2015 30th IEEE/ACM International Conference on Automated Software Engineering

# JaConTeBe: A Benchmark Suite of Real-World Java Concurrency Bugs

Ziyi Lin*, Darko Marinov†, Hao Zhong‡, Yuting Chen‡, and Jianjun Zhao‡
*School Of Software, Shanghai Jiao Tong University, China
†Department of Computer Science, University of Illinois at Urbana-Champaign, USA
‡Department of Computer Science and Engineering, Shanghai Jiao Tong University, China
{linziyi, zhonghao, chenyt, zhao-jj}@sjtu.edu.cn, marinov@illinois.edu

***Abstract*—Researchers have proposed various approaches to detect concurrency bugs and improve multi-threaded programs, but performing evaluations of these approaches still remains a substantial challenge. We survey the existing evaluations and find out that they often use code or bugs not representative of real world. To improve representativeness, we have prepared JaConTeBe, a benchmark suite of 47 confirmed concurrency bugs from 8 popular open-source projects, supplemented with test cases for reproducing buggy behaviors. Running three approaches on JaConTeBe shows that our benchmark suite confirms some limitations of the three approaches. We submitted JaConTeBe to the SIR repository (a software-artifact repository for rigorous controlled experiments), and it was included as a part of SIR.**

***Keywords*—*Java concurrency bugs, evaluations, benchmark suite, SIR, JaConTeBe***

## I. Introduction

Concurrent code is gaining much attention with the growth of multi-core computing. However, developing concurrent code remains challenging and error-prone because threads are non-deterministically scheduled and can inappropriately interact. Concurrency bugs are prevalent, *e.g.*, as reported by Lu *et al.* [54]. Because concurrency bugs decrease the quality of software, researchers have proposed various approaches to detect such bugs. (Section VI presents an overview of these approaches.) Although these approaches can successfully detect some concurrency bugs, their effectiveness in practice is largely unknown. In typical evaluations done by the researchers proposing an approach, the code or bugs used may be selected with a bias and not representative of the real-world code or bugs. In addition, it is difficult to compare these approaches, because their effectiveness may highly depend on the selection of concurrency bugs.

To address the evaluation problem, researchers have used various benchmark suites. (Section II surveys four suites in detail.) Typically, a benchmark suite consists of concurrency bugs and corresponding test cases. Researchers can run different approaches on the benchmarks to evaluate the effectiveness. While several benchmarks have been used, a key concern is whether these benchmarks fully reflect the characteristics of concurrency bugs in the real world. Following Sim *et al.* [76], Lu *et al.* [53] proposed five guidelines for preparing benchmark suites: representative, diverse, portable, accessible, and fair. In particular, representative means both code and bugs should be real ones. The existing benchmark suites do not fully satisfy these guidelines, especially the representative guideline, as discussed in Section IV-C. Trying to directly use real software systems, with no extra artifacts, to evaluate the effectiveness is hard. One main reason is that the complexity of real-world software systems makes it difficult to establish whether the reported issues are true positives, especially for concurrency bugs; much extra effort is required to tell true bugs from various false positives.

In this paper, we first survey the existing benchmark suites for Java concurrency bugs and then introduce a new benchmark suite, called *JaConTeBe* (from "JAva CONcurrency TEsting BEnchmark"). JaConTeBe currently contains 47 Java concurrency bugs taken from real-world code. JaConTeBe can help evaluate approaches for detecting concurrency bugs. First, JaConTeBe allows to evaluate whether an approach is capable of finding the bugs; researchers can run novel approaches on JaConTeBe to evaluate their effectiveness. Second, it allows to evaluate whether an approach is practical for real bugs; although an approach may work well on seeded bugs, it may be insufficient to deal with real bugs, because real bugs may have different characteristics. Third, it allows to compare the strengths and weaknesses among similar approaches; it is important to properly compare and evaluate different tools on real software bugs.

The paper makes the following contributions:

- We present a survey of the existing Java concurrency benchmark suites and investigate the evaluations of approaches for detecting Java concurrency bugs. We have observed that the existing benchmark suites do not contain many real-world bugs, although researchers have the desire to evaluate their work with real bugs. The gap between real bugs and evaluations is not fulfilled yet.
- We build up JaConTeBe, a new benchmark suite for Java concurrency bugs. The initial version of JaConTeBe contains 47 confirmed, real-world concurrency bugs from 8 open-source projects. We submitted JaConTeBe to the SIR repository of software artifacts for rigorous controlled experiments, and it was included at *http://sir.unl.edu/portal/bios/JaConTeBe.php*. We view JaConTeBe as ongoing work and plan to collect more bugs in the future. These bugs can be classified into three types, and in particular the subtype of Java communication deadlock is less studied in the literature. For each bug, we provide detailed documentation that describes the root cause and buggy interleaving of the bug, and we also implement a test case and a test

978-1-5090-0025-8/15 $31.00 © 2015 IEEE
DOI 10.1109/ASE.2015.87

script so that other researchers can easily reproduce the buggy behavior.

- We run JaConTeBe test cases on three approaches for detecting Java concurrency bugs: the Java PathFinder (JPF) research tool [79], the RV-Predict commercial tool [39], and our implementation of the CheckMate approach [45]. The results show that JaConTeBe can uncover the capability of the existing approaches and can help confirm some of their known limitations in tackling real applications.

The rest of this paper is organized as follows. Section II surveys the existing benchmark suites and evaluations of approaches to detecting Java concurrency bugs. Section III presents our new JaConTeBe benchmark suite. Section IV evaluates JaConTeBe. Section V discusses some results and potential future work. Section VI describes related work, and Section VII concludes.

## II. Survey of Benchmark Usage

In this section, we survey first the existing Java concurrency benchmark suites and then how these suites and other programs are used in papers on detecting Java concurrency bugs. Our survey identifies some limitations in the existing evaluations and lists the characteristics of a desirable benchmark suite.

### A. Existing Benchmarks

Many evaluations use some of the programs from the following four Java concurrency benchmark suites.

*Java Parallel Grande (JPG)* [77] is the parallel version of the Java Grande [11] benchmark suite, designed for evaluating parallel Java applications for high-performance computing. JPG contains 20 applications. Although not initially designed for bug detection, it became popular, and many approaches for detecting concurrency bugs used some applications from JPG.

*DaCapo* [8] is a benchmark suite of real-world Java applications that require non-trivial memory loads. There are two releases of the benchmark suite: version 2006 with 11 applications and version 2009 with 14 applications. As for JPG, DaCapo was not originally designed for bug detection and does not explicitly list any concurrency bugs.

The *IBM benchmark suite* [22] contains 41 buggy Java concurrent programs and covers 12 types of concurrency bug patterns [25]. However, 24 of the buggy programs in the IBM benchmark suite are from undergraduate students assignments and rather small [23]. 16 are open source, but only 4 of them are real applications, and the others are small programs simulating concurrency behaviors. 1 program is a commercial product, Joscar, from AOL messenger.

The *Software-artifact Infrastructure Repository (SIR)* [18] is a repository of software-related artifacts that supports rigorous controlled experimentation with program analysis and software testing techniques. Before we contributed JaConTeBe to SIR, SIR contained 36 Java concurrency bugs collected from different sources, including 10 from the IBM benchmark suite and 11 bugs from real applications.

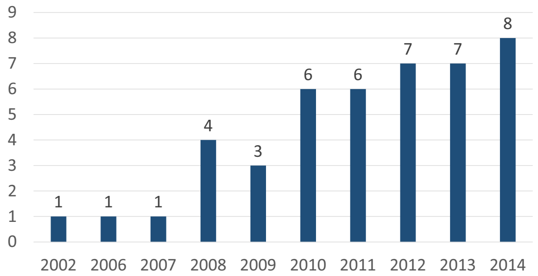

Fig. 1. Surveyed papers distributed by publication year.

TABLE I. Research topics of the surveyed papers.

| Research Topic | Description | #Papers |
|---|---|---|
| atomicity | detect atomicity violations | 9 |
| atomic-set-serializability | detect atomicity violations and races | 3 |
| concurrency debugging | debug concurrency bugs | 3 |
| concurrency testing | techniques and frameworks for testing concurrency bugs | 8 |
| deadlock | detect deadlock bugs | 7 |
| nullpointer dereference | detect nullpointer bugs in concurrent code | 1 |
| race | detect data races | 13 |

After we investigated the existing benchmarks, we came to the following observation:

Observation 1: *The existing benchmarks typically contain only several real-world concurrency bugs.*

### B. Programs Used in Evaluations

We perform a survey to find out what programs are used in empirical evaluations of published papers. We focus on five most relevant conferences: ASE, FSE, ICSE, ISSTA, and PLDI. We search DBLP[1] with the query “`deadlock|race|datarace|concurren|atom |multithreaded|thread|preemption|parallel venue:[conference name]`” to find all candidate papers published in each conference between 2002 and 2014. We start from 2002 because the oldest well-known benchmark suite, JPG, was published in 2001.

We get 256 candidate papers in total. After filtering out papers that are not related to Java or concurrency bug detection, we are left with 59 papers. To reduce our effort for inspecting these papers, we randomly sample only 10 papers published before 2010, but we select all 34 papers published since 2010. We inspect in detail 44 papers, 3 from ASE [30], [31], [72], 8 from FSE [21], [26], [42], [45], [52], [61], [70], [78], 8 from ICSE [14], [34], [48], [57], [59], [60], [63], [82], 9 from ISSTA [5], [35], [40], [43], [62], [67], [74], [81], [84], and 16 from PLDI [7], [9], [12], [12], [15], [17], [27]–[29], [33], [39], [46], [55], [58], [68], [73]. Fig. 1 shows the distribution of the papers by publication year. These papers are divided into seven different research topics related to concurrency bug detection, as shown in Table I.

We investigate what programs are used in evaluations in these 44 papers. To simplify the analysis, we ignore different

[1] http://dblp.uni-trier.de/db/

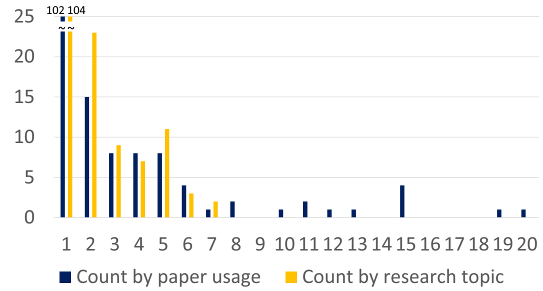

Fig. 2. Usage of programs in surveyed papers. The X-axis represents (1) the number of papers in which a program was used (blue bars) and (2) the number of topics that the papers belong to (yellow bars). The Y-axis represents the number of programs.

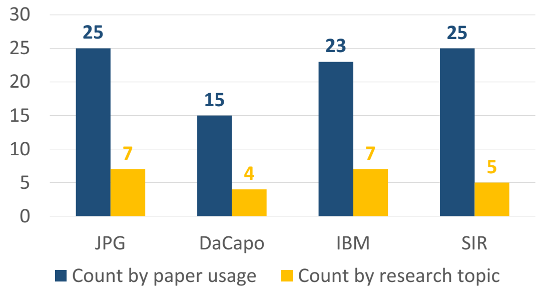

Fig. 3. Usage of benchmark suites in surveyed papers.

versions and bugs from the same program/project[2] in the survey, and count only the number of different programs/projects used in each paper. For example, if a paper uses some code from Apache Pool 1.X for one bug and some code from Apache Pool 2.X for another bug, we count that as one Apache Pool project; if another paper also uses some code from Apache Pool 1.X for two different bugs, we count that as two papers using Apache Pool once each.

These papers altogether use 159 programs in their evaluations. Fig. 2 shows that about two thirds (102 out of 159) of the programs are used in only one paper and on one research topic (104 out of 159), meaning most programs are not shared among evaluations done by different researchers. `JDK` and `RayTracer` are shared the most by, respectively, 20 and 19 papers that cover all seven research topics.

Observation 2: *Researchers often follow an ad-hoc process to choose programs for evaluation.*

Many programs used in the evaluations belong to some of the four benchmark suites: JPG, DaCapo, IBM benchmark, and SIR benchmark. As Fig. 3 shows, JPG is the most widely used, and its programs appear in 25 out of 44 (56.8%) papers that cover all seven research topics. (Note that a paper may use programs from multiple benchmark suites.)

Of the 159 programs used, 9 are from JPG, 12 are from DaCapo (both versions 2006 and 2009), 20 are from the IBM suite, and 11 are from SIR; several of these programs are in multiple suites: 2 programs are in both JPG and IBM, while 5 programs are in both IBM and SIR. The other 114 programs do not belong to any well-known benchmark suite.

Observation 3: *JPG has been the most popular benchmark suite, although it was not designed for concurrency bug detection and is rather outdated.*

We have found out that 104 out of 159 benchmark programs are from real-world projects. Object-relation mapping tools and web-server containers are most represented. This fact suggests that researchers have the desire to use real-world programs in their evaluations. However, many of the real-world programs are not shared among evaluations and researchers: only 30 programs appear in two or more papers, and 28 in two or more research topics. One main reason is that, when evaluating an approach proposed to find only some bugs, researchers prefer to select the benchmark programs that are related to their target bugs. When the target bugs differ, researchers tend to select quite different programs in their evaluations. Although this allows researchers to focus on their target bugs, the results can become narrowly applicable and hard to compare. To fully evaluate the effectiveness, there is a strong need for a more comprehensive benchmark suite.

Observation 4: *Real-world programs are preferably used by researchers but are often selected with bias.*

## III. The JaConTeBe Benchmark Suite

In this section, we present JaConTeBe, a benchmark suite we collected to allow more comprehensive evaluations of approaches for detecting concurrency bugs. The buggy programs in JaConTeBe are from popular open-source systems and confirmed by developers.

### *A. Overview*

We prepare JaConTeBe following Lu *et al.*'s five guidelines [53]. In particular, (1) for representativeness and accessibility, we collect reported and confirmed bugs from real open-source projects; (2) for diversity, we collect bugs that cover various bug types and different importance; (3) for fairness, we choose bugs from projects that are shared among different papers and research topics so they are more general; and (4) for portability, we choose bugs that are portable to different platforms, except two bugs triggered only on Windows.

We use the following process to collect concurrency bugs. First, we select from the open-source projects that have been used in prior evaluations, as described in Section III-B. Then, for each project, we search its online bug database for bug reports related to concurrency bugs, as described in Section III-C. Finally, for each (non-duplicate) bug report that we can reproduce, we create a test case that allows triggering buggy behavior (usually deterministically on every run), as described in Section III-D.

In total, the initial version of JaConTeBe contains 47 concurrency bugs from 8 open-source projects. It covers three types of concurrency bugs: races (*e.g.*, low-level data races and high-level atomicity violations), deadlocks (*e.g.*, resource deadlocks caused by cyclic locks and communication deadlocks caused by missed synchronization signals), and Java memory-model-related bugs (caused by inappropriate assumptions, *e.g.*,

[2] We use the terms "program" and "project" interchangeably, but intuitively one should distinguish small programs from real projects.

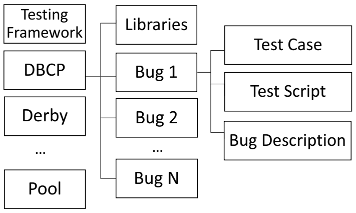

Fig. 4. Overview of the JaConTeBe benchmark suite.

about synchronization activities and reordering of statement executions among threads).

Fig. 4 shows the organization of JaConTeBe. It contains one common directory (with a testing framework and jar libraries shared across the entire benchmark suite) and one directory for each project with concurrency bugs (currently 8 such directories). Each project directory has several jar libraries shared by its bugs and a subdirectory for each bug. Each subdirectory contains the following:

1) A Java test case to reproduce the buggy behavior
2) A shell script to set test parameters and run the test
3) A description of the bug and its root cause

### B. Selecting Projects

It is challenging to sample representative open-source projects from their large population. For example, GitHub[3] hosts more than 100K Java projects, and Apache[4] hosts over 200 Java projects. Furthermore, many projects are added each year with the rapid growth of open-source communities.

However, as we have observed from our survey, some projects[5] have been used more often for evaluations. We simply select from those projects that have been already used instead of creating new rules to find out new projects. We use two requirements for selecting candidate projects:

1) A candidate must have been used in evaluations on two or more research topics. This filters rarely used projects and reduces the bias towards certain approaches on specific research topics. Of 159 programs from our survey, this requirement filters out about two thirds, leaving 55.
2) A candidate must be an open-source, well-maintained project with real bugs. This ensures that the selected projects have real bugs instead of injected or artificial ones, and equally importantly, it allows us to access the bug database to retrieve sufficient information to document and reproduce the concurrency bug. This requirement filters out (1) `RayTracer`, one of the programs most widely used in evaluations, but which is an artificial program written by researchers for benchmarking and not actual code written by developers, and (2) *cache4j*, the popular real-world benchmark that has stopped updating since 2006 and has no bug database. This requirement further filters out about three quarters of candidates, leaving 13.

TABLE II. PROJECTS IN JACONTEBE.

| Project | #Papers/topics in survey | #Bugs in JaCon-TeBe | Description |
|---|---|---|---|
| Apache DBCP | 4 / 2 | 4 | Database connection pool |
| Apache Derby | 6 / 4 | 5 | Relational database |
| Apache Groovy | 3 / 2 | 6 | Dynamic language for JVM |
| OpenJDK | 20 / 7 | 20 | Java Development Kit |
| Apache Log4j | 3 / 3 | 5 | Logging library |
| Apache Lucene | 7 / 4 | 2 | Search library |
| Apache Pool | 8 / 5 | 5 | Object-pooling API |

We select 7 out of 13—Apache DBCP, Apache Derby, Apache Groovy, OpenJDK, Apache Log4j, Apache Lucene, and Apache Pool—for the initial version of JaConTeBe. As mentioned previously, we omit version differences of the same project, but in JaConTeBe, we keep OpenJDK6 and OpenJDK7 separated, because their bugs can be reproduced only with different versions of the Java environment. Table II shows some details of these projects, with OpenJDK6 and OpenJDK7 merged together in this table, because some papers do not clearly mention which version of OpenJDK is used in the evaluation. For each project, we tabulate the number of papers that used the project in evaluation (and the number of research topics that those papers cover), the number of bugs prepared in JaConTeBe (note that this is completely independent of the number of papers and topics but by chance the numbers end up similar), and a brief description. OpenJDK is the most widely used real-world project in our survey and is also the biggest contributor of bugs (42.6%, 20 out of 47) for JaConTeBe.

### C. Searching for Concurrency Bugs

While we selected all the projects for JaConTeBe from the surveyed papers, we could not identify all the bugs because the papers often did not give the bug IDs or URLs. Therefore, we decided to search for concurrency bugs in the bug databases of all the projects. More specifically, we search each bug database with the keywords "`concurren`", "`atom`", "`deadlock`", "`thread`", "`race`", and "`synch`". This simple search returns many false positives, *e.g.*, "`thread`" also matches "`threaded`" or "`threading`", two words that are often used in the discussions of bug reports even when not for concurrency bugs. Of 1065 issues returned by our queries, we manually confirm 206 as related to concurrency bugs.

We select bugs for JaConTeBe using these requirements:

1) The bug should not be a duplicate. We ignore bug reports which are marked "`Duplicate`" or "`clone of`" in the bug database.
2) The bug should be reproducible, *i.e.*, (1) the bug report provides sufficient information to reproduce the bug, *e.g.*, stacktraces and thread dumps when the program crashes, or even better, a test case for triggering the buggy behavior; (2) the necessary resources to reproduce the bug are available, *e.g.*, `GROOVY-1890` is a deadlock bug from version 1.1-beta-1, but neither source code nor binary release of that version is available for download any more, therefore it is not easily reproducible.

[3] https://github.com/
[4] https://projects.apache.org/projects.html?language
[5] While we also called them "programs" before as some are rather small, we will call them "projects" in the remainder as we select larger, real projects.

3) The bug should be relevant. Every real bug represents some problem triggered under certain conditions, but some bugs have little impact on the system or are only triggered under very specific conditions that are not common and may not be worth fixing. To identify these bugs that may not be representative, we check for: (1) the report labeled "won't fix" or having a similar description in the discussion, and/or (2) the priority being marked as the lowest level.

Finally, we collect 47 bugs for our benchmark suite. These bugs were reported between 1999 and 2013. Most (70.2%, 33 out of 47) are important (marked as "blocker", "critical", "major", or some equivalent priority value in the bug report), and the others (14 out of 47) are less important (marked as "minor" or some equivalent value in the bug report). 5 bugs (2 major and 3 minor) have not been fixed until now. The average fix time for all fixed bugs is 22.9 months, and the average fix time for all fixed important bugs is 12.3 months. These long times indicate that these bugs can be highly non-trivial.

As Table III shows, we classify the 47 bugs into three types: races, deadlocks, and memory-model related bugs. We then analyze the root causes of communication deadlock, a subtype less studied in the literature.

*1) Races and Atomicity Violations:* JaConTeBe contains 19 data races and atomicity violations, which are two well-studied types of concurrency bugs [36], [49]. Both represent the cases of concurrently modifying shared variables such that the output can depend on the execution sequence, so in JaConTeBe, we group races and atomicity violations together.

For example, Fig. 5 shows a bug in OpenJDK6. Thread 1 checks two conditions at line 4, while thread 2 can assign a null pointer to `filter` at line 14. If the assignment happens between the checks of condition 1 and condition 2, the checking of condition 1 is invalid.

*2) Deadlocks:* JaConTeBe contains 25 deadlocks, of which 16 are resource deadlocks and 9 are communication deadlocks. A resource deadlock occurs when a thread holds resources and prevents the other threads to proceed [4], [13]. A communication deadlock occurs when synchronization signals (*i.e.*, `wait`, `notify`, and `notifyAll` in Java) among threads are improperly used. It was pointed out even in 2008 [23] that researchers focus on races and atomicity violations more than on deadlocks; 7 years later, although more researchers have focused on resource deadlocks, communication deadlocks are still less studied.

Table IV provides some more details for the 9 communication deadlocks in JaConTeBe. They have three symptoms. The bug `POOL-149` corresponds to a symptom called "notify-before-wait", where `notify` is executed before `wait`, and thus waiting threads will not be notified. The bug `LUCENE-1544` corresponds to a symptom called "always-wait", where the condition for a waiting loop always holds, blocking the program. The others correspond to "notify-not-executed", where a waiting thread cannot get notified, because the corresponding `notify` does not get executed. A previous study [25] reported similar concurrency bug patterns—"losing a notify bug pattern" and "a 'Blocking' critical section bug pattern"—but these patterns are too general, and the study did not show corresponding real bugs. We thus further investigate the communication deadlocks in JaConTeBe and summarize their root causes as follows:

TABLE III. CONCURRENCY BUGS IN BENCHMARK PROGRAMS.

| Bug Types | | #Bug | Description |
|---|---|---|---|
| Races and atomicity violations | | 19 | Low-level data races and high-level atomicity violations |
| Deadlock | Resource deadlock | 16 | Threads compete for the critical resources (locks), and each thread holds a resource (lock) required by others |
| | Communication deadlock | 9 | The wait-notify communication between threads is broken |
| Memory-model related issue | Inconsistent synchronization | 2 | The accesses to the shared variable among threads are not properly synchronized, so a change from one thread is not visible to other threads |
| | SC violation | 1 | The statement execution order among different threads is reordered by JMM, making SC violation |

- Redundant locks: This is a mixed use of holding extra locks while invoking `wait` or `notify` methods. In Java, when `wait` is called on an object, the lock on the object is released. However, if the waiting thread holds an extra lock, and the notify thread waits for it to wake up the waiting thread, the program blocks.
- Misuse of `notify` and `notifyAll`: If `notify` is incorrectly used at the place where `notifyAll` should be used, then the JVM picks only one of the waiting threads (seemingly at random), while all the remaining threads keep waiting.
- Scheduling problem: The threads may not be appropriately orchestrated, making a `notify` in one thread execute before the intended `wait` in another thread. The scheduling problem can also lead to the notify condition being unsatisfied, *e.g.*, in `POOL-162`, the interference among thread executions can change one notify condition to an unexpected state.
- Unhandled exceptions: The waiting threads cannot get notified when an exception is not caught or handled appropriately. Due to exceptions, a working thread may not invoke `notify` or `notifyAll`, or it fails to set the conditions to notify.
- Logical incorrectness: Logical incorrectness is related to the thread scheduling problems, *e.g.*, `JDK-6648001` is caused by a conflict of waiting and notify conditions, and once a thread waits, it is infeasible to notify the thread due to logical problems.
- Inappropriate usage of data structures and algorithms: When data structures and algorithms that do not support concurrency accesses are used in a concurrent setting, deadlocks can happen, *e.g.*, `POOL-146`.

*3) Memory Model Related Bugs:* JaConTeBe contains three bugs related to the Java memory model (JMM) [56], *i.e.*, two inconsistent synchronization bugs [37] and a sequential-consistency (SC) violation bug [66].

An inconsistent synchronization is similar to a race condition; both occur when two or more accesses to a shared variable are not synchronized properly, and at least one access is a write. However, an inconsistent synchronization bug is usually caused by an "invisible" change, which means a shared variable is modified in one thread but the modification is not

TABLE IV. ROOT CAUSE FOR COMMUNICATION DEADLOCKS IN JACONTEBE.

| Bug | Symptom | Root Cause | Bug Description |
|---|---|---|---|
| POOL-149 | Notify-before-wait | Schedule issue | A notify statement may be executed ahead of the corresponding wait statement. See https://issues.apache.org/jira/browse/POOL-149. |
| POOL-146 | Notify-not-executed | Inappropriate usage of data structures and algorithms | Pool uses a `LinkedList` object. One thread A attempts to get the first element from the list and, if the first element is not retrievable, waits. Another thread B retrieves other elements in the list and then notifies thread A to continue. However, B can be blocked, and A then cannot be waken up, because a list must be accessed from its head. See https://issues.apache.org/jira/browse/POOL-146. |
| POOL-162 |  | Unhandled exception | An exception makes a borrowed object to not be returned, while other threads wait for that object. See https://issues.apache.org/jira/browse/POOL-162. |
| LOG4J-50463 |  | Unhandled exception | An exception is thrown in a notify thread, while other threads waits to be notified. See https://issues.apache.org/bugzilla/show_bug.cgi?id=50463. |
| LOG4J-38137 |  | Misuse of notify and notifyAll | The program has one dispatcher and two appenders. After one element is removed from the buffer, the dispatcher notifies only one appender, making the other appender wait. See https://issues.apache.org/bugzilla/show_bug.cgi?id=38137. |
| JDK-8012019 |  | Redundant locks | Some redundant synchronization is enforced on threads, causing a deadlock of the waiting and the notify threads. See https://bugs.openjdk.java.net/browse/JDK-8012019. |
| LUCENE-2783 |  | Redundant locks | The bug is the same as JDK-8012019. See https://issues.apache.org/jira/browse/LUCENE-2783. |
| JDK-6648001 |  | Logical issue | There are conditions for both `wait` and `notify`, but they are contradictory. When the waiting condition is satisfied, it is infeasible to reach the notify statement. See https://bugs.openjdk.java.net/browse/JDK-6648001. |
| LUCENE-1544 | Always wait | Logical issue | The condition in a wait thread (`...while(condition)`{`wait(timeout)`}`...`) is always true. See https://issues.apache.org/jira/browse/LUCENE-1544. |

propagated to another thread. For example, `POOL-46` is an inconsistent synchronization bug in JaConTeBe; one thread is synchronized to reset a loop flag (a shared variable), but another thread is not synchronized to read the flag, resulting in an infinite loop. The inconsistent synchronization can be fixed either by marking accesses from all threads to one shared variable as "`synchronized`" or declaring the shared variable as "`volatile`".

A SC violation bug happens when the execution sequence of two or more statements in one thread is reordered. It can be introduced at different optimization phases (*e.g.*, compiling optimizations, runtime optimizations, or instruction execution optimizations) [56], [66]. Although reordering is allowed by the Java memory model when it does not perturb the single-thread execution, it may break data or control dependencies among threads. Section IV-B shows the SC violation bug from the Groovy project.

### D. Implementing Test Cases

It is non-trivial to supplement a concurrency bug with a test case that triggers the buggy behavior, either deterministically on every run or almost deterministically such that one needs a small number of runs to trigger the buggy behavior. For JaConTeBe, we leverage the bug reports to create a test case for each bug. If a bug report has an attached test case, we modify it to deterministically trigger the bug. Otherwise, we design a new test case according to the description and discussion in the bug report (*e.g.*, stacktraces, buggy interleaving, or bug symptoms), expecting to reproduce the reported bug.

To reproduce a concurrency bug (almost) deterministically, a test case needs to specify appropriate (1) thread interleaving that schedules the execution to the buggy location by following a mostly deterministic interleaving and (2) inputs that lead the execution through the branches to reach the buggy location by following a mostly deterministic execution path. When necessary to achieve this goal, we use instrumentation and mocking to design the test cases.

We can usually enforce the interleaving by adding statements in the test case, *e.g.*, to control that one thread starts before another, but when more elaborated controls are required during the thread execution, we use instrumentation to ensure the threads follow the expected interleaving to trigger the bug. For example, Fig. 5 shows the code snippet for a bug that is very unlikely to be triggered when the test is repeated without instrumentation, because the time for preemption between the two condition checks at line 4 is very short. However, the bug can be triggered easily if we add some instrumentation, either (1) a pause (*e.g.*, sleep for a few milliseconds) between condition 1 and "`&&`", so that the bug triggers much more frequently, or (2) some control statements to enforce the statement at line 14 to execute just between the conditions 1 and 2 at line 4, so that the bug can be deterministically reproduced. We use ASM [1], a Java bytecode manipulation and analysis framework, to instrument the Java binary code. In this example, as we modify the JDK code itself, we need to start the test with the `-Xbootclasspath/p` option to

```
 1          thread 1
 2  public void log(LogRecord record){
 3      ...
 4      if($\underline{filter! = null}^1$ && $\underline{!filter.isLoggable(record)}^2$)
 5      {
 6          return;
 7      }
 8      ...
 9  }
10
11          thread 2
12  public void setFilter(Filter newFilter) ...{
13      ...
14      filter=newFilter;
15  }
```

Fig. 5. An instrumentation example in OpenJDK6. This bug happens when filter is assigned to null at line 14 in one thread just between the two condition checks at line 4 in another thread.

```
1  if (physicalConnection_.isClosed())
2  {
3      ...
4  } else {
5      ...
6      if (!physicalConnection_.isGlobalPending_()) {
7          pooledConnection_.recycleConnection();
8      }
9  }
```

Fig. 6. A mock example in DERBY-5560. The bug is in the method call at line 7 and requires the checks at lines 1 and 6 to be false and true, respectively.

TABLE V. RESULTS FOR THREE APPROACHES.

| Detection | #Succeed | #Fail | #Crash |
|---|---|---|---|
| Approach | JaC + SIR | JaC + SIR | JaC + SIR |
| JPF | 11 + 36 | 10 + 0 | 26 + 0 |
| RV-Predict | 10 + 17 | 2 + 0 | 0 + 0 |
| CheckMate | 8 + 13 | 17 + 2 | 0 + 0 |

TABLE VI. DETAILED RESULTS FOR CHECKMATE ON JACONTEBE.

| Buggy Program | JaConTeBe Result | Deadlock Type | Project Version | Result in [45] |
|---|---|---|---|---|
| POOL-146 | succeed | communication deadlock | POOL 1.5 | 2 |
| POOL-149 | succeed | communication deadlock | | |
| POOL-162 | fail | communication deadlock | | |
| LOG4J-38137 | succeed | communication deadlock | Log4j 1.2.13 | 1 |
| LOG4J-41214 | fail | resource deadlock | | |
| DBCP-65 | fail | resource deadlock | DBCP 1.2 | 2 |
| DBCP-270 | succeed | resource deadlock | | |

preload the instrumented code instead of the original code from JDK.

We also use mocking to ensure that the concurrent code strictly follows a given program trace to reach a buggy program location. For example, Fig. 6 shows the code snippet for bug `DERBY-5560` that is triggered in the method call at line 7. The execution reaches line 7 only when the checks at lines 1 and 6 are false and true, respectively. The real process of setting up and maintaining a `physicalConnection` object is rather complex, so mocking its behavior makes it easier to get the required values. We use `mockito` [3], a popular Java mocking framework, to mock the method call at line 1 to return false directly without calling the real method. The method call at line 6 always return false, so we need no mocking for it.

Strictly speaking, neither instrumentation nor mocking can guarantee deterministic triggering of a bug on every run. But, in practice, using one of them is often effective to help reproduce bugs deterministically. There is only one bug that we could not reproduce very deterministically. That is the SC violation bug, where we cannot easily enforce the execution sequence of statements in one thread at runtime. In addition, any insertion or deletion of statements may affect the behavior of the Java memory model, making the reproduced bug to not be the original one. In this case, we can only execute the test case for a large number of times to reproduce the bug. In particular, using the test case from JaConTeBe, the SC violation bug `GROOVY-5198` can be reproduced within seconds on a typical JVM.

In summary, each bug in JaConTeBe is accompanied by a test case and a description of the bug. Most test cases (36 out of 47) can reproduce the bug deterministically on every run. To achieve that, 6 test cases use mocking, and 3 test cases use instrumentation; no test case uses both mocking and instrumentation. The other 11 test cases need several runs to trigger the bug, and one of those test cases uses mocking; these test cases could have used more mocking or instrumentation to reproduce the bug faster, but it takes only a few seconds for multiple runs to trigger the bug, so it was not worthwhile to make them more deterministic. Only one test case (for the bug `GROOVY-5198`) requires many more runs, as described in more detail in Section IV-B. In addition, we design a framework for the JaConTeBe benchmark suite to provide the common utilities for running the buggy programs and the capability of executing the bugs in their expected scenarios such as controlling the threads, terminating deadlocks, and enforcing thread interleavings.

## IV. EVALUATION

To evaluate JaConTeBe, we perform small experiments with three approaches for detecting Java concurrency bugs: the Java PathFinder (JPF) research tool[6] [79], the RV-Predict commercial tool[7] [39], and our implementation of the CheckMate approach [45]. (Both RV-Predict and CheckMate were proposed in the papers we have surveyed in Section II.) We also run these three implementations on *SIR-old-conc*, which is how we call the old 36 concurrency bugs in SIR[8]) now that JaConTeBe is also a part of SIR. 5 bugs overlap in JaConTeBe and SIR-old-conc, belonging to the Apache Log4j and Apache Pool projects.

We run JPF on all concurrency bugs in JaConTeBe (47 bugs) and SIR-old-conc (36 bugs). We run JPF V7 and JDK 1.7 whenever possible, but JaConTeBe has 14 JDK6 bugs, and JPF V7 is incompatible with lower JDK versions, so we also run JPF V6 and JDK 1.6.0 on these JDK6 bugs. We run RV-Predict 1.4 on the buggy programs known to contain data races: 12 bugs from JaConTeBe (excluding 10 OpenJDK6 and OpenJDK7 bugs because they are incompatible with RV-Predict's requirement of JRE 8) and 17 bugs from SIR-old-conc (excluding atomicity violations that RV-Predict does not detect). We run CheckMate on all deadlock bugs in JaConTeBe (25 bugs) and SIR-old-conc (15 bugs). 3 programs in SIR-old-conc contain both non-deadlock and deadlock bugs that can be triggered by the same test case via different schedules, so we put these programs into non-deadlocks and run on RV-Predict (because detecting a deadlock that actually happened is rather obvious when a program hangs).

We run both JPF and RV-Predict "out-of-the-box", only with their default configurations, except that for JPF we change the option `search.multiple_errors` from default false to true, which instructs JPF to search for multiple bugs in one run. Our CheckMate implementation has no configuration options. The results could differ for different configurations of these tools or for different tools.

### A. Overall Results

Table V summarizes the results. For each approach, we tabulate the numbers of bugs it successfully detects, it fails to detect, and where it crashes. "JaC" is short for JaConTeBe, "SIR" is short for SIR-old-conc, and "JaC+SIR" has the first value for JaConTeBe and the second for SIR-old-conc.

JPF detects all 36 bugs in SIR-old-conc but does not detect most bugs in JaConTeBe. For 10 bugs, JPF terminates exploration but fails to detect the bug, *i.e.*, either reports

[6] http://babelfish.arc.nasa.gov/trac/jpf/

[7] https://runtimeverification.com/predict/

[8] http://sir.unl.edu/portal/bios/concurrency.php

only other bugs not the expected one (although we set the option search.multiple_errors) or does not report any bug. For 26 bugs, JPF conceptually crashes, *i.e.*, does not terminate exploration, mainly due to unsupported (native) methods. JPF is a Java virtual machine implemented in Java itself and requires special implementation for native methods. While JPF implements many native methods, it does not implement all the native methods required by the JVM specification [51]. When the code invokes an unimplemented native method, JPF terminates execution and reports an error message such as "cannot find native (method name)". Some seeming implementations of JDK methods in JPF are also incomplete, *e.g.*, JPF implements java.lang.Class.getProtectionDomain by throwing UnsupportedOperationException whenever invoked. As the test cases in JaConTeBe run unmodified, real-world projects, some of them invoke native and JDK methods unimplemented in JPF, making JPF crash. These results confirm a known limitation of JPF with native methods [75] and show that handling unmodified real code (in JaConTeBe) is different from handling carefully prepared simplified examples (in SIR-old-conc).

RV-Predict is a commercial tool that evolved from a long line of research, with the most recent result by Huang *et al.* [39]. RV-Predict is designed to detect data races for complex applications. It succeeds to detect 10 bugs in JaConTeBe and 17 in SIR-old-conc. The two missed bugs from JaConTeBe are in the java.* packages that RV-Predict by default does not check.

CheckMate is chosen among various deadlock detection approaches because it is designed to detect both communication deadlocks and resource deadlocks (unlike many other approaches focusing on resource deadlocks only). We implement CheckMate following the original description [45] and run it against all deadlock bugs in JaConTeBe and SIR-old-conc. While CheckMate finds almost all deadlocks in SIR-old-conc (and the 2 missed bugs are real-world bugs), it fails to find over two thirds of deadlock bugs in JaConTeBe. Table VI compares the results of running CheckMate on the projects that overlap in JaConTeBe and the original evaluation of CheckMate [45]. For each bug, we tabulate the result of CheckMate on the test case from JaConTeBe, the bug's deadlock type, and the corresponding project version. The last column shows the number of deadlocks detected in the corresponding project in the original evaluation [45] that does not report all bug IDs but only reports how many deadlocks are found in which version of which project. (The evaluation does mention two bug IDs, LOG4J-38137 and GROOVY-1890; we have LOG4J-38137 in Table VI, and Section III-C describes why GROOVY-1890 is not selected.) The results indicate that JaConTebe finds more limitations in CheckMate than SIR-old-conc finds.

### B. Most challenging bugs in JaConTeBe

During the evaluation, we found two particularly challenging bugs: native deadlock and SC violation.

**Native deadlock.** Most resource and communication deadlocks in Java code are caused by the synchronization at the pure Java level. We find a special subtype of resource deadlocks caused by the native locks acquired in the native methods. We refer to this type of deadlock as *native deadlock*; it often mixes execution from two different levels [16], the Java level and the native C/C++ level. As any ordinary resource deadlock, a native deadlock is caused by cyclic locking. However, unlike an ordinary resource deadlock, a native deadlock has some native C/C++ lock(s) acquired by native methods and potentially some Java lock(s) acquired by Java methods. For example, a native deadlock occurs when a thread holds a native lock and acquires a Java lock, while another thread holds the Java lock and waits for the native lock.

We find only one native deadlock in the 8 projects currently in JaConTeBe, but there are likely more such deadlocks in other projects, and they are still harmful and can hang the program. Moreover, they are difficult for developers to even comprehend, *e.g.*, consider a Stackoverflow question[9] about a deadlock caused by a native method. JVM does not provide information about acquisition of native locks in the stacktraces.

The bug GROOVY-4736 in JaConTeBe is a native deadlock bug from Groovy 1.7.9. Fig. 7 shows its simplified stacktrace. The thread dump of Compiling and instantiation 2 shows literately no lock acquisition, but the deadlock information shows the thread Compiling and instantiation 3 waiting for a lock held by thread Compiling and instantiation 2. In fact, the native lock is held in the native method forName0() by the thread Compiling and instantiation 2 and attempted to be acquired in the native method getDeclaredFields0() by the other thread. Therefore, the deadlock is formed by a native lock and a Java lock.

Such native deadlocks are difficult to detect and reproduce, and both JPF and CheckMate fail to detect the deadlock in GROOVY-4736. Static analysis usually considers native methods as "black box" [50] and has a limited support for them [67]. Dynamic deadlock prediction (*e.g.*, [45]) analyzes the execution trace to find potential deadlocks, but the acquisition of locks in native methods is not visible in the program execution trace at the Java level. Similarly, reproduction techniques (*e.g.*, [83]) need to know where the lock has been acquired and released. The bug GROOVY-4736 could motivate a bigger study of native deadlocks in Java.

**SC violation.** JMM [56] allows statement reordering that can be surprising to the developer and lead to bugs. Fig. 8 shows the example bug GROOVY-5198 where two threads execute the same method. If thread 1 executes line 11 after line 10, then thread 2 passes the condition checks at line 4 and returns the value at line 6 without problems. However, JMM allows thread 1 to execute line 11 *before* line 10, and if thread 2 executes in between those lines, it completes the check at line 4 and can return an unintended null value at line 6. Different from order-violation bugs [54] caused by a nondeterministic execution order of statements in two threads, SC violation is caused by the relaxed memory consistency model.

### C. Comparison with other benchmark suites

Compared with the currently used benchmark suites for evaluating bug detection for Java—JPG, DaCapo, IBM, and

[9] http://stackoverflow.com/questions/37551/multiple-threads-stuck-in-native-calls-java

```
"Compiling and instantiation 3": waiting to lock monitor
    0x000000000cc5b3a8 (object 0x00000007d6ce9040,
  a groovy.lang.GroovyClassLoader$InnerLoader), which
      is held by "Compiling and instantiation 2"
"Compiling and instantiation 2": waiting to lock monitor
    0x000000000cc5bc98 (object 0x00000007d7da8070,
  a java.util.HashMap),which is held by "Compiling and
      instantiation 3"

Java stack information for the threads listed above:
===================================================
"Compiling and instantiation 3":
    at java.lang.Class.getDeclaredFields0(Native Method)
    ...
    at groovy.lang.GroovyClassLoader.recompile(
        GroovyClassLoader.java:777)
    - locked <0x00000007d7da8070> (a java.util.HashMap)
    ...
    at Groovy4736$3.run(Groovy4736.java:169)

"Compiling and instantiation 2":
    at groovy.lang.GroovyClassLoader.recompile(
        GroovyClassLoader.java:776)
    - waiting to lock <0x00000007d7da8070> (a java.util.
        HashMap)
    at groovy.lang.GroovyClassLoader.loadClass(
        GroovyClassLoader.java:737)
    at groovy.lang.GroovyClassLoader$InnerLoader.
        loadClass(GroovyClassLoader.java:449)
    at groovy.lang.GroovyClassLoader.loadClass(
        GroovyClassLoader.java:793)
    at java.lang.ClassLoader.loadClass(ClassLoader.java
        :357)
    at java.lang.Class.forName0(Native Method)
    at java.lang.Class.forName(Class.java:190)
    at test.D1.class$(test.D1)
    at test.D1.$get$$class$test$E3(test.D1)
    at test.D1.<init>(test.D1:4)
    at sun.reflect.NativeConstructorAccessorImpl.
        newInstance0(Native Method)
    at sun.reflect.NativeConstructorAccessorImpl.
        newInstance(NativeConstructorAccessorImpl.java
        :57)
    at sun.reflect.DelegatingConstructorAccessorImpl.
        newInstance(DelegatingConstructorAccessorImpl.
        java:45)
    at java.lang.reflect.Constructor.newInstance(
        Constructor.java:526)
    at Groovy4736$3.run(Groovy4736.java:169)
```

Fig. 7. Stacktrace of the native deadlock bug `GROOVY-4736`.

```
1  public MetaMethod retrieveStaticMethod(String
       methodName, Object[] arguments) {
2      ...
3      cacheEntry = e.cachedStaticMethod;
4      if (cacheEntry != null && MetaClassHelper.
           sameClasses(cacheEntry.params, arguments, e.
           staticMethods instanceof MetaMethod))
5      {
6          return cacheEntry.method;
7      }
8      ...
9      cacheEntry.params = classes;
10     cacheEntry.method = pickStaticMethod(methodName,
           classes);
11     e.cachedStaticMethod = cacheEntry;
12     return cacheEntry.method;
13 }
```

Fig. 8. A SC violation example, `GROOVY-5198`. JMM allows the sequence T1.L11→T2.L4→T2.L6→T1.L10 that can return unexpected `null`.

SIR—JaConTeBe provides more bugs, from real open-source projects, with test cases for reproduction, and with detailed documentation of root causes and buggy interleavings.

TABLE VII. COMPOSITION OF SIR-OLD-CONC BENCHMARKS.

| Source | #Programs | Average Size (LOC) |
|---|---|---|
| IBM benchmark suite | 10 | 86.8 |
| Newly designed | 7 | 121.0 |
| Used by research papers | 8 | 495.5 |
| Real-world applications | 11 | 4043.6 |

Although JPG and DaCapo are frequently used in evaluations, they were not initially designed for the purpose of concurrency bug detection. Both consist of a set of multithreaded applications whose concurrency bugs were unknown. As a result, the effectiveness of evaluated techniques may not be precisely calculated when these benchmarks are used. In contrast, JaConTeBe is built up from real-world buggy projects, where the symptom, type, and root cause of each bug have been clearly stated, and a test case is provided. Researchers can use JaConTeBe to evaluate the effectiveness of their approaches in handling real software systems.

Unlike JPG and DaCapo, IBM and SIR benchmark suites provide documented buggy programs and tests to reproduce them. However, 24 out of 41 programs in IBM are written by undergraduate students, 1 is from a real-world application, and the others are open-source programs. The 36 buggy programs in SIR are from 4 sources: bugs from IBM benchmarks, bugs designed based on several bug patterns [24], [25], bugs used by other studies [19], [65], and real-world bugs, as shown in Table VII. These buggy programs cover traditional pedagogical concurrency programs (*e.g.*, dining philosophers and sleeping barbers) and typical concurrency bug idioms (*e.g.*, two-stage locking). Although both IBM and SIR cover various types of concurrency bugs with good diversity, the small code size is insufficient to show the limitations of some approaches, *e.g.*, how model checking can encounter the state-space explosion problem with larger scale. JaConTeBe complements SIR in that it provides more real-world bugs, and JaConTeBe has been accepted as a part of SIR.

## V. DISCUSSION AND FUTURE WORK

**Enriching JaConTeBe with more concurrency bugs.** While we have inspected hundreds of bug reports (potentially) related to concurrency in the bug databases of 8 projects, our inspection is still incomplete. For example, a bug report may not mention any keywords that we queried, even if it is about a concurrency bug. Specifically, `DBCP-379`[10] has been confirmed as a concurrency bug, but it does not mention our keywords. Therefore, we need to improve our searching method, trying to include more true positive matches without substantially enlarging false positives. Additionally, we inspected only 7 of the 13 projects from the surveyed papers (Section III-B), so we plan to inspect the other 6—Apache Xalan, Apache Tomcat, Apache FTP Server, Eclipse, HSQLDB, and PMD—for the future JaConTeBe versions. Moreover, we also plan to analyze more concurrency bugs from other projects to further enrich JaConTeBe.

**Evaluating the effectiveness of JaConTeBe.** We have compared the results of running three approaches against JaConTeBe and SIR-old-conc to determine if JaConTeBe can confirm some known limitations of these approaches. We did not use the IBM benchmark suite for comparison because it is no longer available on its web site, and its current maintainer

[10] https://issues.apache.org/jira/browse/DBCP-379

provided us a distribution that contains only some of the programs from the benchmark suite. We plan to evaluate more detection approaches and tools in the future.

**The effectiveness of real bugs in benchmarks.** Whether (or when) artificial faults can replace real bugs in software testing research is an open question. Recent research [47] finds that the elaborately designed mutants "can be used as a substitute for real faults when comparing (generated) test suites" [47], *i.e.*, the substitution equivalence is conditional on having appropriate mutants. Somewhat older research [32], [85] finds that, for typical mutation operators, operator-based mutant selection may not be better than random mutant selection in several cases, increasing the importance of selecting proper mutants that can represent real bugs.

**The importance of practical testing scenarios.** JaConTeBe follows the methodology established by prior benchmark suites to provide test cases that focus on the bugs and can (almost) deterministically reproduce them with relatively short executions. However, in an actual practical usage of a tool for detection of concurrency bugs, the developers would not have such test cases. In fact, they may even have no test cases (with both program inputs and expected outputs) but only have some program inputs, and/or they may have much longer executions, not focused on the bug. In the future, it would be important to study not only what bugs can be found by various approaches with focused test cases but also how developers actually use the approaches/tools, e.g., following the work of Parnin and Orso on analyzing automated debugging [64].

## VI. Related Work

As a benchmark suite of real-world Java concurrency bugs, JaConTeBe is related to several lines of research.

**Benchmark guidelines.** As pointed out by Bird *et al.* [6], the quality of a benchmark can substantially impact the results of evaluating an approach. To prepare successful benchmarks, Sim *et al.* [76] proposed seven guidelines, and Lu *et al.* [53] further distilled them into five guidelines. Section II-A presents a survey of four existing benchmark suites of concurrency bugs; comparing to these benchmark suites, our JaConTeBe benchmark suite has the following benefits: (1) it has clearly documented root causes for bugs; (2) all bugs are from real-world open-source applications; (3) many of our bugs are difficult to detect; and (4) it includes tests and scripts to reproduce the concurrency bugs deterministically and easily.

Bradbury *et al.* [10] propose a methodology for creating a benchmark suite based on a combinatorial test-design technique. Specifically, they design a matrix that covers 44 combinations of 7 benchmark metrics, and suggest to fill the matrix with programs from three sources: existing benchmarks such as 28 programs from the IBM benchmark suite, bugs from open-source repositories, and artificial mutants. They do not provide a benchmark suite covering the full matrix.

**Bug detection approaches.** Lu *et al*. [54] classify concurrency bugs into deadlock bugs and non-deadlock bugs, and further classify the latter into atomicity violations, order violations, and other bugs. Numerous approaches have been proposed for detecting such bugs. For deadlocks, static approaches (*e.g.*, [59]) typically analyze code for cyclic use, and dynamic approaches (*e.g.*, [21], [71]) analyze execution traces to locate potential deadlocks. As some reports can be false alarms, Cai *et al.* [14] propose an approach that validates detected deadlocks dynamically. For non-deadlock concurrency bugs, such as atomicity violations and races, static approaches (*e.g.*, [58], [80]) analyze code, and dynamic approaches (*e.g.*, Falcon [63]) analyze runtime behavior to check whether threads can access the same memory locations. Most approaches have to predefine the nature of target bugs. For example, FindBugs [2], [37] defines 46 multi-threaded bug patterns, and Farchi *et al.* [25] summarized 8 patterns from Java concurrency bugs, with regard to how bugs are manifested. All these studies reveal some aspects of concurrency bugs, and help us to prepare our benchmark suite. Moreover, while collecting the bugs for JaConTeBe, we also find some less studied concurrency bugs, complementing the above studies.

**Concurrency bug reproduction.** Huang *et al.* [38] proposes LEAP that monitors and records shared-memory dependencies at runtime and replays concurrency bugs deterministically. Typically, record-and-replay suffers from side effects, because instrumented code may change the behavior of the original code. To address the problem, researchers (*e.g.*, [69], [83]) explore offline approaches that reproduce the concurrency bugs based on execution traces and bug reports. Furthermore, hybrid approaches (*e.g.*, CLAP [41]) record little information at runtime and search for executions that reproduce buggy behaviors. Tools such as CalFuzzer [44], Concurrit [20], and IMUnit [42] allow programmers to manipulate thread interleavings to reproduce buggy behaviors. We manually prepare a test case for each bug in JaConTeBe, but the mentioned approaches could reduce the effort for preparing such test cases when we enlarge JaConTeBe in the future.

## VII. Conclusion

We have conducted a survey of the existing benchmarks used for evaluating concurrency bug detection approaches in Java, and we have prepared JaConTeBe, a new benchmark suite of Java concurrency bugs. JaConTeBe currently contains 47 real-world bugs from 8 practical software projects. We believe that JaConTeBe can help researchers to evaluate the effectiveness of their bug detection approaches. The experiment results and survey datasets are available at http://stap.sjtu.edu.cn/index.php?title=JaConTeBe.

## Acknowledgments

We thank Xinxi Chen and Fei Luo for helping us to analyze JaConTeBe concurrency bugs, Wayne Motycka and Gregg Rothermel for inspecting JaConTeBe and accepting it as a part of SIR, Yilong Li for help with RV-Predict, and Rachel Tzoref-Brill for providing the IBM benchmark suite. This research was sponsored in part by 973 Program in China (Grant No. 2015CB352203), the National Nature Science Foundation of China (Grant Nos. 9111800420 and 612721020), and the US NSF (Grant Nos. CCF-1012759 and CCF-1438982). Ziyi Lin and Yuting Chen are supported by China Scholarship Council and partially supported by NSFC (Grant Nos. 61472242 and 61572312), and Hao Zhong is partially supported by NSFC (Grant No. 61572313).

# REFERENCES

[1] ASM homepage.[online].available: http://asm.ow2.org.

[2] FindBugs website.[online].available: http://findbugs.sourceforge.net/bugDescriptions.html.

[3] Mockito homepage.[online].available: http://site.mockito.org.

[4] S. Bensalem and K. Havelund. Dynamic deadlock analysis of multi-threaded programs. In *HVC*, 2005.

[5] S. Bindal, S. Bansal, and A. Lal. Variable and thread bounding for systematic testing of multithreaded programs. In *ISSTA*, 2013.

[6] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced?: Bias in bug-fix datasets. In *ESEC/FSE*, 2009.

[7] S. Biswas, J. Huang, A. Sengupta, and M. D. Bond. DoubleChecker: Efficient sound and precise atomicity checking. In *PLDI*, 2014.

[8] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. L. Hosking, M. Jump, H. B. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, 2006.

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

[10] J. S. Bradbury, I. Segall, E. Farchi, K. Jalbert, and D. Kelk. Using combinatorial benchmark construction to improve the assessment of concurrency bug detection tools. In *PADTAD*, 2012.

[11] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A benchmark suite for high performance Java. *Concurrency - Practice and Experience*, 12(6):375–388, 2000.

[12] J. Burnim, T. Elmas, G. C. Necula, and K. Sen. NDSeq: Runtime checking for nondeterministic sequential specifications of parallel correctness. In *PLDI*, 2011.

[13] Y. Cai and W. Chan. Magiclock: Scalable detection of potential deadlocks in large-scale multithreaded programs. *IEEE Transactions on Software Engineering*, 40(3):266–281, 2014.

[14] Y. Cai, S. Wu, and W. K. Chan. ConLock: A constraint-based approach to dynamic checking on deadlocks in multithreaded programs. In *ICSE*, 2014.

[15] J. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI*, 2002.

[16] M. d'Amorim, A. Sobeih, and D. Marinov. Optimized execution of deterministic blocks in Java PathFinder. In *ICFEM*, 2006.

[17] D. Dimitrov, V. Raychev, M. T. Vechev, and E. Koskinen. Commutativity race detection. In *PLDI*, 2014.

[18] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[19] M. B. Dwyer, S. Person, and S. Elbaum. Controlling factors in evaluating path-sensitive error detection techniques. In *FSE*, 2006.

[20] T. Elmas, J. Burnim, G. Necula, and K. Sen. CONCURRIT: A domain specific language for reproducing concurrency bugs. In *PLDI*, 2013.

[21] M. Eslamimehr and J. Palsberg. Sherlock: Scalable deadlock detection for concurrent programs. In *FSE*, 2014.

[22] Y. Eytani, K. Havelund, S. D. Stoller, and S. Ur. Towards a framework and a benchmark for testing tools for multi-threaded programs. *Concurrency and Computation: Practice and Experience*, 19(3):267–279, 2007.

[23] Y. Eytani, R. Tzoref, and S. Ur. Experience with a concurrency bugs benchmark. In *TESTBENCH*, 2008.

[24] Y. Eytani and S. Ur. Compiling a benchmark of documented multi-threaded bugs. In *PADTAD*, 2004.

[25] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *IPDPS*, 2003.

[26] A. Farzan, P. Madhusudan, N. Razavi, and F. Sorrentino. Predicting null-pointer dereferences in concurrent programs. In *FSE*, 2012.

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

[28] C. Flanagan and S. N. Freund. Adversarial memory for detecting destructive races. In *PLDI*, 2010.

[29] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. In *PLDI*, 2008.

[30] M. K. Ganai. Scalable and precise symbolic analysis for atomicity violations. In *ASE*, 2011.

[31] M. K. Ganai. Efficient data race prediction with incremental reasoning on time-stamped lock history. In *ASE*, 2013.

[32] M. Gligoric, L. Zhang, C. Pereira, and G. Pokam. Selective mutation testing for concurrent code. In *ISSTA*, 2013.

[33] G. Golan-Gueta, G. Ramalingam, M. Sagiv, and E. Yahav. Concurrent libraries with foresight. In *PLDI*, 2013.

[34] C. Hammer, J. Dolby, M. Vaziri, and F. Tip. Dynamic detection of atomic-set-serializability violations. In *ICSE*, 2008.

[35] S. Hong, J. Ahn, S. Park, M. Kim, and M. J. Harrold. Testing concurrent programs to achieve high synchronization coverage. In *ISSTA*, 2012.

[36] S. Hong and M. Kim. A survey of race bug detection techniques for multithreaded programmes. *Software Testing, Verification and Reliability*, 25(3):191–217, 2015.

[37] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *OOPSLA*, 2004.

[38] J. Huang, P. Liu, and C. Zhang. LEAP: Lightweight deterministic multi-processor replay of concurrent Java programs. In *FSE*, 2010.

[39] J. Huang, P. O. Meredith, and G. Rosu. Maximal sound predictive race detection with control flow abstraction. In *PLDI*, 2014.

[40] J. Huang and C. Zhang. Persuasive prediction of concurrency access anomalies. In *ISSTA*, 2011.

[41] J. Huang, C. Zhang, and J. Dolby. CLAP: Recording local executions to reproduce concurrency failures. In *PLDI*, 2013.

[42] V. Jagannath, M. Gligoric, D. Jin, Q. Luo, and G. Rosu. Improved multithreaded unit testing. In *ESEC/FSE*, 2011.

[43] V. Jagannath, Q. Luo, and D. Marinov. Change-aware preemption prioritization. In *ISSTA*, 2011.

[44] P. Joshi, M. Naik, C.-S. Park, and K. Sen. CalFuzzer: An extensible active testing framework for concurrent programs. In *CAV*, 2009.

[45] P. Joshi, M. Naik, K. Sen, and D. Gay. An effective dynamic analysis for detecting generalized deadlocks. In *FSE*, 2010.

[46] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *PLDI*, 2009.

[47] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *FSE*, 2014.

[48] Z. Lai, S. Cheung, and W. K. Chan. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In *ICSE*, 2010.

[49] Z. Letko, T. Vojnar, and B. Křena. AtomRace: Data race and atomicity violation detector and healer. In *PADTAD*, 2008.

[50] S. Li, Y. D. Liu, and G. Tan. JATO: Native code atomicity for Java. In *APLAS*, 2012.

[51] T. Lindholm and F. Yellin. *Java virtual machine specification*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[52] P. Liu, J. Dolby, and C. Zhang. Finding incorrect compositions of atomicity. In *ESEC/FSE*, 2013.

[53] S. Lu, Z. Li, F. Qin, L. Tan, and P. Zhou. BugBench: Benchmarks for evaluating bug detection tools. In *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

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

[55] B. Lucia, B. P. Wood, and L. Ceze. Isolating and understanding concurrency errors using reconstructed execution fragments. In *PLDI*, 2011.

[56] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL*, 2005.

[57] D. Marino, C. Hammer, J. Dolby, M. Vaziri, F. Tip, and J. Vitek. Detecting deadlock in programs with data-centric synchronization. In *ICSE*, 2013.

[58] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *PLDI*, 2006.

[59] M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *ICSE*, 2009.

[60] A. Nistor, Q. Luo, M. Pradel, T. R. Gross, and D. Marinov. Ballerina: Automatic generation and clustering of efficient random unit tests for multithreaded code. In *ICSE*, 2012.

[61] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *FSE*, 2008.

[62] S. Park. Debugging non-deadlock concurrency bugs. In *ISSTA*, 2013.

[63] S. Park, R. W. Vuduc, and M. J. Harrold. Falcon: Fault localization in concurrent programs. In *ICSE*, 2010.

[64] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *ISSTA*, 2011.

[65] C. S. Păsăreanu, M. B. Dwyer, and W. Visser. Finding feasible abstract counter-examples. *International Journal on Software Tools for Technology Transfer*, 5(1):34–48, 2003.

[66] W. Pugh and T. Lindholm. JSR-133: Java memory model and thread specification. *Java Community Process*, 2004.

[67] C. Radoi and D. Dig. Practical static race detection for Java parallel loops. In *ISSTA*, 2013.

[68] R. Raman, J. Zhao, V. Sarkar, M. T. Vechev, and E. Yahav. Scalable and precise dynamic datarace detection for structured parallelism. In *PLDI*, 2012.

[69] J. Rößler, A. Zeller, G. Fraser, C. Zamfir, and G. Candea. Reconstructing core dumps. In *ICST*, 2013.

[70] M. Samak and M. K. Ramanathan. Omen+: A precise dynamic deadlock detector for multithreaded Java libraries. In *FSE*, 2014.

[71] M. Samak and M. K. Ramanathan. Trace driven dynamic deadlock detection and reproduction. In *PPOPP*, 2014.

[72] K. Sen. Effective random testing of concurrent programs. In *ASE*, 2007.

[73] K. Sen. Race directed random testing of concurrent programs. In *PLDI*, 2008.

[74] O. Shacham, E. Yahav, G. Golan-Gueta, A. Aiken, N. G. Bronson, M. Sagiv, and M. T. Vechev. Verifying atomicity via data independence. In *ISSTA*, 2014.

[75] N. Shafiei and F. van Breugel. Automatic handling of native methods in Java PathFinder. In *SPIN*, 2014.

[76] S. E. Sim, S. M. Easterbrook, and R. C. Holt. Using benchmarking to advance research: A challenge to software engineering. In *ICSE*, 2003.

[77] L. A. Smith, J. M. Bull, and J. Obdrizalek. A parallel Java grande benchmark suite. In *Supercomputing*, 2001.

[78] F. Sorrentino, A. Farzan, and P. Madhusudan. PENELOPE: Weaving threads to expose atomicity violations. In *FSE*, 2010.

[79] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[80] J. W. Voung, R. Jhala, and S. Lerner. RELAY: Static race detection on millions of lines of code. In *ESEC/FSE*, 2007.

[81] J. Yi, T. Disney, S. N. Freund, and C. Flanagan. Cooperative types for controlling thread interference in Java. In *ISSTA*, 2012.

[82] T. Yu, W. Srisa-an, and G. Rothermel. SimRT: An automated framework to support regression testing for data races. In *ICSE*, 2014.

[83] C. Zamfir and G. Candea. Execution synthesis: A technique for automated software debugging. In *Eurosys*, 2010.

[84] K. Zhai, B. Xu, W. K. Chan, and T. H. Tse. CARISMA: A context-sensitive approach to race-condition sample-instance selection for multithreaded applications. In *ISSTA*, 2012.

[85] L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei. Is operator-based mutant selection superior to random mutant selection? In *ICSE*, 2010.