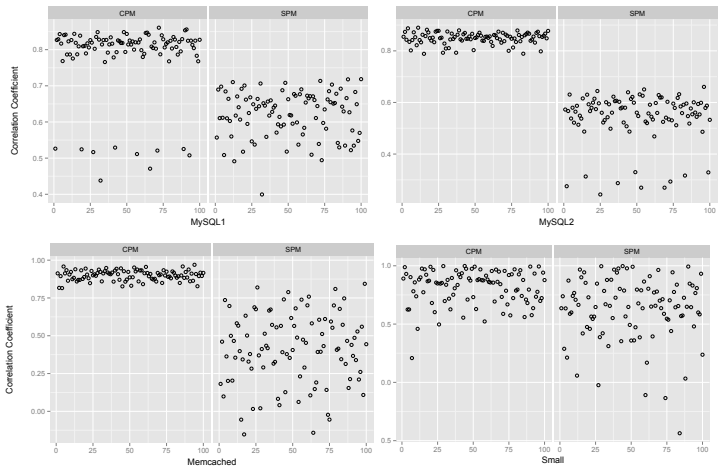

Fig. 3: **Performance Comparison of Linear Regression Models.**

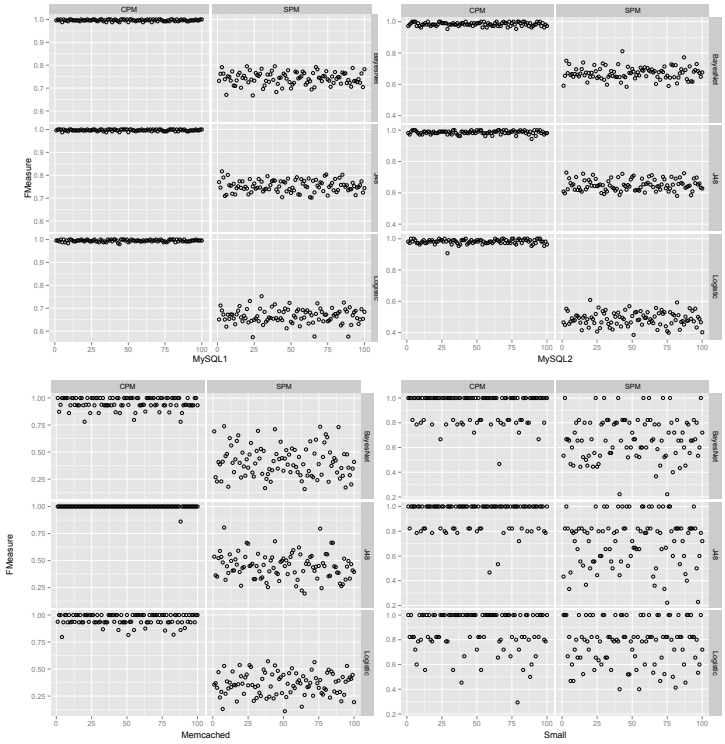

Fig. 4: **Performance Comparison of Classification Models.**

correlation coefficients. For example, in project MySQL1, the higher the concurrency cyclomatic complexity (CCC), the lower the testability score. In fact, four out of five static metrics (i.e., CCC, CSV, CBB CCD) and the test suite metric (SCV) in CPM are each moderately correlated to the testability across all four datasets. Only CSO demonstrates low correlation for MYSQL2 and MEMCACHED.

In contrast, for all four datasets, none of the static metrics in SPM have moderate/strong correlation to testability. Only the test suite metric (SCV) is moderately correlated with testability. However, SCV is still not as effective as CCV. The best static metric in SPM is CB, which is moderately correlated for three out of four datasets (MYSQL1, MEMCACHED, SMALL).

TABLE V: **Performance of CPM and SPM metrics.**

Prog.	MYSQL1		MYSQL2		MEMCACHED		SMALL	
	CPM	SPM	CPM	SPM	CPM	SPM	CPM	SPM
CC	**0.785**	0.622	**0.849**	0.544	**0.899**	0.412	**0.816**	0.629
MAE.	**11.1**	14.6	**15.6**	24.7	**13.4**	29.8	**21.7**	25.7
RMSE	**19.1**	22.5	**18.8**	35.3	**16.8**	33.7	**36.4**	42.5
BayesNet	**0.996**	0.745	**0.984**	0.671	**0.954**	0.412	**0.932**	0.672
Decision Tree	**0.997**	0.753	**0.986**	0.650	**0.999**	0.450	**0.938**	0.720
Logistics	**0.995**	0.666	**0.982**	0.487	**0.960**	0.363	**0.879**	0.771

The worst metric is CP - it is weakly correlated with testability for all four datasets.

Overall, these results indicate that *each of the single CPM metrics was more effective than SPM at predicting concurrent program testability*. For SPM, however, only the test suite metric was effective.

B. RQ2: Effectiveness of CPM vs. SPM

To examine RQ2, we employ both linear regression and classification techniques. We also compare the performance of CPM to that of SPM using both types of techniques.

1) Linear Regression: Figure 3 shows correlation coefficients for CPM and SPM across all four datasets across 100 iterations. The horizontal axis is iteration and the vertical axis is correlation coefficient. In the three large datasets, while the correlation coefficients vary across iterations, the trend clearly indicates that CPM outperforms SPM. The only exception occurs on SMALL, where CPM varies dramatically across iterations. There are two reasons for this, possibly leading to deteriorated quality of prediction. First, the size of the dataset is small (54 instances). Second, while the three applications in SMALL share similar functionality, they are still very diverse (e.g., developed by different teams).

The second row (CC) in Table V shows the average correlation coefficients across 100 iterations for the two metric sets and all four datasets. The numbers marked with bold under CPM indicate that the value difference in comparison to SPM is statistically significant. For example, correlation coefficient (0.785) of CPM in MYSQL1 is significantly better than that (0.622) of SPM. In fact, for all four datasets, CPM is strongly correlated with testability. In addition, CPM is more effective than SPM in terms of correlation coefficients. Where error is concerned, for all datasets, the average mean absolute error (MAE) and average root mean squared error (RMSE) of CPM are lower than those of SPM at the 0.05 significance level.

Overall, these results indicate that *the combination of CPM metrics is more effective and accurate than traditional sequential program metrics at predicting testability of concurrent programs*.

2) Classification: We use F-measure to evaluate prediction performance for the three classification algorithms, as described in Section IV. Figure 4 shows F-measure values for each of classification techniques and four datasets on the two metric sets CPM and SPM. The horizontal axis is iteration and the vertical axis is F-measure. In the three large datasets, the trend clearly indicates that CPM outperforms SPM. Similar to the linear regression model, on the SMALL dataset, the F-measure values dramatically fluctuate across iterations.

Rows 5-7 in Table V show the mean of F-measures from 100 ten-fold cross validations for the three techniques. The F-measure values for CPM are bolded if they are significant ($p\text{-}value < 0.05$) in comparison to SPM. On MYSQL1 using BayesNet, for instance, the F-measure (0.996) of CPM is *statistically* better than that of SPM (0.745). In fact, on all four datasets, CPM is more effective than SPM for each of the three classification techniques. Overall, these results suggest that *when using classification models, CPM outperform the traditional sequential program metrics*.

3) Metrics Effectiveness Analysis: To evaluate the effectiveness of each CPM metric for classification, we measured the information gain ratio [35] of metrics in CPM. The information gain ratio indicates how well a metric distinguishes labels (i.e., *LOW*, *MEDIUM*, *HIGH*) of instances. Specifically, we used the information gain evaluator API (`evaluateAttribute`) in Weka to get the information gain score for each metric. Figure 5 shows these information gain scores. While the effectiveness of metrics vary for different datasets, SCV was ranked first in the three large datasets and second in the small dataset. CSV ranked third for MYSQL2, but lower for all other datasets. While CCC ranked second in MYSQL2, it ranked much lower for all other datasets. In the small dataset, CSO ranked first. It also ranked second for MYSQL1. This suggests that *each CPM metric can play an important role in testability prediction, and the quality of the test suite is particularly important*.

C. RQ3: Across Projects Prediction

RQ3 investigates whether a predictor for one application group (dataset) can be used for other applications. We applied linear regression and classification models built from each dataset to the instances of each of the other three datasets. We then checked how accurate the prediction is by assessing the performance of each model. The prediction results are shown in Table VI. For example, columns 2-4 show the performance values of using models learned from MYSQL2, MEMCACHED and SMALL to predict MYSQL1.

For the linear regression technique, 11 out of the 12 models have strong correlation coefficients. Only the model learned from MYSQL2 yielded moderate correlation on SMALL. For the classification techniques, the F-measure values for 35 out of 36 models were greater than 0.5, indicating that the cross project classification is effective. Only the BayesNet model learned from SMALL had low effectiveness (F-measure = 0.359) in predicting MYSQL2. Overall, these results suggest that *CPM is effective at predicting testability across projects*. In addition, it is not surprising to see that the models are even more successful across similar projects (i.e., MYSQL1 and MYSQL2).

D. RQ4: Connection to Real Concurrency Faults

Just et. al [29] study whether mutants are indeed a valid substitute for real faults. The results show that mutation testing is appealing because large numbers of mutants can be automatically generated and used as a substitution for real faults. While their results are promising, we further investigate whether this conclusion holds in the context of testing concurrent programs.

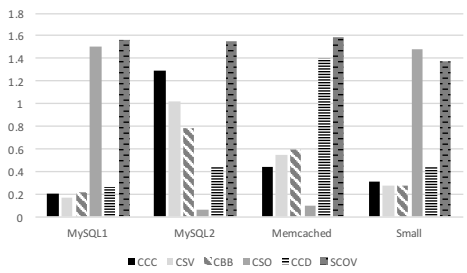

Fig. 5: **CPM Metrics based on the Information Gain Ratio.**

We located and isolated real faults that have been previously found and fixed by analyzing projects that have version control and bug tracking systems. We study only the three large datasets, as the SMALL dataset does not have any such systems. Specifically, we obtained real faults from a project's version control history by identifying commits that corrected a failure in the program's source code. We also examined the bug repository and identified the functions that were reported as containing concurrency faults. As a result, MYSQL1 yields two deadlock faults involving four functions (e.g., `lock_deadlock_recursive`), and MYSQL2 contains two deadlock faults and one data race, where the deadlock faults involve two functions and the data race fault involves three functions. MEMCACHED contains a data race in the function `process_arithmetic_command`.

We next correlated these functions with the testability scores. The results show that eight out of 10 functions fall into the *LOW* testability category. The only two exceptions were on `lock_deadlock_occurs` and `init_failsafe_rpl_thread`; these were categorized as HIGH and MEDIUM. However, further examination indicates that each of the two functions has communication edges with other functions that were all classified as LOW. While the data is anecdotal, if these results generalize to other real datasets and prediction models, then *if one function f is labeled as LOW testable, other functions that can be reached through a communication edge from f should also be flagged as possible LOW testable.*

VI. Discussion and Implications

As presented in the previous section, we were able to demonstrate that CPM is useful for predicting testability of concurrent programs. Specifically, we showed (subject to stated threats to validity) that individual metrics in CPM are more effective at predicting concurrent program testability than SPM based on correlation analysis, linear regression, and classification. We also showed that test suite quality is important (due to information gain ratio of SCV for all four datasets). Using cross project prediction, we showed that models built on three datasets can be used to predict testability for the fourth dataset with good effectiveness (with one exception). We also showed that our LOW testable label was indeed tied to real concurrency faults in three of the

TABLE VI: **Performance on Cross Project Prediction using CPM.**

Prog.	MYSQL1			MYSQL2			MEMCACHED			SMALL		
	MYSQL2	MEMCACHED	SMALL	MYSQL1	MEMCACHED	SMALL	MYSQL1	MYSQL2	SMALL	MYSQL1	MYSQL2	MEMCACHED
CC	0.851	0.725	0.764	0.881	0.755	0.773	0.752	0.804	0.745	0.757	0.565	0.824
MAE	18.1	24.1	13	16	20.5	19.6	22.8	16.6	18.2	11.1	21.3	19.6
RMSE	28.9	32.5	22.3	18.8	25.8	28.7	29.2	20	26.3	17.7	32.1	30.2
BayesNet	0.823	0.746	0.766	0.796	0.582	0.359	0.6	0.859	0.508	0.714	0.526	0.528
Decision Tree	0.711	0.673	0.599	0.975	0.967	0.779	0.784	0.751	0.645	0.583	0.365	0.33
Logistics	0.693	0.687	0.519	0.975	0.967	0.779	0.753	0.762	0.571	0.67	0.602	0.603

four datasets, and found a possible link between functions in concurrent programs that should be examined further.

The two findings that might not be intuitive are that the test suite metric (SCV) has high predictability power and that functions that can be reached via communication edges from LOW testable functions might also be LOW testable.

Our results have implications for practitioners and researchers, discussed below.

Implications for Practitioners. Results indicate that concurrency-related code metrics can be effective and are more effective than SCM predictors learned from sequential program attributes. Practitioners can apply this finding by building a CCFG, obtaining the CPM metrics (we plan to provide a tool in the future to simplify this), and substituting their metrics into our learned model in order to predict LOW testable functions. In addition, our results showed that metric CCC was a good predictor of testability. As it increases, program testability decreases. Industry practitioners can use the CCFG to calculate CCC and examine its distribution for their project. Functions that have higher values of CCC should be examined and possibly subjected to additional code review and/or unit testing to lower the risk of concurrency faults. Also, functions that share communication edges with LOW testable functions warrant additional attention during software assurance activities.

Implications for Researchers. Our study shows that concurrent metrics can be used to predict testability of concurrent programs. Researchers should consider adding the CCFG to their arsenal of program representations. The CPM metrics may have other applications, such as for predicting fault prone components in concurrent programs, predicting change prone components in concurrent programs, and predicting the number of test that are needed to achieve coverage of concurrent programs. Additionally, the finding on communication edges implies that researchers should examine concurrent edge coverage more carefully as an important test criterion. Identification and test of functions that can reach LOW testable functions may hold promise for further reduction of concurrency faults and could be the focus of future work.

VII. Related Work

There has been some research on developing various software metrics to assess software quality [15]. For example, Lee et al. [38] proposed a set of micro-interaction metrics (MIMs) that leverage developers' interaction information combined with source code metrics to predict defects. Meneely [46] et al. examine structure of developer collaboration and use developer network derived from code information to predict defects. However, none of the above work has considered concurrency-related code metrics. The only related work found is the work by Mathews et al. [43] based on Ada programs. This work considers only the number of synchronizations and conditional branches that contain synchronization points without utilizing them to perform testability/defect prediction.

Although prediction of mutation scores of concurrent programs has not been previously researched, the related topic of using software metrics to predict faults in source code has been well researched. For example, Koru and Liu utilized static software measures along with defect data at the class level to predict bugs using machine learning. Machine learning techniques are popular for predicting software defects and testability [4], [11], [34], [49]. For example, Menzies et al. [47] conclude that static code metrics are useful in predicting defects under specific learning methods. Khoshgoftaar et al. [32] use a neural network to learn models from source code metrics to predict testability based on mutation analysis. Jalbert et al. [25] predict mutation scores by using source code metrics combined with coverage information. These techniques, however, focus on sequential programs while ignoring code attributes and testability for concurrent programs.

There has been a great deal of research on mutation testing for sequential programs [12], [42], [53]. Jia and Harman [27] provide a recent survey. In this work, we focus on techniques that share similarities with ours. There has also been some work on mutation testing for concurrent programs [6], [21], [31], which has been discussed in Section II. Other tools such as MuTMuT [20] have been used to optimize the execution of mutants by reducing interleaving space that has to be explored. None of the techniques, however, attempt to predict testability using software metrics.

VIII. Conclusions

This paper presents an approach to predict the testability of concurrent programs at the function level. We proposed five novel static code metrics and combined them with a dynamic test suite metric to learn four prediction models. We applied the models to four data sets from large and small real-world programs. Our results showed that our approach of using CPM significantly improved testability prediction and classification. In the future, we will consider other code metrics and test suite attributes and investigate their effectiveness.

IX. Acknowledgments

This work has been partially funded by NSF under grant CCF-1511117.

REFERENCES

[1] AGET. Multithreaded HTTP Download Accelerator. Web page. http://www.enderunix.org/aget/.

[2] E. Alpaydin. *Introduction to Machine Learning*. The MIT Press, 2004.

[3] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[4] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761.

[5] M. D. Bond, K. E. Coons, and K. S. McKinley. PACER: Proportional detection of data races. In *PLDI*, June 2010.

[6] J. Bradbury, J. Cordy, and J. Dingel. Mutation operators for concurrent java (j2se 5.0). In *International Workshop on Mutation Analysis*, pages 11–11, 2006.

[7] A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur. Applications of synchronization coverage. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 206–212, 2005.

[8] BZIP2. Parallel BZIP2 . Web page. http://compression.ca/pbzip2/.

[9] BZIP2SMP. Parallelizing BZIP2 for SMP machines. Web page. http://bzip2smp.sourceforge.net.

[10] A. Causevic, D. Sundmark, and S. Punnekkat. An industrial survey on contemporary aspects of software testing. In *International Conference on Software Testing, Verification and Validation*, pages 393–401, 2010.

[11] V. U. B. Challagulla, F. B. Bastani, I.-L. Yen, and R. A. Paul. Empirical assessment of machine learning based software defect prediction techniques. *International Journal on Artificial Intelligence Tools*, 17(02):389–400.

[12] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur. Interface mutation: An approach for integration testing. *IEEE Transactions on Software Engineering*, 27(3):228–247.

[13] D. Deng, W. Zhang, and S. Lu. Efficient concurrency-bug detection across inputs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, pages 785–802, 2013.

[14] D. Dimitrov, V. Raychev, M. Vechev, and E. Koskinen. Commutativity race detection. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 305–315, 2014.

[15] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. CRC Press, 1998.

[16] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *PLDI*, 2009.

[17] M. K. Ganai and C. Wang. Interval analysis for concurrent trace programs using transaction sequence graphs. In *Proceedings of the International Conference on Runtime Verification*, pages 253–269, 2010.

[18] S. Ghosh. Towards measurement of testability of concurrent object-oriented programs using fault insertion: A preliminary investigation. In *Proceedings of the IEEE International Workshop on Source Code Analysis and Manipulation*, pages 17–, 2002.

[19] B. Ghotra, S. McIntosh, and A. E. Hassan. Revisiting the impact of classification techniques on the performance of defect prediction models. In *Proceedings of the International Conference on Software Engineering - Volume 1*, pages 789–800, 2015.

[20] M. Gligoric, V. Jagannath, and D. Marinov. Mutmut: Efficient exploration for mutation testing of multithreaded code. In *International Conference on Software Testing, Verification and Validation*, pages 55–64, 2010.

[21] M. Gligoric, L. Zhang, C. Pereira, and G. Pokam. Selective mutation testing for concurrent code. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 224–234, 2013.

[22] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: An update. *Special Interest Group on Knowledge Discovery and Data Mining Explorations Newsletter*, 11(1):10–18, Nov. 2009.

[23] M. Harman, Y. Jia, and W. B. Langdon. Strong higher order mutation-based test data generation. In *Proceedings of the ACM SIGSOFT Symposium and the European Conference on Foundations of Software Engineering*, pages 212–222, 2011.

[24] S. Hong, J. Ahn, S. Park, M. Kim, and M. J. Harrold. Testing concurrent programs to achieve high synchronization coverage. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 210–220, 2012.

[25] K. Jalbert and J. S. Bradbury. Predicting mutation score using source code and test suite metrics. In *Proceedings of the International Workshop on Realizing AI Synergies in Software Engineering*, pages 42–46, 2012.

[26] Y. Jia and M. Harman. Higher order mutation testing. *Information and Software Technology Journal*, 51(10):1379–1393.

[27] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

[28] B. F. Joy Paul Guilford. *Fundamental Statistics in Psychology and Education*. McGraw-Hill, 1978.

[29] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 654–665, 2014.

[30] V. Kahlon, N. Sinha, E. Kruus, and Y. Zhang. Static data race detection for concurrent programs with asynchronous calls. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 13–22, 2009.

[31] L. W. G. Kaiser. Constructing subtle concurrency bugs using synchronization-centric second-order mutation operators. In *International Conference on Software Engineering and Knowledge Engineering*, 2011.

[32] T. Khoshgoftaar, E. Allen, and Z. Xu. Predicting testability of program modules using a neural network. In *IEEE Symposium on Application-Specific Systems and Software Engineering Technology.*, pages 57–62, 2000.

[33] T. Khoshgoftaar, R. Szabo, and J. Voas. Detecting program modules with low testability. In *Proceedings of International Conference on Software Maintenance*, pages 242–250, 1995.

[34] S. Kim, E. J. Whitehead, Jr., and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196.

[35] S. Kullback. *Information Theory and Statistics*. A Wiley publication in mathematical statistics. Dover Publications, 1997.

[36] M. Kusano and C. Wang. CCmutator: A mutation generator for concurrency constructs in multithreaded C/C++ applications. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 722–725, 2013.

[37] C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis transformation. In *International Symposium on Code Generation and Optimization.*, pages 75–86, 2004.

[38] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In. Micro interaction metrics for defect prediction. In *Proceedings of the ACM SIGSOFT Symposium and the European Conference on Foundations of Software Engineering*, pages 311–321, 2011.

[39] S. Lu, W. Jiang, and Y. Zhou. A study of interleaving coverage criteria. In *The International Symposium on the Foundations of Software Engineering: Companion Papers*, pages 533–536, 2007.

[40] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, Mar. 2008.

[41] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM Special Interest Group on Programming Languages conference on Programming language design and implementation*, 2005.

[42] Y.-S. Ma, J. Offutt, and Y.-R. Kwon. Mujava: A mutation system for java. In *Proceedings of the International Conference on Software Engineering*, pages 827–830, 2006.

[43] M. E. Mathews and S. Tu. Metrics measuring control flow complexity in concurrent programs. *IEEE Transactions on Computers*, 3(5):5.

[44] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320.

[45] memcached. memcached - a distributed memory object caching system. Web page. http://memcached.org.

[46] A. Meneely, L. Williams, W. Snipes, and J. Osborne. Predicting failures with developer networks and social network analysis. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 13–23, 2008.

[47] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33(1):2–13.

[48] T. Menzies, O. Jalali, J. Hihn, D. Baker, and K. Lum. Stable rankings for different effort models. *Automated Software Engineering*, 17(4):409–437, Dec. 2010.

[49] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *ACM/IEEE International Conference on Software Engineering.*, pages 181–190, 2008.

[50] MySQL. MySQL. Web page. https://www.mysql.com/.

[51] A. Orso and G. Rothermel. Software testing: A research travelogue (2000–2014). In *Proceedings of the on Future of Software Engineering*, pages 117–132, 2014.

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

[53] D. Schuler, V. Dallmeier, and A. Zeller. Efficient mutation testing by checking invariant violations. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 69–80, 2009.

[54] J. M. Voas. Pie: A dynamic failure-based technique. *IEEE Transactions on Software Engineering*, 18(8):717–727.

[55] J. M. Voas and K. W. Miller. Putting assertions in their place. In *Proceedings of International Symposium on Software Reliability Engineering*, pages 152–157, 1994.

[56] J. M. Voas, K. W. Miller, and J. E. Payne. An empirical comparison of a dynamic software testability metric to static cyclomatic complexity. In *Proceedings of the International Conference on Software Quality Management*, pages 431–445, 1994.

[57] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition*. Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, 2005.

[58] W. E. Wong, editor. *Mutation Testing for the New Century*. Advances in Database Systems. Springer, 2001.

[59] H. Zhang, X. Zhang, and M. Gu. Predicting defective software components from code complexity measures. In *Pacific Rim International Symposium on Dependable Computing*, pages 93–96, 2007.

[60] W. Zhang, C. Sun, and S. Lu. Conmem: Detecting severe concurrency bugs through an effect-oriented approach. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 179–192, 2010.