Differential Testing of Concurrent Classes

Valerio Terragni
University of Auckland
Auckland, New Zealand
v.terragni@auckland.ac.nz

Shing-Chi Cheung
The Hong Kong University of Science and Technology
Hong Kong, China
scc@cse.ust.hk

***Abstract*—Concurrent programs are pervasive, yet difficult to write. The inherent complexity of thread synchronization makes the evolution of concurrent programs prone to concurrency faults. Previous work on regression testing concurrent programs focused on reducing the cost of re-run the existing tests. However, existing tests may not be able to expose the regression faults in the modified program. In this paper, we present CONDIFF a differential testing technique that generates concurrent tests and oracles to expose behavioral differences between two versions of a given concurrent class. Since concurrent programs are non-deterministic, this involves exploring all possible non-deterministic thread interleavings of each generated test on both versions. However, we can afford to analyze only a few concurrent tests due to the high cost of exhaustive interleaving exploration. To address the challenge, CONDIFF leverages the information of code changes and trace analysis to analyze only those concurrent tests that are likely to expose behavioral differences (if they exist). We evaluated CONDIFF on a set of Java classes. Our results show that CONDIFF can effectively generate concurrent tests that expose behavioral differences.**

***Index Terms*—differential testing, concurrency, regression testing, test generation, oracle generation, thread-safety**

I. INTRODUCTION

The ubiquity of multi-core processors has led to the pervasive adoption of concurrency programming to leverage the available parallelism. The inherent complexity of concurrency and synchronization makes the evolution of concurrent programs an error-prone activity [1] as code modification could introduce concurrency faults. Such faults are difficult to expose at runtime since they only manifest under some specific – usually few – non-deterministic thread interleavings [2], [3].

Regression testing is the activity to guard against such faults by identifying tests that behave differently before and after code modifications [4], [5]. Developers usually start regression testing by re-running on the modified version the existing tests that run successfully before the code modification [5]. This to check if any of such tests fail unexpectedly.

A fundamental problem of such an approach is that the time required to re-execute all existing tests can easily exceed the affordable time budget [5]–[8]. This problem is exacerbated when regression testing concurrent programs, as concurrent (multi-thread) tests are usually long-running in order to explore a large number of thread interleavings [9], [10]. Researchers have proposed techniques that tackle this problem in the context of regression testing of concurrent programs [10], [11]. Given a set of existing concurrent tests, these techniques perform selection at test (SIMRT [11]) or at interleaving (RECONTEST [10]) levels. However, the existing tests may not necessarily expose all regression faults in the modified program [12]–[14], as tests are often written without considering how the program will be modified [15]. They may miss either the scenarios that exhibit behavioral differences or the oracles capable of detecting such differences [16], [17]. Regression testing often requires defining additional regression tests and oracles [13].

Differential Testing [18] is an automated approach that exposes various errors by analyzing two or more comparable systems [19]–[28]. One can employ differential testing to generate new regression tests by treating two versions of the same program as comparable systems [15], [16], [29]–[31].

In a nutshell, differential testing for regression testing works in two phases: *test generation* and *behavioral checking*. The first phase generates a large number of tests that exercise the modified parts of the program. The second phase executes each generated test on both program versions to collect the test outputs (e.g., method return values, program states, exceptions) and reports a behavioral difference (oracle violation) if the outputs of a test diverge across the versions [16], [29]. By using the two versions as cross-referencing oracles, differential testing is able to detect also those regression faults that do not exhibit explicit erroneous behaviors like crashes or assertion failures [16]. Note that differential-testing cannot (nor aim to) guess developers' intentions [15], [16], [29]–[31]. After behavioral differences are exposed, developers need to identify the unintentional differences from the intentional ones.

In this paper, we propose the use of differential testing for generating tests and oracles to expose concurrency faults introduced by code modifications. More specifically, we target regression testing of Object-Oriented (OO) concurrent programs at class level. Existing differential testing techniques for OO programs mostly target sequential faults (e.g., [16], [17], [29], [30], [32]), which makes them inadequate. First, they generate sequential tests [33] (i.e., single-threaded method call sequences), which cannot manifest concurrency faults. Second, they assume deterministic executions, while multi-thread executions are intrinsically non-deterministic [34], [35].

To address this gap, this paper presents a novel differential testing technique called *CONcurrent DIFFerential testing* (**CONDIFF**) aiming at exposing non-deterministic behavioral differences between two versions of a concurrent class. CONDIFF adapts the two phases of traditional differential testing as follows. In the test generation phase, CONDIFF generates concurrent unit tests that invoke the changed methods of the class.

A *concurrent test* consists of multiple concurrently executing threads that exercise a shared instance of the class [36]–[38]. In the behavioral checking phase, ConDiff executes each generated test on both program versions under all possible non-deterministic thread interleavings (using the stateful model-checker JPF [39]) while collecting the test outputs. ConDiff needs to explore all interleavings to guarantee that behavioral differences are caused by code changes and not by the non-determinism of concurrent executions.

These adaptations, although necessary, introduce a fundamental challenge. The behavioral checking phase is often computational expensive since there is a huge number of possible interleavings for a concurrent test [40], [41]. As such, the behavioral checking can analyze, within a reasonable time budget, only a few concurrent tests. This is an issue because there are a lot of possible concurrent tests that exercise a given concurrent class [36]. Test generation often needs to generate many concurrent tests before finding a test that manifests the concurrency fault (behavioral difference) [36], [38].

To address the challenge, our intuition is that manifesting different interleavings across the two program versions before and after modification is a pre-requisite for a concurrent test to expose non-deterministic behavioral differences. Based on this intuition, ConDiff performs an additional *filtering phase* after the test generation and before the behavioral checking phase. In this phase, ConDiff adapts the change impact analysis from our previous work ReConTest [10] to identify whether each generated test can expose different interleavings across program versions. ConDiff performs the (expensive) behavioral checking only for tests meeting this criterion. The analysis is precise and inexpensive. Its precision comes from using information gathered during a test's real execution, and it avoids high costs by inferring different interleavings without enumerating them (which is exactly the cost to avoid).

We evaluate ConDiff with 24 revisions of Java concurrent classes that manifest behavioral differences only under particular thread interleavings. ConDiff is able to expose all of them in four minutes on average. Compared with a version of ConDiff without the filtering phase, ConDiff reduces the number of tests for behavioral checking as much as 96.5x (15.5x on average), and it reduces the time for revealing the behavioral differences as much as 9.9x (3.2x on average). In summary, this paper makes four contributions:

- We present the first formal definition of differential testing with the goal of exposing concurrency faults that manifest only under specific non-deterministic thread interleavings;
- We prove that, under a certain assumption, concurrent tests that exhibit behavioral differences are included in the class of concurrent tests that exhibit different interleavings;
- We leverage this result and propose ConDiff, the first technique to effectively expose behavioral differences between two versions of a concurrent class;
- We implement the proposed technique for Java programs and we demonstrate its effectiveness on 24 faulty revisions.
- We released our experimental data to facilitate future work in this area [42].

II. Background and Problem Formulation

An *Object-Oriented* OO program is composed of a set of classes. Each class defines a set of fields and methods. During a program execution, each object (i.e., an instance of a class) has a *state* that encompasses all of the current values of each of its fields (e.g., `3`, `true`, `"hi!"` for primitive fields). The value of a non-primitive field is the state of the object referenced by the field in a deep-copy semantic [43]. Each method in a class has possibly empty sequence of input parameters, which can be primitive values or object references. We treat the object receiver of an instance method as the method's first parameter [33]. Since the behavior of method executions often depends on the value/state of its input parameters [44], a test for OO programs is no longer a set of input values but a sequence of method calls that instantiates non-primitive parameters and brings them to certain states [33], [44], [45].

Definition 1. *A **sequential test (st)** is an ordered sequence of method calls $st = \langle mc_1, \dots, mc_n \rangle$ that exercises from a single thread the public interface of a program.*

Differential testing for OO sequential classes [16], [29], [30] aims to determine via dynamic analysis how the behaviors of a program $\mathcal{P}$ differ from the behaviors of $\mathcal{P}'$, a modified version of $\mathcal{P}$. Previous work mostly quantify behaviors by considering the *observable outputs* of *outer-most*[1] *method calls*.

Definition 2. *The **observable output** $\mathrm{O_i}$ of a method call $mc_i \in st$ consists of two elements:*

(i) the states of the non-primitive input parameters (including the object receiver state) when mc exits

(ii) mc return value (if any)

If the return type is non-primitive, the return value is the state of the returned object. If a method terminates with an exception, its return value is the stack trace of the exception [16].

Proving behavioral equivalence is undecidable in general (Rice's theorem [46]) and intractable in practice [47]. This is because, given any non-trivial OO program the are infinitely many sequential tests, since the length of method call sequences can be arbitrary long [33]. On the contrary, differential testing can prove that two program versions are *behaviorally different* by proving the existence of a difference-revealing sequential test. Following previous work, we define such tests as follows.

Executing a sequential test $st = \langle mc_1, \dots, mc_n \rangle$ induces a sequence of state transitions $S_0 \xrightarrow{mc_1} S_1 \xrightarrow{mc_2} S_2 \dots \xrightarrow{mc_n} S_n$. Let S_{i-1} denote the state of the program before mc_i is invoked, and S_i the state after. Let $I_i \subseteq S_{i-1}$ denote the states/values of mc_i's parameters before mc_i is invoked. Let $\mathrm{O}_i \subseteq S_i$ denote the observable output of mc_i when it exists. Let $\mathrm{O}_i^{\mathcal{P}}$ and $\mathrm{O}_i^{\mathcal{P}'}$ denote O_i when m_i is executed with $\mathcal{P}$ and $\mathcal{P}'$, respectively.

Definition 3. *A sequential test st is **difference-revealing** if and only if $\exists\ mc_i \in st$, $O_i^{\mathcal{P}}(I) \neq O_i^{\mathcal{P}'}(I)$, where $I = I_i^{\mathcal{P}} = I_i^{\mathcal{P}'}$.*

[1]Each method call execution in *st* could transitively invoke other method calls. We refer as *outer-most* the method calls in *st*, while we refer as *nested* those that are transitively invoked by the execution of outer-most method calls.

Differential testing for OO concurrent classes. In this paper, we study *the problem of exposing non-deterministic behavioral differences between two versions of a concurrent class*. A class is called *concurrent* if instances of such a class are meant to be accessed concurrently from multiple threads. Figure 2 shows an example of a Java concurrent class. A concurrent class is *thread-safe* if it encapsulates synchronization mechanisms (e.g., synchronized blocks in `Java` and mutexes and semaphores in `C`) that guarantee that the same instance of the class can be correctly accessed from multiple threads without additional synchronization mechanisms other than the ones implemented in the class [36], [38], [48]. A class is *thread-unsafe* otherwise. Differential testing can be an effective approach to expose thread-safety violations introduced by code changes. The key idea is to assume that the previous version of a concurrent class is thread-safe. Behavioral differences with a modified version of the class represent thread-safety violations.

Traditional differential testing techniques for sequential programs are inadequate for concurrent classes. They can only reveal those differences that manifest sequentially, but not those that manifest concurrently. The reason is twofolds.

Reason 1) *They generate sequential (single-threaded) tests, which unlikely exercise concurrency behaviors.* Threads are rarely spawned by the methods of concurrent classes [36], [49]. Instead, the logistics of creating and running new threads to invoke the methods of such class concurrently are often taken care of by client code (e.g., test code). Therefore, we need (multi-threaded) concurrent tests [36], [37], [49]–[51] to exercise and validate concurrency behaviors.

Definition 4. *A **concurrent test (ct)** is a series of method call sequences ct=$\langle$prefix, suffix$_1$, ..., suffix$_n\rangle$ ($n \geq 2$) that exercise from multiple threads a shared instance of a given class [36].*

A concurrent test executes each call sequences in a distinct thread. The *prefix* is a call sequence to be executed before all suffixes. The prefix contains method calls that create shared instances of a given class that will be accessed concurrently by the suffixes. The prefix also contains method calls to bring the shared instances to certain states that may allow the suffixes to behave differently. A suffix is a call sequence supposed to be executed concurrently with other suffixes after executing the common prefix. The suffixes share instances created by the prefix and use them as input parameters. Intuitively, we need at least two concurrent threads (suffixes) ($n \geq 2$) to create meaningful concurrent tests [36], [41], [49].

Reason 2) *They assume deterministic executions, while concurrent executions are intrinsically non-deterministic,* i.e., the execution of a method call mc_i is deterministic given I_i [43]. Under such assumption, these techniques can guarantee that the reported behavioral differences is caused by the code revision. This is a reasonable assumption for sequential programs[2]. However, this assumption is not realistic for concurrent programs, as concurrent executions are intrinsically non-deterministic [34].

[2]Examples of non-deterministic sequential programs are those that behave differently based on the current time, resource consumptions, network resources, and randomness. Programs with such characteristics are fairly uncommon [52].

The behavior of a method call when executed concurrently depend not only on the inputs, but also on the manifested non-deterministic execution order of shared-memory accesses across multiple threads, called *interleaving* [40], denoted by σ.

We now define a difference-revealing concurrent test by taking into account the non-determinism of multi-threaded executions. We use *IS*($\mathcal{P}$, ct) to denote the set of all *feasible* interleavings that can be manifested when ct executes on $\mathcal{P}$. Let $O_i^{\mathcal{P}}(I_i^{\mathcal{P}}, \sigma)$ denote the observable outputs of executing a method call mc_i on program $\mathcal{P}$ with input parameters $I_i^{\mathcal{P}}$ and interleaving $\sigma \in$ *IS*($\mathcal{P}$, ct).

Definition 5. *A concurrent test ct is **difference-revealing** if and only if $\exists mc_i \in$ suffix$_j$ of ct such that $I = I_i^{\mathcal{P}} = I_i^{\mathcal{P}'}$ and*

$$\left\{ \bigcup_{\sigma \in IS(\mathcal{P}, ct)} O_i^{\mathcal{P}}(I, \sigma) \right\} \triangle \left\{ \bigcup_{\sigma' \in IS(\mathcal{P}', ct)} O_i^{\mathcal{P}'}(I, \sigma') \right\} \neq \emptyset$$

$\triangle$ is the symmetric difference $A \triangle B = (A - B) \cup (B - A)$. A concurrent test is difference-revealing if executed on $\mathcal{P}'$ (or $\mathcal{P}$) under a certain interleaving, it produces an observable output that cannot be produced under any of the interleavings that can be manifested by the test when executes on $\mathcal{P}$ (or $\mathcal{P}'$).

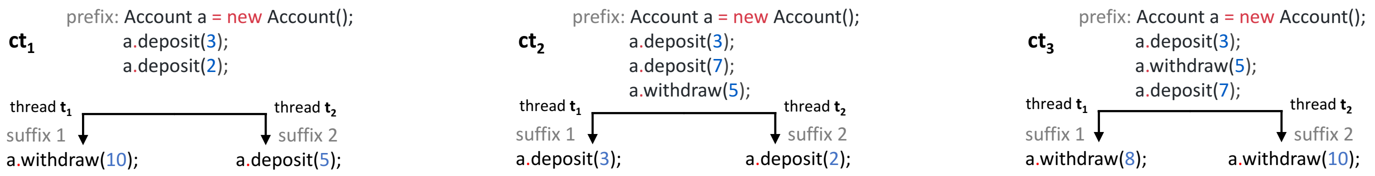
Problem Formulation. *Given two versions $\mathcal{P}$ and $\mathcal{P}'$ of a concurrent class and a time budget $\mathcal{B}$, our research goal is to automatically infer within $\mathcal{B}$ if $\mathcal{P}$ and $\mathcal{P}'$ are behavioral different by generating a difference-revealing concurrent test.*

III. Motivating Example

Figure 2 shows a simple example that we use to illustrate the limitations of traditional approaches and the challenges of differential testing concurrent classes. It shows two consecutive versions ($\mathcal{P}$ and $\mathcal{P}'$) of a Java concurrent class by marking the + added, − deleted lines. The public interface exposes two operations: the withdraw and deposit of a positive amount.

In $\mathcal{P}$ every public method is declared `synchronized`. Therefore, when a thread invokes one of these methods, it acquires the lock associated with the object instance before entering the method and releases it after. As such, if two threads invoke these methods concurrently on the same instance their executions are mutually exclusive. This makes the original version of the class thread-safe.

To boost execution parallelism, developers reduce the size of the critical sections in $\mathcal{P}'$ by postponing the lock acquisition from the method declaration to the method body. The change promotes concurrency: multiple threads can now check in parallel if the thread-local variable `amount` is positive (line 7 and 13 in Figure 2). However, the developers erroneously left the statement at line 14 outside the critical section. Line 14 performs an access to the thread-shared variable `balance`, which could read a stale value of `balance` while another thread is updating it. This makes the new version of the class no longer thread-safe. In fact, concurrent tests exist that behave differently when executed on $\mathcal{P}$ and $\mathcal{P}'$. The goal of differential testing is to generate such tests automatically and to report their behavioral differences to the developers.



```
prefix: Account a = new Account();
        a.deposit(3);
        a.deposit(2);
suffix 1: a.withdraw(10);
suffix 2: a.deposit(5);
```

```
prefix: Account a = new Account();
        a.deposit(3);
        a.deposit(7);
        a.withdraw(5);
suffix 1: a.deposit(3);
suffix 2: a.deposit(2);
```

```
prefix: Account a = new Account();
        a.deposit(3);
        a.withdraw(5);
        a.deposit(7);
suffix 1: a.withdraw(8);
suffix 2: a.withdraw(10);
```

Fig. 1. Examples of concurrent tests for the class in Figure 2.

```
1      public class Account {
2       private int balance;
3       public Account() {
4            this.balance = 0;
5       }
    -   public synchronized void deposit(int amount) {
6   +   public void deposit(int amount) {
7          if (amount <= 0) {return;}
8   +      synchronized(this){
9            balance = balance + amount;
10  +      }
11      }
    -   public synchronized void withdraw(int amount) {
12  +   public void withdraw(int amount) {
13           if (amount <= 0) {return;}
14           if (balance >= amount) {
15  +        synchronized(this) {
16               balance = balance - amount;
17           }
18  +        }
19      }
20     }
```

Fig. 2. Two versions of a concurrent class that exhibit behavioral differences when the class is accessed concurrently from multiple threads.

We now discuss three concurrent tests (ct_1, ct_2, and ct_3) shown in Figure 1. Among the three concurrent tests, ct_3 is the only difference-revealing test (Definition 5). Each test has two concurrent threads t_1 and t_2 executing the *suffix*$_1$ and *suffix*$_2$, respectively. All suffixes are composed of one method call. We use mc_{S1} to denote the one of *suffix*$_1$ and mc_{S2} the one of *suffix*$_2$. The input of the method calls mc_{S1} and mc_{S2} (I_{S1} and I_{S2}) comprises (i) the state of the object receiver `a` after executing the common prefix and (ii) the value of the input parameter `amount`. For any test, $I^{P}_{S1} = I^{\mathcal{P}'}_{S1}$ and $I^{P}_{S2} = I^{\mathcal{P}'}_{S2}$ as the behavioral difference cannot manifest sequentially.

To ease the following discussion, Table I shows for each concurrent test (ct_1, ct_2 and ct_3) and program version ($\mathcal{P}$ and $\mathcal{P}'$) the possible interleavings and the corresponding observable outputs of mc_{S1} and mc_{S2} (O_{S1}, O_{S2}). For describing the interleavings we use $W^t_l(x)$ and $R^t_l(x)$ to denote the write and read accesses, respectively, on the shared-memory `balance` with value x, triggered by the execution of line l, performed by thread t. For example, $R^{t1}_{14}(5)$, represents that thread $t1$ reads `5` from the shared-memory `balance` by executing the statement at line 14[3]. The observable outputs of mc_{S1} and mc_{S2} (O_{S1} and O_{S2}) are composed of the state of the object receiver `a` when the method calls exits.

There are two possible interleavings when the first test (ct_1) executes on $\mathcal{P}$ (see Table I). In σ_1, t_1 evaluates the if condition

[3]Note that we do not show the execution order of shared-memory accesses in the *prefix*. By being executed sequentially before all the concurrent suffix the execution order is always the same.

TABLE I
INTERLEAVINGS AND OUTPUTS OF THE TESTS IN FIG. 1

		σ **interleaving**	$\mathbf{O}_{S1}$	$\mathbf{O}_{S2}$
$\mathbf{ct_1}$	$\mathcal{P}$	1 $\langle R^{t1}_{14}(5), R^{t2}_{9}(5), W^{t2}_{9}(10)\rangle$	5	10
		2 $\langle R^{t2}_{9}(5), W^{t2}_{9}(10), R^{t1}_{14}(10), R^{t1}_{16}(10), W^{t1}_{16}(0)\rangle$	0	10
	$\mathcal{P}'$	3 same as σ_1	5	10
		4 same as σ_2	0	10
		5 $\langle R^{t2}_{9}(5), R^{t1}_{14}(5), W^{t2}_{9}(10)\rangle$	5	10
$\mathbf{ct_2}$	$\mathcal{P}$	6 $\langle R^{t1}_{9}(5), W^{t1}_{9}(8), R^{t2}_{9}(8), W^{t2}_{9}(10)\rangle$	8	10
		7 $\langle R^{t2}_{9}(5), W^{t2}_{9}(7), R^{t1}_{9}(7), W^{t1}_{9}(10)\rangle$	10	7
	$\mathcal{P}'$	8 same as σ_6	8	10
		9 same as σ_7	10	7
$\mathbf{ct_3}$	$\mathcal{P}$	10 $\langle R^{t1}_{14}(10), R^{t1}_{16}(10), W^{t1}_{16}(2), R^{t2}_{14}(2)\rangle$	2	2
		11 $\langle R^{t2}_{14}(10), R^{t2}_{16}(10), W^{t2}_{16}(0), R^{t1}_{14}(0)\rangle$	0	0
	$\mathcal{P}'$	12 same as σ_{10}	2	2
		13 same as σ_{11}	0	0
		14 $\langle R^{t1}_{14}(10), R^{t2}_{14}(10), R^{t1}_{16}(10), W^{t1}_{16}(2), R^{t2}_{16}(2), W^{t2}_{16}(-8)\rangle$	2	-8
		15 $\langle R^{t2}_{14}(10), R^{t1}_{14}(10), R^{t1}_{16}(10), W^{t1}_{16}(2), R^{t2}_{16}(2), W^{t2}_{16}(-8)\rangle$	2	-8
		16 $\langle R^{t1}_{14}(10), R^{t1}_{16}(10), R^{t2}_{14}(10), W^{t1}_{16}(2), R^{t2}_{16}(2), W^{t2}_{16}(-8)\rangle$	2	-8
		17 $\langle R^{t2}_{14}(10), R^{t1}_{14}(10), R^{t2}_{16}(10), W^{t2}_{16}(0), R^{t1}_{16}(0), W^{t1}_{16}(-8)\rangle$	-8	0
		18 $\langle R^{t1}_{14}(10), R^{t2}_{14}(10), R^{t2}_{16}(10), W^{t2}_{16}(0), R^{t1}_{16}(0), W^{t1}_{16}(-8)\rangle$	-8	0
		19 $\langle R^{t2}_{14}(10), R^{t2}_{16}(10), R^{t1}_{14}(10), W^{t2}_{16}(0), R^{t1}_{16}(0), W^{t1}_{16}(-8)\rangle$	-8	0

at line 14 to false and therefore it does not perform the withdraw operation. Conversely, in σ_2, t_1 evaluates the condition to true since line 14 is executed after t_2 terminates the deposit operation. The test has a non-deterministic behavior because t_1 can perform the withdraw only under σ_1. $O^{\mathcal{P}}_{S1}(I, \sigma_1) = 5$, while $O^{\mathcal{P}}_{S1}(I, \sigma_2) = 0$. There are three different interleavings when the test executes on $\mathcal{P}'$ ($\sigma_3, \sigma_4, \sigma_5$ Table I). σ_5 is a new interleaving introduced by the code changes, which does not lead to new observable outputs. As a result, the concurrent test ct_1 is not difference-revealing. Note that under Def. 3, ct_1 would be erroneously marked difference-revealing if σ_1 is observed when ct_1 executes on $\mathcal{P}$ and σ_3 on $\mathcal{P}'$.

The second test (ct_2) performs two concurrent deposits on a shared account with balance 5. There are two possible interleavings when the test executes on both $\mathcal{P}$ and $\mathcal{P}'$. All of these interleavings produce the same observable outputs.

The third test (ct_3) performs two concurrent withdrawals on a shared account with balance `10`. Only one can be completed as the balance is not enough to perform both of them. There are two possible interleavings when the test executes on $\mathcal{P}$. When the test executes on $\mathcal{P}'$ it can manifest eight different interleavings, six of them cannot be manifested in $\mathcal{P}$. In $\mathcal{P}'$ the synchronization mechanism is changed and now the read access at line 14 can interleave between the read and write accesses at line 16. For such new interleavings the observable outputs of mc_{S1} or mc_{S2} is `-8`. This behavior cannot be observed in $\mathcal{P}$ under all possible interleavings. As a result, ct_3 is a difference-revealing test under Definition 5.

From this motivating example, we can identify two main challenges of differential testing for concurrent classes.

Challenge 1*: Differential testing needs to explore a large number of concurrent tests*, since only particular combinations of sequential prefix, concurrent suffixes and input parameter values manifest behavioral differences. For example, changing the order between `a.deposit(7)` and `a.withdraw(5)` in the prefix of ct_3 or changing the input parameter `amount` to `5` on both mc_x and mc_y would make $ct_3 \notin CT_{dr}$. If we set a bound l on the maximum call sequence length, and a bound k on the number of parameters values for each method call in a test. Given a class with m public methods there are at most $(k \cdot m)^{(n+1) \cdot l}$ concurrent tests, where n (≥ 2) is the number of suffixes (see Definition 5). For illustration, let us consider the class in Figure 2 ($m = 2$). With a relatively small values of l ($= 10$) and k ($= 5$) there are 10^{30} possible concurrent tests.

Challenge 2: *High cost of behavioral checking.* To infer if a concurrent test ct is difference-revealing by Definition 5 requires to explore all interleavings in $IS(\mathcal{P}, ct)$ and all interleavings in $IS(\mathcal{P}', ct)$. There are $\frac{(M_1+M_2)!}{M_1! \cdot M_2!}$ possible interleavings for a concurrent test with two concurrent threads, where M_1 is the number of shared-memory accesses triggered by the first thread, and M_2 that of the second. For illustration, consider ct_3 (M_1=M_2=3). In the worst case (when all interleavings are feasible on both program versions), ConDiff needs to execute ct on 40 different interleavings while collecting the observable outputs of the method calls. With the increase of M_1 and M_2 this number grows rapidly. For example, it becomes $\sim 7 \cdot 10^{11}$ for M_1=M_2=10.

IV. ConDiff

These challenges highlight the need for a differential testing approach that efficiently identifies difference-revealing concurrent tests while minimizing the search space. A common strategy, widely used in differential testing for sequential programs [16], [29], [30], [53], is to generate tests that invoke only the changed methods. However, this alone does not significantly reduce the search space, as traditional approaches still require generating many sequential tests to reveal behavioral differences [29]. Generating many tests is manageable for sequential programs but problematic for concurrent programs due to the high cost of behavioral checking.

Our intuition to further reduce the search space is that the set of difference-revealing concurrent tests ($\mathbf{CT_{dr}}$) can be over-approximated, under a certain assumption, by $\mathbf{CT_{\Delta IS}}$ the set of tests that manifest different interleavings across $\mathcal{P}$ and $\mathcal{P}'$, i.e., $\mathbf{CT_{\Delta IS}} = \{ct \in CT : IS(\mathcal{P}, ct) \bigtriangleup IS(\mathcal{P}', ct) \neq \emptyset\} \subseteq CT$.

The following theorem and proof assume that if a test follows the same interleaving (i.e., same memory accesses, same values, and same order of accesses) it will produce the same observable outputs. This is a common assumption used in many deterministic record-replay techniques for concurrent programs [54], [55]. This implies that (i) the call sequences in a concurrent test do not reveal differences when executed sequentially (Definition 3), which can be easily verified by traditional methods, and (ii) the thread scheduler is the only source of non-determinism[2].

Theorem 1. *If a concurrent test ct is difference-revealing ($ct \in CT_{dr}$) then ct exhibits different interleavings if executed on $\mathcal{P}$ and on $\mathcal{P}'$ ($ct \in CT_{\Delta IS}$). i.e.,* $\mathbf{CT_{dr}} \subseteq \mathbf{CT_{\Delta IS}}$

Proof. It will be proven by contradiction. Let assume that $\exists ct$ such that $ct \in CT_{dr}$ and $ct \notin CT_{\Delta IS}$. Because $ct \in CT_{dr}$ from Definition 5 we have that one or both of the following conditions are true ① $\exists mc_i \in ct : \exists \sigma' \in IS(\mathcal{P}', ct)$ such that $\forall \sigma \in IS(\mathcal{P}, ct)$, $O_i^{\mathcal{P}}(I, \sigma) \neq O_i^{\mathcal{P}'}(I, \sigma')$ ② $\exists mc_i \in ct : \exists \sigma \in IS(\mathcal{P}, ct)$ such that $\forall \sigma' \in IS(\mathcal{P}', ct)$, $O_i^{\mathcal{P}'}(I, \sigma') \neq O_i^{\mathcal{P}}(I, \sigma)$. Let consider case ①. $IS(\mathcal{P}, ct) = IS(\mathcal{P}', ct)$ because we assume that $ct \notin CT_{\Delta IS}$, i.e., ct exhibits the same interleavings if executed on $\mathcal{P}$ and $\mathcal{P}'$. This means that the interleaving $\sigma' \in IS(\mathcal{P}', ct)$ (from ①) can also manifest when ct is executed on $\mathcal{P}$, i.e., $\sigma' \in IS(\mathcal{P}, ct)$. Thus $\exists \sigma \in IS(\mathcal{P}, ct)$ ($\sigma = \sigma'$) such that $O_i^{\mathcal{P}}(I, \sigma) = O_i^{\mathcal{P}'}(I, \sigma')$. This is a contradiction because we assumed that such an interleaving does not exist ($ct \in CT_{dr}$). The proof is analogous for the symmetric case ② □.

Theorem 1 implies that the set $CT_{\Delta IS}$ gives a precise, though not minimal, approximation of CT_{dr}. It is not minimal because different interleavings can result in equivalent behaviors, as seen with σ_5 in Table I, which produces the same outputs as the old interleaving σ_1. However, ConDiff can safely skip behavioral checks for tests not in $CT_{\Delta IS}$ (e.g., ct_2 in Figure 1) without missing behavioral differences. We now explain how ConDiff generates tests and filters those in $CT_{\Delta IS}$.

A. ConDiff *Algorithm*

Function ConDiff in Figure 3 summarizes the ConDiff approach. Given two versions of a concurrent class, $\mathcal{P}$ and $\mathcal{P}'$, ConDiff alternates between generating concurrent tests and exploring their interleaving space until it finds a difference-revealing test or the time limit expires. To reduce the cost of behavioral checks, ConDiff limits tests to two concurrent threads (i.e., *suffix*1 and *suffix*2), a common practice [41] as most concurrency faults occur with just two threads [56].

ConDiff starts by statically analyzing $\mathcal{P}$ and $\mathcal{P}'$ (line 2) to identify the set of public methods (CM) that contain at least one changed statement (regardless of the change type). Note that, considering only changes to synchronization-related statements (e.g, synchronization blocks) would likely miss behavioral differences. This is because, in principle, any code change could introduce concurrency faults [1], [10], [11] (see Column 8 Table II). Given CM, ConDiff computes the set of changed method pairs (CMP) (line 3) that constitutes the coverage targets of ConDiff. CMP is an adaptation of the *concurrent method pairs metric* [57] recently exploited by Choudhary et al. [38] to improve the effectiveness of concurrent test generation. The original metric is defined as the set of all possible pairs of public methods of a class ($\langle$`m1, m2`$\rangle$ and $\langle$`m2, m1`$\rangle$ is the same pair). ConDiff adapts this metric for differential testing by considering only those method pairs composed of at least one changed method. For example, in Figure 2 all public methods are changed, thus CMP = $\{\langle$`w, w`$\rangle$, $\langle$`w, d`$\rangle$, $\langle$`d, d`$\rangle\}$, where `w` is the method withdraw and `d` is the method deposit.

CONDIFF generates a random prefix to instantiate the class under test and performs up to l method calls (GENRANDOMPREFIX, line 5) to set the object in a state where suffixes may behave differently. Following OO test generators, we assume method calls that do not throw exceptions in sequential execution result in valid states. If the prefix shows behavioral differences sequentially (Definition 3), CONDIFF skips it and reports a sequential regression warning (ISDIFFSEQ, lines 6-7). For each method pair $\langle m_1, m_2 \rangle$ in CMP, CONDIFF generates k pairs of random suffixes (function GENRANDOMSUFFIX lines 10-11) using a different combination of randomly chosen parameters values (e.g., value of `amount` in Figure 2). The suffixes use the same the same object under test created by the prefix as an input parameter (e.g., the object receiver `a` in Figure 1) to trigger shared-memory accesses. If the suffixes reveal differences after the prefix in sequential execution, CONDIFF skips them (lines 13-14). Then it creates a new concurrent test *ct* by assembling the prefix and suffixes (line 15). Lines 16-17 perform the filtering phase. For only the unfiltered tests, CONDIFF performs the behavioral checking, which follows Definition 5 (lines 18-24).

B. Filtering Phase

Recent regression testing techniques use change impact analysis (CIA) [58] to reduce the cost of re-running concurrent tests on $\mathcal{P}'$ [10], [11], [59]. These techniques show that new interleavings (i.e., new behaviors) can be detected through *impacted* shared-memory accesses [10]. Impact occurs in two ways: (i) a *new* access triggered by added/changed statements or execution paths, or (ii) an *old* access with changes to its *Concurrency Context* (CC) [10] (i.e., *lockset* or happens-before relations) enabling new interleavings. For example, R^{t1}_{14} of ct_3 in Figure 1 is an impacted access of type (ii). For example, when ct_3 runs on $\mathcal{P}$, the *lockset* of R^{t1}_{14} (the locks held by thread t_1 during access [60]) includes the object's lock (`this`). On $\mathcal{P}'$, however, the lockset is empty. In contrast, ct_2 triggers no impacted shared-memory accesses on $\mathcal{P}'$. For example, even though the deposit method is no longer synchronized in $\mathcal{P}'$, the *lockset* of R^{t1}_{14} remains the same in both $\mathcal{P}$ and $\mathcal{P}'$. CONDIFF leverages this key result to determine whether a concurrent test belongs to $CT_{\Delta IS}$ (ISINΔIS, lines 25–29). To achieve this, CONDIFF adapts the CIA of RECONTEST as follows:

Given a concurrent test *ct*, CONDIFF executes[4] *ct* on an instrumented version of $\mathcal{P}$ and $\mathcal{P}'$ and collects two execution traces $\mathcal{E}^{\mathcal{P}}$ and $\mathcal{E}^{\mathcal{P}'}$, respectively (line 26 Figure 3). An execution trace $\mathcal{E}{=}\langle e_i \rangle$ is an ordered sequence of shared-memory accesses and synchronization events [61]. To avoid redundant analysis, CONDIFF before computing the CC it checks if the pair of traces $\mathcal{E}^{\mathcal{P}}$, $\mathcal{E}^{\mathcal{P}'}$ has been already witnessed in previous iterations. If yes, it discards *ct* since the resulting interleaving spaces would be identical to a previously generated test [37].

CONDIFF scans both execution traces individually to pre-compute two maps $\mathbb{CC}^{\mathcal{P}}$ and $\mathbb{CC}^{\mathcal{P}'}$ (lines 27-28). Each map

[4]Like most differential testing techniques we assume that the method signatures (name and parameters) have not changed across $\mathcal{P}$ and $\mathcal{P}'$. As such, *ct* can be compiled and executed on both versions without adaptation.

```
input  : two versions of a conc. class $\mathcal{P}$ and $\mathcal{P}'$, time-budget $\mathcal{B}$
         $l$ max prefix length, $k$ max # of parameters values
output : difference-revealing concurrent test $ct$
         $\mathbb{O}^{\mathcal{P}} \triangle \mathbb{O}^{\mathcal{P}'}$ behavioral differences of $ct$
1  function CONDIFF
2      CM ← GETCHANGEDMETHODS( $\mathcal{P}$, $\mathcal{P}'$)
3      CMP ← GETCHANGEDMETHODPAIRS( $\mathcal{P}$, $\mathcal{P}'$, CM)
4      while time budget $\mathcal{B}$ not expired do
           /*          Test Generation                  */
5          prefix = GENRANDOMPREFIX($l$)
6          if ISDIFFSEQ(prefix, $\mathcal{P}$, $\mathcal{P}'$) then
7              skip prefix and continue while loop at line 4
8          for each $\langle m_1, m_2 \rangle \in$ CMP do
9              for 1 to $k$ do
10                 suffix$_1$ ← GENRANDOMSUFFIX($m_1$)
11                 suffix$_2$ ← GENRANDOMSUFFIX($m_2$)
12                 if ISDIFFSEQ(prefix $\oplus$ suffix$_1$, $\mathcal{P}$, $\mathcal{P}'$) or
13                 ISDIFFSEQ(prefix $\oplus$ suffix$_2$, $\mathcal{P}$, $\mathcal{P}'$) then
14                     skip suffixes, continue for loop at line 9
15                 $ct$ ← ⟨prefix, suffix$_1$, suffix$_2$⟩      // #CT++ (§V)
                   /*          Filtering                  */
16                 if ISIN$\Delta$IS($ct$, $\mathcal{P}$, $\mathcal{P}'$) = false then
17                     skip $ct$, continue for loop at line 9
                   /*      Behavioral Checking            */
18                 $\langle \mathbb{O}^{\mathcal{P}}, \mathbb{O}^{\mathcal{P}'} \rangle \leftarrow \langle \varnothing, \varnothing \rangle$      // #BC++ (§V)
19                 for each $\sigma \in IS(ct, \mathcal{P})$ do
20                     add $O^{\mathcal{P}}(\sigma)$ to $\mathbb{O}^{\mathcal{P}}$
21                 for each $\sigma' \in IS(ct, \mathcal{P}')$ do
22                     add $O^{\mathcal{P}'}(\sigma')$ to $\mathbb{O}^{\mathcal{P}'}$
23                 if $\mathbb{O}^{\mathcal{P}} \neq \mathbb{O}^{\mathcal{P}'}$ then // difference found!
24                     return $ct$ and $\mathbb{O}^{\mathcal{P}} \triangle \mathbb{O}^{\mathcal{P}'}$

input  : newly generated concurrent test $ct$, $\mathcal{P}$ and $\mathcal{P}'$
output : true if $ct \in CT_{\Delta IS}$, false otherwise
25 function ISIN$\Delta$IS
26     $\langle \mathcal{E}^{\mathcal{P}}, \mathcal{E}^{\mathcal{P}'} \rangle$ ← COLLECTTRACE($ct$, $\mathcal{P}$, $\mathcal{P}'$)
27     $\mathbb{CC}^{\mathcal{P}}$ ← COMPUTECC($\mathcal{E}^{\mathcal{P}}$)
28     $\mathbb{CC}^{\mathcal{P}'}$ ← COMPUTECC($\mathcal{E}^{\mathcal{P}'}$)
29     return ISIMPACTED($\mathcal{E}^{\mathcal{P}}, \mathcal{E}^{\mathcal{P}'}, \mathbb{CC}^{\mathcal{P}}, \mathbb{CC}^{\mathcal{P}'}$) or
       ISIMPACTED($\mathcal{E}^{\mathcal{P}'}, \mathcal{E}^{\mathcal{P}}, \mathbb{CC}^{\mathcal{P}'}, \mathbb{CC}^{\mathcal{P}}$)

input  : execution traces $\mathcal{E}^{V1}$, $\mathcal{E}^{V2}$ and $\mathbb{CC}^{V1}$, $\mathbb{CC}^{V2}$
output : true if $\mathcal{E}^{V2}$ has an impacted access, false otherwise.
30 function ISIMPACTED
31     for each $e_x \in \mathcal{E}^{V2}$ do
32         $s^* \leftarrow s \in V_1 \mid s_x \sim s$
33         if $s^*$ = null or $\mathbb{CC}^{V1}(s^*) = \varnothing$ then
34             return true
35         else if [ ISDIFFCC($CC_x$, $\mathbb{CC}^{V1}(s^*)$) or
           ISNEWVALUE ($e_x$, $\mathcal{E}^{V1}$)] and ISCONFLICT($e_x$,
           $\mathcal{E}^{V2}$) then
36             return true
37     return false
```

Fig. 3. CONDIFF Algorithm.

contains all the unique concurrency contexts of the shared-memory accesses in the trace triggered by statement s, i.e., $\mathbb{CC}^{\mathcal{P}}(s) = \{CC_x \mid e_x \in \mathcal{E}^{\mathcal{P}}, s_x = s\}$, where CC_x is the context of e_x and s_x is the statement triggering e_x.

RECONTEST needs to compute all impacted accesses in $\mathcal{E}^{\mathcal{P}'}$ to identify all problematic interleavings [10]. In contrast, ISIMPACTED function returns true if the trace has at least one impacted access, and false otherwise. Since even a single impacted access makes $IS(ct, \mathcal{P}') \neq IS(ct, \mathcal{P})$, this approach

TABLE II
DESCRIPTION OF THE CORRECT AND BUGGY VERSIONS OF THE CONCURRENT CLASSES CONSIDERED IN OUR EXPERIMENTS

Class ID	Class Name	LOC	# Public Methods	# Method Pairs	Code Base	Buggy Version	Code Change that Fixes the Fault	Type of Behavioral Difference	Bug Report
C1	AbstractMultiMap$AsMap	1,647	26	351	Google Commons	1.0	fix "if" condition	wrong return value/`NullPointerException`	GUAVA-339
C2	BufferedInputStream	404	10	55	JDK	1.1	add synchronization	`NullPointerException`	JDK-4728096
C3	IntRange	423	26	351	Apache Commons	2.4	use a temp variable	corrupted object receiver state	LANG-481
C4	AppenderAttachableImpl	172	8	36	Log4J	1.2.17	add synchronization	`ArrayIndexOutOfBoundsException`	LOG4J-54325
C5	Logger	1,297	44	990	JDK	1.4.1	add synchronization	`NullPointerException`	JDK-4779253
C6	ObjectPool	63	4	10	java-design-patterns	1.17.0	add synchronization	wrong return value	ISSUE-621

reduces analysis costs when many impacted accesses exist.

Even if the impact set of $\mathcal{E}^{\mathcal{P}'}$ is empty, meaning ct shows no *new* interleavings in $\mathcal{P}'$, the concurrent test can still reveal behavioral differences. This occurs when *old* interleavings (i.e., behaviors) appear in $\mathcal{P}$ but not in $\mathcal{P}'$. Regression faults arise not only from *new faulty* behaviors in $\mathcal{P}'$, but also when *old correct* behaviors in $\mathcal{P}$ no longer occur in $\mathcal{P}'$ [4], [62]. Therefore, CONDIFF calls ISIMPACTED twice to check if $\mathcal{E}^{V2}$ (either $\mathcal{E}^{\mathcal{P}'}$ or $\mathcal{E}^{\mathcal{P}}$) contains an impacted access (see line 29).

For each shared-memory access event e_x in $\mathcal{E}^{V2}$, ISIMPACTED finds the statement s^* in V_1 that corresponds to s_x (using the $\sim$ relation [10]). For instance, in Figure 2, line 9 in $\mathcal{P}'$ corresponds to line 8 in $\mathcal{P}$. Access e_x is new if (i) s^* does not exist (i.e., s_x is new or modified in $\mathcal{P}'$) or (ii) $\mathbb{CC}^{V1}(s^*)$ is empty, meaning e_x arises from a new execution path. ISDIFFCC determines if e_x's concurrency context (CC_x) differs from any in $\mathbb{CC}^{V1}(s^*)$. Differences are clear when cardinalities vary, e.g., R^{t1}_{14} in ct_3 has a lockset cardinality of zero in $\mathcal{P}'$ but one in $\mathcal{P}$. For matching CCs with the same cardinality, CONDIFF uses change-resilient object abstractions from RECONTEST to align lock objects cross the two executions [10].

CONDIFF introduces two additional analyses to improve results. First, the ISNEWVALUE function checks if the value accessed by e_x in $\mathcal{E}^{V2}$ was never accessed by the corresponding statement s^* in $\mathcal{E}^{V1}$. Unlike RECONTEST, which assumes the *value-independent assumption* [63], CONDIFF considers an access impacted if it reads or writes a *new* value. This is achieved by instrumenting the values read or written by shared-memory accesses. Second, the ISCONFLICT function checks if an impacted access $e_x \in \mathcal{E}^{V2}$ by thread t_1 (or t_2) accesses a memory location also accessed by its concurrent thread t_2 (or t_1) in $\mathcal{E}^{V2}$. If so, ISIMPACTED returns TRUE; otherwise, FALSE. Intuitively, if the thread does not access the same memory location accessed by e_x any new interleaving involving e_x cannot affect the execution behavior [64].

V. EVALUATION

This section describes the experiments that we conducted to evaluate CONDIFF. We study three research questions.

- **RQ1 - Effectiveness.** Can CONDIFF detect non-deterministic behavioral differences within time budget $\mathcal{B}$?
- **RQ2 - Efficiency.** How *fast* is CONDIFF in detecting the behavioral differences?
- **RQ3 - Evaluation of the filtering phase ΔIS.** What is the contribution of the *filtering phase* based on Theorem 1 in reducing the search space?

A. Implementation

We created a prototype of CONDIFF for Java classes. Concurrent test generation builds on COVCON [38], while instrumentation for execution traces and shared-memory values uses ASM [65]. Static change analysis adapts CHANGEDISTILLING [66], and the filtering phase is based on RECONTEST [10]. CONDIFF systematically explores interleavings with the stateful model-checker JPF [39], and Java reflection collects observable outputs from outermost method calls.

B. Subjects

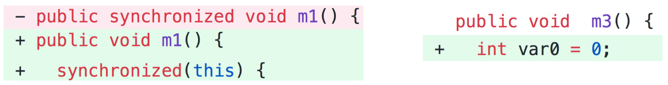
We conducted experiments on six known concurrency faults in open-source Java concurrent classes, with details in Table II. In all cases, the faults are in a single method, covering a range of behavioral differences: runtime exceptions, incorrect return values, and invalid object states. The limited number of classes is due to JPF compatibility issues. For each *faulty version*, we obtained the *correct version* that fixes the fault, as done in prior regression testing studies [10], [17], [59], given the lack of benchmarks for Java concurrent regression faults.

However, this approach may not generate scenarios that accurately represent regression faults [17]. Each correct version reflects only the minimized difference between the faulty and fixed versions, which may not represent the actual code changes that caused the fault. This minimized difference is advantageous for CONDIFF and differential testing since only the fault-inducing statements are modified. For instance, in the fixed version of $C5$, only one method is changed, limiting the usable method pairs to 44 out of 990. To address this and create more realistic scenarios, we developed various faulty versions with additional semantics-preserving modifications that do not alter the class's sequential or concurrent behavior. If the method to modify is synchronized, we enclosed the method body inside a synchronized block with `this` as the lock (Fig. 4 left). Otherwise, we added a local variable (Fig. 4 right).

For each of the six classes, we considered four possible scenarios (see Table III). The scenario C-B involves changes only to the *faulty method*. In contrast, C-$BR\%25$, C-$BR\%50$, and C-$BR\%100$ include modifications to 25%, 50%, and all public methods in the faulty version, respectively. For example, $C1$ has 26 public methods, and in C-$BR\%25$, we manually modified six randomly chosen methods using semantics-preserving changes, resulting in a total of seven changed methods. To obtain C-$BR\%50$, 12 methods, for C-$BR\%100$ 25 methods. In C-$BR\%100$, all methods are changed, but only one change causes a behavioral difference.

TABLE III
RQ1 AND RQ3 - EFFECTIVENESS RESULTS

			RQ1: CONDIFF										RQ3: BASELINE					
Subject	**# Changed**	**# Changed**	**SR**	**#CT**			**#BC**			**#CT − #BC**			**SR**	**#BC (#CT)**			**SR**	**#BC**
ID	**Methods**	**M. Pairs**	**%**	**avg**	**max**	**min**	**avg**	**max**	**min**	**avg**	**max**	**min**	**%**	**avg**	**max**	**min**	**+%**	**↘**
C1-B	1	26	100%	3	5	1	1	1	1	2	4	0	100%	3	5	1	-	2.5x
C1-BR25%	7	161	100%	4	8	1	1	1	1	3	7	0	100%	4	8	1	-	3.7x
C1-BR50%	13	260	100%	4	9	1	1	1	1	3	8	0	100%	4	9	1	-	3.9x
C1-BR100%	26	351	100%	8	23	1	1	1	1	7	22	0	100%	8	23	1	-	7.6x
C2-B	1	10	100%	7	12	1	3	6	1	4	8	0	100%	7	12	1	-	2.2x
C2-BR25%	3	27	100%	18	27	7	6	10	2	12	24	4	100%	18	27	7	-	3.0x
C2-BR50%	5	34	100%	25	52	7	8	15	2	17	37	4	100%	25	52	7	-	3.3x
C2-BR100%	10	55	100%	41	62	8	11	18	2	30	45	6	100%	41	62	8	-	3.8x
C3-B	1	26	100%	19	32	6	9	13	4	10	19	2	100%	19	32	6	-	2.0x
C3-BR25%	7	161	100%	115	243	3	8	16	2	107	227	1	100%	115	243	3	-	14.5x
C3-BR50%	13	260	100%	285	411	204	11	15	6	274	397	198	100%	285	411	204	-	25.7x
C3-BR100%	26	351	100%	223	430	8	6	12	1	217	418	7	100%	223	430	8	-	35.4x
C4-B	1	8	100%	2	3	1	1	2	1	1	1	0	100%	2	3	1	-	1.3x
C4-BR25%	2	15	100%	23	45	8	2	2	1	22	43	6	100%	22	45	8	-	14.3x
C4-BR50%	4	26	100%	90	230	29	2	2	1	88	229	27	100%	90	230	29	-	55.9x
C4-BR100%	8	36	100%	163	442	49	2	3	1	162	441	48	100%	154	442	49	-	96.5x
C5-B	1	44	100%	60	143	10	10	18	3	50	128	6	100%	54	101	10	-	5.5x
C5-BR25%	11	429	90%	274	638	47	10	21	4	264	617	43	60%	229	356	47	+30%	22.4x
C5-BR50%	22	665	70%	204	385	7	5	8	1	199	379	6	0%	178	307	29	+70%	34.9x
C5-BR100%	44	990	60%	904	1,360	382	8	12	6	896	1,353	376	0%	209	498	110	+60%	26.4x
C6-B	1	4	100%	4	4	4	3	3	3	1	1	1	100%	4	4	4	-	1.3x
C6-BR25%	1	4	100%	4	4	4	3	3	3	1	1	1	100%	4	4	4	-	1.3x
C6-BR50%	2	6	100%	5	7	3	2	3	1	3	4	2	100%	5	7	3	-	2.5x
C6-BR100%	4	10	100%	6	9	2	2	3	1	4	6	1	100%	6	9	2	-	2.7x
AVG			**96%**	**104**	**191**	**33**	**5**	**8**	**2**	**99**	**184**	**31**	**90%**	**71**	**138**	**23**		**15.5x**



```
- public synchronized void m1() {
+ public void m1() {
+   synchronized(this) {
```

```
  public void  m3() {
+   int var0 = 0;
```

Fig. 4. Semantics-preserving code modifications.

Table III shows, for each of the 24 subjects (scenarios), the number of changed methods and the number of method pairs that involve at least one changed method. These scenarios align with recent findings that faulty code is often more ”unnatural” (improbable) than correct code [67], [68], as most code changes in a revision do not introduce faults [67]. Additionally, these scenarios simulate faults arising from common activities, such as refactoring or performance improvements, where all changes should be semantics-preserving.

C. Setup

We ran CONDIFF on each of the 24 subjects with a time budget $\mathcal{B}$ of one hour per subject [37], [38], [64]. We used the upper bounds $l = 10$ and $k = 5$ (see Figure 3). We repeated the experiments ten times to cope with the randomness of the test generation process. To obtain replicable results, CONDIFF generated tests pseudo-deterministically given a random seed. To avoid biases in selecting the seeds, we used the numbers from 1 to 10. We evaluate CONDIFF with the following metrics:

- *Success Rate* (**SR**) whose value is true if CONDIFF generates and reports a difference-revealing concurrent test within the given time budget $\mathcal{B}$, false otherwise.
- *Number of Concurrent Tests generated* (**#CT**) and *Number of Behavioral Checking performed* (**#BC**) when the time budget expired or when the technique found the first difference-revealing concurrent test.
- *Difference Revealing Time* (**DRT**), which is given by the overall time in seconds to identify the first difference-revealing concurrent test. If SR $=$ false, DRT $= \mathcal{B}$.

D. Results

RQ1- Effectiveness. (Columns 4 to 13 in Table III). The success rate *SR* for 21 out of 24 subjects is 100%, meaning that CONDIFF is able to report their behavioral differences within $\mathcal{B}$ in all the ten runs. We manually validated that the results of all runs were correctly reported. The three subjects with *SR* $<$ 100% are *C5-BR%25*, *C5-BR%50* and *C5-BR%100*. These are the subjects with the highest number of changed method pairs (Column 3 Table III). Manifesting behavioral differences for *C5* requires a peculiar concurrent test that was not generated in some runs within the given time budget. The average number of concurrent tests *#CT* ranges from 2 to 904, (104 on average). As expected, for each subject, *#CT* increases when the number of changed methods increases. The average *#BC* ranges from 1 to 10, with an avg of 5 concurrent tests.

The difference between *#CT* and *#BC* quantifies the effectiveness of the filtering phase. CONDIFF prunes a large number of concurrent tests that execute changed methods but do not yield to different interleavings across $\mathcal{P}$ and $\mathcal{P}'$. Columns 8-10 show the number of filtered tests for each subject. In many cases, the concurrent behavioral checker analyses only the reported difference-revealing concurrent test (i.e., *#BC* $= 1$). In other cases *#BC* > 1, meaning that it analyzes concurrent tests that are not difference-revealing. This result is expected since the manifestation of different interleavings across $\mathcal{P}$ and $\mathcal{P}'$ is a necessary but not sufficient condition for a concurrent test to reveal behavioral differences (see Theorem 1).

RQ2 - Efficiency (Columns 2 to 10 in Table IV).The difference-revealing time *DRT* ranges from 6 to 2,260 seconds (245 seconds on average). This shows that CONDIFF rapidly exposes the behavioral differences in our subjects. Columns 5-9 in Table IV details the time spent on various parts of CONDIFF: test generation (*TIME GEN*), concurrent behavioral

TABLE IV
RQ2 AND RQ3 - EFFICIENCY RESULTS

Subject	RQ2: CONDIFF									RQ3: BASELINE							
	DRT			**TIME GEN**		**TIME BC**		**TIME ΔIS**		**DRT**			**TIME GEN**		**TIME BC**		**DRT**
ID	**avg**	**max**	**min**	**avg**	**%**	**avg**	**%**	**avg**	**%**	**avg**	**max**	**min**	**avg**	**%**	**avg**	**%**	↘
C1-B	6	7	5	<1	13%	5	73%	<1	<1%	10	14	6	1	11%	7	78%	1.5x
C1-BR25%	6	8	5	1	22%	4	63%	<1	1%	12	23	5	2	19%	9	73%	1.9x
C1-BR50%	6	7	5	<1	17%	4	67%	<1	1%	10	21	5	2	17%	8	75%	1.8x
C1-BR100%	6	7	5	2	26%	4	59%	<1	<1%	15	36	7	3	18%	12	76%	2.5x
C2-B	7	7	6	<1	8%	5	78%	<1	1%	20	31	6	15	77%	4	19%	2.9x
C2-BR25%	38	83	8	33	86%	4	10%	<1	1%	122	192	59	118	97%	3	2%	3.2x
C2-BR50%	58	145	7	51	88%	6	9%	<1	1%	164	381	64	147	90%	16	10%	2.8x
C2-BR100%	78	161	8	68	87%	8	10%	<1	1%	260	442	48	236	91%	22	9%	3.3x
C3-B	126	151	110	<1	1%	123	98%	<1	<1%	156	211	117	<1	1%	154	99%	1.2x
C3-BR25%	135	179	99	6	5%	124	91%	1	1%	488	899	108	4	1%	481	99%	3.6x
C3-BR50%	169	201	133	18	11%	137	81%	4	2%	1,046	1,513	691	13	1%	1,026	98%	6.2x
C3-BR100%	149	202	93	20	13%	117	79%	3	2%	832	1,601	113	17	2%	807	97%	5.6x
C4-B	7	13	3	2	31%	4	58%	<1	1%	8	13	3	2	29%	5	62%	1.1x
C4-BR25%	10	17	6	4	35%	5	49%	<1	3%	41	82	15	4	9%	36	88%	4.0x
C4-BR50%	17	54	5	8	45%	5	28%	1	7%	173	452	53	9	5%	160	93%	9.9x
C4-BR100%	31	104	5	17	56%	5	17%	2	7%	271	795	78	16	6%	248	92%	8.8x
C5-B	146	245	60	38	26%	102	70%	2	1%	534	1,049	153	27	5%	503	94%	3.7x
C5-BR25%	649	3,600	110	521	80%	100	15%	8	1%	2,720	3,600	727	246	9%	2,462	91%	4.2x
C5-BR50%	1,908	3,600	365	1,833	96%	56	3%	7	<1%	3,600	3,600	3,600	1,263	35%	2,341	65%	1.9x
C5-BR100%	2,260	3,600	658	2,042	90%	103	5%	48	2%	3,600	3,600	3,600	134	4%	3,487	97%	1.6x
C6-B	18	18	17	<1	3%	17	94%	<1	<1%	20	21	19	<1	2%	19	95%	1.1x
C6-BR25%	18	19	17	<1	3%	17	94%	<1	<1%	20	21	19	<1	2%	19	95%	1.1x
C6-BR50%	14	18	10	<1	3%	13	93%	<1	<1%	27	35	18	<1	2%	26	96%	1.9x
C6-BR100%	14	18	10	<1	3%	13	92%	<1	<1%	27	37	13	<1	2%	26	96%	1.9x
AVG	**245**	**519**	**73**	**195**	**35%**	**41**	**56%**	**3**	**1%**	**591**	**778**	**397**	**94**	**22%**	**495**	**75%**	**3.2x**

checking (*TIME BC*), and filtering (*TIME* Δ*IS*). While the time spent on initial static change analysis and sequential behavioral checking is included in *DRT*, it is not reported here due to its low cost (averaging less than 1 sec). Notably, *TIME* Δ*IS* averages only three seconds, accounting for less than 1% of *DRT* in determining if each of the *#CT* generated tests belongs to *CT*$_{\Delta IS}$, demonstrating its low computational cost.

RQ3 - Evaluation of the filtering phase ΔIS. We created a variant of CONDIFF, that we call BASELINE which skips the filtering phase (i.e., bypassing lines 16 and 17 in Figure 3). We ran BASELINE ten times with the same time budget, random seeds and upper bounds, and compared it with CONDIFF. Columns 14-19 in Table III and Columns 11-18 in Table IV show the results. BASELINE failed to expose the behavioral differences in any of the ten runs (*SR* = 0%) for the subjects $C5$-$BR\%50$ and $C5$-$BR\%100$ (see Column 14 in Table III). Moreover, *SR* of $C5$-$BR\%25$ is 30% less. Column "*SR +%*" in Table III shows the increase in success rate of CONDIFF over BASELINE. The average *#BC* of BASELINE ranges from 2 to 209, with an average of 71 concurrent tests. CONDIFF performed 15.5× less behavioral checking than BASELINE, on average (last Column in Table III)[5]. For each run, CONDIFF and BASELINE generate the same concurrent tests in the same order, but BASELINE skips the filtering phase, i.e., *#CT* = *#BC*. When the number of changed methods in each class increases, *#BC* of BASELINE rapidly increases while *#BC* of CONDIFF remains almost unchanged. This shows that CONDIFF can correctly identify overlapping interleaving spaces across program versions under semantically equivalent code changes. The average *DRT* of BASELINE ranges from 8 to 3,600 seconds (591 seconds on average). CONDIFF detects behavioral differences on average 3.2× faster than BASELINE (last Column in Table IV).

CONDIFF is always faster than BASELINE in detecting behavioral differences because (i) CONDIFF performs the expensive concurrent behavioral checking in fewer tests than BASELINE; and (ii) *TIME* Δ*IS* has a negligible cost. Moreover, for these two reasons, CONDIFF allocates more time to the test generator component (see the *TIME GEN* Columns in Table IV), thus it explores a larger number of concurrent tests given the same time. This is the reason why for subjects $C5$-$BR\%50$ and $C5$-$BR\%100$, CONDIFF can detect the behavioral difference in some runs while BASELINE cannot. In fact, for these subjects the avg *#CT* of CONDIFF (Column 4 in Table III) is higher than the avg *#CT* of BASELINE (Column 15 in Table III).

Moreover, these results indicate that Theorem 1 is empirically sound, as none of the filtered tests was difference-revealing. This can be demonstrated by the result that when both CONDIFF and BASELINE have *SR* =100%, their *#CT* is identical[6].

VI. THREATS TO VALIDITY

A threat to external validity is that we evaluated CONDIFF using only six Java open-source projects, which may not generalize to all programming languages and characteristics. The limited number of subjects is due to compatibility issues with JPF, which we experienced while testing other Java projects. We plan to replace JPF with a different exhaustive interleaving explorer and evaluate CONDIFF on a larger set of subjects. Additionally, although it is standard practice [10], [17], [59], a potential threat is the lack of real regression faults. We address this with semantics-preserving code changes.

[5]The cell background in the last columns of the two tables indicates the degrees of reduction with respect to the BASELINE: LOW (>1.0x and <2.0x), MEDIUM ($\geq$ 2.0x and $<$ 3.0x), or HIGH ($\geq$ 3.0x).

[6]There are three exceptions $C4$-$BR\%25$, $C4$-$BR\%100$ and $C5$-B. A manual investigation revealed that discrepancy was due to JPF crashes.

VII. Related Work

To the best of our knowledge, ConDiff is the first technique that generates concurrent tests and oracles to expose behavioral differences between two versions of a concurrent class. We discuss related work in the realms of regression testing, differential testing, and concurrency testing.

Regression testing for concurrent programs is not a well-studied problem yet [1]. Pioneer studies focus on reducing the cost of re-executing existing concurrent tests [10], [11], [59] (e.g., ReConTest). They can detect concurrent faults only if the given tests contain fault-revealing interleavings that can be caught by predefined oracles. ConDiff addresses this limitation by generating new tests and oracles to expose behavioral differences between two versions.

There are also studies to reduce the incremental cost of *software verification* for modified sequential or concurrent software [59], [69]–[73]. All of these studies aim at detecting regression faults faster with a given set of tests and oracles. In contrast, ConDiff aims at generating new tests and oracles to detect regression faults that cannot be revealed by the existing ones. Recently, researchers have presented the dynamic symbolic techniques ConTesa [74], Conc-iSE [75] and Taco [76] to generate test inputs that exposes new interleavings for regression testing concurrent programs. These techniques differ substantially from ConDiff. First, they do not tackle the differential testing problem as they do not check if the program behavior of the new interleavings is equivalent to the behavior of the old interleavings. Second, they generate test inputs but no concurrent tests (multi-threated method call sequences, see Figure 1). In principle, these techniques could be used to generate additional test inputs for the concurrent tests generated by ConDiff. Third, they detect faults by relying on the assertions encoded in the program or software crashes, while ConDiff generates regression oracles.

Differential testing for sequential programs either generate difference-revealing test inputs relying on various symbolic analyzes [12]–[14], [77], [78] or generate difference-revealing sequential tests (Definition 1). The main techniques of the latter type are: Diffut [16], BeRT [29], [32], [79], Diffgen [30], and EvoSuiteR [17], [53]. Unlike ConDiff, all of these techniques do not generate concurrent tests and they do not address the issues of non-determinism and interleaving space exploration. There are only two differential testing techniques for concurrent classes, proposed by Pradel et al [80], [81]. They differ from ConDiff, as they either target regression performance issues [80] or incorrect substitutability faults [81].

SpeedGun [80] compares the execution time of automatically generated concurrent tests across program revisions to expose *performance* issues introduced by code modifications. Conversely, ConDiff targets correctness problem, and thus it complements SpeedGun. Nevertheless, the tools could work in synergy. After modifying a concurrent class, developers could use ConDiff and SpeedGun to detect correctness and performance issues, respectively. Note that SpeedGun generates long-running performance tests that have a high degree of concurrency [80]. Using these tests to identify behavioral differences would be expensive as the number of interleavings grows exponentially with respect to execution length and number of concurrent threads [3], [40].

The other differential testing technique generates both sequential and concurrent tests to identify behavioral differences between a subclass and its superclass [81]. While its behavioral checking resembles that of ConDiff, its test generation focuses on *a single* program version. Adapting this technique would likely produce results similar to Baseline, the version of ConDiff without the filtering phase (see RQ3).

Concurrency testing. ConDiff builds on top of recent concurrent test generators [36]–[38], [49]–[51], [82], [83]. Unlike ConDiff, they analyze a single program version and do not consider program changes while generating concurrent tests, since their goal is not regression testing. Thus, many of the generated tests are likely to explore interleavings involving unmodified code and program behaviors, and thus not effective in exposing regression faults. Moreover, these techniques can detect only those concurrency faults that manifest visible oracle violations (i.e., exceptions or deadlocks) [36], [38], [49] or that trigger interleavings matching a given set of problematic access patterns [37], [50], [82], [84]. In contrast, ConDiff can also detect regression faults that manifest wrong return values or incorrect program states (see C1, C3 and C6 in Table II).

VIII. Conclusion and Future Work

In this paper we presented ConDiff, a differential testing technique to expose non-deterministic behavioral differences across two versions of a concurrent class. We released the benchmarks and the results to ease future work in this area [42].

ConDiff makes only the first steps towards differential testing for concurrent classes. There are several opportunities for future work. We discuss the three most promising ones.

Program level differential testing. Currently, ConDiff focuses on class-level differential testing, handling modified concurrent classes individually. This may miss behavioral differences when an unchanged method invokes a changed method in another class. Future work includes updating the change analysis and test generation to address such cases.

Incremental analysis. A common usage scenario of ConDiff is to analyze the same version of a class multiple times. For example, one time with its preceding version and another time with its subsequent version. Currently, ConDiff does not save analysis results for future use. Enabling incremental analysis [70], [73] could reduce the high cost of behavioral checking. How to effectively re-use the analysis results of ConDiff is an interesting research problem.

Test augmentation. ConDiff is intended for cases where existing regression tests fail to expose faults in the modified version. However, it could leverage these tests [4], [14] to improve effectiveness by targeting test generation only on those classes or methods where current tests are inadequate.

Acknowledgments

The research is partially supported by the HKSAR RGC/GRF grant 16205821.

References

[1] R. Gu, G. Jin, L. Song, L. Zhu, and S. Lu, "What Change History Tells Us About Thread Synchronization," in *FSE*, 2015, pp. 426–438.

[2] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and Reproducing Heisenbugs in Concurrent Programs," in *OSDI*, 2008, pp. 267–280.

[3] S. Lu, S. Park, and Y. Zhou, "Finding Atomicity-Violation Bugs through Unserializable Interleaving Testing," *IEEE TSE*, vol. 38, no. 4, pp. 844–860, 2012.

[4] S. Yoo and M. Harman, "Regression Testing Minimization, Selection and Prioritization: A Survey," *STVR*, vol. 22, no. 2, pp. 67–120, 2012.

[5] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi, "Regression Test Selection for Java Software," in *OOPSLA*, 2001.

[6] M. Gligoric, L. Eloussi, and D. Marinov, "Practical Regression Test Selection with Dynamic File Dependencies," in *ISSTA*, 2015, pp. 211–222.

[7] M. Gligoric, S. Negara, O. Legunsen, and D. Marinov, "An Empirical Evaluation and Comparison of Manual and Automated Test Selection," in *ASE*, 2014, pp. 361–372.

[8] T. Xie and D. Notkin, "Checking Inside the Black Box: Regression Testing By Comparing Value Spectra," *IEEE TSE*, vol. 31, no. 10, pp. 869–883, 2005.

[9] S. Park, S. Lu, and Y. Zhou, "CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places," in *ASPLOS*, 2009, pp. 25–36.

[10] V. Terragni, S.-C. Cheung, and C. Zhang, "Recontest: Effective regression testing of concurrent programs," in *ICSE*, 2015.

[11] T. Yu, W. Srisa-an, and G. Rothermel, "SimRT: An Automated Framework to Support Regression Testing for Data Races," in *ICSE*, 2014, pp. 48–59.

[12] K. Taneja, T. Xie, N. Tillmann, and J. de Halleux, "eXpress: Guided Path Exploration for Efficient Regression Test Generation," in *ISSTA 2011*, 2011.

[13] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold, "Test-suite Augmentation for Evolving Software," in *ASE*, 2008, pp. 218–227.

[14] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen, "Directed Test Suite Augmentation: Techniques and Tradeoffs," in *FSE*, 2010, pp. 257–266.

[15] S. Shamshiri, "Automated Unit Test Generation for Evolving Software," in *FSE*, 2015, pp. 1038–1041.

[16] T. Xie, K. Taneja, S. Kale, and D. Marinov, "Towards a Framework for Differential Unit Testing of Object-oriented Programs," in *AST*, 2007, p. 5.

[17] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges ," in *ASE*, 2015, pp. 201–211.

[18] W. M. McKeeman, "Differential Testing for Software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[19] M. M. Almasi, H. Hemmati, G. Fraser, P. McMinn, and J. Benefelds, "Search-based Detection of Deviation Failures in the Migration of Legacy Spreadsheet Applications," in *ISSTA*, 2018, pp. 266–275.

[20] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and Understanding Bugs in C Compilers," in *PLDI*, 2011, pp. 283–294.

[21] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, "Coverage-directed Differential Testing of JVM Implementations," in *PLDI*, 2016, pp. 85–99.

[22] G. Argyros, I. Stais, S. Jana, A. D. Keromytis, and A. Kiayias, "SFADiff: Automated Evasion Attacks and Fingerprinting Using Black-Box Differential Automata Learning," in *SIGSAC*, 2016, pp. 1690–1701.

[23] A. Groce, G. Holzmann, and R. Joshi, "Randomized Differential Testing As a Prelude to Formal Verification," in *ICSE*, 2007, pp. 621–631.

[24] B. Daniel, D. Dig, K. Garcia, and D. Marinov, "Automated Testing of Refactoring Engines," in *FSE*, 2007, pp. 185–194.

[25] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana, "Nezha: Efficient Domain-Independent Differential Testing," in *SP*, 2017, pp. 615–632.

[26] T. Kapus and C. Cadar, "Automatic Testing of Symbolic Execution Engines via Program Generation and Differential Testing," in *ASE*, 2017, pp. 590–600.

[27] D. Lehmann and M. Pradel, "Feedback-Directed Differential Testing of Interactive Debuggers," in *ESEC/FSE*, 2018, pp. 610–620.

[28] T. Zhang and M. Kim, "Automated Transplantation and Differential Testing for Clones," in *ICSE*, 2017, pp. 665–676.

[29] W. Jin, A. Orso, and T. Xie., "Automated Behavioral Regression Testing," in *ICST*, 2010, pp. 137–146.

[30] K. Taneja and T. Xie, "DiffGen: Automated Regression Unit-Test Generation," in *ASE*, 2008, pp. 407–410.

[31] R. B. Evans and A. Savoia, "Differential Testing: A New Approach to Change Detection," in *FSE*, 2007, pp. 549–552.

[32] A. Orso and T. Xie, "BERT: BEhavioral Regression Testing," in *WODA*, 2008, pp. 36–42.

[33] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed Random Test Generation," in *ICSE*, 2007, pp. 75–84.

[34] A. Pnueli, "The Temporal Semantics of Concurrent Programs," *Theoretical computer science*, vol. 13, no. 1, pp. 45–60, 1981.

[35] V. Terragni, P. Salza, and F. Ferrucci, "A Container-Based Infrastructure for Fuzzy-Driven Root Causing of Flaky Tests," in *ICSE-NIER*, 2020, pp. 69–72.

[36] M. Pradel and T. R. Gross, "Fully Automatic and Precise Detection of Thread Safety Violations," in *PLDI*, 2012, pp. 521–530.

[37] V. Terragni and S.-C. Cheung, "Coverage Driven Test Code Generation for Concurrent Classes," in *ICSE*, 2016, pp. 1121–1132.

[38] A. Choudhary, S. Lu, and M. Pradel, "Efficient Detection of Thread Safety Violations via Coverage-Guided Generation of Concurrent Tests," in *ICSE*, 2017, pp. 266–277.

[39] K. Havelund and T. Pressburger, " Model checking JAVA programs using JAVA PathFinder," *STTT*, vol. 2, no. 4, pp. 366–381, 2000.

[40] S. Lu, W. Jiang, and Y. Zhou, "A Study of Interleaving Coverage Criteria," in *FSE*, 2007, pp. 533–536.

[41] V. Terragni and M. Pezzè, "Effectiveness and Challenges in Generating Concurrent Tests for Thread-Safe Classes," in *ASE*, 2018, pp. 64–75.

[42] V. Terragni and S.-C. Cheung. (2024) ConDiff- experimental data and subjects https://doi.org/10.5281/zenodo.14722292.

[43] T. Xie, D. Marinov, and D. Notkin, "Rostra: A Framework for Detecting Redundant Object-Oriented Unit Tests," in *ASE*, 2004, pp. 196–205.

[44] G. Fraser and A. Arcuri, "Evosuite: Automatic Test Suite Generation for Object-oriented Software," in *FSE*, 2011, pp. 416–419.

[45] G. Jahangirova and V. Terragni, "Sbft tool competition 2023 - java test case generation track," 2023, pp. 61–64.

[46] H. Rogers and H. Rogers, *Theory of Recursive Functions and Effective Computability*. McGraw-Hill New York, 1967, vol. 126.

[47] A. Carzaniga, A. Mattavelli, and M. Pezzè, "Measuring Software Redundancy," in *ICSE*, 2015, pp. 156–166.

[48] T. Peierls, B. Goetz, J. Bloch, J. Bowbeer, D. Lea, and D. Holmes, *Java Concurrency in Practice*. Pearson Education, 2006.

[49] A. Nistor, Q. Luo, M. Pradel, T. R. Gross, and D. Marinov, "Ballerina: Automatic Generation and Clustering of Efficient Random Unit Tests for Multithreaded Code," in *ICSE*, 2012, pp. 727–737.

[50] S. Steenbuck and G. Fraser, "Generating Unit Tests for Concurrent Classes," in *ICST*, 2013, pp. 144–153.

[51] V. Terragni and M. Pezzè, "Statically driven generation of concurrent tests for thread-safe classes," *STVR*, vol. 31, no. 4, p. e1774, 2021.

[52] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An Empirical Analysis of Flaky Tests," in *FSE*, 2014, pp. 643–653.

[53] S. Shamshiri, G. Fraser, P. McMinn, and A. Orso, "Search-based Propagation of Regression Faults in Automated Regression Testing," in *ICSTW*, 2013, pp. 396–399.

[54] Y. Jiang, T. Gu, C. Xu, X. Ma, and J. Lu, "CARE: Cache Guided Deterministic Replay for Concurrent Java Programs," in *ICSE*, 2014, pp. 457–467.

[55] J. Huang, P. Liu, and C. Zhang, "LEAP: Lightweight Deterministic Multi-processor Replay of Concurrent Java Programs," in *FSE*, 2010, pp. 207–216.

[56] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics," in *ASPLOS*, 2008, pp. 329–339.

[57] D. Deng, W. Zhang, and S. Lu, "Efficient Concurrency-bug Detection Across Inputs," in *OOPSLA*, 2013, pp. 785–802.

[58] R. S. Arnold, *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[59] V. Jagannath, Q. Luo, and D. Marinov, "Change-aware Preemption Prioritization," in *ISSTA*, 2011, pp. 133–143.

[60] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A Dynamic Data Race Detector for Multithreaded Programs," *ACM TOCS*, vol. 15, no. 4, pp. 391–411, 1997.

[61] Z. Lai, S. C. Cheung, and W. K. Chan, "Detecting Atomic-set Serializability Violations in Multithreaded Programs Through Active Randomized Testing," in *ICSE*, 2010, pp. 235–244.

[62] G. Rothermel and M. J. Harrold, "Analyzing Regression Test Selection Techniques," *IEEE TSE*, vol. 22, no. 8, pp. 529–551, 1996.

[63] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam, "Maple: A Coverage-driven Testing Tool for Multithreaded Programs," in *OOPSLA*, 2012, pp. 485–502.

[64] F. A. Bianchi, M. Pezzè, and V. Terragni, "Reproducing Concurrency Failures from Crash Stacks," in *FSE*, 2017, pp. 705–716.

[65] E. B. et al. (2018) ASM Java Bytecode Manipulation Framework - http://asm.objectweb.org/.

[66] B. Fluri, M. Wursch, M. PInzger, and H. C. Gall, "Change Distilling: Tree Differencing for Fine-grained Source Code Change Extraction," *IEEE TSE*, vol. 33, no. 11, pp. 725–743, 2007.

[67] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the "Naturalness" of Buggy Code," in *ICSE*, 2016, pp. 428–439.

[68] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the Naturalness of Software," in *ICSE*, 2012, pp. 837–847.

[69] S. Lauterburg, A. Sobeih, D. Marinov, and M. Viswanathan, "Incremental State-Space Exploration for Programs with Dynamically Allocated Data," in *ICSE*, 2008, pp. 291–300.

[70] G. Yang, M. B. Dwyer, and G. Rothermel, "Regression Model Checking," in *ICSM*, 2009, pp. 115–124.

[71] D. Beyer, S. Löwe, E. Novikov, A. Stahlbauer, and P. Wendler, "Precision Reuse for Efficient Regression Verification," in *FSE*, 2013, pp. 389–399.

[72] J. Backes, S. Person, N. Rungta, and O. Tkachuk, "Regression Verification Using Impact Summaries," in *Model Checking Software*, 2013, pp. 99–116.

[73] S. Chaki, A. Gurfinkel, and O. Strichman, "Regression Verification for Multi-threaded Programs," in *VMCAI*, 2012, pp. 119–135.

[74] T. Yu, Z. Huang, and C. Wang, "ConTesa: Directed Test Suite Augmentation for Concurrent Software," *IEEE Transactions on Software Engineering*, 2018.

[75] S. Guo, M. Kusano, and C. Wang, "Conc-iSE: Incremental Symbolic Execution of Concurrent Software," in *ASE*, 2016, pp. 531–542.

[76] T. Yu, "TACO: Test Suite Augmentation for Concurrent Programs," in *FSE*, 2015, pp. 918–921.

[77] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu, "Differential Symbolic Execution," in *FSE*, 2008, pp. 226–237.

[78] H. Palikareva, T. Kuchta, and C. Cadar, "Shadow of a Doubt: Testing for Divergences Between Software Versions," in *ICSE*, 2016, pp. 1181–1192.

[79] W. Jin, A. Orso, and T. Xie., "BERT: A Tool for Behavioral Regression Testing," in *FSE*, 2010, pp. 361–362.

[80] M. Pradel, M. Huggler, and T. R. Gross, "Performance Regression Testing of Concurrent Classes," in *ISSTA*, 2014, pp. 13–25.

[81] M. Pradel and T. R. Gross, "Automatic Testing of Sequential and Concurrent Substitutability," in *ICSE*, 2013, pp. 282–291.

[82] M. Samak, M. K. Ramanathan, and S. Jagannathan, "Synthesizing Racy Tests," in *PLDI*, 2015, pp. 175–185.

[83] V. Terragni, M. Pezzè, and F. A. Bianchi, "Coverage-driven test generation for thread-safe classes via parallel and conflict dependencies," in *ICST*, 2019, pp. 264–275.

[84] T. Yu, W. Srisa-an, and G. Rothermel, "An Empirical Comparison of the Fault-detection Capabilities of Internal Oracles," in *ISSRE*, 2013, pp. 11–20.