Coverage-Driven Test Code Generation for Concurrent Classes

Valerio Terragni and Shing-Chi Cheung
Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
Hong Kong
{vterragni, scc}@cse.ust.hk

ABSTRACT

Previous techniques on concurrency testing have mainly focused on exploring the interleaving space of manually written test code to expose faulty interleavings of shared memory accesses. These techniques assume the availability of failure-inducing tests. In this paper, we present AutoConTest, a coverage-driven approach to generate effective concurrent test code that achieve high interleaving coverage. AutoConTest consists of three components. First, it computes the coverage requirements dynamically and iteratively during sequential test code generation, using a coverage metric that captures the execution context of shared memory accesses. Second, it smartly selects these sequential codes based on the computed result and assembles them for concurrent tests, achieving increased context-sensitive interleaving coverage. Third, it explores the newly covered interleavings. We have implemented AutoConTest as an automated tool and evaluated it using 6 real-world concurrent Java subjects. The results show that AutoConTest is able to generate effective concurrent tests that achieve high interleaving coverage and expose concurrency faults quickly. AutoConTest took less than 65 seconds (including program analysis, test generation and execution) to expose the faults in the program subjects.

CCS Concepts

•Software and its engineering → Software testing and debugging; Synchronization; Dynamic analysis;

Keywords

Automated test generation; Interleaving coverage criteria

1. INTRODUCTION

Rapid advances in multi-core chip technology have lead to pervasive adoption of concurrency programming, where software is jointly executed by multiple threads in a shared memory space [39]. Due to the inherent complexity of thread synchronization, concurrent programs are error-prone.

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in:

ICSE '16, May 14-22, 2016, Austin, TX, USA

© 2016 ACM. ISBN 978-1-4503-3900-1/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2884781.2884876

While writing failure-inducing tests is cumbersome and labour-intensive, their availability is essential to expose software faults. Therefore, automated generation of test code for sequential programs is an active research topic [14, 21, 22, 40, 59, 60]. Adapting these techniques for concurrency testing [38, 43, 56] presents both opportunities and challenges.

A major obstacle to the adoption of automated test code generation for sequential programs is the automatic derivation of effective test oracles that can accurately differentiate between failing and successful tests [13, 19, 42]. Fortunately, this obstacle is much alleviated for concurrency testing. Recent characteristic studies on real faults show that 56-70% of the examined concurrency faults lead to visible oracle violations, such as crashes or hangs [29, 33, 57]. Besides, the use of effective *concurrency correctness criteria* (e.g., serializability [61]) can help detect those concurrency faults that do not manifest visible oracle violations [68].

Unlike their sequential counterpart, the challenges of concurrency testing lie in the non-determinism of thread scheduling. Even for a test that can trigger a concurrency fault, we often need to execute the test many times in order to expose a faulty (i.e., oracle-violating) thread interleaving. Although there are techniques proposed to facilitate the interleaving exploration [17, 37, 41, 64], we can only afford exploring the interleaving spaces of a small number of tests in practice due to the enormity of interleaving spaces. This imposes a critical constraint of using random test generation techniques [38, 43] because effective concurrency fault detection typically requires a lot of randomly generated tests [38, 43]. For instance, an existing random technique took on average 8.2 hours to generate and run millions of tests before exposing a concurrency fault [7], where 99.5% of the time was spent on interleaving space exploration [43].

Existing works on concurrent testing [25, 41, 64, 67] mostly study how to guide the interleaving space exploration of a given concurrent test with respect to an interleaving coverage criterion (e.g., [32, 58, 66]). However, the generation of concurrent test code for an interleaving coverage criterion has rarely been studied [56]. Indeed, it is an important problem because test effectiveness can be much increased if we are able to generate concurrent test code that effectively triggers the interleavings of shared memory accesses relevant to a given interleaving coverage criterion. It saves redundant effort in exploring a large amount of interleavings that are irrelevant to concurrency fault detection or ineffective for enhancing interleaving coverage.

Automating test generation for interleaving coverage imposes two major challenges. First, it needs to estimate the

coverage requirements (i.e., interleavings) with respect to all possible concurrent tests. However, calculating the executable domain of interleaving coverage criteria precisely requires context-sensitive and synchronization-sensitive analysis that is machine undecidable for concurrent programs [44]. Second, it needs to estimate the set of interleavings that can be covered by a generated concurrent test in order to avoid generating tests that explore interleavings redundantly. However, the cost of precise estimation amounts to that of actually exploring the test's interleaving space due to the need for thread-sensitive analysis, which is intractable [54]. Previous work addressed the challenges by approximating the interleaving coverage using a context-insensitive analysis [56]. However, the approximation yields to both infeasible and missing requirements.

In this paper, we present a technique called *AUTOmated CONcurrency TESTing* (AutoConTest) that automatically generates and runs a suite of concurrent tests for a given class under test. The concurrent tests are obtained by first generating single-threaded sequences of method calls and then assembling them into concurrent (i.e., multi-threaded) tests. Our intuition is that context-sensitive information can be computed efficiently and precisely during dynamic sequential test generation. This observation relieves us from computing the entire coverage requirements before test generation. Instead, we can generate concurrent tests iteratively so that each test increases context-sensitive interleaving coverage based on the coverage data that are collected during the generation of the single-threaded sequences of method calls. In summary, this paper makes the following four contributions:

- We propose a coverage metric (Section 4.1) that captures context-sensitive information using single-threaded execution without requiring thread-sensitive analysis.
- We present a greedy algorithm (Section 4.2) to iteratively explore the search space of possible method call sequences and identify the optimal sequence that achieves the highest coverage with respect to our coverage metric in each iteration.
- We present a thread scheduler algorithm (Section 4.4) to increase the context-sensitive interleaving coverage.
- We implemented a prototype tool of our approach and evaluated it on 6 real-world Java subjects (Section 5). The evaluation results show that AutoConTest detected all concurrency faults with an average time of 38 seconds.

2. PROBLEM FORMULATION

This section formulates the problem of generating concurrent tests guided by an interleaving coverage criterion. It also introduces the background and terminology of this work.

Preliminaries. Consider a concurrent program $\mathcal{P}$, its multi-threaded execution under a test (input) t, denoted by $\mathcal{P}(t)$, is modelled as a sequence of shared memory accesses. Each execution of $\mathcal{P}(t)$ follows some non-deterministic thread schedule that determines the order of memory accesses. In each execution of $\mathcal{P}(t)$, the order of shared memory accesses within each thread is fixed while their global order across multiple threads can vary. Let Σ denote the set of all shared memory accesses triggered by $\mathcal{P}(t)$. An **interleaving** of $\mathcal{P}(t)$ is a total order relation on a set in Σ [32] (i.e., an ordered sequence of shared memory accesses). Whether $\mathcal{P}(t)$ can exhibit a concurrency failure wholly depends on the possible interleavings it prescribes. The concurrency testing of $\mathcal{P}(t)$ requires exploring the possible interleavings that jointly form the *interleaving space* of $\mathcal{P}(t)$.

2.1 Object-Oriented Concurrent Test

In this work, $\mathcal{P}$ is an object-oriented program composed of a set of classes, each defining a set of methods and fields that can be, respectively, executed and accessed concurrently by multiple threads. The *test inputs* are unit-level concurrent tests that validate the correctness of a given class-under-test (CUT) by concurrently invoking a shared CUT object's public methods using multiple threads [38, 43, 56].

A **concurrent test** $\mathbf{t} = \alpha \bullet (\beta \mid\mid \gamma)$ consists of three call sequences, namely α, β and γ. Each of them is a sequence $\delta = \langle c_1, ..., c_n \rangle$ of method calls to be executed by one thread. Each *method call* c_i consists of a method signature and input parameters, which can be primitive values (e.g., integers or booleans) or object references. We treat the object receiver of an instance method call as the call's first parameter.

The sequences β and γ are to be executed concurrently by two different threads. Examples of β and γ sequences can be found in Figure 2. Like existing works [38, 43, 56], we adopt a minimum configuration of two concurrent threads per test. This is because previous studies show that 96% of the concurrency bugs examined are guaranteed to manifest if a certain partial order between two threads is enforced [33].

To allow $\mathcal{P}(t)$ trigger shared memory accesses, we confine the method call parameters of type CUT (including the object receiver) in β and γ to a single *Shared Object Under Test* (SOUT) of type CUT so that the method calls made by the two sequences likely access SOUT's fields and trigger shared memory accesses. The role of α is to initialize SOUT before executing β and γ concurrently. Note that the call sequences can create and mutate other objects, as for example, those referenced by a method parameter of type different from CUT. These objects will not be shared across threads.

2.2 Coverage-Driven Test Code Generation

Various **interleaving coverage criteria** have been proposed to help select representative interleavings from the interleaving space induced by a test (e.g., [32, 58, 66]). An *interleaving coverage criterion* prescribes a set of properties that an interleaving has to satisfy to be considered a test coverage requirement. The properties are mostly derived based on specific fault models or bug characteristics [34]. Examples of interleaving-based requirements include those interleavings that constitute a data race [52], cover a specific location pair [25, 58] or match a problematic access pattern, which violates a certain form of serializability [34, 61].

Our approach deals with the problem of **coverage-driven test code generation**, that is to iteratively generate a sequence of concurrent tests $\langle t_1, t_2, \ldots, t_N \rangle$ such that the interleaving space of each $\mathcal{P}(t_i)$ contains those interleavings that 1) match predefined coverage requirements and 2) cannot be induced by any of concurrent tests generated before t_i.

We next formally present the problem definition. Let $\mathcal{TSP}$ denote the union of the interleavings spaces of all possible concurrent tests that can be performed on $\mathcal{P}$. Let $\mathcal{RQ}$ denote the subset of $\mathcal{TSP}$ containing only those interleavings that are requirements of a given interleaving coverage criterion. Let $cov(t)$ denote the set of requirements covered by a concurrent test t, i.e., $cov(t) = \mathcal{RQ} \cap \{\textit{the interleaving space of } \mathcal{P}(t)\}$. For each test t, we denote $cost(t) (= cost_{gen}(t) + cost_{exp}(t))$ as the cumulative cost in time to generate t and explore the interleaving space of $\mathcal{P}(t)$ to select and test the coverage requirements (i.e., interleavings).

```
public class CUT{
 int x= 0, y= 0, z =0;
 synchronized void m1(int x){
   this.x = x;                    a1: W(x)
 }
synchronized void m2(int y){
   this.y = y;                    a2: W(y)
 }
 void m3(int v1){
   if(z ==0){                     a3: R(z)
     synchronized (this){         l1: lock
       m4(v1); }                  l2: unlock
   } else
       m4(v1*-1); //BUG1 }
private void m4(int v1){
     x = x + v1;                  a4: R(x)
 }                                a5: W(x)
void m5(){
   for(int i = 0; i<= y; i++)     a6: R(y)
     z = i; //BUG2                a7: W(z)
 }
void m6(B b){
    CUT o = b.m();
    o.m1(-1);
}
synchronized void m7(){
    y = y + 1;                    a8: R(y)
 } }                              a9: W(y)
```

Figure 1: Source code of the motivating example

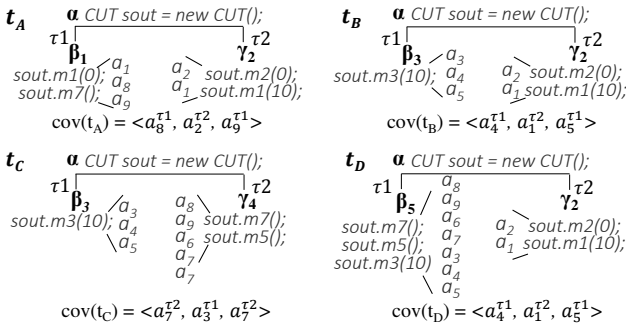

Figure 2: Examples of concurrent tests

Problem Definition: *Given an interleaving coverage criterion, CUT and cost budget $\mathcal{B}$, generate a concurrent test suite $\mathcal{T} = \langle t_1, t_2, ..., t_N \rangle$ with the following objective:*

$$\textit{maximize} \quad \Big|\bigcup_{i=1}^{N} cov(t_i)\Big| \quad \textit{subject to:}$$

$$\sum_{i=1}^{N} cost(t_i) \leq \mathcal{B}, \quad \{cov(t_i) \backslash \bigcup_{j=1}^{i-1} cov(t_j)\} \neq \varnothing^1 \quad \forall i \in [1 \ldots N]$$

Our research problem is to generate tests that collectively achieve the highest coverage within a given time budget.

3. MOTIVATING EXAMPLE

In this section we explain two research challenges of the problem and illustrate how they can be addressed with a running example (Figure 1).

Challenge 1: *Deriving the set of coverage requirements $\mathcal{RQ}$ statically (prior to testing) is generally infeasible* [44]. While interleaving coverage criteria are generally computed with respect to a given test input [32], coverage-guided test input generation needs to compute the coverage requirements with respect to all possible test inputs. However, calculating the executable domain of interleaving coverage criteria precisely requires context-sensitive and synchronization-sensitive analysis that is machine undecidable for concurrent programs [44].

Challenge 2: *Coverage-driven test code generation requires the estimation of the additional interleaving coverage contributed by each candidate test examined during test generation.* Intuitively, only candidate tests that increase test coverage have to be outputted. Simply inferring if a candidate test t_i is redundant by estimating $cov(t_i)$ is too expensive [54]. This is because the estimation involves exploring exhaustively all feasible t_i's interleavings that satisfies the intended coverage requirements. This simple approach brings no benefits over the coverage-oblivious approach, which outputs each candidate t_i and explores its interleaving space regardless if t_i is redundant or not.

Recently, Steenbuck and Fraser proposed **ConSuite [56]**, a coverage-guided test generation technique for concurrent classes. First, it derives an approximated set of coverage requirements $\mathcal{RQ}$ with a context-insensitive and synchronization-insensitive analysis, by permuting the byte-code instructions that access the fields of the CUT according to a given parameterized interleaving coverage similar as *partial interleaving* [32]. For example, in Figure 1 there are nine such instructions $a_1, ..., a_9$. The interleavings $\sigma_1 = \langle a_8^{ta}, a_2^{tb}, a_9^{ta} \rangle$ and $\sigma_2 = \langle a_4^{ta}, a_1^{tb}, a_5^{ta} \rangle$ are two examples of statically computed interleaving coverage requirements. Second, it uses sequential test generation guided by structural coverage to obtain concurrent tests that increase interleaving coverage. The structural coverage is measured by the coverage of those instructions (i.e., $a_1, .., a_9$) that compose the interleaving coverage requirements. For example, to cover the requirement σ_1, *ConSuite* uses sequential test generation [18] to obtain two call sequences: β_1 that covers the accesses a_8 and a_9 (in the given order) and γ_2 that covers the access a_2. Then β_1 and γ_2 are assembled in the concurrent test t_A in Figure 2 covering the interleaving $\sigma_1 = \langle a_8^{t1} a_2^{t2}, a_9^{t1} \rangle$. Unlike the computation of interleaving coverage, computing the structural coverage of a call sequence can be done precisely and efficiently by keeping track of which instructions are executed by the call sequence [54]. Concurrency testing is conducted to determine whether the coverage requirements covered by the concurrent test are feasible and fault-inducing. Similarly, *ConSuite* generates the concurrent test t_B to cover σ_2.

ConSuite's approach partially addresses the two above mentioned challenges as it adopts a context-insensitive solution that likely misses the failing tests t_C and t_D in Figure 2. Context-sensitivity is especially important for object-oriented programs as the same instruction may be executed in multiple method call contexts [64]. For example, a_1 accesses a different memory location if it is triggered from `m1` or `m6`.

First, the omission of context sensitivity can lead to both *infeasible requirements* and *missing requirements*. While attempting to cover infeasible requirements only wastes testing resources, missing important coverage requirements is more alerting because it compromises fault-detection capabilities. An example of missing requirement is the faulty interleaving $\langle a_7^{t2}, a_3^{t1}, a_7^{t2} \rangle$, which is covered by the test t_C. *ConSuite* misses this requirement because it computes $\mathcal{RQ}$ using context-insensitive analysis, which does not consider that the instruction a_7 might trigger more than one shared memory access when the value of the field y is greater than zero. Missing the requirement $\langle a_7^{t2}, a_3^{t1}, a_7^{t2} \rangle$, test t_C in Figure 2 is unlikely generated.

Second, the omission of context sensitivity can also result in failure to trigger the faulty interleavings even when they are covered by a test. For example, the faulty interleaving $\sigma_2 = \langle a_4^{ta}, a_1^{tb}, a_5^{ta} \rangle$ is covered by both tests t_B and t_D in Figure 2. However, σ_2 cannot be manifested by t_B because it accesses a_4 and a_5 inside a synchronized block during which a_1 cannot occur. Incorrect generation of t_B to cover σ_2 would prohibit subsequent generation of t_D and hence fail to detect the faulty interleaving. As such, we need to consider the context of synchronization in generating concurrent tests.

Our intuition. As discussed, concurrent test generation can be effectively guided by interleaving coverage if those covered interleavings are context-sensitive. Although

[1] $\backslash$ is the set-theoretic difference operator.

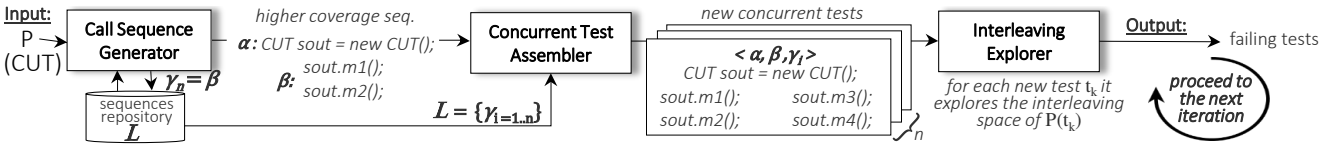

Figure 3: Overview of AutoConTest process at the n^{th} iteration. It iterates until reaching a time budget $\mathcal{B}$

static context-sensitive analysis is undecidable in general [44], context-sensitive information can be computed efficiently and precisely during dynamic sequential test generation. This observation relieves us from computing the entire coverage requirements before test generation. Instead, we can generate tests iteratively so that each test increases interleaving coverage based on the coverage data that are collected during the call sequence generation for the test. To facilitate the collection of coverage data that contain the context-sensitive information (e.g., flow information and execution of synchronization statements) of each method call, we propose a coverage metric $\mathcal{M}$ called *sequential coverage*. $\mathcal{M}$ is so defined that it can be readily measured based on a call sequence executed by a single thread. Coverage data that affect $\mathcal{M}$ are collected. While we will discuss $\mathcal{M}$ in the next section, let us illustrate our intuition using the motivating example.

Figure 4 shows how AutoConTest generates concurrent tests for the motivating example. At the first iteration, it generates a call sequence δ_1 that increases the sequential coverage of test suite $\mathcal{T}$. Since $\mathcal{T}$ is initially empty, all method calls in δ_1 increase the coverage. AutoConTest generates the test $t_1 = (\delta_1||\delta_1)$ that covers the interleaving $\langle a_8^{ta}, a_2^{tb}, a_9^{ta}\rangle$. At the second iteration, AutoConTest identifies a call sequence δ_2 that increases the sequential coverage of $\mathcal{T}$ since the method `m3` was not invoked in δ_1 and the invocation of `m5` in δ_2 triggers one more shared memory access than in δ_1. The new concurrent tests $t_2 = (\delta_2||\delta_2)$ and $t_3 = (\delta_2||\delta_1)$ increase interleaving coverage. At the third iteration, AutoConTest generates δ_3 that increases the sequential coverage of $\mathcal{T}$ because the invocation of the method `m5` executes different synchronization statements with respect to `m5`'s invocation in δ_2. Thus, δ_3 is used to generate new tests, including the failing test $t_5 = (\delta_3||\delta_2)$. We will explain in Section 4.2 how to find such high coverage sequences systematically.

4. METHODOLOGY

AutoConTest is an iterative process consisting of three major components (Figure 3)

1) The Call Sequence Generator (Section 4.2) navigates the space of possible call sequences and selects one (denoted by β in Figure 3) that improves $\mathcal{T}$'s sequential coverage $\mathcal{M}$ (Section 4.1). It adds the selected sequence β (treated as γ_n) to the generated sequence repository $\mathcal{L}$. As a result, at the n^{th} iteration, $\mathcal{L}$ contains n call sequences $\{\gamma_{i=1\ldots n}\}$.

2) The Concurrent Test Assembler (Section 4.3) accepts a sequence β from the call sequence generator and weaves it with all the n sequences $\{\gamma_{i=1\ldots n}\}$ in the repository $\mathcal{L}$ one by one, assembling n new concurrent tests.

3) The Interleaving Explorer (Section 4.4) navigates the interleaving space of the tests generated at the current iteration and reports any failing tests.

AutoConTest iterates until reaching a given time budget. Next we present the sequential coverage, followed by the description of the three components.

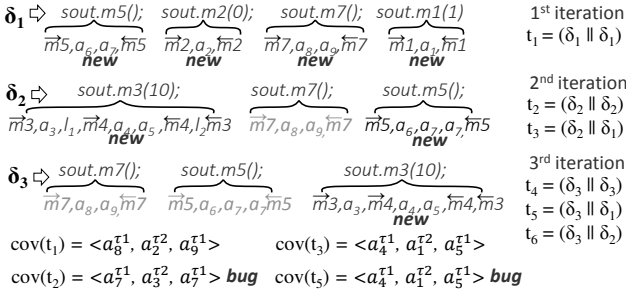

Figure 4: AutoConTest running example

4.1 Sequential Coverage

This section presents a context-sensitive coverage metric $\mathcal{M}$ on call sequences, referred to as sequential coverage. Let E denote the **trace** of a call sequence δ, i.e., the ordered sequence of events exhibited by a sequential (single-threaded) execution of δ. An event can be one of the following:

- write and read accesses to object fields a_i
- lock acquire and release events l_j
- method enter $\overrightarrow{m}$ and exit $\overleftarrow{m}$ events.

An access a_i is encoded by the code location (at bytecode level) that triggers a_i. This abstraction ignores the values of the shared memory accesses. We follow existing work on interleaving coverage [32, 58, 67] and assume that the manifestation of fault-inducing interleavings depends on the exposure of erroneous inter-thread memory dependencies, which are independent of the data values of the shared memories involved (*value-independent assumption* [67]). In Java, a lock acquire event is generated when a synchronized block or method is entered, and a lock release event is generated when exiting the block or method. We encode these events by their code location and type (i.e., lock, unlock). The enter/exit events of a method m are encoded with m's signature.

Definition 1. Given a call sequence $\delta=\langle c_1, \ldots, c_n\rangle$, the **trace of a method call** $c_i \in \delta$ is the non-empty segment E_i of E such that E_i contains only the events triggered by the invocation of c_i.

Definition 2. Given a call sequence δ, its **sequential coverage $\mathcal{M}(\delta)$** is defined as the partition $\{E_1, E_2, \ldots, E_n\}$ of E, i.e., the unordered set composed by the n method call traces of E.

For example, $\mathcal{M}(\delta_2) = \{\langle\overrightarrow{m}3, a_3, l_1, \overrightarrow{m}4, a_4, a_5, \overleftarrow{m}4, l_2, \overleftarrow{m}3\rangle, \langle\overrightarrow{m}7, a_8, a_9, \overleftarrow{m}7\rangle, \langle\overrightarrow{m}5, a_6, a_7, a_7, \overleftarrow{m}5\rangle\}$ in Figure 4. Note that the trace of a method call c_i include all indirectly covered events, i.e., those covered by all callees of c_i. We now study the situation in which a call sequence increases the sequential coverage of a suite $\mathcal{T}$. Let $\mathcal{L}$ denote the cumulative set of call sequences generated to assemble concurrent tests at previous iterations, i.e., $\mathcal{L}=\{\beta, \gamma \mid (\beta \mid\mid \gamma) \in \mathcal{T}\}$. We consider two method call traces to be equivalent iff they are constituted by the same events in the same order.

Definition 3. A call sequence δ increases $\mathcal{T}$'s sequential coverage iff δ's **coverage improvement** over $\mathcal{L}$ i.e.,

$$\boldsymbol{\Delta}_{\mathcal{M}}(\delta, \mathcal{L}) = \{\mathcal{M}(\delta) \setminus \cup_{i=1}^{|\mathcal{L}|}\mathcal{M}(\delta_i)\} \text{ is non-empty.}$$

We drop $\mathcal{L}$ from the notation $\Delta_{\mathcal{M}}(\delta, \mathcal{L})$ becoming $\Delta_{\mathcal{M}}(\delta)$ if $\mathcal{L}$ refers to the cumulative set of call sequences generated to assemble the tests before checking δ's coverage improvement.

Consider the example in Figure 4. At the third iteration, $\mathcal{T} = \{t_1 = (\delta_1||\delta_1), t_2 = (\delta_2||\delta_2), t_3 = (\delta_2||\delta_1)\}$ and $\mathcal{L}= \{\delta_1, \delta_2\}$. The call sequence δ_3 increases the sequential coverage of $\mathcal{T}$ because $\mathcal{M}(\delta_3)$ contains one new method call trace that is neither contained in $\mathcal{M}(\delta_1)$ nor $\mathcal{M}(\delta_2)$, i.e., $\Delta_{\mathcal{M}}(\delta_3) = \{\langle \overrightarrow{m}3, a_3, \overrightarrow{m}4, a_4, a_5, \overleftarrow{m}4, \overleftarrow{m}3\rangle\}$. Thus, δ_3 is used to assemble new concurrent tests with the call sequences in $\mathcal{L}$, yielding to the fault-inducing test $t_5 = (\delta_3||\delta_1)$.

To distinguish a method call $c_i \in \delta$ from the ones that c_i directly or indirectly invokes, we refer to c_i as an *outermost* method call in the following discussion. The definition of $\mathcal{M}(\delta)$ leverages an observation that the coverage can be computed by omitting the sequential execution order of the outermost method calls in δ without affecting concurrency fault detection capabilities. This is because the general form of *thread-safety* [23] does not guarantee that multiple calls to a shared object of a thread-safe class are executed atomically in the same thread [43]. These multiple calls require external synchronizations [30, 31] to preserve atomicity. For example, Figure 5 shows a concurrent test for the JDK Vector class, a_1, a_2 and a_3 are accesses to the field `elementData`. The interleaving $\langle a_1^{t1}, a_3^{t2}, a_2^{t1}\rangle$ throws an exception. However, it does not expose a thread-safety violation or a concurrency fault because the operations in *thread1* are supposed to be composed in the same atomic operation [30, 31]. Pradel et al. [43] prune all of such interleavings during concurrent testing by checking if every caught exception can also be triggered by a *linearization of the calls* [12], e.g., $\langle a_1^{t1}, a_3^{t1}, a_2^{t1}\rangle$ in Figure 5. Instead, we avoid these false positives altogether by assuming the interleavings that contain accesses triggered by the same thread but different outermost method calls (e.g., the interleaving in Figure 5) should not be tested, i.e., they are not coverage requirements. An advantage of ignoring the order of outermost method calls in a sequential call sequence execution is that the coverage $\mathcal{M}$ likely attains saturation faster. For example, given $\delta_1 = \langle$`sout.m1(1),sout.m2(1)`$\rangle$ and $\delta_2 = \langle$`sout.m2(1),sout.m1(1)`$\rangle$, $\mathcal{M}(\delta_1) = \mathcal{M}(\delta_2)$ even if the execution traces of δ_1 and δ_2 are different as they trigger the same events but in a different order.

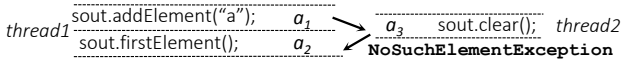

Figure 5: Example of invalid atomic composition

4.2 Call Sequence Generation

At each iteration, the call sequence generation component systematically explores the space of possible call sequences, and it returns one, denoted by β, with the highest sequential coverage improvement among those explored at the current iteration. This sequence will be used to assemble new concurrent tests. As the cost of interleaving exploration increases with execution length, when the component compares sequences with same degree of coverage improvement, it opts for the shortest one (measured by the number of outermost method calls). More formally, let us define the following relation on calls sequences. Given two sequences δ_1 and δ_2, δ_1 is "*better than*" δ_2 in improving the sequential coverage, denoted by $\delta_1 \succ \delta_2$, iff $|\Delta_{\mathcal{M}}(\delta_1)| > |\Delta_{\mathcal{M}}(\delta_2)|$, or $(|\Delta_{\mathcal{M}}(\delta_1)| = |\Delta_{\mathcal{M}}(\delta_2)| \wedge |\delta_1| < |\delta_2|)$. Let $\mathcal{Q}$ denote the set of call sequences explored by the component at a given iteration.

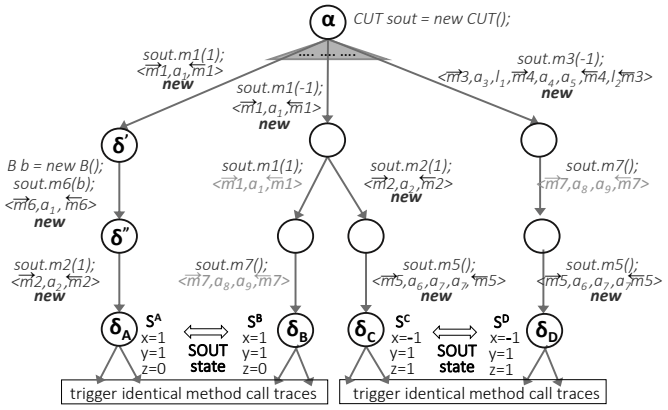

Figure 6: Examples of redundant sequence δ_B

Definition 4. The component's output is an **optimal call sequence** $\boldsymbol{\beta} \in \mathcal{Q}$ such that $\Delta_{\mathcal{M}}(\beta) \neq \emptyset \wedge \nexists\, \delta \in \mathcal{Q}, \delta \succ \beta$

The search space is represented as a rooted, directed and potentially infinite tree $\mathbb{T}$ whose root node is a call sequence α that instantiates the object SOUT. The instantiation involves making a constructor call and additional method calls if necessary to setup non primitive parameters. The edges represent method calls. Each node in the tree represents a call sequence that corresponds to the ordered sequence of the method calls along the path from the root to the node. Figure 6 shows the portion of the search space representing four sequences δ_A, δ_B, δ_C and δ_D for the motivating example in Figure 1. For instance, the node δ_B represents the sequence $\langle$ `CUT sout = new CUT();sout.m1(-1);sout.m1(1);sout.m7()`$\rangle$. The search space is explored by dynamically building the tree starting from the root.

To dynamically build the tree, we adapt the *sequence extension operator* of *Randoop* [40] to the *node traversal operator*, which traverses from a node to its child as follows. Given a method m and a node representing sequence δ, the node traversal operator produces a child node representing a new sequence δ'. The new sequence is formed by appending to δ a sequence of method calls (i.e., an edge) with m being the last method call. Other method calls are added before m in the appending sequence to create m's non-primitive parameters, if any. Note that alternative extension operators like those presented in *Evosuite* [21] are unsuitable. This is because they change or insert method calls at random sequence position, which compromises the tree traversal due to tree structure likely changes at each extension.

The *candidate methods* (CM) to extend a sequence (node) are those methods in $\mathcal{P}$ such that they have at least one parameter p of type CUT, which can be binded to the shared object SOUT (see Section 2.1). Each input parameter p_i is confined to a finite set of possible values. The tree traversal operator selects a parameter's value in the following way. If p_i is a primitive type, its value is chosen from a *pool* of representative values. If p_i is a non-primitive type, there are two possibilities: if p_i is of type CUT, it is always binded to the object reference of SOUT; otherwise it is always binded to a new object of type p_i, which is created by appending appropriate method calls chosen from a *pool* of representative sequences. For example, in Figure 6 the node δ' is the extension of α with the method call `sout.m1()`. While δ'' is the extension of δ' with method calls "`B b = new B(); sout.m6(b);`". Since `m6` has a non-primitive parameter of type `B`, the tree traversal operator adds the method calls to

create it. AutoConTest systematically explores the possible extensions of a node with different combination of parameter values. The pools of primitive and non-primitive parameter values are constructed by *Randoop* pseudo-deterministically (using the same random seed [40]) at each iteration. AutoConTest uses different random seeds across iterations to achieve diversity. As a result, the set of out-going edges of every node is the same at each iteration, this property is essential to our tree traversal.

A key challenge is how to effectively construct the search space of possible call sequences and compare their coverage improvements. However, an exhaustive construction of such a search space is infeasible even by bounding the length of call sequences to a given value [11, 36]. To address the challenge, AutoConTest deploys a **coverage-driven** search traversal that predicts before extending a node to its children if these children can have a descendant that represents an optimal call sequence β. If not, these children are skipped. The prediction is made by reasoning about the program state and the coverage reached by each visited call sequence.

Each execution of a call sequence $\delta{=}\langle c_1, \ldots, c_n\rangle$ induces a sequence of state transitions $S_0 \xrightarrow{c_1} S_1 \xrightarrow{c_2} S_2 \ldots \xrightarrow{c_n} S_n$. Let S_{i-1} denote the state of c_i's parameters before c_i is invoked, and S_i the state after. Like Xie et al. [65], we assume that the sequential execution of a method call c_i is deterministic given the state S_{i-1} Under this assumption, the method call c_{n+1} triggers the same method call trace if it is appended at the end of two different call sequences that reach the same state S_n. For example, the method call `sout.m5()` triggers the same method call trace $\langle \overrightarrow{m}5, a_6, a_7, \overleftarrow{m}5\rangle$ if it is appended after δ_A or after δ_B. This is because δ_A and δ_B reach the same state of `m5`'s parameters, denoted by $S(\delta_A)$ and $S(\delta_B)$, respectively. Therefore, exploring the possible extensions of a call sequence is redundant if we have already explored the possible extensions of another call sequence that reaches the same state and with better coverage improvement.

Definition 5. A call sequence δ is **redundant** iff $\exists \delta^* \in \mathcal{Q}$ such that $S(\delta) = S(\delta^*) \wedge (\Delta_{\mathcal{M}}(\delta) \subset \Delta_{\mathcal{M}}(\delta^*) \vee (\Delta_{\mathcal{M}}(\delta) = \Delta_{\mathcal{M}}(\delta^*) \wedge |\delta^*| \leq |\delta|))$.

Theorem 1. *The unexplored descendants of a node representing a redundant call sequence do not need to be explored in order to reach the optimal solution β.*

Proof. It is proved by contradiction. Let assume that a sequence δ is redundant and the optimal β is a descendant of δ. From Def. 5, it exists a sequence δ^* such that $\Delta_{\mathcal{M}}(\delta) \subseteq \Delta_{\mathcal{M}}(\delta^*)$, which implies (1). Because each node has the same set of out-going edges, $S(\delta) = S(\delta^*)$ and by assuming that sequential executions are deterministic, it exists a descendant of δ^* denoted by β^* such that the set of method call traces triggered along the path from δ^* to β^*, denoted by $\mathcal{M}(\overline{\beta})$, is identical to the set of method call traces triggered along the path from δ to β, denoted by $\mathcal{M}(\overline{\beta^*})$. This implies $\Delta_{\mathcal{M}}(\overline{\beta}) = \Delta_{\mathcal{M}}(\overline{\beta^*})$. Together with (1) we obtain (2). Because $\Delta_{\mathcal{M}}(\beta) = \{\Delta_{\mathcal{M}}(\delta) \cup \Delta_{\mathcal{M}}(\overline{\beta})\}$ and $\Delta_{\mathcal{M}}(\beta^*) = \{\Delta_{\mathcal{M}}(\delta^*) \cup \Delta_{\mathcal{M}}(\overline{\beta^*})\}$, (2) and $\Delta_{\mathcal{M}}(\delta) \subseteq \Delta_{\mathcal{M}}(\delta^*)$ we obtain (3). If $|\Delta_{\mathcal{M}}(\beta)| < |\Delta_{\mathcal{M}}(\beta^*)|$ $\beta \not\succ \beta^*$ (contradiction). If $|\Delta_{\mathcal{M}}(\beta)| = |\Delta_{\mathcal{M}}(\beta^*)|$, from (3) we have that $|\Delta_{\mathcal{M}}(\delta)| = |\Delta_{\mathcal{M}}(\delta^*)|$ but $|\delta^*| \leq |\delta|$ (Def. 5), thus $\beta \not\succ \beta^*$ (contradiction). □

$$|\Delta_{\mathcal{M}}(\delta)| \leq |\Delta_{\mathcal{M}}(\delta^*)| \quad (1)$$

$$|\Delta_{\mathcal{M}}(\delta)| + |\Delta_{\mathcal{M}}(\overline{\beta})| \leq |\Delta_{\mathcal{M}}(\overline{\beta^*})| + |\Delta_{\mathcal{M}}(\delta^*)| \quad (2)$$

$$|\Delta_{\mathcal{M}}(\beta)| \leq |\Delta_{\mathcal{M}}(\beta^*)| \quad (3)$$

```
Function callGen(δ)
    if Δ_M(δ) is saturated then
        return δ
    β ← δ          // init best seq with prefix δ
    for each m ∈ CM do
        δ' ← δ ∪ m          // sequence extension
        execute δ'          // gathering runtime data
        if DepthPruning == false then
            β' ← callGen(δ')          // recursion
            if β' ≻ β then
                β ← β'
        if BreadthPruning == true then
            break for loop          // stop extending δ
    return β
```

Figure 7: Call sequence generation algorithm

For example, assume that the call sequences δ_A, δ_B, δ_C and δ_D in Figure 6 are explored in the given order at an iteration. Let $S(\delta_A)$ denote the SOUT's state after executing δ_A. S_n^A and S_n^B are equivalent as SOUT's fields have the same values. Thus, for each path from δ_A to one of its descendants, there exists a corresponding path from δ_B to one of its descendants such that the method call traces triggered along these two paths are identical. Since $\Delta_{\mathcal{M}}(\delta_B) \subset \Delta_{\mathcal{M}}(\delta_A)$, the descendants of δ_B cannot lead to a sequence with a coverage improvement higher than any of the call sequences extended from δ_A. Thus, children of δ_B can be skipped.

Considering the additional method call traces covered $\Delta_{\mathcal{M}}(\delta)$ rather than the degree of coverage improvement $|\Delta_{\mathcal{M}}(\delta)|$ by a call sequence is crucial to guarantee a lossless pruning. Consider the call sequences δ_C and δ_D yielding to the same state of the object SOUT. Although δ_D has a lower degree of coverage improvement than δ_C, $\Delta_{\mathcal{M}}(\delta_D)$ contains the method call trace $\langle \overrightarrow{m}3, a_3, l_1, \overrightarrow{m}4, a_4, a_5, \overleftarrow{m}4, l_2, \overleftarrow{m}3\rangle$ which is not in $\Delta_{\mathcal{M}}(\delta_C)$, i.e., $\Delta_{\mathcal{M}}(\delta_D) \not\subset \Delta_{\mathcal{M}}(\delta_C)$. Thus δ_D's subtree has to be explored as it could contain the optimal call sequence β. For instance, the SOUT's state of S_n^C and S_n^D has $z = 1$ and it cannot be set at zero by any subsequent method call. As a result the method call trace triggered by the method `m3` in δ_D cannot be triggered by any edge in the subtree of δ_C and δ_D (see Figure 1). However, the method call traces in $\Delta_{\mathcal{M}}(\delta_C)$ can be triggered in the subtree of δ_D, thus there exists a descendant of δ_D with higher coverage than any descendants of δ_C. We now describe our call sequence generation algorithm (Figure 7) and the search space pruning strategies (Figure 8).

Search order. We choose to traverse the tree using *depth-first search* (DFS). Since coverage tends to increase with sequence length, the DFS strategy likely finds a higher coverage sequence faster than the *breadth-first search* (BFS).

Stopping criterion. There may exist some sequences that can be continuously extended to give new non-redundant sequences. Thus, without a well-defined stopping criterion the algorithm could easily exhaust all the available testing time budget. A possible solution is to impose an upper bound on the sequence length [11, 36]. The issue of this approach is that there is no pragmatic way to choose the best bound for a given CUT. Instead, the algorithm adopts a saturation based stopping criterion. Let δ' denote an extended call sequence of δ with a method. The algorithm stops extending δ' but continues to extend δ with another pending method if none of the latest k extensions of δ' exhibits coverage improvement. If the algorithm terminates without finding a sequence β that improves the sequential coverage, it is re-launched with the saturation level k incremented.

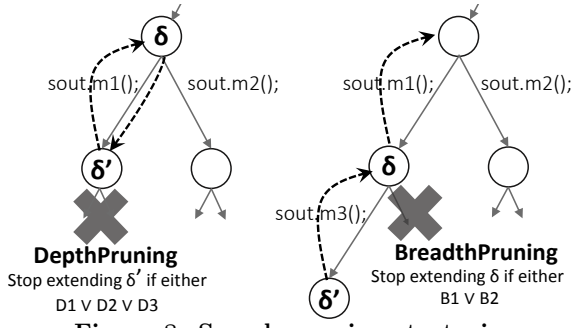

Figure 8: Search pruning strategies

Gathering runtime data. Each newly explored sequence is executed to collect the following information.

1. $\boldsymbol{\delta}$**.excp**, a binary value that is *true* if the execution of δ throws a caught or uncaught exception; *false* otherwise.
2. $\boldsymbol{\Delta_{\mathcal{M}}(\delta)}$, which is computed by executing an instrumented version of the program under test.
3. $\boldsymbol{S(\delta)}$, the state of the object SOUT after executing δ. It is obtained by serializing SOUT in a deep copy semantic.

Note that our search space exploration strategy is carefully designed so that we need only to consider the object SOUT for representing the state of call sequences. This is because SOUT is the only object that will be reused in later extensions. There are two useful properties relating a sequence δ and its extension δ'. $\Delta_{\mathcal{M}}(\delta) \subseteq \Delta_{\mathcal{M}}(\delta')$ (*property 1*) and $|\ \delta\ | < |\ \delta'\ |$ (*property 2*). These properties enable us to optimize the coverage computation. First, there is no need to compute the entire sequential coverage of δ' from scratch. Instead, the coverage can be computed by combining the previously computed coverage of δ and the covered method call traces of the newly added extension. This significantly reduces the computation cost. Second, if the coverage of δ' does not improve, we can conclude $\delta' \not\succ \delta$ by property 2.

Depth Pruning The algorithm stops to extend δ' but continues to extend δ with another pending method, if any of the following conditions holds.

D1: $\delta'.excp = true$. If δ' throws an exception, it is not extended because all sequences with δ' as prefix would also throw the exception. This pruning strategy was first presented by Pacheco et al. [40]. The occurrence of an exception likely indicates an illegal sequence, which are more likely to be generated than legal sequences because only a small set of all possible sequences is legal [10]. Fortunately, conscious developers mostly anticipate possible misuses of public methods and protect their code against such misuses by checking the object state or the parameter values at the beginning of these methods, throwing an exception upon misuses.

D2: $|\Delta_{\mathcal{M}}(\delta')|=|\Delta_{\mathcal{M}}(\delta)| \wedge S(\delta')=S(\delta)$. If δ' neither increases the coverage nor changes the program state, it is not extended because δ' brings no extra benefit over its prefix δ. In fact, δ' is redundant with respect to δ by properties 1 and 2.

D3: $|\Delta_{\mathcal{M}}(\delta')|=|\Delta_{\mathcal{M}}(\delta)| \wedge \delta'$ is *redundant*. Although δ' is non-redundant with respect to δ if D2 is false, δ' could still be redundant with respect to other sequences that have been explored at the current iteration. We check if δ is redundant by tracking the mapping between each observed state S and the sequential coverage of those sequences with state S.

Breadth Pruning. The algorithm stops the "breadth" visit of a node, i.e., it avoids extending δ with any pending methods if either of the following conditions is met.

B1: $|\Delta_{\mathcal{M}}(\delta')|>|\Delta_{\mathcal{M}}(\delta)| \wedge S(\delta')=S(\delta)$. If this is true, δ becomes redundant with respect δ' by property 1 and 2. Note that the first time the algorithm extended δ, δ was not redundant (otherwise δ would have not been extended).

B2: δ is *redundant*. This check is analogous to D3.

4.3 Concurrent Test Assembler

Given a call sequence β returned by the Call Sequence Generator component at the current iteration and the set of returned sequences at previous iterations $\mathcal{L}=\{\gamma_i\}$, the Concurrent Test Assembler component creates new concurrent tests as follows. It updates $\mathcal{L}$ with β and then creates $|\mathcal{L}|$ new concurrent tests by assembling β with each γ_i in $\mathcal{L}$, i.e., $\{t_i = (\beta \mid\mid \gamma_i)\ \forall \gamma_i \in \mathcal{L}\}$. For instance, at the third iteration in the running example in Figure 4, AutoConTest creates three new tests t_4, t_5 and t_6 by assembling δ_3 (i.e., β at the third iteration) with δ_3 (itself), δ_1 and δ_2. As a result, by construction all new method call traces covered by β are concurrently pair-wise tested with all method call traces in $\mathcal{L}$. This facilitates the improvement of interleaving coverage because covering a new pair of method call traces is a condition necessary to cover a new feasible and fault-inducing interleaving. For example, to cover the feasible and fault-inducing interleaving $\langle a_4^{t1}, a_1^{t2}, a_5^{t1}\rangle$ in Figure 1, a concurrent test needs to execute concurrently the pair of method call traces $\{\langle \overrightarrow{m}3, a_3, \overrightarrow{m}4, a_4, a_5, \overleftarrow{m}4, \overleftarrow{m}3\rangle\}$ and $\{\langle \overrightarrow{m}1, a_1, \overleftarrow{m}1\rangle\}$.

Theorem 2. *A concurrent test $t_x = (\beta \mid\mid \gamma)$ covers a new pair of concurrent method call traces, if t_x increases the interleaving coverage of $\mathcal{T}$, i.e., $\exists$ feasible and fault-inducing interleaving $x \in cov(t_x)$ such that $x \notin \cup_{i=1}^{|\mathcal{L}|} cov(t_i)$.*

Proof sketch. By contradiction, assume that t_x exists and it does not cover a new pair of method call traces. For x being a feasible and fault-inducing interleaving not covered in $\mathcal{T}$ there could be only two cases: (1) x contains a newly covered shared memory access with respect to all sequences in $\mathcal{T}$ (2) x is already covered in $\mathcal{T}$ but the memory accesses in x are executed either by a new intra-invocation path or under a different sequence of synchronization operations, which makes x feasible and fault-inducing in t_x but not in $\mathcal{T}$. Both cases (1) and (2) require that either β or γ covers a new method call trace. (contradiction). □

An additional check could be performed on each new pair to infer whether they are relevant for the given interleaving coverage criterion. To let AutoConTest be applicable to any criteria, we choose to not perform this check. Nevertheless, covering all possible pairs is a sufficient condition to test adequacy, regardless the criterion considered.

4.4 Interleaving Explorer

A concurrent test can exhibit many interleavings. Given the concurrent tests assembled at a given iteration, the Interleaving Explorer component, therefore, identifies and tests the interleavings that match the specified interleaving coverage criterion. A key challenge is to identify and test all coverage requirements covered by each concurrent test. It is because, when a call sequence is executed concurrently with other sequences, it can manifest a different behavior (i.e., triggers different method call traces) from its sequential one. This phenomenon is referred to as **concurrent interference**, which occurs when multiple sequences access and mutate the shared object SOUT concurrently. When this phenomenon occurs, the method call trace that can increase the interleaving coverage might not be executed. A method is subject to concurrent interference if the SOUT's state

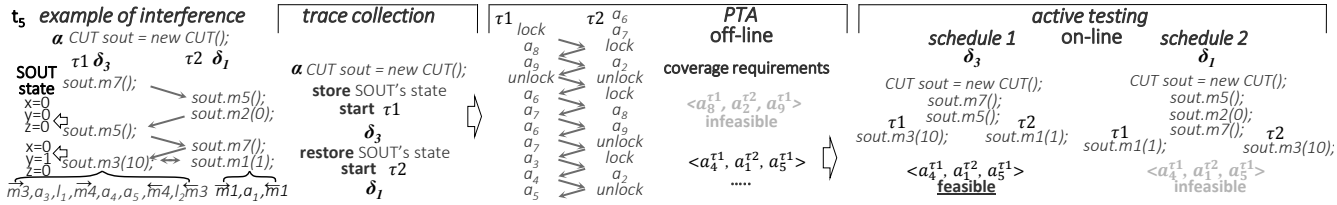

Figure 9: How AutoConTest addresses the problem of interference during concurrent execution

when, the method is called in a concurrent test, differs from that when the method is called in a sequential execution.

For example, consider the sequences δ_1 and δ_3 in Figure 4, the sequential execution of δ_3 triggers the method call trace $\langle \overrightarrow{m}3, a_3, \overrightarrow{m}4, a_4, a_5, \overleftarrow{m}4, \overleftarrow{m}3\rangle$, while that one of δ_1 triggers $\langle \overrightarrow{m}1, a_1, \overleftarrow{m}3\rangle$. The fault-inducing interleaving $\langle a_4^{t1}, a_1^{t2}, a_5^{t1}\rangle$ can be triggered (i.e., covered) only if these two method call traces are concurrently executed. Figure 9 (left) shows a possible concurrent interference during the concurrent execution of the test $t_5 = (\delta_3 \mid\mid \delta_1)$ that makes δ_3 trigger a different method call trace $\langle \overrightarrow{m}3, a_3, l_1, \overrightarrow{m}4, a_4, a_5, \overleftarrow{m}4, l_2, \overleftarrow{m}3\rangle$ that does not cover the faulty interleaving, i.e., $\langle a_4^{t1}, a_1^{t2}, a_5^{t1}\rangle$ is infeasible because the accesses a_4 and a_5 are executed inside a synchronized block. The concurrent interference from δ_1 modifies the value of the SOUT's field y before δ_3's method invocation of `m5` uses the old value of y to set the field z to a value different from zero. If z is zero, the invocation of `m3` takes a different execution path, triggering a different method call trace.

To address the challenge, we adopt Predictive Trace Analysis (PTA) [53] to identify the coverage requirements, and we use a dedicate thread scheduler algorithm to avoid concurrent interferences while testing these requirements.

PTA (e.g., [26, 28, 41, 55, 63]) runs a concurrent test t with an instrumented version of the program, generating an execution trace of memory accesses and concurrency operations. Then, it explores the interleaving space of a test t off-line by re-shuffling the order of the memory accesses in the trace to match those interleavings required by a given coverage criterion. Then, it prunes the interleavings that are infeasible using a *lockset/happens-before checker* [28]. For example, $\langle a_8^{t1}, a_2^{t2}, a_9^{t1}\rangle$ is infeasible because the three accesses are protected by the same lock. Our intuition is that PTA can still explore t's interleaving space even if the execution trace is collected by executing the two threads sequentially, one after the other. Thus, it can effectively identify the coverage requirements without suffering from concurrent interference. Figure 9 shows how AutoConTest executes the test for collecting the execution trace to be analyzed by PTA: (1) AutoConTest creates the object SOUT and saves its state. (2) Thread τ_1 executes δ_3 until completion. (3) AutoConTest restores the initial state of the object SOUT to guarantee that the execution of δ_1 delivers the same sequential coverage as that observed during the call sequence generation. (4) Finally, thread τ_2 executes δ_1.

The retained interleavings (e.g., $\langle a_4^{t1}, a_1^{t2}, a_5^{t1}\rangle$) are then tested with real executions to check if they manifest faulty behaviors (e.g., crashes, hangs or assertion violations). The interleavings are enforced using a thread scheduler algorithm (i.e., active testing). Upon calling the methods that trigger the shared memory accesses involved in the interleaving to be tested (e.g., `sout.m3(10)` and `sout.m1(1)`), the algorithm ensures that the state of SOUT is the one observed when the associated call sequence is executed sequentially. This is to avoid concurrent interference to binding the state of SOUT arbitrarily. Intuitively, there are two candidates of SOUT's state. The SOUT's state reached by the sequential execution of δ_3 before invoking `sout.m3(10)`, and the state reached by the sequential execution of δ_1 before invoking `sout.m1(1)`. So, for each interleaving to be tested, we run active testing on two concurrent test schedules. Figure 9 shows the two schedules for the interleaving $\langle a_8^{t1}, a_2^{t2}, a_9^{t1}\rangle$. In one schedule, we concurrently invoke `sout.m3(10)` and `sout.m1(1)` on the SOUT's state reached by the sequential execution of δ_3 before invoking `sout.m3(10)`. In the other schedule, we concurrently invoke `sout.m3(10)` and `sout.m1(1)` on the SOUT's state reached by the sequential execution of δ_1 before invoking `sout.m1(1)`. In the running example in Figure 9, the first schedule succeeded to trigger the faulty interleaving.

5. EVALUATION

In this paper, we propose AutoConTest, a context-sensitive and coverage driven test code generation technique for concurrent classes. We evaluate if AutoConTest is effective in exposing concurrency faults and able to achieve high interleaving coverage. The interleaving coverage considered in our experiments is the set of problematic interleavings that match the access patterns violating atomic-set serializability [61]. We carried out a series of experiments to answer the following research questions:

RQ1: Given a time budget, how effective is AutoConTest in detecting concurrency faults and achieving high interleaving coverage?

RQ2: Is AutoConTest more effective than random-based test generation [43]?

RQ3: Are AutoConTest's search pruning strategies effective in exploring the search space? Can they mostly achieve a lossless pruning?

Implementation. To evaluate our technique, we implemented AutoConTest into an automated tool for Java classes. The test generation phase adapts *Randoop* [40] to extend call sequences and to generate runnable Java test code. The adaptation is presented in Section 4.2. To compute the coverage metric in the execution-feedback process, we instrumented the program at load-time using the ASM framework [1]. AutoConTest uses the XStream framework [9] to serialize object states. The framework can serialize any Java objects without requiring their classes to implement the `java.io.Serializable` interface. To check if a method invocation modifies the state of its object receiver, we compared the serializations obtained by XStream before and after the invocation. We adopted an efficient and conservative comparison approach: two object states are equivalent if the two serializations are identical; otherwise they are not equivalent. The interleaving explorer is built upon the PTA technique *AssetFuzzer* [28]. We modified its active-testing phase according to the modifications discussed in Section 4.4.

Table 1: Subjects description and the experimental results with a time budget of one hour

Subject Description					AutoConTest				ConTeGen*				
BUG ID	Code base	Class Under Test (CUT)	CUT SLOC	Issue ID	Time First Fault (ms)	# Tests	Suite Size	|cov(t)| Tot.	Random Seed	Time First Fault (ms)	# Tests	Suite Size	|cov(t)| Tot.
1	Apache Commons 2.4	[..].lang.math.IntRange	278	[2]	22,438	7	2157	19	1	66,357	110	1742	91
2	Google Commons 1.0	[...]AbstractMultiMap$AsMap	1125	[6]	28,967	3	19	1	0	*1hr	130	1898	0
3	Java JDK 1.1.7	java.util.Vector	216	[4]	64,506	17	1437	2	4	1,014,452	185	1232	2
4	JFreeChart 0.9	[...]chart.axis.NumberAxis	1298	[3]	35,075	1	114	23	0	156,846	99	1351	3
5	Java JDK 1.4.1	java.util.logging.Logger	992	[7]	45,141	3	105	1	0	*1hr	230	3161	0
6	Java JDK 1.4.2	java.util.Vector	326	[5]	29,588	1	104	33	0	2,254,045	167	2729	1

Subjects. We performed our experiments on six known real-world concurrency faults from four popular codebases, which have been used in the evaluation of related works [43, 56]. For each fault, Table 1 gives the following information: the subject ID that is used in the rest of the paper, the name and version of the code base, the Class Under Test (CUT), the number of Source Lines Of Code (SLOC) of the CUT (including CUT's non-abstract super class if any) and the bug report links from the issue-tracking systems. Note that all these subjects have a single fault. However a single concurrency fault might trigger multiple faulty interleavings.

5.1 RQ1 - Cost/Effectiveness

For each subject, we ran AutoConTest with a time budget of one hour. The measured elapse time includes the time for test generation and interleaving exploration. The Column "Time First Fault" in Table 1 indicates the elapsed time to trigger the first faulty interleaving with active testing. The Column "# Tests" indicates the number of concurrent tests generated and explored within the given time budget. The Column "Suite Size" shows the size of the tests generated for each subject. Like previous works, test size was measured by the number of (non-native libraries) method calls [20, 24]. The Column "|cov(t)| Tot." indicates the cumulative number of *feasible* coverage requirements that have been successfully triggered by active testing. AutoConTest detected the first fault for all subjects with an average time of 38 seconds. On average for all subjects, 17% of the time budget was spent on test generation while the remaining time on interleaving exploration. In the experiment, the first test generated by AutoConTest for each subject successfully revealed the first fault. This demonstrates that AutoConTest is effective in generating a small number of effective concurrent tests that expose concurrency faults.

5.2 RQ2 - Comparison with ConTeGen

We compared AutoConTest with *ConTeGen* [43], a state-of-the-arts technique for the random generation of concurrent test code. The comparison is based on *ConTeGen*'s original implementation [8], which is publicly available. We ran *ConTeGen* with the same time budget of one hour, five times using different random seeds [43]. All runs of *ConTeGen* did not detect any concurrency fault within the time budget. *ConTeGen* uses stress-testing as interleaving exploration methodology, which is known to be ineffective in exploring interleaving spaces [41, 64]. For fair comparison, we conducted further experiments to compare AutoConTest and an approach that explores the interleaving space of the tests generated by *ConTeGen* using the same interleaving exploration component of AutoConTest. We call this approach *ConTeGen**. It enables us to compute the coverage requirements covered by *ConTeGen* tests. Table 1 shows the best result among the five different runs of *ConTeGen**. Column

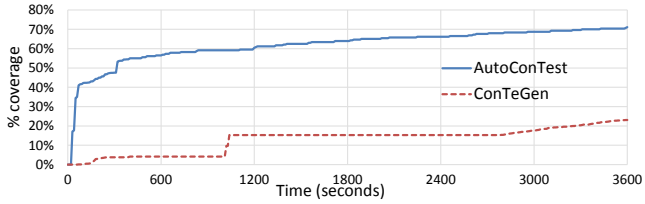

Figure 10: Coverage comparison

"Random Seed" indicates the seed value that achieved the best result (in terms of first fault detected). All runs of *ConTeGen** failed to detect the two faults with ID 5 and 2. The first faults of the remaining subjects were detected between 1 to 37 minutes, with an average of 14.5 minutes. We calculated the cumulative coverage in percentage at the same interval of time and we compute the average for each subject. Figure 10 shows the comparison results. On average, AutoConTest achieved in less than 40 seconds the same percentage of coverage achieved by *ConTeGen** in one hour. We notice that our approach achieved higher coverage than much larger test suite computed randomly. This suggests that the effectiveness of AutoConTest mainly arises from the ability to cover corner program behaviors, which are difficult to cover by randomly generating more tests. In fact, randomly generated tests are subject to redundancy as reflected by the slow increase in *ConTeGen**'s cumulative coverage over time in Figure 10.

We were not able to directly compare with *ConSuite* [56] as the tool is not publicly available. However, we noticed that 58% of the tested interleavings involve the same set of static instructions under different calling contexts. This suggests the importance of context sensitivity.

5.3 RQ3 - Search Pruning Strategies

The goal of this research question is to evaluate the Call Sequence Generation Component. For this set of experiments we ran the component in isolation, i.e., without exploring the interleaving space of the generated tests. We ran two versions of the component. One version is obtained by enabling all the search pruning strategies (see Figure 8). The other version is obtained by disabling all pruning strategies except $D1$, which is already supported by *Randoop* [40]. We ran both versions on our subjects with a time budget of one hour using the same initial saturation stopping criterion, i.e., saturation level of $k = 3$ see Section 4.2. Table 2 shows the results. The first iteration of all subjects terminated in a few seconds (3 sec. on average) for the version with pruning strategies enabled whereas the first iteration of all subjects failed to terminate within an hour for the version with pruning strategies disabled. The second column of Table 2 shows the saturation time of the enabled version, i.e., the time of the last returned sequence. The saturation level attained at the last returned sequence was $k = 84$ on average for all subjects, while the average number of iterations (i.e.,

Table 2: Search pruning strategies evaluation

	AutoConTest pruning strategies enabled					Disabled		
BUG	Time	Cum.	Optimal β first iteration			Optimal β first iteration		
ID	Saturation	$\lvert\Delta_{\mathcal{M}}\rvert$	Time (ms)	$\lvert\Delta_{\mathcal{M}}(\beta)\rvert$	$\lvert\beta\rvert$	Time (ms)	$\lvert\Delta_{\mathcal{M}}(\beta)\rvert$	$\lvert\beta\rvert$
1	8,151	30	1,598	25	43	*1hr	25	43
2	4,505	7	1,741	6	11	*1hr	6	11
3	14,070	71	1,931	23	43	*1hr	24	48
4	301,605	70	7,098	56	91	*1hr	56	91
5	1,329,381	53	2,911	24	60	*1hr	24	60
6	452,560	344	2,866	44	88	*1hr	44	87

returned sequences) was on average 20. Recall that if the component terminates the exploration without finding a call sequence that improves the coverage, it is re-launched with the saturation level k incremented. The third column shows the number of distinct method call traces covered by the returned sequences.

We also evaluated if the sequences returned by the enabled version at the first iteration were optimal, by comparing them with the optimal sequences returned by the disabled version upon the expiry of time budget. For only two subjects, the sequence returned by the enabled version was sub-optimal. For the subject with ID 3, the coverage improvement was 23 instead of 24. For the subject with ID 6, both disabled and enabled versions returned a call sequence with the same coverage, but the disabled versions returned a sequence with one method call shorter. This is likely a negligible cost as the component terminated 1530× faster, and the first fault was detected by the first generated test for all subjects (see RQ1). We conjecture that the lossy pruning could have been caused by the imprecision of the serialization library.

6. RELATED WORK

Concurrency testing. Existing techniques on test automation for concurrent programs focus mostly on improving the effectiveness of concurrency testing by exposing the interleavings that cause concurrency anomalies (e.g., [16, 17, 26, 28, 37, 41, 50, 55, 64]). Instead of generating test code, these techniques explore the possible interleavings arising from the concurrent executions of a given test code. The fault detection effectiveness of these techniques depends on the capability of the given test code to cover faulty interleavings. AutoConTest aims to improve their effectiveness by automatically generating suites of concurrent tests that achieve high interleaving coverage. In fact, our interleaving explorer component can be replaced with any of these techniques.

Like AutoConTest's coverage metric $\mathcal{M}$, the metric HaPSet [64] also captures the execution context of shared memory accesses. However, unlike $\mathcal{M}$, which analyzes single-threaded executions, HaPSet analyzes the interleavings induced by multi-threaded executions. HaPSet was intended to guide interleaving exploration, this metric would not be effective to guide test code generation as one should infer if a concurrent test increase interleaving coverage before the expensive interleaving exploration.

Automated test code generation for sequential programs has been an active research topic, leading to the development of various fully-fledged tools [14, 21, 22, 40, 59]. Since they aim primarily at sequential programs, they do not address the issues of shared memory accesses and interleaving coverage. Efforts have recently been made to extend popular techniques to generate concurrent tests [38, 43, 56]. These efforts can be broadly divided in two categories: *random-based* [38, 43] and *search-based* [56]. Random-based techniques generate tests by randomly assembling method calls into concurrent tests. A major drawback of these techniques is that randomly generated tests are likely to cover interleavings redundantly (see Section 5.2). Instead, AutoConTest generates tests guided by a coverage criterion that helps generate tests that increase interleaving coverage. Search-based techniques generates tests by first estimating coverage requirements and then by guiding the test generation towards these requirements. As discussed in Section 3, collecting the entire set of coverage requirements statically is infeasible for concurrent programs [44]. Instead, AutoConTest collects coverage requirements dynamically during systematic exploration of method call sequences.

Automated test input generation. Symbolic execution [27] analyses a program source code to automatically generate test inputs (data values) to improve code coverage and expose software bugs. Studies were conducted to extend Dynamic Symbolic Execution (DSE) for concurrent programs [15, 35, 45, 51]. However, DSE alone does not generate test code. We believe that AutoConTest's effectiveness could be further improved by incorporating DSE into our call sequence generation component.

Systematic exploration of method sequences. Techniques have been proposed to systematically explore method call sequences for testing sequential programs [11, 36, 62, 65]. Like AutoConTest, these techniques also leverage object state matching to prune the search space [62, 65]. Since it is not their goal to find an optimal sequence, they ignore the coverage during state matching. Ignoring such coverage would miss the optimal sequence as explained in Figure 6.

Synthesis of failing tests. Recently, Samak et al. proposed a series of techniques for constructing test drivers that expose concurrency faults [46, 47, 48, 49]. Unlike AutoConTest, these techniques do not generate test code from scratch. Instead, they require a test suite of sequential tests as an input (i.e., *seeds*). These techniques infer if there exists a particular (concurrent) combination of two seeds that could lead to a deadlock [46, 47, 48] or data race [49]. If so, they are assembled into a concurrent test. The effectiveness of this approach relies on the given seeds. The fault-detection capability of these techniques could be further improved by seeding the sequences generated by AutoConTest.

7. CONCLUSION

In this paper, we presented AutoConTest, an automated coverage-driven approach to generate code for effective concurrent tests that achieve high interleaving coverage and expose concurrency faults quickly. Our experimental results showed that the coverage requirements dynamically obtained via systematic exploration of call sequences are effective in increasing interleaving coverage. Moreover, our experimental results showed that this systematic exploration is tractable with the use of our search space pruning strategies.

In the future, we plan to evaluate AutoConTest on other types of interleaving coverage criteria and to build better pools of the parameters values used to explore the space of call sequences. For instance, using DSE to generate primitive-type values based on a method's control flow.

8. ACKNOWLEDGMENTS

The research was partially funded by Research Grants Council (General Research Fund 611813) of Hong Kong.

9. REFERENCES

[1] ASM Java Bytecode Manipulation Framework. http://asm.objectweb.org/

[2] LANG-481. https://issues.apache.org/jira/browse/LANG-481

[3] Jfreechart-278. http://sourceforge.net/p/jfreechart/bugs/278/

[4] JDK-4334376. https://bugs.openjdk.java.net/browse/JDK-4334376

[5] JDK-4791557. http://bugs.java.com/bugdatabase/view_bug.do?bug_id=4791557

[6] Guava-339. https://code.google.com/p/guava-libraries/issues/detail?id=339

[7] JDK-4779253 http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4779253

[8] ConTeGen. http://thread-safe.org/download

[9] XStream framework. http://xstream.codehaus.org/

[10] S. Artzi, M. D. Ernst, A. Kiezun, C. Pacheco, and J. H. Perkins. Finding the Needles in the Haystack: Generating Legal Test Inputs for Object-oriented Programs. MIT-CSAIL-TR-2006-056, 2006.

[11] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated Testing Based on Java Predicates. In *ISSTA*, pages 123–133, 2002.

[12] S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan. Line-up: A Complete and Automatic Linearizability Checker. In *PLDI*, pages 330–340, 2010.

[13] A. Carzaniga, A. Goffi, A. Gorla, A. Mattavelli, and M. Pezzè. Cross-checking Oracles from Intrinsic Software Redundancy. In *ICSE*, pages 931–942, 2014.

[14] C. Csallner and Y. Smaragdakis. Jcrasher: An Automatic Robustness Tester for Java. *Software Pract and Exper*, 34(11):1025–1050, 2004.

[15] A. Farzan, A. Holzer, N. Razavi, and H. Veith. Con2colic Testing. In *FSE*, pages 37–47, 2013.

[16] C. Flanagan and S. N. Freund. Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs. In *POPL*, pages 256–267, 2004.

[17] C. Flanagan and S. N. Freund. Fasttrack: Efficient and Precise Dynamic Race Detection. In *PLDI*, pages 121–133, 2009.

[18] G. Fraser and A. Arcuri. Whole Test Suite Generation. *IEEE TSE*, 39(2):276–291, 2013.

[19] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg. Does Automated White-box Test Generation Really Help Software Testers? In *ISSTA*, pages 291–301, 2013.

[20] G. Fraser and F. Wotawa. Redundancy Based Test-suite Reduction. In *FASE*, pages 291–305, 2007.

[21] G. Fraser and A. Arcuri. Evosuite: Automatic Test Suite Generation for Object-oriented Software. In *FSE*, pages 416–419, 2011.

[22] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *PLDI*, pages 213–223, 2005.

[23] B. Göetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.

[24] M. Harder, J. Mellen, and M. D. Ernst. Improving Test Suites via Operational Abstraction. In *ICSE*, pages 60–71, 2003.

[25] S. Hong, J. Ahn, S. Park, M. Kim, and M. J. Harrold. Testing Concurrent Programs to Achieve High Synchronization Coverage. In *ISSTA*, pages , 2012.

[26] J. Huang and C. Zhang. Persuasive Prediction of Concurrency Access Anomalies. In *ISSTA*, pages 144–154, 2011.

[27] J. C. King. Symbolic Execution and Program Testing. *CACM*, 19(7):385–394, 1976.

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

[29] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have Things Changed Now?: An Empirical Study of Bug Characteristics in Modern Open Source Software. In *ASID*, pages 25–33, 2006.

[30] Y. Lin and D. Dig. Check-then-act Misuse of Java Concurrent Collections. In *ICST*, pages 164–173, 2013.

[31] P. Liu, J. Dolby, and C. Zhang. Finding Incorrect Compositions of Atomicity. In *FSE*, pages 158–168, 2013.

[32] S. Lu, W. Jiang, and Y. Zhou. A Study of Interleaving Coverage Criteria. In *FSE*, pages 533–536, 2007.

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

[34] S. Lu, S. Park, and Y. Zhou. Finding Atomicity-violation Bugs Through Unserializable Interleaving Testing. *IEEE TSE*, 38(4):844–860, 2012.

[35] R. Majumdar and K. Sen. Hybrid Concolic Testing. In *ICSE*, pages 416–426, 2007.

[36] D. Marinov and S. Khurshid. Testera: A Novel Framework for Automated Testing of Java Programs. In *AS*E, pages 22–31, 2001.

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

[38] 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*, pages 727–737, 2012.

[39] S. Okur and D. Dig. How Do Developers Use Parallel Libraries? In *FSE*, pages 54:1–54:11, 2012.

[40] C. Pacheco, S. Lahiri, M. Ernst, and T. Ball. Feedback-directed Random Test Generation. In *ICSE*, pages 75–84, 2007.

[41] S. Park, S. Lu, and Y. Zhou. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. In *ASPLOS*, pages 25–36, 2009.

[42] F. Pastore, L. Mariani, and G. Fraser. Crowdoracles: Can the Crowd Solve the Oracle Problem? In *ICST*, pages 342–351, 2013.

[43] M. Pradel and T. R. Gross. Fully Automatic and Precise Detection of Thread Safety Violations. In *PLDI*, pages 521–530, 2012.

[44] G. Ramalingam. Context-sensitive Synchronization-sensitive Analysis Is Undecidable. *TOPLAS*, 22(2):416–430, 2000.

[45] N. Razavi, F. Ivančić, V. Kahlon, and A. Gupta. Concurrent Test Generation Using Concolic Multi-trace Analysis. In *APLAS*, pages 239–255, 2012.

[46] M. Samak and M. K. Ramanathan. Multithreaded Test Synthesis for Deadlock Detection. In *OOPSLA*, pages 473–489, 2014.

[47] M. Samak and M. K. Ramanathan. Omen+: A Precise Dynamic Deadlock Detector for Multithreaded Java Libraries. In *FSE*, pages 735–738, 2014.

[48] M. Samak and M. K. Ramanathan. Omen: A Tool for Synthesizing Tests for Deadlock Detection. In *SPLASH*, pages 37–38, 2014.

[49] M. Samak, M. K. Ramanathan, and S. Jagannathan. Synthesizing Racy Tests. In *PLDI*, pages 175–185, 2015.

[50] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[51] K. Sen and G. Agha. Cute and Jcute: Concolic Unit Testing and Explicit Path Model-checking Tools. In *CAV*, pages 419–423, 2006.

[52] K. Sen and G. Agha. A Race-detection and Flipping Algorithm for Automated Testing of Multi-threaded Programs. In *HVC*, pages 166–182, 2006.

[53] K. Sen, G. Roşu, and G. Agha. Detecting Errors in Multithreaded Programs by Generalized Predictive Analysis of Executions. In *FMOODS*, pages 211–226, 2005.

[54] E. Sherman, M. B. Dwyer, and S. Elbaum. Saturation-based Testing of Concurrent Programs. In *FSE*, pages 53–62, 2009.

[55] F. Sorrentino, A. Farzan, and P. Madhusudan. PENELOPE: Weaving Threads to Expose Atomicity violations. In *FSE*, pages 37–46, 2010.

[56] S. Steenbuck and G. Fraser. Generating Unit Tests for Concurrent Classes. In *ICST*, pages 144–153, 2013.

[57] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai. Bug Characteristics in Open Source Software. *ESE*, 19(6):1665–1705, 2014.

[58] S. Tasiran, M. E. Keremouglu, and K. Mucslu. Location Pairs: A Test Coverage Metric for Shared-memory Concurrent Programs. *ESE*, 17(3):129–165, 2012.

[59] N. Tillmann and J. De Halleux. Pex–white Box Test Generation for Net. In *TAP*, pages 134–153, 2008.

[60] P. Tonella. Evolutionary Testing of Classes. In *ISSTA*, pages 119–128, 2004.

[61] M. Vaziri, F. Tip, and J. Dolby. Associating Synchronization Constraints with Data in an Object-oriented Language. In *POPL*, pages 334–345, 2006.

[62] W. Visser, C. S. Păsăreanu, and R. Pelánek. Test Input Generation for Java Containers Using State Matching. In *ISSTA*, pages 37–48, 2006.

[63] C. Wang and M. Ganai. Predicting Concurrency Failures in the Generalized Execution Traces of x86 Executables. In *RV*, pages 4–18, 2012.

[64] C. Wang, M. Said, and A. Gupta. Coverage Guided Systematic Concurrency Testing. In *ICSE*, pages 221–230, 2011.

[65] T. Xie, D. Marinov, and D. Notkin. Rostra: A Framework for Detecting Redundant Object-oriented Unit Tests. In *ASE*, pages 196–205, 2004.

[66] C.-S. D. Yang, A. L. Souter, and L. L. Pollock. All-du-path Coverage for Parallel Programs. In *ISSTA*, pages 153-162, 1998.

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

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