Effectiveness and Challenges in Generating Concurrent Tests for Thread-Safe Classes

Valerio Terragni[†] and Mauro Pezzè[†*]

[†]USI Università della Svizzera italiana, Lugano, Switzerland
[*]University of Milano-Bicocca, Milan, Italy
{valerio.terragni, mauro.pezze}@usi.ch

ABSTRACT

Developing correct and efficient concurrent programs is difficult and error-prone, due to the complexity of thread synchronization. Often, developers alleviate such problem by relying on thread-safe classes, which encapsulate most synchronization-related challenges. Thus, testing such classes is crucial to ensure the reliability of the concurrency aspects of programs. Some recent techniques and corresponding tools tackle the problem of testing thread-safe classes by automatically generating concurrent tests. In this paper, we present a comprehensive study of the state-of-the-art techniques and an independent empirical evaluation of the publicly available tools. We conducted the study by executing all tools on the JaConTeBe benchmark that contains 47 well-documented concurrency faults. Our results show that 8 out of 47 faults (17%) were detected by at least one tool. By studying the issues of the tools and the generated tests, we derive insights to guide future research on improving the effectiveness of automated concurrent test generation.

CCS CONCEPTS

• **Software and its engineering → Concurrent programming languages**; **Software testing and debugging**;

KEYWORDS

Test generation, Concurrency faults, Thread-safety

ACM Reference Format:
Terragni Valerio and Mauro Pezzè. 2018. Effectiveness and Challenges in Generating Concurrent Tests for Thread-Safe Classes. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3238147.3238224

1 INTRODUCTION

Concurrent programming is pervasive across application domains due to the widespread of multi-core chip technology. Developing correct and efficient concurrent programs is hard due to the complexity of thread synchronization that suffers from under- and over-synchronization problems. Under-synchronization introduces subtle concurrency faults, like data races and atomicity violations, that are difficult to expose at testing time since they manifest under specific non-deterministic thread interleavings. Over-synchronization causes deadlocks and affects performance by introducing overhead and reducing parallelism [36, 42].

Often developers reduce the complexity of developing reliable concurrent programs in object-oriented shared-memory languages, for instance `Java` and `C++`, by relying on *thread-safe* classes [22], which address the important challenge of synchronizing concurrent memory accesses in a correct and efficient way [42]. By delegating the burden of thread synchronization to thread-safe classes, developers can use the same instance of such classes from multiple threads without additional synchronization [22], thus relying on the correctness of *thread-safe* classes to avoid concurrency failures [36]. Ensuring the correctness of thread-safe classes is important. It identifies concurrency faults in the implementation of the thread-safe classes, and thus in the programs that rely on them.

An effective approach to validate the correctness of thread-safe classes consists in automatically generating concurrent unit tests, and checking if the generated tests trigger fault-revealing thread interleavings, by relying on one of the many interleaving exploration techniques [10, 17, 51, 60]. A concurrent unit test, *concurrent test* hereafter, consists of multiple concurrently executing threads that exercise the public interface of a class under test. Figure 2 shows an example of concurrent test that triggers a concurrency fault in the thread-safe class `AppenderAttachableImpl` of the `Log4J` library, shown in Figure 1.

In recent years, researches have proposed techniques for automatically generating concurrent tests [12, 35, 40, 45, 48–50, 57, 59], inspired by recent advances in sequential unit test generation [20, 37]. Generating concurrent tests faces new challenges that are not present when generating sequential tests, like multi-threading, non-determinism, shared states, and huge interleaving spaces [12, 59].

Current generators of concurrent tests address the new challenges with different approaches, for which there is no clear evidence of their effectiveness and limitations yet. Although, the authors of these approaches have performed experiments to evaluate and compare them [12, 59], such experiments are too narrow in the criteria to select subjects and also in the analysis of the results to provide solid evidence of effectiveness and limitations. To address this gap, this paper reports an empirical study of the six currently openly-accessible concurrent test generators for thread-safe classes. The goal of the study is to (i) assess the effectiveness of the techniques in generating fault-revealing tests, (ii) identify and better

```
1  /** faulty AppenderAttachableImpl class - Log4j v. 1.2.17
2  https://bz.apache.org/bugzilla/show_bug.cgi?id=54325 **/
3    public void removeAllAppenders() {
4  
5      if(appenderList != null) {
6        int len = appenderList.size();
7        for(int i = 0; i < len; i++) {
8          Appender a = (Appender) appenderList.elementAt(i);
9          a.close();
10       }
11       appenderList.removeAllElements();
12       appenderList = null;
13     }
14   }
```

Figure 1: Code related to the Log4j concurrency bug 54325.

```
 // Sequential Prefix
AppenderAttachableImpl var0 = new AppenderAttachableImpl();
ConsoleAppender var1 = new ConsoleAppender();
var0.addAppender((Appender) var1);

    new Thread(new Runnable() {
      public void run() {
             var0.removeAllAppenders();         // Suffix 1
      }}).start();

    new Thread(new Runnable() {
      public void run() {
             var0.removeAllAppenders();         // Suffix 2
      }}).start();
```

Figure 2: Concurrent test that exposes the bug in Figure 1.

understand their limitations, and (iii) shed light on future research directions identifying ways in which existing techniques can be improved or new techniques be defined. In our study, we refer to the recent JaConTeBe benchmark of concurrency faults in thread-safe classes, published in ASE 2015, a repository of 47 concurrency faults experienced in the field [31]. We would like to notice that we selected the benchmark independently from the studied tools, to avoid biases in selecting the subjects. Our results indicate that only 17% of the faults can be detected by at least one of the generators of concurrent tests evaluated in this paper. This indicates both an impressive effectiveness and a large space for improvements. We analyzed the results of both the automatically-generated and the JaConTeBe test suites, and examined the characteristics of both detected and undetected faults to study the effectiveness and limitations of the concurrent test generators. We identified three main issues that prevent them from generating fault-revealing tests, and we observed that the issues are shared among the different techniques. We devise future research directions to address these problems and increase the overall effectiveness of concurrent test generation. To ease reproducibility, our experimental data are available in our website [1].

In summary, this paper makes the following contributions:

- A survey on the existing techniques for generating concurrent unit tests for thread-safe classes;
- A large-scale experimental and comparative evaluation of the six generators of concurrent tests for Java thread-safe classes conducted on the JaConTeBe benchmark [31], which includes 47 concurrency faults;
- An analysis of the effectiveness of these six techniques in detecting concurrency faults;
- A discussion of a set of insights that we gained from the study, and that give some guidelines for future research in this area;

2 GENERATING CONCURRENT TESTS

This section presents the preliminaries and background of test generation for exposing concurrency faults in thread-safe classes, in the context of concurrent object-oriented programs that implement the shared-memory programming paradigm.

A concurrent shared-memory object-oriented program is composed of a set of classes, each composed of a set of methods and fields that can be executed and accessed concurrently by multiple threads, respectively. A class is *thread-safe* if it encapsulates synchronization mechanisms that prevent incorrect accesses to the class from multiple threads [22]. A class is thread-unsafe otherwise. Example of synchronization mechanisms are synchronized blocks in `Java`, and locks, mutexes and semaphores in `C`. Thread-safety guarantees that the same instance of a thread-safe class can be correctly accessed by multiple threads without additional synchronization other than the one implemented in the class [40].

Writing correct, efficient and reliable thread-safe classes is hard, due to the non-deterministic order of memory accesses across threads, which can lead to *thread-safety violations*, that is, deviations from the expected behavior of the class when concurrently accessed. A key characteristic of such violations is that they manifest non-deterministically, due to the non-determinism of the scheduler that determines the execution order of threads. The order of accesses to shared memory locations is fixed within one thread, but can vary across threads. An *interleaving* is a total order relation of shared memory accesses among threads [32]. Concurrent executions can manifest many different interleavings, and only some –usually few– of them expose thread-safety violations [38].

As an example of thread-safety violation, Figure 1 shows the code snippet of class `AppenderAttachableImpl` of the `Log4j` library, part of the JaConTeBe benchmark [31]. Method `removeAllAppenders` checks whether the object instance field `appenderList` is initialized (line 5) before dereferencing it (line 6), and setting the reference to `null` (line 12). These accesses are not properly synchronized: two threads that concurrently invoke `removeAllAppenders` may cause a `NullPointerException` both in a specific program state and with a particular thread interleaving (line 6). In a thread interleaving that triggers the exception, a thread t_1 executes line 12 after a thread t_2 executed line 5 and before t_2 executes line 6.

An effective approach for validating the correctness of thread-safe classes is the automated generation of concurrent tests. Figure 3 shows a logical architecture of such approach, which is shared among all the surveyed techniques.

In a nutshell, given a class under test and, optionally, a set of auxiliary classes that the class under test depends on, the techniques automatically exposes thread-safety violations in four consecutive steps: (i) they generate sequential (single-threaded) method call sequences that exercise the public interface of the class under test, (ii) assemble such sequences in a concurrent test that executes concurrently the method call sequences from multiple threads, (iii) explore the interleaving space of the generated concurrent tests by means of state-of-the-art interleaving explorers, (iv) check if any of the explored interleavings exposes a thread-safety violation. Figure 2 shows an example of a concurrent test that can reveal the fault-revealing interleaving described above. In this paper, we refer to the general definition of concurrent test presented in the seminal paper of Pradel and Gross [40].

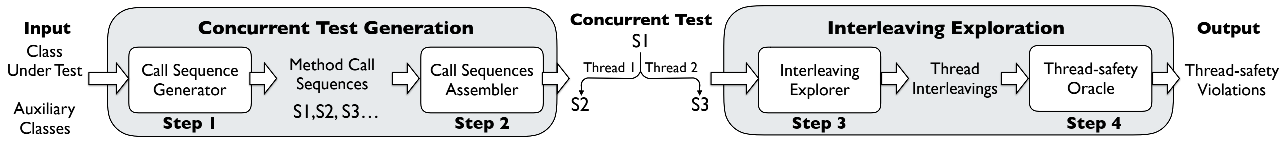

Figure 3: Logical architecture of concurrent test generation.

A *concurrent test* is a set of method call sequences, where each call in a sequence consists of a method signature, a possibly empty list of input variables (method parameters), and an optional output variable (method return value). In an instance method call, the first input variable is the object that receives the call, for example, `var0` in the call at line 8 Figure 2. A test consists of a prefix and a set of suffixes. A *prefix* is a call sequence to be executed in a single thread that instantiates the class under test, to create the object instances that will be accessed concurrently from multiple threads. A prefix may need to invoke additional methods to bring an object instance into a particular state that may enable the suffixes to trigger a thread-safety violation. For instance, the method call `var0.addAppender((Appender) var1);` at line 4 in Figure 2 instantiates the shared object field `appenderList`, in order to satisfy the condition of the if statement at line 5 in Figure 1, during the execution of the concurrent suffixes, to trigger a `NullPointerException` when executing the statement at line 6. A *suffix* is a call sequence to be executed concurrently with other suffixes, after executing the common prefix. All suffixes share the object instances created by the prefix, and can use them as input variables for suffix calls, to invoke methods that access the shared instances concurrently. For example, all the suffixes in Figure 2 use the same shared object instance `var0` of the class under test as an object receiver.

3 STATE-OF-THE-ART GENERATORS OF CONCURRENT TESTS

For a complete survey of the main techniques to generate concurrent tests, we identified a list of relevant papers, by querying scholarly web engines (Google Scholar, ACM and IEEE Digital Libraries) with the query: `test generation + concurrency`, and refined the list, by ignoring the papers unrelated to concurrent test generation and concurrency fault detection. We discarded papers that present techniques for generating sequential tests [20, 37, 62], for generating concurrent tests to expose performance issues [42], incorrect substitutability faults [41], or for reproducing concurrency failures from crash stacks [7], which are outside the focus of this study. We discarded techniques presented only in short workshop papers [52, 53]. We excluded techniques that generate test inputs for concurrent programs, notably techniques that use concolic execution or symbolic analysis [16, 24, 44], since these techniques generates only inputs and not concurrent tests, as the one shown in Figure 2.

Table 1 summarizes the nine relevant techniques that we identified in our study. The table indicates the name of the tool, the main reference, the venue and year of publication, and the category of the technique, according to the taxonomy of Choudhary et al.', who classify concurrent test generators as *random-based, coverage-based* and *sequential-test-based* [12].

Table 1: State-of-the-art Generators of Concurrent Tests

Tool name	Reference	Venue	Year	Category
BALLERINA	[35]	ICSE	2012	random-based
CONTEGE	[40]	PLDI	2012	
CONSUITE	[57]	ICST	2013	coverage-based
AUTOCONTEST	[59]	ICSE	2016	
COVCON	[12]	ICSE	2017	
OMEN	[45–47]	FSE	2014	sequential-test based
NARADA	[49]	PLDI	2015	
INTRUDER	[48]	FSE	2015	
MINION	[50]	OOPSLA	2016	

In Sections 3.1, 3.2 and 3.3, we overview the test generation techniques, grouped per category, corresponding to Steps 1 and 2 in Figure 3. In Section 3.4, we discuss the interleaving explorers and thread-safety oracles used by the different techniques, corresponding to Step 3 and 4 in Figure 3.

3.1 Random-Based Techniques

The pioneer random-based techniques are BALLERINA [35] and CONTEGE [40], which generate concurrent tests by randomly combining randomly generated method call sequences with random input parameters. Both BALLERINA and CONTEGE rely on existing interleaving explorers and use linearizability [26] as test oracle. *Linearizability* reports a violation whenever a thread interleaving produces a behavior that cannot be produced in any linearized test execution where all methods are execute atomically (sequentially).

BALLERINA ICSE 2012 [35]. Nistor et al. identify as a major challenge of concurrent test generation the high computational cost of exploring the interleaving spaces of the generated tests, which size grows factorially with the number of shared memory accesses executed by the concurrent suffixes [32]. BALLERINA addresses this challenge by confining each concurrent suffix to a single method call, limiting the test to a single shared object under test, and generating tests with exactly two concurrent threads. BALLERINA clusters oracle violations to reduce the cost of inspecting them: oracle violations belonging to the same cluster are likely to be either all false alarms or all true errors [35].

CONTEGE PLDI 2012 [40]. CONTEGE addresses the same challenges of BALLERINA, with a novel linearizability checker that improves efficiency and reduces false alarms over BALLERINA [9].

Pros: Random-based techniques can efficiently generate concurrent tests, since they do not require complex analysis; they can generate thousands of concurrent tests in few seconds [12], and can effectively expose easy-to-find concurrency faults [35], whose manifestation does not depend on a particular program state.

Cons: Random-based techniques are less effective in revealing hard-to-find concurrency faults, because randomly generated tests tend to repetitively test similar program behaviors [12]. We need to randomly generate thousands or even millions of concurrent tests to effectively detect hard-to-find faults, due to the low probability of randomly generating a failure-inducing test [40]. For example, ConTeGe requires more than a million tests to expose a single concurrency fault [40]. This is an issue, because of the high computational cost of exploring the interleaving space of all generated tests. In practice, we can explore only the interleaving space of few tests within an affordable time budget [12, 35, 59].

3.2 Coverage-Based Techniques

Coverage-based techniques address the limitations of random-based techniques by driving the generation of concurrent tests with interleaving coverage criteria [12, 57, 59]. These techniques identify and prune concurrent tests that lead to redundant behaviors (thread interleavings) to steer test generation towards new program behaviors, thus avoiding the high cost of exploring the interleaving spaces of redundant tests. A major challenge faced by coverage-based techniques is the high cost of computing the precise executable domain of interleaving coverage criteria for concurrent programs [43, 55, 59]. To address this challenge coverage-based techniques rely on coverage criteria that are efficient to compute, at the cost of approximating the coverage of thread interleavings.

ConSuite ICST 2013 [57]. The seminal coverage-based concurrent test generation approach, ConSuite, extends EvoSuite [20] to generate concurrent tests. ConSuite statically estimates the coverage requirements as a set of thread interleavings, and selects candidate concurrent tests for covering uncovered interleavings, by checking if the sequential executions of the suffixes of the tests cover the memory accesses that comprise the target interleaving. ConSuite does not consider the execution context of shared memory accesses, and thus ignores concurrency faults that manifest failure-inducing interleavings only under specific execution contexts. Moreover, even if the concurrent suffixes cover the target shared-memory accesses when executed sequentially, there is no guarantee that the suffixes execute the accesses in the specific order prescribed by the interleaving coverage target, when executed concurrently.

AutoConTest ICSE 2016 [59]. AutoConTest improves over ConSuite by introducing a context-sensitivity coverage metric that can be efficiently computed, as it analyses sequential executions, and that includes synchronization sensitive (lock acquisitions and releases) and calling context information. AutoConTest overcomes the limitations of computing an approximated set of coverage requirements statically (prior testing), by generating concurrent tests iteratively, so that each test increases the coverage based on the coverage data that are collected during the test generation.

CovCon ICSE 2017 [12]. CovCon exploits the concept of concurrent method pairs, proposed by Deng et al. [14], that is, the set of pairs of methods that execute concurrently [14]. CovCon measures the frequency of concurrent executions of pairs of methods, to focus the test generation on infrequently or not at all covered pairs [12].

Pros: Coverage-based techniques limit redundant tests, thus reducing the number of generated tests, and consequently the interleaving exploration costs, as in principle only the tests that increase interleaving coverage will be analyzed by the (computationally expensive) interleaving explorer.

Cons: The effectiveness of coverage-based techniques depends on the coverage criterion, which might either be too expensive to compute or it might miss relevant coverage requirements. The optimal trade-off between analysis computational cost and precision of the computed requirements is still unclear. For example, the ConSuite coverage criterion likely misses coverage requirements because of being both statically computed and specific to predefined faulty interleaving patterns. AutoConTest efficiently learns coverage requirements while generating method call sequences, but can miss coverage requirements if specific input parameters are needed to observe new coverage requirements. The CovCon coverage criterion is computed efficiently at method level, but does not capture the many interleaving coverage requirements that involve the shared memory accesses triggered by the methods.

3.3 Sequential-Test-Based Techniques

The sequential-test-based techniques proposed by Samak et al. apply the same overall approach to different kinds of concurrency faults [46, 48–50]: Omen targets deadlocks, Narada data races, Intruder atomicity violations, and Minion assertion violations. They analyse concurrent programs starting from a suite of sequential (single-threaded) tests, which can be either manually-written or generated by existing sequential test generators [20, 37]. They analyze the execution traces obtained by executing the initial test suites sequentially, to identify concurrency faults that may occur when combining multiple sequential tests into concurrent tests. If such faults are identified, these techniques generate concurrent tests, which they analyze with interleaving explorers to check if they indeed expose the fault during a concurrent execution.

Pros: Sequential-test-based techniques do not generate concurrent tests that are irrelevant with respect to the considered type of concurrency fault.

Cons: Their effectiveness depends on the initial set of sequential tests. The hypothesis that sequential tests executed concurrently are always adequate to expose concurrency faults is not always valid. Sequential tests do not refer to the concurrency structure: manually written sequential tests are designed without considering concurrency issues, while automatically generated sequential tests are produced referring to sequential-based coverage criteria, for example, branch coverage [20]. Moreover, each sequential-test-based techniques imposes a relatively high computational cost [12].

3.4 Interleaving Explorers and Thread-Safety Oracles

Generators of concurrent tests check if the generated tests expose thread-safety violations (Step 3 and 4 in Figure 3) by adopting different interleaving explorers and thread-safety oracles, which come with different computational costs, present diverse proneness to false positives, and target different types of faults (data races, atomicity violations) and related failure types (exceptions, endless hang, and logical issues).

Table 2 summarizes the main interleaving explorers: selective, random and exhaustive techniques. *Selective* techniques limit the interleaving space exploration of the generated tests to interleavings that match predefined patterns of concurrency faults, like data

Table 2: Interleaving Explorers and Thread-safety Oracles

Interleaving Explorer	Implicit Oracle (exceptions, endless hang)	Internal Oracle (data races, atomicity violations)
Random	ConTeGe, CovCon	Narada
Selective	Omen	AutoConTest, Intruder
Exhaustive	Ballerina	

races and atomicity violations [30, 38]. *Random* techniques randomly explore the space of possible thread interleavings, while *exhaustive* techniques exhaustively explore all non-redundant interleavings [25, 34, 63]. Redundant interleavings are often identified with partial order reduction techniques [18]. Both random and systematic techniques do not restrict the interleavings to be explored, and therefore are not specific to any particular kind of concurrency faults, thus differing from selective techniques. Both random and exhaustive techniques hardly scale to concurrent tests with huge interleaving spaces, since exhaustive techniques are computationally expensive, and random techniques have a low probability of detecting concurrency faults [38]. In contrast, selective techniques are more efficient, because they focus on a small portion of the interleaving space [38]. ConTeGe, CovCon and Narada rely on random interleaving explorers. Ballerina relies on two different exhaustive interleaving explorers. AutoConTest, Omen and Intruder rely on selective techniques, AssetFuzzer [30], iGoodLock [28], and CTrigger [38], respectively.

Thread safety oracles are either implicit or internal. *Implicit oracles* report a concurrency fault if an explored thread-interleaving manifests an "obvious" and visible oracle violation like runtime exceptions or endless hangs [4]. Implicit oracles cannot detect faults that manifest as logical errors (wrong output). *Internal oracles* detect faults by monitoring the internal program states. For example, they report an oracle violation if an explored thread interleaving matches a predefined pattern of concurrency faults [67], regardless of observing a runtime exception or an endless hang. Thus, internal oracles can both fail to detect the presence of faults (false negatives), and signal the presence of anomalies that are not faults (false positives) [67]. ConTeGe, CovCon, Omen and Ballerina rely on implicit oracles, while the other techniques on internal oracles.

Generating tests and exploring interleaving are two orthogonal problems. As such, in principle, the tests generated whit any test generator could be analyzed by any interleaving explorer [12, 35, 40]. In practice, this is not always the case. For example, AutoConTest generates concurrent tests with tens of method calls in the concurrent suffixes. This may be incompatible with both exhaustive and random interleaving explorer, since exhaustive exploration may become too expensive, while random exploration may be ineffective as the probability of triggering faulty-interleavings decreases with the increase of test length [38].

4 EXPERIMENTS

In this section, we describe the experiments that we designed to evaluate and compare the six openly-accessible state-of-the-art concurrent test generators that we selected as a result of our analysis of the literature. The goal of the experiments is to evaluate and compare how effectively and efficiently these generators find concurrency faults.

Table 3: JaConTeBe Benchmark [31]

Code base (label)	# Subjects	Description
Apache DBCP (dbcp)	4	Database connection pool
Apache Derby (derby)	5	Relational database
Apache Groovy (groovy)	6	Dynamic language for JVM
OpenJDK (jdk)	20	Java Development Kit
Apache Log4j (log4j)	5	Logging library
Apache Lucene (lucene)	2	Search library
Apache Pool (pool)	5	Object-pooling API

4.1 Tool Selection

We experimented with all state-of-the-art techniques for which there exists a publicly available tool: ConTeGe, CovCon, AutoConTest, Omen, Narada and Intruder. We were not able to experiment with Ballerina, ConSuite and Minion because the tools were not publicly available at the time of conducting the experiments.[1] The empirical comparison of the tools is facilitated by the fact that all tools generate tests for programs written in Java.

We experimented ConTeGe and CovCon both with the default configuration that uses stress testing as interleaving explorer and with the configuration that uses the *exhaustive* interleaving explorer JPF [25]. Such configuration is supported by both tools, and we indicate the variants of ConTeGe and CovCon that rely on JPF as ConTeGeJPF and CovConJPF, respectively.

4.2 Subject Selection

The criteria for selecting subjects play a crucial role in evaluating and comparing testing techniques [5, 21, 65]. Most of the experiments reported in the surveyed papers refer to previously published papers as a basis for selecting subjects. In their seminal papers, both Pradel and Nistor propose an interesting set of experiments, but do not discuss the criteria for selecting the experimental subjects [35, 40]. In the context of sequential test generation, Fraser and Arcuri argue that "*if the subject selection is unclear, in principle it could mean that the presented set of classes represents the entire set of classes on which the particular tool was ever tried, but it could also mean it is a subset on which the technique performs particularly well*", and suggest to randomly sample open-source code repositories in order to avoid biases [21]. Fraser and Arcuri's approach is suitable to evaluate test generator techniques with respect to coverage, since this does not require that the selected subjects contain concurrency faults. The approach is not well suited for evaluating test generation techniques that aim to expose concurrency faults quickly, without aiming to maximise coverage, like random-based and sequential-test-based techniques. Our experiments aims to evaluate the different techniques referring to an unbiased benchmark, that is a benchmark of thread-safe classes with documented concurrency faults selected without neither concurrent test generation nor some specific techniques in mind. A benchmark that satisfies this requirement is the JaConTeBe benchmark, recently proposed by Lin et al. [31]. JaConTeBe is composed of 47 concurrency faults from seven Java popular open source projects (see Table 3), and considers a wide range of concurrency faults types: data races, atomicity violations, resource and `wait-notify` deadlocks. The readers can find detailed description of the subjects in the original paper [31] and

[1]The Minion website (https://sites.google.com/site/miniontool/) is under construction at the time of conducting the experiments

companion website [2], where the authors claim that JaConTeBe is the most advisable benchmark for evaluating concurrency testing techniques. Competing benchmarks are either composed of toy concurrent programs (IBM benchmark suite [15]) or not specifically designed for concurrency fault detection (JPG [56] and DaCapo [8]), thus without documented concurrency faults.

For each of the 47 concurrency faults (subjects), the JaConTeBe benchmark provides the following artifacts: (i) the binaries of the code base's buggy version; (ii) the link to the bug report; (iii) the source code of a manually-written fault-revealing concurrent test; JaConTeBe was built specifically to evaluate interleaving exploration techniques (Step 3 and 4 in Figure 3), and thus it includes fault-revealing concurrent tests. These characteristics make JaConTeBe an excellent benchmark for our study, since our goal is to assess the capability of test generators to synthesize concurrent tests (Step 1 and 2 in Figure 3) that can expose the concurrency faults that can be revealed with manually-written tests.

4.3 Subject and Tool Preparation

For each subject, we collected the inputs of the tools: the class under test and the set of auxiliary classes (see Figure 3). We retrieved the class under test by examining the bug report and the manually-written test. As discussed in Section 2, the *auxiliary classes* are those in which the class under test depends upon. For example, if a method `m` of the class under test has a parameter of non-primitive type `A`, then `A` is an auxiliary class. Most tools require auxiliary classes as input to create objects of type `A` to be used as a parameter for invoking `m`. We identified a proper set of auxiliary classes by relying on the corresponding manually-written tests of JaConTeBe. We set the auxiliary classes as all the classes of the program under test (excluding the class under test itself) that have been referenced to in the manually-written test.

While most of the evaluated tools accept binaries as an input, CovCon needs the source code, because it instruments the source code with an eclipse plug-in[2]. Therefore, we retrieved the relevant source code from the corresponding code repositories, and we confirmed that the JaConTeBe tests fail if executed on the source code. We instrumented the source code of the class under test and all of its superclasses[3].

The sequential-test-based tools, Omen, Narada and Intruder, require a set of sequential tests in input as they do not perform Step 1 of Figure 3. We could obtain such tests by extracting human-written sequential tests from public repositories, but it would be unfair with respect to coverage-based and random-based techniques that generate concurrent tests relying exclusively on automatically generated codes. To avoid biases, we generated the tests with a random-based generator of sequential tests, following the experiments of Choudhary et al. [12]. We opted for Randoop [37], the most popular tool of this type[4]. An important configuration choice is the number of sequential tests for the three sequential-test based tools. Too many tests could introduce overhead during the analysis phase, while too few tests could be insufficient to find the fault. We addressed this issue by iteratively alternating the execution of Randoop and of the sequential-test based tools. At each iteration we doubled the time budget t and the maximum number of tests k of Randoop. We chose an initial value of $t = 30$ *seconds* and $k = 50$, as a recent empirical study indicates that Randoop saturates code coverage in less than 60 seconds [54].

[2]https://github.com/michaelpradel/ConTeGe/tree/CovCon
[3]Excluding `java.lang.Object`.
[4]We ran Randoop ignoring all tests that do not instantiate the CUT at least once by including the option `--include-if-classname-appears`

4.4 Evaluation Setup

We executed the eight tools (six techniques and two alternative configurations) with a time-budget of an hour for each of the 47 JaConTeBe subjects. We repeated the experiments ten times to cope with the randomness of the choices of the tools while generating tests and exploring interleaving spaces. The overall machine time of the experiments was 156 days. We executed our experiment on a server Ubuntu 16.04.2 with 64 octa-core CPUs Intel® Xeon® E5-4650L @ 2.60 GHz and ~529 GB of RAM.

We evaluated the effectiveness of the tools in terms of bug finding capability, and their efficiency in terms of the time required to detect the faults, which includes the time of both generating the tests and exploring their interleaving spaces (Step 1 to 4 in Figure 3). We did not use coverage as an evaluation criterion, since in the context of concurrency testing the only pertinent coverage criteria are those related to thread interleavings [32]. Computing the complete set of dynamically covered interleavings for all the generated tests would be too computationally expensive. Furthermore, instrumenting shared memory accesses to collect the covered interleavings would introduce delays that could prevent the manifestation of the fault-revealing interleavings. We do not compare the number of tests generated by each technique, because it can be misleading, since the length of method call sequences in a concurrent test can vary a lot across techniques. For example, AutoConTest generates long suffixes, while ConTeGe short ones. Moreover, sequential-test-based tools do not report all the concurrent tests that they generate, but only those that they deem as fault-revealing.

We manually analyzed the results of each run, to identify unsuccessful and successful runs. We checked the relations of all thread-safety violations reported by the tools with the concurrency faults under analysis, by relying on both the bug report and the manually-written concurrent test. It is important to clarify that we do not require the automated-generated concurrent tests and the corresponding manually-written tests to be identical, since often the same concurrency fault can be manifested with different concurrent tests.

4.5 Results

Table 4 summarises the results of our experiments by providing fault-finding and execution time data for all subjects. The first and third columns present the category and the ID of each fault, respectively, as defined by the curators of the JaConTeBe benchmark [31]. The second column indicates the type of failure that manifests the fault when executing the manually-written test.

The subjects are ordered by the first and second columns. The remaining columns show the results for each tool by indicating four possible outcomes: (i) the tool detects the fault (✓) with the number of successful runs that detected the fault, and the average detection time of successful runs; (ii) the tool crashes (×); (iii) the tool does not generate any test (●); (iv) the tool generates tests but

Table 4: Summary of Fault-finding Results for Each Tool and JaConTeBe Fault

Fault Category	Failure Type	Fault ID	ConTeGe	ConTeGeJPF	AutoConTest	CovCon	CovConJPF	Omen	Narada	Intruder
inconsistent synch.	endless loop	dbcp4	–	–	×	–	–	–	–	–
	logic	pool5	–	–	×	–	–	–	–	–
race/atomicity violations	endless loop	**groovy6**	✓(1/10) 231 sec.	–	×	–	–	–	–	–
	runtime exception	dbcp3	–	–	×	–	–	●	●	●
		derby3	●	●	●	●	●	●	●	●
		groovy1	●	●	×	●	●	–	–	–
		groovy3	–	–	●	–	–	–	–	–
		groovy4	–	–	●	–	–	–	×	×
		jdk6_3	–	–	✓(10/10) 58 sec	✓(10/10) 54 sec	✓(7/10) 312 sec	–	–	✓(10/10) 307 sec
		jdk6_4	✓(8/10) 710 sec	✓(10/10) 59 sec	×	✓(10/10) 75 sec	✓(2/10) 300 sec	–	×	×
		jdk6_13	✓(1/10) 2,617 sec	✓(3/10) 1,092 sec	×	✓(6/10) 916 sec	✓(3/10) 1,549 sec	–	×	×
		jdk7_3	–	–	–	–	–	–	–	–
		log4j_3	✓(10/10) 22 sec	✓(10/10) 15 sec	✓(10/10) 37 sec	✓ (10/10) 58 sec	✓ (10/10) 22 sec	–	–	–
		pool1	–	–	×	–	–	–	–	–
	logic	groovy5	–	–	–	–	–	–	×	×
		jdk6_1	–	–	●	–	–	●	●	●
		jdk6_2	–	–	–	–	–	–	–	–
		jdk6_5	–	–	–	–	–	–	–	–
		jdk6_14	–	–	–	–	–	–	–	–
		jdk7_1	–	–	–	–	–	–	–	–
		jdk7_6	–	–	–	–	–	–	–	–
		log4j_1	–	–	×	–	–	–	✓ (10/10) 35 sec.	–
resource deadlock	endless hang	dbcp1	–	–	●	–	–	●	●	●
		dbcp2	–	–	●	–	–	●	●	●
		derby1	●	●	●	●	●	●	●	●
		derby2	●	●	●	●	●	●	●	●
		derby4	●	●	●	●	●	●	●	●
		derby5	–	–	–	–	–	–	–	–
		groovy2	–	–	–	–	–	×	×	×
		jdk6_6	–	–	–	–	–	–	–	–
		jdk6_7	–	–	●	✓ (3/10) 17 sec.	–	●	●	●
		jdk6_8	–	–	–	–	–	–	–	–
		jdk6_10	✓ (2/10) 321 sec.	–	●	✓ (5/10) 69 sec.	–	●	●	●
		jdk6_11	–	–	×	–	–	–	–	–
		jdk6_12	–	–	–	–	–	–	–	–
		jdk7_2	–	–	–	–	–	–	–	–
		jdk7_4	–	–	–	–	–	–	–	–
		log4j_2	–	–	–	–	–	–	–	–
wait-notify deadlock	endless hang	jdk6_9	–	–	–	–	–	–	–	–
		jdk7_5	–	–	–	–	–	–	–	–
		log4j_4	–	–	–	–	–	–	–	–
		log4j_5	–	–	–	–	–	–	–	–
		lucene1	–	–	●	–	–	–	–	–
		lucene2	–	–	●	–	–	–	–	–
		pool2	–	–	–	–	–	–	–	–
		pool3	–	–	–	–	–	–	–	–
		pool4	–	–	–	–	–	–	–	–
summary (# faults detected, average time)			(5, 780 sec)	(3, 388 sec)	(2, 48 sec)	(6, 198 sec)	(4, 546 sec)	(0, –)	(1, 35 sec)	(1, 307 sec)

does not detect the fault (–); The last row in Table 4 summarize the results for each tool as number of detected faults and average detection time.

As discussed in Section 3.4, the surveyed techniques rely on different interleaving explorers and thread-safety oracles to detect faults of specific categories and failure types. In Table 4, a gray background indicates the faults that are not detectable with some tools by either the interleaving explorer or the thread-safety oracle adopted by the tool. CovCon, ConTeGe, CovConJPF and ConTeGeJPF use interleaving explorers that are not limited to any fault category, thus they can potentially detect all faults in JaConTeBe, but rely on implicit oracles that can detect only faults that manifest exceptions or endless hangs as type of failure. Therefore, these tools could detect at most ~80% of the JaConTeBe faults. Omen can detect only "resource deadlocks" faults and "endless hang" failures, which amount of 34% of the JaConTeBe faults. AutoConTest, Narada and Intruder rely on internal oracles, which do not impose any restrictions on the failure type, but rely on interleaving explorers and oracles that are limited to "race/atomicity" faults,which amount of ~43% of the JaConTeBe faults. We executed all tools on all the subjects (even the combinations with the gray background), since some concurrency faults could manifest with failure types different from the ones specified in the JaConTeBe benchmark.

Effectiveness. ConTeGe detects five faults that belong to two different categories, and that lead to failures of three types. It detects `groovy6`, which is not detected by any other tools. ConTeGeJPF detects three of the faults detected with ConTeGe. AutoConTest detects two atomicity violations. CovCon detects six faults of two categories and two failure types, detecting more faults than any other tool. CovConJPF detects four of the faults detected by CovCon. Omen does not detect any fault. Narada detects one fault `log4j_1`, which is not detected by any other tool. Intruder detects the fault `log6_3`, which is also detected by three other tools. AutoConTest, Narada and Intruder detect the faults in all the ten runs, showing stability with respect to non-determinism comparing with the other tools, which have a lower fault detection rate. It is worth mentioning that, for some faults, AutoConTest crashes due to incompatibility issues with the instrumentation framework used to compute the coverage. For subjects `dbcp1`, `dbcp2`, `derby1`, `derby2` and `derby4`, most of the tools (including Randoop) fail to generate any tests that successfully instantiate the class under test, which requires a complex method call sequence.

Efficiency. The overall detection time varies from 15 seconds for subject `log4j_3` to 2,617 seconds for `jdk6_13`. CovCon is the fastest among the four tools that detected at least three faults, with an average detection time of 198 seconds (see last row in Table 4), followed by ConTeGeJPF (388 seconds), CovConJPF (546 seconds) and ConTeGe (780 seconds). The readers should notice that ConTeGeJPF and CovConJPF detect the faults faster than the original configurations, despite the fact that JPF is computationally more expensive than the random interleaving exploration of ConTeGe and CovCon. This is because ConTeGeJPF and CovConJPF explored

Table 5: Common Issues and Their Distribution Across The 47 Subjects

Problems	Percentage	Faults IDs
Invalid assumptions (1, 2, 3)	40.42%	dbcp{1-2-3}, derby{2-4-5}, groovy{1-2-6}, log4j{2-4-5}, lucene1, jdk6_{7-8-9-12}, jdk7_{2-4}
|-Invalid assumption 1 (two threads only)	12.77%	groovy{1-2}, log4j{2-4-5}, lucene1
|-Invalid assumption 2 (one shared object only)	25.53%	dbcp{1-2-3}, derby{2-4-5}, groovy6, jdk6_7, jdk7_4, log4_2
|-Invalid assumption 3 (no static invocations)	17.02%	dbcp3, groovy6, jdk6_{8-9-12}, jdk7_{2-4}, log4_2
Environmental dependencies	25.53%	derby1, groovy{1-2-3}, jdk6_{2-9-10-12}, jdk7_{1,2}, lucene{1-2}
Inadequacy for wait-notify	19.15%	jdk6_9, jdk7_5, log4j{4-5}, lucene{1-2}, pool{2-3-4}

the interleaving space of fewer tests before exposing the fault. This suggests that the random interleaving exploration of ConTeGe and CovCon easily miss failure-inducing interleavings even if the generated test can manifest one.

All tools report some thread-safety violations unrelated to the faults under analysis, which might or might not be true concurrency faults. ConTeGe, CovCon, ConTeGeJPF and CovConJPF report a unrelated `ConcurrentModificationException` for `jdk6_7`, `jdk6_10`, and `jdk6_11`. Intruder and AutoConTest detect the `jdk6_3` fault, and also report unrelated atomicity violations when analyzing this subject. Omen detect an unrelated deadlock for `jdk6_4`.

Automated concurrent test generators find **17%** of the JaConTeBe faults, and none of them alone finds more than **13%** of the faults. The average fault detection time ranges from **15** to **2,617** seconds.

5 RESULTS ANALYSIS AND DISCUSSION

The experimental results that we discussed in the previous section indicate limited complementarity among the concurrent test generators: half of the detected faults are revealed by at least four different tools. The results also indicate some overall incompleteness of the current approaches, since 83% of the concurrency faults in the JaConTeBe benchmark are not detected by any tool.

The absence of effective oracles is not the main reason of many undetected faults. In fact, the combinations of failure types and fault categories that characterise each subject of the JaConTeBe benchmark (with the only exception of `pool5`) can be revealed by the interleaving explorer and thread-safety oracle of at least three tools (see Table 4). This confirms that while automatically generating effective test oracles is a major challenges faced by sequential test generators [11, 23, 39, 62], generating oracles for concurrent tests is less critical [33, 67]: Lu et al. shown that 70% of concurrency faults lead to exceptions or hangs [33]. Moreover, Yu et al. proved that internal oracles are effective in detecting those concurrency faults that do not manifest visible oracle violations [67].

We manually analyzed the results of the experiments to identify and better understand why the tools fail to expose the considered concurrency faults. We relied on the manually-written tests in JaConTeBe to learn the characteristics of the concurrent tests that find faults that are not revealed with automatically generated tests.

We identified three main issues related to concurrent test generation: invalid assumptions, environmental dependencies, and inadequacy for `wait-notify`, as shown in Table 5. Each subject can suffer from more than one issue.

5.1 Invalid Assumptions

All surveyed techniques reduce the search space and facilitate fault detection by relying on some predefined assumptions on the concurrent tests that they generate: two threads only, one shared object only and no static invocations. Our results indicate that about 40% of the faults in the JaConTeBe benchmark cannot be revealed without violating at least one of such assumptions (see Table 5).

Assumption 1: exactly two concurrent threads. All surveyed techniques generate concurrent tests with exactly two concurrent suffixes, following the Lu et al.'s study that shows that 96% of concurrency faults can be manifested by enforcing a certain partial order between two concurrent threads only [33]. By inspecting the 47 failure-inducing tests in the JaConTeBe benchmark, we observe that 70.21% spawn two concurrent threads only. This is an under-approximation, because some tests are designed according to a stress-test methodology, which spawns many identical threads for increasing the chance to trigger a fault-revealing interleaving. Thus, some of the faults revealed with tests with more than two concurrent threads may be also revealed with tests with only two concurrent threads. We refined the study by manually identifying the fault-revealing tests that adopt a stress-test methodology, and concluded that 12.77% of the faults cannot be revealed with tests with two concurrent threads only. It is fairly easy to modify the tools for generating concurrent tests with an arbitrary number of suffixes, to target also faults that can be revealed only with more than two concurrent threads, but it may dramatically impact on the performance and effectiveness of the tools.

Assumption 2: at most a shared object instance. Most surveyed techniques generate concurrent tests that access from multiple threads only one shared object of the class under test, according to the intuition that accessing a single shared object is enough to trigger faults related to concurrent accesses to shared memory locations[5]. However, in some cases two (or more) object instances, although distinct, may access the same memory locations, and thus they could trigger concurrency failures. Such failures cannot be exposed with tests that instantiate a single shared object. For instance, in the fault-revealing concurrent test of the subject `derby5` in Figure 4, the concurrent execution of the suffixes leads to a concurrency failure even if the two suffixes do not access the same shared object, but two different objects (`baseContainerHandle` and `storedPage`). This happens because the *setter methods* in the sequential prefix mutually set each other references (line 4 and 5 in Figure 4). By inspecting the JaConTeBe benchmark, we observe that Assumption 2 is violated in 25.53% faults.

[5]Only ConTeGe and ConTeGeJPF generate tests that access more than one shared object.

```
// Sequential Prefix
... // omitted for brevity
...
storedPage.setExclusive(baseContainerHandle);
baseContainerHandle.addObserver(storedPage);

new Thread(new Runnable() {
    public void run() {
      baseContainerHandle.close();        // Suffix 1
  }}).start();

new Thread(new Runnable() {
  public void run() {
      storedPage.releaseExclusive();      // Suffix 2
  }}).start();
```

Figure 4: Manually-written test of the subject *derby5*, showing the case of two shared variables.

```
 // Sequential Prefix
final String dirA = projectBase + "/base/a";
final String dirB = projectBase + "/base/b";

new Thread(new Runnable() {
   public void run() {
         File file = new File(dirA);      // Suffix 1
         file.mkdirs();
   }}).start();

new Thread(new Runnable() {
    public void run() {
         File file = new File(dirB);      // Suffix 2
         file.mkdirs();
  }}).start();
```

Figure 5: Manually-written test of the subject *jdk6_2*, showing the case of the environmental dependencies problem.

Assumption 3: only non-static methods. All techniques but ConTeGe and ConTeGeJPF generate concurrent tests that invoke non-static methods of shared object instances only, following the intuition that most concurrency faults derive from incorrect accesses to dynamic instances. However, some JaConTeBe concurrency faults are exposed only with concurrent tests that either invoke static methods or access public static fields. By inspecting the JaConTeBe benchmark, we observe that Assumption 3 is violated in 17.02% faults. Only ConTeGe detects the fault groovy6, which requires the invoking of static methods (see Table 4).

5.2 Environmental Dependencies

A major challenge in generating both sequential and concurrent tests is to properly instantiate environmental dependencies to suitably exercise method call sequences, for instance, create certain files or database connections. For example, the fault-revealing concurrent test for subject jdk6_6 in Figure 5 requires a specific folder structure in the file system. Classic tools for generating sequential tests simulate dependencies on files and database connections with mocking techniques [3, 58]. Combining concurrent test generators with such techniques may address the environment dependency issues that account for 25.53% of the JaConTeBe faults.

5.3 Inadequacy for Wait-Notify

A relevant aspect of shared memory synchronization is the use of shared objects to pass messages: threads block their execution waiting to receive messages from other threads. This mechanism is implemented with primitives `wait()`, `notify()` and `notifyAll()` in Java[6] and `wait()`, `signal()` and `broadcast()` in C++.

None of the surveyed techniques supports such mechanism, and therefore none generates concurrency tests that can expose failures involving the execution of `wait` and `notify`. The two-step concurrent test generation approach illustrated in Figure 3 and implemented by all the techniques cannot address `wait` and `notify` related failures. The general approach works as follows: Step 1 generates single-threaded method call sequences, and Step 2 assembles such sequences in concurrent tests. Step 1 executes each newly generated sequence, and discards those that either throw exceptions or hang indefinitely. Such sequences are neither extended nor used to assemble new concurrent tests since they are likely illegal [37]. Discarding sequences that hang indefinitely on a single thread raises an issue in the presence of `wait` invocations: if a method invocation of the class under test puts the executor thread on an object wait, then the sequential test generator cannot call a new method on a different thread that will wake up the executor, a time-out exception is raised, and the sequence is discarded. As a result, all the concurrent tests that contain invocations of `wait` are discarded. However, the execution of `wait` is essential to trigger `wait-notify` deadlocks. `Wait-notify` faults amount to 19% of the JaConTeBe subjects.

[6]From Java 1.5 the mechanism is also implemented with `await()`, `signal()` in the `java.util.concurrent` package.

6 THREATS TO VALIDITY

Relevant threats to the validity of the results derive from the choice of the subjects, the time budget for the experiments, the selection of auxiliary classes and the choice of configuration options.

Choice of the subjects. We refer to faults present in open-source Java projects, and this may not generalize to all programming languages and program characteristics. Generalising the results would require extending the techniques originally defined for Java programs to a suitable variety of concurrent programming languages.

Time budget for the experiments. The time budgets of the experiment may impact on the number of detected faults: increasing the time budget may lead to detect more faults. The nature of the undetected faults discussed in Table 5 and the characteristics of the evaluated techniques indicate that for many cases no technique would detect any additional faults even with an unlimited time budget. In our experiments, we set the time budget to an hour per subject, according to the experiments of the most recent of the surveyed techniques [12].

Selection of auxiliary classes. We select the auxiliary classes by relying on fault-triggering manually-written tests provided in the JaConTeBe benchmark. The existence of such tests nullifies the need of running concurrent test generators, but it is useful in the evaluation, since it provides a set of auxiliary classes that are sufficient for revealing the fault.

Configuration options. The configuration options of the tools and the setup of the execution environments may impact on the effectiveness of the considered tools. We mitigated this threat by comparing the possible configuration options and inspecting the feedback from the execution environment. We also verified that the tools when executed in our environment were able to detect the concurrency failures reported in the corresponding papers.

7 RELATED WORK

To our knowledge, this paper presents the first independent empirical study that evaluates, compares and discusses the state-of-the-art generators of concurrent tests. Bianchi et al.'s recent survey proposes a comprehensive view of the state-of-the-art techniques for concurrent testing, and discusses the techniques based on the published papers [6]. Conversely, this paper focuses on concurrent test generators only and provides important additional empirical data. Related empirical studies focus either on techniques that explore the interleaving spaces of manually-written concurrent tests or on test generators for sequential programs.

Thomson et al. evaluated schedule bounding techniques for systematic concurrency testing, preparing a benchmark of 52 faulty concurrent programs [61]. Lin et al. evaluated interleaving exploration techniques (Step 3 and 4 in Figure 3) on the JaConTeBe benchmark [31] given the manually-written tests in input. Hong and Kim empirically evaluated detectors of data races [27].

Several empirical studies evaluated test generation techniques for sequential object-oriented programs [13, 19, 21, 66]. Xiao et al. [66] inspected the issues that limit test generators tools in obtaining high structural coverage. They found that the main issues are (i) dependencies on external methods, and (ii) finding a suitable method sequence to derive desired input object states. Shamshiri et al. [54] compared three popular generators of sequential tests: Randoop [37], Evosuite [20] and AgitarOne (www.agitar.com) using the Defects4J [29] benchmark. They found that the oracle problem remains the major obstacle, as 63.3% of the undetected faults were covered by automatically generated tests at least once. Fraser and Arcuri evaluated Evosuite [20] on a set of randomly selected open source projects [21]. Unsurprisingly, since a concurrent test is a concatenation of sequential tests, some of the challenges faced by concurrent test generators are inherited from those faced by sequential test generators. For example, the object creation problem in the presence of complex inputs [66]. The study presented in this paper raises additional insights and challenges that are specific to concurrent test generation.

8 CONCLUSION AND FUTURE RESEARCH DIRECTIONS

In this paper, we surveyed the main state-of-the-art techniques for generating concurrent tests, and we empirically evaluate and compare the ones for which we have been able to access a publicly available tool. We surveyed nine techniques and compared six of them, by referring to the 47 JaConTeBe benchmark. Our results show that overall the evaluated techniques can detect 17% of the JaConTeBe faults, which is an impressive result, if we consider that the faults remained undetected for years in popular code bases (many detected faults are in Java, which runs in over 15 billions devices). Current test generators could have revealed the faults and avoided their manifestation in the field.

The generation of concurrent tests is a relatively young but promising approach. Our evaluation results indicate a large space for improving its overall effectiveness. The analysis of the faults that current techniques do not detect yields insights into the main limitations, and indicates future research directions: adaptive configuration, search space reduction, and `wait-notify` handling.

Adaptive configuration. Current generators of concurrent tests work under some predefined assumptions on the tests being generated, as discussed in Section 5.1. These assumptions can drastically reduce the search space, thus improving the efficiency of the techniques, but can also prevent the generation of fault-revealing tests. Some tools allow users to enable/disable some of these assumptions in the form of configuration parameters [40]. However, no tool provide support to understand the impact of different configurations on the fault-detection capability for a given program under analysis. Interesting research directions are both studying techniques to automatically identify a proper configuration and defining self-adaptive interleaving explorers. CovCon and ConTeGe are examples of tools that can benefit from self-adaptive strategies. Both of them execute each test a fixed number of iterations independently from the nature of the test. The choice of the number of iterations is important, since too few iterations may miss a fault-revealing interleaving while too many could waste testing resources. A self-adaptive strategy could identify an optimal number of iterations for a test by approximating the number of thread interleavings that can be manifested when executing the test itself. Intuitively, the number of iterations should grow proportionally with respect to such number. Lu et al.'s formulas could provide useful hints for a cost-effective way to compute such approximations [32].

Search space reduction. Concurrent test generators explore a huge space of tests when generating concurrent tests. Given a class under test, it often exists a myriad of possible combinations of method invocations and input parameters. Generating all possible tests and exploring their interleaving spaces within an affordable time-budget remain infeasible. An interesting research direction is to define approaches that identify methods that cannot lead to a thread-safety violation if assembled in the same concurrent test, before generating the tests. Generating concurrent tests that involve such methods can be avoided without affecting fault-detection capabilities.

Wait-notify handling. As discusses in Section 5.3, current techniques cannot generate tests that expose `wait-notify` concurrency faults. To size the impact of this limitation, we observe that searching for `.wait() AND .notify()` in `github.com` produces ~60 millions code results. Using `wait-notify` synchronization primitives is not a matter of code style, since their synchronization behaviour cannot be simulated using other synchronization primitives like locks [64]. An important research direction is to update the four-steps framework for concurrent test generation presented in Figure 3. For instance, by recombining all steps to enable simultaneous test generation and interleaving exploration. Test generators could generate and execute method call sequences on multiple threads simultaneously so that, if a `wait` is executed putting the executing thread in a suspended state, another thread can unblock the suspended thread by generating and executing another method call sequence.

ACKNOWLEDGMENTS

This work is partially supported by the Swiss SNF project *ASTERIx: Automatic System TEsting of inteRactive software applications* (SNF 200021_178742).

REFERENCES

[1] 2018. Effectiveness and Challenges in Generating Concurrent Tests for Thread-safe Classes. http://star.inf.usi.ch/star/software/contest2018/index.htm. (2018).

[2] 2018. JaConTeBe. http://sir.unl.edu/portal/bios/JaConTeBe.php. (2018).

[3] Andrea Arcuri, Gordon Fraser, and Juan Pablo Galeotti. 2014. Automated Unit Test Generation for Classes with Environment Dependencies. In *Proceedings of the International Conference on Automated Software Engineering (ASE '14)*. ACM, 79–90.

[4] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525.

[5] Victor R Basili, Richard W Selby, and David H Hutchens. 1986. Experimentation in Software Engineering. *IEEE Transactions on Software Engineering* 7 (1986), 733–743.

[6] Francesco A. Bianchi, Alessandro Margara, and Mauro Pezzè. 2017. A Survey of Recent Trends in Testing Concurrent Software Systems. *IEEE Transactions on Software Engineering* (2017).

[7] Francesco A. Bianchi, Mauro Pezzè, and Valerio Terragni. 2017. Reproducing Concurrency Failures from Crash Stacks. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE '17)*. ACM, 705–716.

[8] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the Conference on Object-Oriented Programming Systems and Applications (OOPSLA '06)*. ACM, 169–190.

[9] Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. 2010. Line-up: A Complete and Automatic Linearizability Checker. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, 330–340.

[10] Yan Cai, Shangru Wu, and W. K. Chan. 2014. ConLock: A Constraint-based Approach to Dynamic Checking on Deadlocks in Multithreaded Programs. In *Proceedings of the International Conference on Software Engineering (ICSE '14)*. ACM, 491–502.

[11] Antonio Carzaniga, Alberto Goffi, Alessandra Gorla, Andrea Mattavelli, and Mauro Pezzè. 2014. Cross-checking Oracles from Intrinsic Software Redundancy. In *Proceedings of the International Conference on Software Engineering (ICSE '14)*. ACM, 931–942.

[12] Ankit Choudhary, Shan Lu, and Michael Pradel. 2017. Efficient Detection of Thread Safety Violations via Coverage-Guided Generation of Concurrent Tests. In *Proceedings of the International Conference on Software Engineering (ICSE '17)*. IEEE Computer Society, 266–277.

[13] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet?. In *Proceedings of the International Conference on Automated Software Engineering (ASE '16)*. IEEE Computer Society, 429–440.

[14] Dongdong Deng, Wei Zhang, and Shan Lu. 2013. Efficient Concurrency-bug Detection Across Inputs. In *Proceedings of the Conference on Object-Oriented Programming Systems and Applications (OOPSLA '13)*. ACM, 785–802.

[15] Yaniv Eytani, Klaus Havelund, Scott D Stoller, and Shmuel Ur. 2007. Towards a Framework and a Benchmark for Testing Tools for Multi-threaded Programs. *Concurrency and Computation: Practice and Experience* 19, 3 (2007), 267–279.

[16] Azadeh Farzan, Andreas Holzer, Niloofar Razavi, and Helmut Veith. 2013. Con2colic Testing. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '13)*. ACM, 37–47.

[17] Cormac Flanagan and Stephen N. Freund. 2004. Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs. In *Proceedings of the Symposium on Principles of Programming Languages (POPL '04)*. ACM, 256–267.

[18] Cormac Flanagan and Patrice Godefroid. 2005. Dynamic Partial-order Reduction for Model Checking Software. In *Proceedings of the Symposium on Principles of Programming Languages (POPL '05)*. ACM, 110–121.

[19] Gordon Fraser and Andrea Arcuri. 2013. EvoSuite: On the Challenges of Test Case Generation in the Real World. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST '13)*. IEEE Computer Society, 362–369.

[20] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Transactions on Software Engineering* 39, 2 (2013), 276–291.

[21] Gordon Fraser and Andrea Arcuri. 2014. A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Transactions on Software Engineering and Methodology* 24, 2, Article 8 (Dec. 2014), 42 pages.

[22] Brian Goetz and Tim Peierls. 2006. *Java Concurrency in Practice*. Pearson Education.

[23] Alberto Goffi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. 2016. Automatic Generation of Oracles for Exceptional Behaviors. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '16)*. ACM, 213–224.

[24] Shengjian Guo, Markus Kusano, Chao Wang, Zijiang Yang, and Aarti Gupta. 2015. Assertion Guided Symbolic Execution of Multithreaded Programs. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '13)*. ACM, 854–865.

[25] Klaus Havelund and Thomas Pressburger. 2000. Model Checking Java Programs Using Java Pathfinder. *International Journal on Software Tools for Technology Transfer* 2, 4 (2000), 366–381.

[26] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems* 12, 3 (1990), 463–492.

[27] Shin Hong and Moonzoo Kim. 2015. A Survey of Race Bug Detection Techniques for Multithreaded Programmes. *Software Testing, Verification and Reliability* 25, 3 (2015), 191–217.

[28] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. 2009. A Randomized Dynamic Program Analysis Technique for Detecting Real Deadlocks. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, 110–120.

[29] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '14)*. ACM, 437–440.

[30] Zhifeng Lai, S. C. Cheung, and W. K. Chan. 2010. Detecting Atomic-set Serializability Violations in Multithreaded Programs Through Active Randomized Testing. In *Proceedings of the International Conference on Software Engineering (ICSE '10)*. ACM, 235–244.

[31] Ziyi Lin, Darko Marinov, Hao Zhong, Yuting Chen, and Jianjun Zhao. 2015. JaConTeBe: A Benchmark Suite of Real-World Java Concurrency Bugs (T). In *Proceedings of the International Conference on Automated Software Engineering (ASE '15)*. IEEE Computer Society, 178–189.

[32] Shan Lu, Weihang Jiang, and Yuanyuan Zhou. 2007. A Study of Interleaving Coverage Criteria. In *Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC-FSE companion '07)*. ACM, 533–536.

[33] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)*. ACM, 329–339.

[34] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI '08)*. USENIX Association, 267–280.

[35] Adrian Nistor, Qingzhou Luo, Michael Pradel, Thomas R. Gross, and Darko Marinov. 2012. BALLERINA: Automatic Generation and Clustering of Efficient Random Unit Tests for Multithreaded Code. In *Proceedings of the International Conference on Software Engineering (ICSE '12)*. IEEE Computer Society, 727–737.

[36] Semih Okur and Danny Dig. 2012. How Do Developers Use Parallel Libraries?. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '12)*. ACM, 54:1–54:11.

[37] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *Proceedings of the International Conference on Software Engineering (ICSE '07)*. ACM, 75–84.

[38] Soyeon Park, Shan Lu, and Yuanyuan Zhou. 2009. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09)*. ACM, 25–36.

[39] Mauro Pezzè and Cheng Zhang. 2015. Automated Test Oracles: A Survey. In *Advances in Computers*. Vol. 95. Elsevier, 1–48.

[40] Michael Pradel and Thomas R. Gross. 2012. Fully Automatic and Precise Detection of Thread Safety Violations. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, 521–530.

[41] Michael Pradel and Thomas R. Gross. 2013. Automatic Testing of Sequential and Concurrent Substitutability. In *Proceedings of the International Conference on Software Engineering (ICSE '13)*. IEEE Computer Society, 282–291.

[42] Michael Pradel, Markus Huggler, and Thomas R. Gross. 2014. Performance Regression Testing of Concurrent Classes. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2014)*. ACM, 13–25.

[43] Ganesan Ramalingam. 2000. Context-sensitive Synchronization-sensitive Analysis is Undecidable. *ACM Transactions on Programming Languages and Systems* 22, 2 (2000), 416–430.

[44] Niloofar Razavi, Franjo Ivančić, Vineet Kahlon, and Aarti Gupta. 2012. Concurrent Test Generation Using Concolic Multi-trace Analysis. In *Asian Symposium on Programming Languages and Systems (ASPLS '10)*. Springer, 239–255.

[45] Malavika Samak and Murali Krishna Ramanathan. 2014. Multithreaded Test Synthesis for Deadlock Detection. In *Proceedings of the Conference on Object-Oriented Programming Systems and Applications (OOPSLA '14)*. ACM, 473–489.

[46] Malavika Samak and Murali Krishna Ramanathan. 2014. Omen+: A Precise Dynamic Deadlock Detector for Multithreaded Java Libraries. In *Proceedings of the*

ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, 735–738.

[47] Malavika Samak and Murali Krishna Ramanathan. 2014. Trace Driven Dynamic Deadlock Detection and Reproduction. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP '14)*. ACM, 29–42.

[48] Malavika Samak and Murali Krishna Ramanathan. 2015. Synthesizing Tests for Detecting Atomicity Violations. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '15)*. ACM.

[49] Malavika Samak, Murali Krishna Ramanathan, and Suresh Jagannathan. 2015. Synthesizing Racy Tests. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, 175–185.

[50] Malavika Samak, Omer Tripp, and Murali Krishna Ramanathan. 2016. Directed Synthesis of Failing Concurrent Executions. In *Proceedings of the Conference on Object-Oriented Programming Systems and Applications (OOPSLA '16)*. ACM, 430–446.

[51] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems* 15, 4 (1997), 391–411.

[52] Jochen Schimmel, Korbinian Molitorisz, Ali Jannesari, and Walter F Tichy. 2013. Automatic Generation of Parallel Unit Tests. In *Proceedings of the International Workshop on Automation of Software Test (AST '10)*. IEEE Computer Society, 40–46.

[53] Jochen Schimmel, Korbinian Molitorisz, Ali Jannesari, and Walter F Tichy. 2015. Combining Unit Tests for Data Race Detection. In *Proceedings of the International Workshop on Automation of Software Test (AST '15)*. IEEE Computer Society, 43–47.

[54] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. 2015. Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges. In *Proceedings of the International Conference on Automated Software Engineering (ASE '15)*. IEEE Computer Society, 201–211.

[55] Elena Sherman, Matthew B. Dwyer, and Sebastian Elbaum. 2009. Saturation-based Testing of Concurrent Programs. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '09)*. ACM, 53–62.

[56] L. A. Smith, J. M. Bull, and J. Obdrizalek. 2001. A Parallel Java Grande Benchmark Suite. In *Supercomputing, ACM/IEEE 2001 Conference*. 6–6.

[57] Sebastian Steenbuck and Gordon Fraser. 2013. Generating Unit Tests for Concurrent Classes. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST '13)*. IEEE Computer Society, 144–153.

[58] Kunal Taneja, Yi Zhang, and Tao Xie. 2010. MODA: Automated Test Generation for Database Applications via Mock Objects. In *Proceedings of the International Conference on Automated Software Engineering (ASE '10)*. ACM, 289–292.

[59] Valerio Terragni and Shing-Chi Cheung. 2016. Coverage-driven Test Code Generation for Concurrent Classes. In *Proceedings of the International Conference on Software Engineering (ICSE '16)*. ACM, 1121–1132.

[60] Valerio Terragni, Shing-Chi Cheung, and Charles Zhang. 2015. RECONTEST: Effective Regression Testing of Concurrent Programs. In *Proceedings of the International Conference on Software Engineering (ICSE '15)*. IEEE Computer Society, 246–256.

[61] Paul Thomson, Alastair F. Donaldson, and Adam Betts. 2014. Concurrency Testing Using Schedule Bounding: An Empirical Study. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP '14)*. ACM, 15–28.

[62] Paolo Tonella. 2004. Evolutionary Testing of Classes. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '04)*. ACM, 119–128.

[63] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. 2003. Model Checking Programs. *Automated Software Engineering* 10, 2 (2003), 203–232.

[64] Chao Wang, Rhishikesh Limaye, Malay Ganai, and Aarti Gupta. 2010. Trace-Based Symbolic Analysis for Atomicity Violations. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS '10)*. Springer, 328–342.

[65] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in Software Engineering*. Springer Science & Business Media.

[66] Xusheng Xiao, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. 2011. Precise Identification of Problems for Structural Test Generation. In *Proceedings of the International Conference on Software Engineering (ICSE '11)*. ACM, 611–620.

[67] Tingting Yu, Witawas Srisa-an, and Gregg Rothermel. 2013. An empirical comparison of the fault-detection capabilities of internal oracles. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE '13)*. IEEE Computer Society, 11–20.