SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2021; **00**:1–40
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

Statically Driven Generation of Concurrent Tests for Thread-Safe Classes

Valerio Terragni[1*] and Mauro Pezzè[2,3]

[1] *University of Auckland, Auckland, New Zealand*
[2] *Università della Svizzera italiana, Lugano, Switzerland*
[3] *Schaffhausen Institute of Technology, Schaffhausen, Switzerland*

SUMMARY

Concurrency testing is an important activity to expose concurrency faults in thread-safe classes. A concurrent test for a thread-safe class is a set of method call sequences that exercise the public interface of the class from multiple threads. Automatically generating fault-revealing concurrent tests within an affordable time-budget is difficult due to the huge search space of possible concurrent tests.

In this paper, we present DEPCON$^+$, a novel approach that reduces the search space of concurrent tests by leveraging statically computed dependencies among public methods. DEPCON$^+$ exploits the intuition that concurrent tests can expose thread-safety violations that manifest exceptions or deadlocks, only if they exercise some specific method dependencies. DEPCON$^+$ provides an efficient way to identify such dependencies by statically analyzing the code, and relies on the computed dependencies to steer the test generation towards those concurrent tests that exhibit the computed dependencies.

We developed a prototype DEPCON$^+$ implementation for Java, and evaluated the approach on 19 known concurrency faults of thread-safe classes that lead to thread-safety violations of either exception or deadlock type. The results presented in this paper show that DEPCON$^+$ is more effective than state-of-the-art approaches in exposing the concurrency faults. The search space pruning of DEPCON$^+$ dramatically reduces the search space of possible concurrent tests, without missing any thread-safety violations.

Received . . .

KEY WORDS: Test Generation, Thread-safety, Concurrency, Non-Determinism, Race Conditions, Deadlocks, Java, Object Oriented Programs

1. INTRODUCTION

Concurrent programming allows developers to take advantage of widely-spread multi-core and parallel architectures. Synchronizing concurrent threads can be extremely complex, and can lead to subtle concurrency failures that are hard to reveal and prevent [1]. The incorrect synchronization of concurrent threads that access shared memory locations may lead to race conditions, atomicity violations and deadlocks. Race conditions and atomicity violations occur when different threads access shared memory locations in a wrong order, and produce erroneous results. Deadlocks occur when different threads acquire shared locks in a wrong order, and cannot progress.

Concurrent object-oriented programs often rely on thread-safe classes, which encapsulate most synchronization-related challenges [2, 3, 4]. Thread-safe classes guarantee that their synchronization

*Correspondence to: Valerio Terragni, Department of Electrical, Computer and Software Engineering, Engineering Block 5 - Bldg 405, Level 6, Room 663, 5 Grafton Rd, Auckland Central, Auckland, 1010, New Zealand. E-mail: v.terragni@auckland.ac.nz

mechanism prevents multiple threads incorrectly access shared instances of the class [5]. Concurrency faults in thread-safe classes lead to thread-safety violations, which can cause serious issues in concurrent programs that rely on such classes. Preventing failures in programs that handle concurrency with thread-safe classes amounts to prevent thread-safety violations in the thread-safe classes [6, 7]. Exposing thread-safety violations is difficult because they often manifest non-deterministically, under specific execution orders of shared memory accesses (thread interleavings) [8, 9].

An effective approach to expose thread-safety violations is the automated generation of concurrent tests [3, 8, 9, 10, 11, 12]. Concurrent tests expose thread-safety violations by exploring different combinations of test inputs and thread interleavings [10]. Concurrent tests for thread-safe classes consist of multiple threads that concurrently invoke public methods on shared instances of the class under test [3, 12, 13]. Figure 2 on page 7 shows a concurrent test for BufferedInputStream, a JDK (buggy) class shown in Figure 1.

The combinatorial explosion of both test inputs and thread interleavings makes it extremely difficult to identify critical combinations of inputs and interleavings that expose thread-safety violations [3, 8, 9, 10]. Approaches to automatically generate concurrent tests aim to identify combinations of method call invocations that can expose thread-safety violations [8, 9, 10, 14]. Finding such combinations is challenging because there exists a myriad of possible concurrent tests for any non-trivial thread-safe class. Generating a large number of concurrent tests is often infeasible because of the high cost of exploring the interleaving space of many concurrent tests [12, 15, 16]. Popular approaches address this challenge with coverage-driven test generation strategies that steer test generation towards not-yet-explored interleavings, thus avoiding the high cost of exploring the interleaving space of redundant tests [8, 9, 10, 11].

Recently, we provide empirical evidence that many concurrent tests that increase interleaving coverage cannot expose thread-safety violations, and proposed the concurrent test generator *DEPendency-driven CONcurrency Testing* (DEPCON) [17]. DEPCON exploits the intuition that only methods that can both interleave and access the same shared memory locations (*parallel* and *conflict* dependent, respectively) can lead to (non-deadlock) thread-safety violations when executed concurrently. DEPCON improves the effectiveness of concurrent test generation because it avoids generating and exploring the interleaving space of concurrent tests that do not manifest both parallel and conflict dependencies.

DEPCON statically analyses the class under test to compute a summary of the public methods of the class (*pre-processing phase*). These summaries contain information about both the shared-memory accesses and the synchronization operations of the corresponding methods. DEPCON uses such summaries to compute the parallel and conflict dependencies among the public methods of the class under test. The results that we report in our ICST paper [17] show that DEPCON dramatically reduces the number of generated tests as much as 35.35× (7.17× on average), and exposes thread-safety violations as much as 33.50× (4.87× on average) faster than state-of-the-art approaches [8]. The results also show that the pre-processing phase is efficient (it requires less than two seconds on average) and complete (it does not prevent the detection of any thread-safety violation).

A major limitation of DEPCON is that it targets only concurrency faults in thread-safe classes that manifest thread-safety violations as runtime exceptions [17], namely race conditions and atomicity violations. DEPCON ignores resource deadlocks, which is an important class of concurrency faults that manifest thread-safety violations with endless hangs [18]. A deadlock occurs when the progress of a program is halted as a running thread attempts to acquire a lock already held by another thread [19]. DEPCON is inadequate to expose thread-safety violations of resource deadlock type for two reasons. First, the parallel and conflict dependencies that we presented in the earlier version of this work do not capture the deadlock condition. Second, DEPCON generates concurrent tests with exactly one instance of the class under test shared among the concurrent threads. While this assumption is reasonable for race conditions and atomicity violations [17], it is not adequate for deadlocks. In fact, to expose deadlocks, it is often necessary to share different instances of the class under test among the concurrent threads [19, 20].

In this paper, we present DEPCON$^+$, a new approach to automatically generate concurrent tests to reveal deadlocks. DEPCON$^+$ extends the initial idea of DEPCON of pruning the search space

DOI: 10.1002/stvr

with parallel and conflict dependency analysis, by proposing a new dependency analysis that we call *double-lock* dependency analysis. This analysis captures the necessary (but not sufficient) condition of deadlock in thread-safe classes: the concurrent execution of methods can acquire the same pair of locks in the opposite order. We present an effective and efficient way to statically compute double-lock dependencies among methods, and show how DEPCON$^+$ steers the concurrent test generation towards concurrent tests that manifest the computed double-lock dependencies. We carefully design DEPCON$^+$ to ensure that the generated tests share multiple instances of the class under test, and properly exercise the circumstances for deadlocks (as captured by the double-lock dependency).

We evaluated DEPCON$^+$ on four known deadlock faults in popular thread-safe classes. Our results show that DEPCON$^+$ can effectively expose the deadlocks within 15 seconds on average, while the state-of-that-art techniques COVCON [8] and DEPCON cannot expose the deadlocks with a time-budget of one hour.

This paper contributes to the state of the art by presenting

(i) a holistic approach called DEPCON$^+$ that combines static parallel and conflict dependency analysis with double-lock analysis to generate tests that efficiently target both data races and deadlocks in thread-safe classes,

(ii) the insight that an efficient static computation of parallel, conflict and double lock dependencies can effectively steer dynamic test generation towards thread-safety violations,

(iii) a prototype implementation of the approach,

(iv) a set of experimental data that show the advantages and limitations of DEPCON$^+$ with respect to DEPCON, and confirm the effectiveness of DEPCON$^+$ in exposing deadlocks in thread-safe classes.

The remainder of this paper is organized as follows. Section 2 introduces preliminary concepts to make the paper self-contained. Section 3 introduces the DEPCON$^+$ approach. Section 4 defines the parallel, conflict and double-lock dependencies based on the method summaries. Section 5 describes how DEPCON$^+$ computes the method summaries (preprocessing phase). Section 6 explains how DEPCON$^+$ generates concurrent tests by leveraging the computed parallel, conflict and double-lock dependencies. Section 7 presents the experiments that we conducted to evaluate our approach. Section 8 discusses the related work on generating concurrent tests. Section 9 summarizes the main contribution of this paper and discusses promising research directions.

2. PRELIMINARIES: GENERATING CONCURRENT TESTS FOR THREAD-SAFE CLASSES

This section presents the background information on generating concurrent tests for exposing concurrency faults in thread-safe classes, which is needed to make the paper self contained. It also defines the type of concurrency faults that DEPCON$^+$ targets.

2.1. Testing Concurrent Object-Oriented Programs

In this paper we consider **concurrent object-oriented programs** that implement the shared-memory programming paradigm. A concurrent object-oriented program is composed of a set of classes, whose methods and fields can be concurrently executed and accessed from multiple threads, respectively. Shared-memory concurrent object-oriented programs, hereafter **concurrent programs**, concurrently execute threads that interact, exchange data, and synchronize with one another by accessing shared memory locations [21, 22].

The execution of a concurrent program with a given input can be modeled as an ordered sequence of shared-memory write and read accesses. A non-deterministic scheduler determines the execution order of threads. The order of accesses to shared-memory locations is fixed within one thread, but can vary across threads. Such non-deterministic orders of shared-memory accesses are called **interleavings** [14]. A concurrent program executed with the same input often manifests

Prepared using stvrauth.cls
DOI: 10.1002/stvr

different interleavings that may result in different program behaviors. This makes testing shared-memory concurrent programs challenging because exposing erroneous behaviors with testing requires exploring both the input and interleaving spaces [7, 10].

2.2. Thread Synchronization

Developers prevent undesirable concurrent interleavings, hereafter *faulty interleavings*, by means of **synchronization mechanisms** that limit the way in which concurrent threads can interleave at runtime. Examples of synchronization mechanisms are locks, mutexes and semaphores. The complexity of thread synchronization challenges the development of concurrent programs that are both correct (avoid all faulty interleavings) and efficient (guarantee a high degree of parallelism) [14].

Thread synchronization often suffers from **concurrency faults** due to both under and over-synchronization issues. Under-synchronization allows more interleavings than it should, introducing subtle concurrency faults, like race conditions [23], atomicity violations [24], atomic-set serializability violations [25] and order violations [7]. Over-synchronization denies more interleavings than it should, introducing deadlocks [26] and performance degradation. Exposing concurrency faults at testing time is difficult because they often manifest under specific and often rare non-deterministic thread interleavings [6].

2.3. Thread-Safe Classes

Developers of concurrent programs often rely on thread-safe classes to avoid the complexity of implementing a correct and efficient synchronization.

Definition 1 (from Goetz et al. [5]). *An object-oriented class is **thread-safe** if it behaves correctly when the same instance of the class is accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment, and with no additional synchronization or other coordination on the part of the calling code (i.e., client code).*

Client code accesses a class instance via the public interface of the class, that is, by invoking its public methods. Thread-safe classes address the important challenge of synchronizing concurrent memory accesses in a correct and efficient way. In other words, thread-safety is a contract that the developers of thread-safe classes make to the client code.

By delegating the burden of thread synchronization to thread-safe classes, developers of client code can concurrently use the same instance of a thread-safe class without additional synchronization. Relying on thread-safe classes simplifies the development of client code, but concurrency faults in thread-safe classes may lead to thread-safety violations.

Definition 2. *A **thread-safety violation** of a thread-safe class is a deviation from the expected behavior of the class when concurrently accessed.*

Thread-safety violations are hard to find, reproduce, and debug. Concurrency faults may hide in rare thread interleavings that do not occur during testing but manifest in production. It is common for some concurrency faults to escape rigorous testing and lead to failures in the field [3, 6]. Ensuring the correctness of thread-safe classes is of paramount importance. It identifies concurrency faults in the implementation of the thread-safe classes, and thus in the programs that rely on them.

2.4. Generating Tests for Thread-Safe Classes

Recently, researchers proposed *concurrent test generation* approaches [3, 8, 9, 10, 11, 12, 27] to automatically expose thread-safety violations. These techniques alternatively generate concurrent tests and explore their interleaving spaces, to expose thread-safety violations at runtime. The core idea is to pair a test generator that creates concurrent tests with an interleaving explorer that checks if the generated tests can manifest thread interleavings that trigger thread-safety violations.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Definition 3. *A **concurrent test (ct)** for a thread-safe class under test consists of multiple concurrently executing threads that invoke the public methods of the class under test accessing the same shared objects.*

More specifically, a concurrent test is a set of sequences of method calls that exercise the public interface of the class under test from multiple threads [3]. 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). A concurrent test consists of a *prefix* and a set of *suffixes*.

A *prefix* is a method call sequence that creates instances of the class under test to be accessed concurrently from multiple threads. A prefix can invoke additional methods to bring the instances into states that may expose errors. A *suffix* is a call sequence that is executed concurrently with other suffixes, after executing the common prefix. All suffixes share the object instances that the prefix creates, and use them as input parameters for calls in the suffix. Accessing the same shared instances is necessary to trigger shared-memory accesses. In this paper, we generate tests with exactly two concurrent suffixes, in line with state-of-the-art concurrent test generators [10]. This is motivated by Lu et al.'s bug characteristic study [28] that reports that 96% of the considered concurrency faults can manifest by executing only two threads.

Linearizability [29, 30] is a popular correctness criterion for object-oriented concurrent programs. In the context of concurrent test generation, *linearizability* provides effective **automated oracles** for exposing thread-safety violations. Linearizability compares the behavior of a concurrent execution of a test ct with the behavior of all permutations of the outermost method calls in ct that maintain the order of calls in each thread [29, 30]. Linearizability has shown to be very effective in exposing thread-safety violations, when applied to concurrent test generation for thread-safe classes [3, 8].

Definition 4. ***Thread-safety oracle** [3] A concurrent test exposes a thread-safety violation if and only if one of its interleavings manifests an erroneous behavior that is not manifested in any method-level linearization of the test.*

Such erroneous behavior can be visible, as in the case of uncaught runtime exceptions and endless hangs, or subtle, as in the case of wrong output and corrupted or invalid program states [31]. In this paper, we consider only visible types of thread-safety violations: uncaught runtime exceptions and endless hangs.

In this context, thread-safety oracles are precise (do not report spurious thread-safety violations) but incomplete (may miss some thread-safety violations). They are precise because uncaught runtime exceptions and endless hangs that occur in a concurrent but not in a sequential usage of the class are thread-safety violations under the commonly accepted definition of thread-safety [5]. They are incomplete as not every thread-safety violation manifests an exception or an endless hang. Thread-safety violations can also manifest in a wrong output or in a corrupted or invalid program state. As such, the thread-safety oracle of Definition 4 may classify a concurrent execution as correct even though it exposes a thread-safety violation. However, Lu et al.'s study indicates that $\sim$70% of concurrency faults lead to exceptions or endless hangs [28]. Thus, we expect that this oracle detects a relevant amount of concurrency faults.

In the following we discuss the uncaught runtime exceptions and endless hangs faults that we consider in this work, and present two examples of faulty thread-safe classes. For each example we show a concurrent test that exposes the thread-safety violation, a runtime exception in the first example and an endless hang in the second.

2.5. Thread-safety Violations of Exception Type

Targeted concurrency faults Thread-safety violations of exception type refer to a wide range of concurrency faults that include data races [23], atomicity violations [24] and atomic-set serializability violations [25]. Indeed, our concurrent test generation for thread-safety violations of exception type

DOI: 10.1002/stvr

```
public class BufferedInputStream extends FilterInputStream {

[…] // omitted methods

1  private void ensureOpen() throws IOException {
2    if (in == null)
3      throw new IOException("Stream closed");
4  }

5  public synchronized void mark(int readlimit) {
6    marklimit = readlimit;
7    markpos = pos;
8  }

9  public synchronized int available() throws IOException {
10   ensureOpen();
11   return (count - pos) + in.available();
12 }

13 public synchronized int read() throws IOException {
14   ensureOpen();
15   if (pos >= count) {
16     fill();
17     if (pos >= count)
18       return -1;
19     }
20   return buf[pos++] & 0xff;
21 }
     // missing synchronization
22 public void close() throws IOException{
23   if (in == null)
24     return;
25   in.close();
26   in = null;
27   buf = null;
28 }
```

Figure 1. Buggy version of the JDK class BufferedInputStream, Bug id 4728096 [32] that leads to a thread-safety violation. Accesses to the instance fields of the class are in **bold**

is agnostic to the type of concurrency fault. However, to reduce the search space of concurrent tests, we are making the following three assumptions on the concurrent tests being generated.

I. *Exactly two concurrent threads (two concurrent suffixes)* As discussed above, this is in line with state-of-the-art concurrent test generators.

II. *Exactly one shared object under test* Accessing a single shared object under test from multiple threads is enough to expose most concurrency faults [10]. Intuitively concurrent suffixes must access the same memory locations to trigger concurrency faults. Because a single object might have multiple instance fields, considering exactly one shared object under test can expose multi-variable atomicity violations as well [25]. However, in some cases two (or more) object instances, although distinct, may access the same memory locations. Revealing thread-safety violations in these cases may require concurrent tests with two or more shared objects [10]. For instance, when the instance fields of two objects are referencing each other.

III. *Only non-static methods* We do not consider invocations of static methods, following the intuition that most concurrency faults derive from incorrect accesses to dynamic instances. This is a valid assumption that often holds in practice [10].

It is important to clarify that we can easily drop these three assumptions and modify the test generation algorithms of DEPCON$^+$ accordingly.

Allowing more than two concurrent threads (concurrent suffixes). Instead of generating two suffixes only, DEPCON$^+$ could create n suffixes by invoking GETSUFFIX function n times (see Algorithms 3). Then, the Function ASSEMBLETEST can be easily parameterized in a way that instantiates and runs n threads in parallel, each executing one of the n suffixes.

Considering more than one shared object under test. DEPCON$^+$ can consider n shared objects under test, by invoking Function GETRANDOMCONSTRUCTOR n times (see Algorithm 3). Function APPENDSTATECHANGER can append method calls that use each of these n objects.

Allowing static methods. We define the coverage targets of DEPCON$^+$ by considering the pairs of public (non-static) methods of the class under test. We could consider static methods as well. For static methods, the computation of the method summaries will not change as DEPCON$^+$ also considers static fields (accessed by `getstatic` and `putstatic` instructions) as shared-memory locations.

However, these three assumptions allow a fair comparison because they are shared among state-of-the-art approaches (such as, COVCON [8]).

Example Figure 1 shows a portion of Class `BufferedInpuStream` of *JDK 1.4*. The class contains a known concurrency fault: an atomicity violation (BUG ID: 4728096), which leads to a thread-safety

DOI: 10.1002/stvr

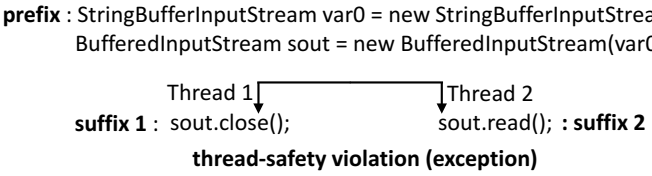

Figure 2. Concurrent test for class BufferedInputStream shown in Figure 1. The test manifests a thread-safety violation (exception). The interleaving that manifests the exception is Thread 2 executes line 17, Thread 1 line 27, Thread 2 line 20

violation of type exception. The JDK developers omitted the synchronization of method `close()` (presumably for performance reasons [32]). Such fault leads to a thread-safety violation when two threads Thread 1 and Thread 2 invoke methods `close()` and `read()`, respectively. For some specific thread interleavings, Thread 2 triggers a `NullPointerException` at line 20. An interleaving that can trigger the exception is: Thread 1 executes the statement at line 27 of method `close` after Thread 2 executes the statement at line 17 and before Thread 2 executes the statement at line 20 of the method `read`. The interested readers can refer to the bug report for more information [32].

Figure 2 shows an example of concurrent test for class `BufferedInpuStream` of Figure 1. The prefix creates two object instances: `var0` and `sout`. Variable `var0` is needed to correctly execute the constructor of `BufferedInputStream`, while variable `sout` (shared object under test) is an object instance of the class under test (`BufferedInputStream`) shared between Thread 1 and Thread 2. In fact, the method invocations of both Thread 1 and Thread 2 use `sout` as the object receiver. Being `sout` shared between the two threads, accessing the instance fields of the class (`in`, `count`, `pos`, and `buf`) triggers accesses to shared-memory locations.

The concurrent test in Figure 2 can manifest the faulty interleaving described above, triggering a `NullPointerException` at line 20 in Figure 1 (thread-safety violation). Such an exception is not thrown by any linearizations of the calls in the concurrent test. There are two possible linearizations [29, 30] of the outermost method calls of the test: $\langle$*prefix*, `sout.read()`, `sout.close()`$\rangle$ and $\langle$*prefix*, `sout.close()`, `sout.read()`$\rangle$. The first linearization does not throw any exception, while the second linearization throws an expected `IOException` at line 3, which is different from the `NullPointerException` manifested during the concurrent execution. Thus, the thread-safety oracle (correctly) reports a thread-safety violation.

Indeed, a thread-safe oracle based on linearizability reports a violation only if the class suffers from a concurrency fault. This is because it considers the (sequential) linearizations of the concurrent test as the expected behaviors [3, 29, 30]. Indeed, the manifestation of a runtime exception could be an expected behavior during a linearized execution [3, 8, 29, 30]. For example, calling `availble()` on a closed input stream triggers a runtime exception (`IOException`) at line 3 in Figure 1, which is an expected behavior. The thread-safety oracle reports a violation only if a runtime exception does not manifest in any linearization of the call.

2.6. Thread-safety Violation of Endless Hang Type

Targeted concurrency faults Thread-safety violations leading to endless hangs can manifest as infinite loops and different forms of deadlocks, commonly classified as communication and resource deadlocks [34].

Communication deadlocks [34] occur when two or more threads mutually wait to communicate with other concurrently executing threads. Each waiting thread can proceed when receiving a communication from the threads it is waiting for. More formally, a communication-deadlock occurs when given a set of concurrently executing threads $\mathcal{T} = \langle \tau_1, \tau_2, \tau_3 \cdots \tau_n \rangle$, each thread in $\mathcal{T}$ waits to communicate with another thread in $\mathcal{T}$ and no thread ever initiates any further communication until it receives the communication it is waiting for [34]. An example of a Java communication deadlock is a thread that invokes a `wait()` on an object after the concurrent thread executed the corresponding `notify()` on the same object.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

```
public class Hashtable<K, V> extends Dictionary<K, V>
implements Map<K, V>, Cloneable, java.io.Serializable {

[…] // omitted methods

public synchronized int size() { return count; }

1  public synchronized boolean equals(final Object o) {
2    if (o == this) {
3      return true;
4    }
5    if (!(o instanceof Map)) {
6      return false;
7    }
8    final Map<K, V> t = (Map<K, V>) o;
9    if (t.size() != size()) { // size is synchronized
10     return false;
11   }
12 […] // rest of the method is omitted
```

Figure 3. Buggy version of the JDK class Hashtable, Bug id 6582568 [33] that leads to a thread-safety violation (resource deadlock).

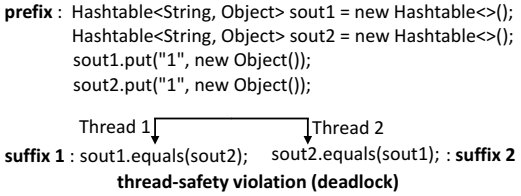

Figure 4. Concurrent test for the class in Figure 3 that manifests a thread-safety violation (endless hang). The interleaving that manifests the deadlock is Thread 1 executes line 8, Thread 2 line 2, Thread 1 line 9 (wait), Thread 2 line 9 (wait)

Resource deadlocks [34] occur when two or more threads simultaneously wait for two or more resources (such as, locks) and cannot proceed until they acquire all the needed resources. More formally, a resource-deadlock occurs when given a set of concurrent threads $\mathcal{T} = \langle \tau_1, \tau_2, \tau_3 \cdots \tau_n \rangle$, each thread in $\mathcal{T}$ requests some resources held by some other threads in $\mathcal{T}$, and needs all the requested resources to proceed [34]. An example of a Java resource deadlocks is when two threads attempt to acquire two or more locks in a different order.

In this paper, we consider resource deadlocks, and more specifically we target *resource deadlocks with exactly two threads that trigger cyclic deadlocks involving exactly two locks* (ABBA deadlocks). This is the most frequent type of resource deadlock. In fact, the bug characteristic study of Lu et al. shows that almost all (97%) deadlock faults involve two threads circularly waiting for at most two resources [28].

More specifically, we are making the following three assumptions on the concurrent tests being generated: (i) exactly two concurrent threads (two concurrent suffixes), (ii) one or two shared objects under test, (iii) only non-static methods. Differently from test generation for thread-safety violations of exception type (Section 2.5), we consider tests with one or two shared objects under test. This is because developers often implement synchronization mechanisms that use as locks the object receiver of method invocations. In such a situation, having two shared objects under test is a necessary condition to have two shared locks (we need two locks for ABBA deadlocks).

Example Figure 3 shows methods `equals` and `size` of the JDK `Hashtable` class of the *JDK 1.6*. The class contains a known concurrency fault: a resource deadlock (BUG ID: 6582568) involving a cyclic wait on two locks. The JDK developers added the synchronized keyword to method `equals`, and synchronized Method `size`, to prevent atomicity violations and race conditions. However, synchronizing both methods `equals` and `size` introduces a resource deadlock that leads to a thread-safety violation. Such violation occurs when two threads concurrently

DOI: 10.1002/stvr

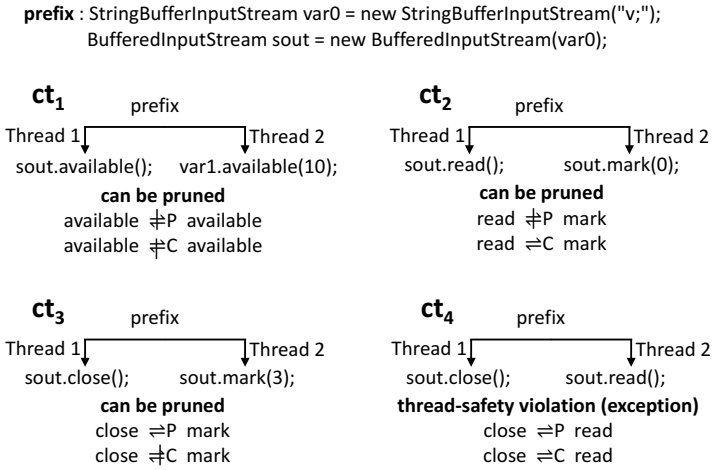

Figure 5. Concurrent tests for `BufferedInputStream` class in Figure 1, only ct_4 can trigger a thread-safety violation of exception type

execute `sout1.equals(sout2)` and `sout2.equals(sout1)`, where *sout1* and *sout2* are two instances of `Hashtable` shared across the two threads. An interleaving that triggers the deadlock is the following. Thread 1 invokes `sout1.equals(sout2)`, and when entering the method `equals` it acquires the lock associated to *sout1* because `equals` is a synchronized method. Before Thread 1 reaches line 9, Thread 2 invokes `sout2.equals(sout1)` and acquires the lock associate to *sout2*. Then, before Thread 2 executes line 9, Thread 1 attempts to execute `t.size()` at line 9, but it has to wait because the method `size` is synchronized, and thus invoking `t.size()` needs to acquire the lock associated to the object `t`. The aliasing at line 8 implies that `t` references `o`, which is the input parameter of the method `equals`. Object `o` is the hashtable `sout2`, since Thread 1 executed `sout1.equals(sout2)`. Therefore, Thread 1 waits at line 9, because Thread 2 currently holds the lock associated with Object `sout2`. While Thread 1 is waiting, Thread 2 reaches line 9. Thread 2 also waits because Thread 1 is holding the lock associated with Object `sout1`. This is a faulty ABBA resource deadlock. The interested readers can refer to the bug report for more information [33].

The concurrent test in Figure 4 can expose the interleaving described above, by triggering an endless hang that is not triggered by any linearization of the calls in the concurrent test. Neither of the two possible linearizations [29, 30] of outermost method calls of the test ($\langle$*prefix*, `sout1.equals(sout2)`, `sout2.equals(sout1)`$\rangle$ and $\langle$*prefix*, `sout2.equals(sout1)`, `sout1.equals(sout2)`$\rangle$) trigger the deadlock. Thus, the thread-safety oracle considered in this paper would (correctly) reports a thread-safety violation.

3. DEPCON$^+$

This paper presents DEPCON$^+$, a concurrent test generator for exposing thread-safety violations in classes designed to be thread-safe. DEPCON$^+$ targets violations that lead to uncaught runtime exceptions and ABBA resource deadlocks.

The challenge of the huge search space of concurrent tests As we discussed in our recent empirical study [10], a major challenge of automatically generating tests that expose thread-safety violations is the huge search space of possible concurrent tests [10]. There exists a myriad of possible combinations of sequential prefix, concurrent suffixes and input parameter values that could constitute a concurrent test. To give an idea of the search space size, let len be the call sequence length, and p the maximum number of parameters values for each method call in a test. Given a class with $|M|$

DOI: 10.1002/stvr

public methods, there are at most $(p \cdot |M|)^{len}$ method call sequences [35, 36]. Assuming concurrent tests with three method call sequences (a sequential prefix and two concurrent suffixes), there are $(p \cdot |M|)^{3 \cdot len}$ possible concurrent tests. Even if we consider the small `BufferedInputStream` class in Figure 1, which has only ten public methods ($|M| = 10$), there are a myriad of possible concurrent tests. With relatively small values of len and p ($len = 10$ and $p = 5$), there are $\sim 10^{50}$ possible concurrent tests.

Because only few concurrent tests manifest thread-safety violations [8], it is challenging to find them in such a huge search space [3, 10]. The cost of navigating a huge search space of possible concurrent tests is a major challenge of concurrent test generation. This is because it is costly to explore the interleaving space of many concurrent tests [37]. While current concurrent test generators could generate millions of concurrent tests in a few minutes [10], exploring the interleaving space of millions of concurrent tests is prohibitively expensive.

As an example, the random-based concurrent test generator ConTeGe [3] took 8.2 hours on average to generate and explore the interleaving space of millions of tests before exposing a single thread-safety violation [3]. 99.5% of this time was spent on exploring the interleaving space of the generated tests [3].

Our key intuition To address the challenge, DepCon$^+$ leverages static dependency analysis to reduce the search space of concurrent tests without missing failure-inducing tests. DepCon$^+$ is based on the core intuition that the method calls in a concurrent test need to satisfy specific requirements to expose thread-safety violations that manifest exceptions or deadlocks. DepCon$^+$ benefits from an efficient and precise approach that statically identifies such requirements before generating concurrent tests.

Only methods that meet the following two requirements can lead to **thread-safety violations of exception type** when concurrently executed:

- their executions can interleave (**parallel dependent $\rightleftharpoons$P**);
- their executions can access (with a write and read) at least one shared-memory location in common (**conflict dependent $\rightleftharpoons$C**).

Figure 5 shows four concurrent tests for the class in Figure 1: ct_1, ct_2, ct_3 and ct_4. All tests share the same prefix (on top of Figure 5) but have different concurrent suffixes. Only ct_4 exposes the thread-safety violation of exception type. There are many concurrent tests for the class in Figure 1 that do not trigger the thread-safety violation (for example, ct_1, ct_2 and ct_3 in Figure 5). A concurrent test generator could waste all the available time-budget generating such tests and exploring their interleaving spaces, missing the few failure-inducing ones [8]. However, the concurrent tests ct_1, ct_2 and ct_3 in Figure 5 do not meet our two requirements, and thus DepCon$^+$ safely avoids generating them:
(ct_1): `available` $\not\rightleftharpoons$P `available` because the method is synchronized; `available` $\not\rightleftharpoons$C `available` because the method does not perform write accesses to shared-memory locations.
(ct_2): `read` $\not\rightleftharpoons$P `mark` because both methods are synchronized, and thus their instructions are protected by the same lock (*current-object*); `read` $\rightleftharpoons$C `mark` because the method `read` writes and the method `mark` reads the same field `pos`.
(ct_3): `close` $\rightleftharpoons$P `mark` because `close` is not synchronized; `read` $\not\rightleftharpoons$C `mark` because the methods do not access the same shared-memory locations.

However, DepCon$^+$ would generate the failure-inducing test ct_4 as the methods in the concurrent suffixes have both parallel and conflict dependencies: `close` $\rightleftharpoons$P `read` because the instructions of `close` are not protected by locks; `close` $\rightleftharpoons$C `read` because `close` writes and `read` reads the same shared memory location `buf`.

Only methods that meet the following requirement can lead to **thread-safety violations of resource deadlock type** when concurrently executed:

- their executions can acquire the same pair of locks in the opposite order (**double-lock dependent $\rightleftharpoons$D**)

DOI: 10.1002/stvr

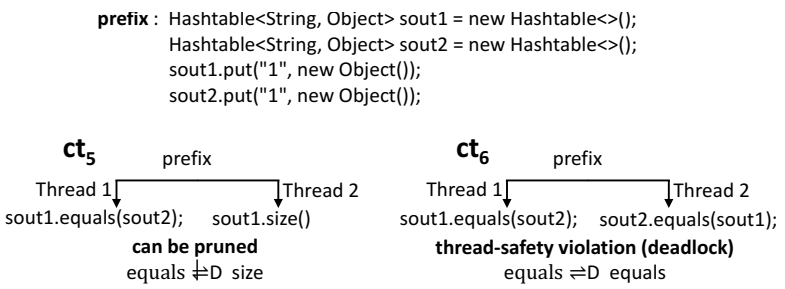

Figure 6. Concurrent tests for the `Hashtable` class in Figure 3, only ct_6 can trigger a thread-safety violation of deadlock type

Figure 6 shows two test cases ct_5 and ct_6 for the class `Hashtable` in Figure 3. Only ct_6 meets the necessary requirement to manifest a resource deadlock. In fact, the pair of two methods `equals` in Figure 3 are double lock dependent (`equals` ⇌D `equals`). Method `equals` can acquire the lock associated with the *current-object* (because `equals` is synchronized), and can acquire the lock associated with the parameter `o` at line 9 in Figure 3 as well. Because the identity of these two locks can be switched, `equals` is *double-lock dependent* with itself. Conversely, method `size` only acquires the lock associated with the *current-object*. As such, `size` is not double lock dependent with method `equals`, thus ct_5 cannot expose a ABBA deadlock. As a result, DEPCON$^+$ safely avoids the cost of generating ct_5 and exploring its interleaving space.

In a nutshell, DEPCON$^+$ exposes thread-safety violations for a Class Under Test (CUT) as follows: (i) it computes the **method summaries** of each public method of the CUT, such summaries encapsulates the possible access and synchronization operations of the method, (ii) it computes the parallel, conflict and double-lock dependencies among CUT methods by relying on the summaries, (iii) it steers the generation of tests towards concurrent tests that exhibit the computed dependencies, and thus could reveal thread-safety violations, (iv) it explores the interleaving space of each generated test relying on an available interleaving explorer.

4. METHOD DEPENDENCIES

DEPCON$^+$ statically computes three method summaries for each public method *m* declared in a (thread-safe) Class Under Test (CUT): access (*AS(m)*), lock (*LS(m)*), and double-lock (*DS(m)*) summaries. DEPCON$^+$ relies on such summaries to perform the conflict, parallel and double-lock dependency analysis, respectively.

This section defines the summaries and the dependencies, and presents two theorems that prove that exhibiting such dependencies is a necessary (but not sufficient) condition for a test to expose the target types of thread-safety violations. DEPCON$^+$ relies on such finding to generate effective coverage-driven concurrent tests. It drives test generation towards concurrent tests that satisfy this necessary condition.

We now present some preliminary information for computing the method summaries. A method *m* is composed of an ordered sequence of instructions: $inst_i$ denotes the i^{th} instruction in *m*. Each instruction has a type: *type(*$inst_i$*)* denotes the type of the instruction $inst_i$. We consider five types of instructions to define the method summaries:

- ***R(x)***, a read access to the memory location *x*;
- ***W(x)***, a write access to the memory location *x*;
- ***ACQ(l)***, the acquisition of the lock *l*;
- ***REL(l)***, the release of the lock *l*;

DOI: 10.1002/stvr

- ***INVOKE(callee)***, the invocation of method *callee*.

A method *m* can contain invocation of other methods (direct invocation), and these methods can invoke other methods, and so on (indirect invocation). The set *callees(m)* denote the set of methods that can be directly or indirectly invoked by *m*, that is, *callees(m)* contains all methods m' for which there exists a chain of method invocation from m to m'.

4.1. Conflict and Parallel Dependencies (Exception Violations)

DEPCON$^+$ computes the conflict and parallel dependencies among the public methods of the CUT by relying on the access and lock summaries of the methods. We now formally define the access *AS(m)* and lock *LS(m)* summaries of a method *m*.

Definition 5. *The **access summary AS(m)** of a method m is the set of shared memory accesses that can be triggered by m and by all of its callees.*

AS(m) $\stackrel{\text{def}}{=} \{inst_i \in \{m \cup callees(m)\} : (type(inst_i) = R(x)) \vee (type(inst_i) = W(x))$, *where x is a shared-memory location*$\}$.

For example, the access summary *AS(*`mark`*)* of method `mark` in Figure 1 is {*W(marklimit), R(pos), W(markpos)*}. The access summary of a method *m* it represents an over-approximation of all the possible accesses of shared-memory locations performed by all possible invocations of *m* under all possible execution paths.

Definition 6. *The **lock summary LS(m)** of a method m is the set of locks that protect every shared-memory access that can be triggered by an invocation of m:*

LS(m) $\stackrel{\text{def}}{=} \{$*locks l such that*

- $\exists\ inst_i \in m$: (*type*$(inst_i)$= *ACQ(l)*) $\wedge$ ($i < k$), *and*
- $\nexists$ *inst*$_j \in m$: (*type(inst*$_j$*)* = *REL(l)*) $\wedge$ ($j <$ *w)*

where k *and* w *are the indexes of the first and last shared-memory accesses in m, respectively*$\}$.

For example, the lock summary *LS(*`mark`*)* of method `mark` in Figure 1 is {*this*}, where *this* is the *current-object* of the class.

LS(m) considers only locks that protect shared-memory accesses because we are not interested in locks that protect accesses to thread-local memory locations. This is in line with classic static and dynamic analyses of concurrent programs [38, 39, 40, 41]. However, Our notion of lock summary slightly differs from the classic notion of *lockset* [42], largely adopted by dynamic and static concurrency bug detectors [23, 24, 38, 39, 43, 44, 45]. The classic lockset is defined at the granularity level of single instructions, while we define lock summary at the granularity level of sets of instructions. The classic lockset can tell us if two instructions executed by multiple threads are protected by the same lock, but cannot infer if a block of instructions (in our case, the shared-memory accesses in a method) can interleave with other blocks [46].

The reader should notice that we define *AS(m)* with information about the callees of *m* (inter-procedural), while we define *LS(m)* without looking at the callees of *m* (intra-procedural). This is because any lock acquired with internal method calls of *m* does not protect every shared-memory access in *m*, and thus it can be ignored. This reduces the cost of computing lock summaries. For instance, let us consider the method call `in.available()` in Figure 1 at line 11, where `in` is a shared-memory location. Being `in` a shared-memory location, `in.available()` could trigger other shared-memory accesses. However, even if the invocation `in.available()` acquires some locks, such locks will not protect the access of `in` at line 11. Thus, any lock acquired by the callees of *m* can be excluded, because it does not protect `in`, and thus it does not protect every shared memory access in *m*.

DOI: 10.1002/stvr

Conflict and parallel dependencies are relations between methods. Such relations are symmetric ($m_1 \rightleftharpoons m_2$ implies $m_2 \rightleftharpoons m_1$) but neither reflexive (not necessary each method relates with itself) nor transitive ($m_1 \rightleftharpoons m_2$ and $m_2 \rightleftharpoons m_3$ does not imply that $m_1 \rightleftharpoons m_3$). For both conflict and parallel dependencies we use the symbol $\equiv$ to denote the equivalence relation between shared-memory locations. It is defined as the equivalence of the memory location identifiers (with field sensitivity, i.e., `o.field`). For instance, instruction *W(buf)* at line 27 in Figure 1 writes the same memory location read by instruction *R(buf)* at line 20. Indeed, the two memory locations refer to the same identifier within the same object scope.

Definition 7. *Methods m_1 and m_2 are **conflict dependent**, $m_1 \rightleftharpoons$**C** m_2, if and only if their access summaries contain instructions that read and write the same shared-memory location:*
$m_1 \rightleftharpoons \text{C}\ m_2$ iff $(\exists W(x) \in AS(m_1) \wedge \exists R(y) \in AS(m_2)) \vee (\exists R(x) \in AS(m_1) \wedge \exists W(y) \in AS(m_2))$, where $x \equiv y$.

Our notion of conflict dependency resembles the classic notion of *race condition* [47, 48], but with an important difference. *Race condition* includes the case in which two thread writes the same memory location [42, 49], i.e., $\exists W(x) \in AS(m_1)$ and $\exists W(y) \in AS(m_2)$, where $x \equiv y$. Instead, our definition of conflict dependency excludes such a case. This reduces the number of conflict dependencies, yielding a further reduction of the search space, without missing any faulty executions. The rationale of this choice is that if both m_1 and m_2 write but never read a common memory location, their executions cannot interfere [46, 50], and their behavior would be the same of that of a sequential execution [51]. Being the same as a sequential execution, any concurrent execution (interleaving) cannot violate the linearizability correctness criterion [30].

Note that the conflict dependent requirement includes multi-variable concurrency faults. In fact, all problematic access patterns of data races [23], atomicity violations [24] and atomic-set serializability violations [25] derive from two concurrent threads that access at least one memory location in common [52], including multi-variable problematic access patterns.

Definition 8. *Methods m_1 and m_2 are **parallel dependent**, $m_1 \rightleftharpoons$**P** m_2, if and only if their corresponding lock summaries do not share common locks:*
$m_1 \rightleftharpoons \text{P}\ m_2$ iff $\forall l_1 \in \mathit{LS}(m_1), \forall l_2 \in \mathit{LS}(m_2), l_1 \not\equiv l_2$.

Our notion of parallel dependency is weaker than the concept of mutual exclusive methods, where none of the instructions of m_1 can interleave with those in m_2. As such, parallel dependency admits interleaving between the two methods, but only if the interleaved memory accesses are thread-local.

Our notion of parallel dependency resembles the notion of May-Happen-in-Parallel (MHP) [53, 54, 55, 56]. MHP analysis computes whether two statements in a multi-threaded program may execute concurrently or not [56]. However MHP is defined at the granularity of statements [55, 56], while we define the parallel analysis of DEPCON$^+$ at the granularity of methods.

Theorem 1. *Given a concurrent test $ct = \langle$prefix, suffix$_1$, suffix$_2\rangle$, if ct violates thread-safety, then the two suffixes contain two methods $m_1 \in$ suffix$_1$ and $m_2 \in$ suffix$_2$ with both parallel and conflict dependencies. That is, $\exists m_1 \in$ suffix$_1 \wedge \exists m_2 \in$ suffix$_2 : m_1 \rightleftharpoons P\ m_2 \wedge m_1 \rightleftharpoons C\ m_2$.*

Proof by contradiction
Let us assume that a concurrent test ct violates thread-safety and no pair of methods from the two concurrent suffixes has both parallel and conflict dependencies. That is $\forall\langle m_1 \in \text{suffix}_1, m_2 \in \text{suffix}_2\rangle : m_1 \not\rightleftharpoons \text{P}\ m_2 \vee m_1 \not\rightleftharpoons \text{C}\ m_2$. If $\forall\langle m_1, m_2\rangle\ m_1 \not\rightleftharpoons \text{P}\ m_2$, than the shared-memory accesses of the two methods do not interleave. Then, according to the definition of linearizability [29], all the concurrent executions of ct are equivalent to the executions of the linearizations of the method calls of ct. Thus, ct cannot violate thread-safety with an uncaught exception (contradiction). If $\forall\langle m_1, m_2\rangle$ $m_1 \not\rightleftharpoons \text{C}\ m_2$ means that each pair of methods do not access common shared-memory locations. Thus,

DOI: 10.1002/stvr

regardless whether their executions interleave or not, the behaviors of all concurrent executions of ct are equivalent to the executions obtained by the linearizations of the method calls of ct [29]. As a result, ct cannot violate thread-safety with an uncaught exception (contradiction). This proves that the presence of two methods m_1 and m_2 in the suffixes such that $m_1 \rightleftharpoons \text{P}\ m_2 \wedge m_1 \rightleftharpoons \text{C}\ m_2$ is a necessary condition for exposing thread-safety violations of exception type. □

Theorem 1 implies that the existence of parallel and conflict dependencies among methods in the suffixes of a concurrent test is a necessary condition for exposing thread-safety violations that leads to runtime exceptions.

4.2. Double-Lock Dependencies (Deadlock Violations)

DEPCON$^+$ computes the double-lock dependencies among the public methods of the CUT relying on the access and lock summaries of the methods. We now formally define the double-lock summary *DS(m)* of a method *m*.

Definition 9. *The **double-lock summary DS(m)** of a method m is the set of all pairs of nested lock acquisitions (excluding re-entrant locks) that can be triggered by m and by all of its callees.*

DS(m) $\stackrel{\text{def}}{=} \{\langle l_1, l_2\rangle : \exists inst_i, inst_j \in \{m \cup callees(m)\}$ *such that*

- *type(*$inst_i$*)* $= ACQ(l_1) \wedge$ *type(*$inst_j$*)* $= ACQ(l_2) \wedge l_1 \not\equiv l_2 \wedge i < j$, and
- $\nexists inst_k \in \{m \cup callees(m)\}$: *type(*$inst_k$*)* $= REL(l_1) \wedge k < j \wedge k > i$

where all instructions in {m ∪ callees(m)} are ordered following the program order of m *"unrolling" each method invocation in* m *and in all of its callees}*

Intuitively, the double-lock summary of a method *m* contains all the pairs of locks $\langle l_1, l_2\rangle$ such that in some executions a thread can acquire lock l_2 while still holding lock l_1. The reader should notice that we added the condition $l_1 \not\equiv l_2$ to exclude the trivial case of reentrant locks. Indeed, in many programming languages, such as JAVA, locks are reentrant: a thread in JAVA can acquire the same lock as often as it wants without creating a deadlock with itself. Computing double-lock summaries requires inter-procedural analysis as locks can be acquired by all methods that *m* can directly or indirectly invoke. For example, the double-lock summary of method `equals` in Figure 3 (*DS(equals)*) is $\{\langle this, o\rangle\}$. Because `equals` is synchronized the lock associated with the *current-object* is acquired at the method entry and released at the method exit. Before the method exists, and thus before the lock *this* is released, line 9 can acquire another lock by invoking `t.size()`. The lock that `t.size()` acquires is `o` because the method `size` is synchronized and `t` is aliased to `o`.

Definition 10. *Methods* m_1 *and* m_2 *are **double-lock dependent**,* $m_1 \rightleftharpoons \boldsymbol{D}\ m_2$*, if and only if both of their double-lock summaries contain a pair of compatible locks in opposite order:*

$m_1 \rightleftharpoons D\ m_2$ iff $\exists \langle l_a, l_b\rangle \in DS(m_1) \wedge \exists \langle l_c, l_d\rangle \in DS(m_2) : l_a \doteq l_d \wedge l_b \doteq l_c$.

The compatible relation ($\doteq$) between locks specifies whether the locks can substitute each other. For instance, recall that the double-lock summary of method `equals` in Figure 3, *DS(*`equals`*)*, is $\{\langle this, o\rangle\}$. The method `equals` is double-lock dependent ($\rightleftharpoons D$) with itself because *this* $\doteq$ *o* (and thus also *o* $\doteq$ *this*) even if the identifiers of these two locks *this* and *o* are different (*this* $\not\equiv$ *o*). This is why we must define the compatible relation $\doteq$ for locks in $\rightleftharpoons D$ as either the equivalence of their identifiers or locks that can substitute each other, if the locks are objects. Recall that in JAVA locks are either objects (such as, `synchronized(this)`) or class identifiers (such as, `synchronized(A.class)`). More specifically, given two locks that are either class identifier or objects, we say that a lock l_a can be a substitute of a lock l_b if and only if the class of lock l_a and the class of the lock l_b are either the same class or one is the superclass of the other. For example, in the double-lock summary of method `equals` in Figure 3, *this* $\equiv$ *o* and *o* $\equiv$ *this* ($l_a \equiv l_d \wedge l_b \equiv l_c$). This is because the class of *this* is `Hashtable` and the class of *o* is `java.lang.Object`, which is a super

Prepared using stvrauth.cls
DOI: 10.1002/stvr

class of `Hashtable`. Therefore, our definition of compatible relation $\doteq$ for locks in $\rightleftharpoons D$ includes both the simple case of ABBA resource deadlock (e.g., `synchronized(A){synchronized(B){...}}` and `synchronized(B){synchronized(A){...}}`) and also more complex deadlocks such as the one shown in Figure 3.

Theorem 2. *Given a concurrent test $ct = \langle prefix, suffix_1, suffix_2 \rangle$, if ct violates thread-safety (with respect to Definition 4), then the two suffixes $suffix_1$, $suffix_2$ contain two methods $m_1 \in suffix_1$ and $m_2 \in suffix_2$ with double-lock dependency. That is, $\exists m_1 \in suffix_1 \wedge \exists m_2 \in suffix_2 : m_1 \rightleftharpoons D\ m_2$*

Proof by contradiction
Let us assume that a concurrent test ct violates thread-safety with a resource ABBA deadlock and no pair of methods from the two concurrent suffixes has double-lock dependencies. That is $\forall \langle m_1 \in \text{suffix}_1, m_2 \in \text{suffix}_2 \rangle : m_1 \nrightleftharpoons \text{D}\ m_2$. If $\forall \langle m_1, m_2 \rangle\ m_1 \nrightleftharpoons \text{D}\ m_2$, then each pair of methods cannot acquire at least two common locks simultaneously, which is a necessary condition to trigger a resource deadlock involving two locks [19, 57]. As a result, ct cannot violate thread-safety with a resource deadlock (contradiction). This proves that having two methods m_1 and m_2 in the suffixes such that $(m_1 \rightleftharpoons \text{D}\ m_2)$ is a necessary condition to expose thread-safety violations of ABBA resource deadlock type. □

Theorem 2 implies that the existence of double-lock dependencies among methods in the suffixes of a concurrent test is a necessary condition for exposing thread-safety violations that leads to endless hangs caused by ABBA resource deadlocks.

5. PREPROCESSING PHASE: COMPUTING THE METHOD SUMMARIES

Algorithm 1 and Algorithm 2 show how DEPCON$^+$ computes the method summaries of a method *m*. Algorithm 1 computes the access and lock summaries, while Algorithm 2 computes the double-lock summary of *m*. The following subsection describes the alias analysis used by both Algorithm 1 and Algorithm 2 to recognize aliases of shared-memory locations and locks.

5.1. Alias Analysis

In program analysis, *aliasing* refers to the situations when there exist multiple references to the same memory location [58]. In other words, an *alias* is a reference to the same memory location. Both Algorithm 1 and 2 rely on an alias analysis to infer the identity of thread-shared variables and locks.

Instead of computing alias information for all references in the method under analysis, DEPCON$^+$ adopts an *on-demand approach*, which performs the alias analysis only when needed. More specifically, our alias analysis works at Java bytecode level and identifies the outermost reference in the reference chain with a backward analysis on the *JVM stack* [59].

The Java Virtual Machine (JVM) [59] is a stack-based abstract machine, in which JAVA bytecode instructions pop their arguments off the stack and push their results on the stack [59]. During a thread execution, the stack stores local variables, method arguments, and return values. Some bytecode instructions (for instance `aload`) load a reference or value onto the stack, while other bytecode instructions (for instance `astore`) store a reference or value. For example, `astore #index` stores a reference into local variable `#index`, while `aload #index` loads a reference onto the stack from a local variable `#index`.

DEPCON$^+$ triggers the alias analysis when it encounters `aload` and `astore` instructions. Because we are computing aliases among method variables, the outermost reference in the reference chain could be (i) a local variable, (ii) a method parameter (including the object receiver), or (iii) a static object. DEPCON$^+$ needs alias information to infer whether `aload` and `astore` instructions are aliasing method parameters or static objects. This is because, by construction, in a concurrent test

DOI: 10.1002/stvr

only the objects passed as parameters to the method calls or static objects can be memory locations that are shared across multiple threads (such as locks).

We now exemplify how the alias analysis works. Consider line 8 of Figure 3, which creates an alias between variable `t` and `o` (where `o` is a method parameter).

```
final Map<K, V> t = (Map<K, V>) o;
```

The bytecode instructions of this source code line are:

```
ALOAD 1
CHECKCAST java/util/Map
ASTORE 2
```

The alias analysis infers that the two variables point to the same memory location, by observing that `ASTORE 2` is preceded by `ALOAD 1`.

In the literature exists two main types of aliasing. The may-alias analysis identifies aliases that *can* occur during some execution of the program. The must-alias analysis identifies aliases that occur in all executions of the program. To guarantee a low computational cost, we implemented a **path-insensitive may-alias analysis** that does not consider each individual execution path. As such, our goal is not to final *all* may aliases (for each execution path). DEPCON$^+$ computes one possible *may* alias for each analyzed instruction. Even if the computed alias is only one, it is not necessarily a *must* alias because the alias might occur for some execution path only.

Our alias analysis also considers **field-sensitivity** when retrieving the identity of variables. For example, given the following class:

```
public class A {
Object field = new Object();

public void m1() {
Object o1 = this.field;
}
}
```

The bytecode of method `m1` is:

```
public m1()V
L0
LINENUMBER 5 L0
ALOAD 0
GETFIELD A.field : Ljava/lang/Object;
ASTORE 1
L1
LINENUMBER 6 L1
RETURN
L2
LOCALVARIABLE this LA; L0 L2 0
LOCALVARIABLE o1 Ljava/lang/Object; L1 L2 1
MAXSTACK = 1
MAXLOCALS = 2
```

The backward alias analysis starting from the instruction `ASTORE 1` tells us that the `LOCALVARIABLE o1 Ljava/lang/Object` aliases `this.field`, as `ASTORE 1` is preceded by the instructions `GETFIELD A.field : Ljava/lang/Object;` and `ALOAD 0`. Note that, in every non-static method the memory location with index 0 represents the object receiver `this` (see `LOCALVARIABLE this LA; L0 L2 0`).

Our analysis considers an array as a single memory location. It does not consider each individual cell of the array as a memory location. This is because a static analysis cannot know the values of the indices, as their values depend on the calling context of the method under analysis (i.e., the particular inputs used by the concurrent test). This might lead to unsound results.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Algorithm 1: COMPUTEACCESSANDLOCKSUMMARIES

input : a public method of CUT (m)

output : access and lock summaries of m ($\langle AS(m), LS(m)\rangle$)

```
1  ⟨AS(m), LS(m)⟩ ← ⟨∅, ∅⟩                                  // init access summary of m
2  tmp-lock-releases ← ∅                                     // temporary lock releases
3  for each inst_i ∈ m do
4      switch type(inst_i) do
5          case Shared-memory accesses: R(x) or W(x) do
6              ref ← GETOUTERMOSTREF(inst_i)
7              if ISSHAREDLOCATION(ref) then
8                  add R(ref.f) or W(ref.f) to AS(m)
9                  remove tmp-lock-releases from LS(m)
10                 tmp-lock-releases ← ∅
11         case Lock acquire: ACQ(l) do
12             if AS(m) = ∅ then
13                 l ← GETLOCK(inst_i)
14                 add l to LS(m)
15         case Lock release: REL(l) do
16             l ← GETLOCK(inst_i)
17             if AS(m) = ∅ then
18                 remove l from LS(m)
19             else
20                 add l to tmp-lock-releases
21         case Method invocation: INVOKE(callee) do
22             callee ← GETINVOKEDMETHOD(inst_i)
23             if callee is a method declared in the CUT then
24                 AS(callee) ← COMPUTEACCESSANDLOCKSUMMARIES(callee)
25                 add AS(callee) to AS(m)
26             else
27                 if ISPURE(callee) is false then
28                     for each parameter p of callee do
29                         if ISSHAREDLOCATION(p) then
30                             add W(p) to AS(m)
31 return ⟨AS(m), LS(m)⟩

32 function ISPURE(m)  // returns true if the method m is pure, false otherwise.
33     for each inst_i in m do
34         switch type(inst_i) do
35             case Write access: W(x) do
36                 x ← GETOUTERMOSTREF(inst_i)
37                 if x is a parameter of m then
38                     return false
39             case Method invocation: INVOKE(callee) do
40                 callee ← GETINVOKEDMETHOD(inst_i)
41                 return ISPURE(callee)
42     return true
```

5.2. Computing the Access and Lock Summaries

Algorithm 1 computes the access and lock summaries of a method *m*. The algorithm implements a fine-grained analysis that incrementally populates the access and lock summaries of each method *m* declared in the CUT. The algorithm invokes the recursive function ISPURE, every time it detects an

Prepared using stvrauth.cls DOI: 10.1002/stvr

invocation to a method that is not declared in the CUT but is directly or indirectly invoked from CUT methods. Function ISPURE analyses the callee methods to infer if they modify the value of method parameters that represent shared-memory locations. ISPURE analysis is coarse-grained to guarantee a low computational cost in the common case of many invocations to non-CUT methods.

Algorithm 1 starts by initializing the access summary *AS(m)*, the lock summary *LS(m)* and *tmp-lock-releases* to empty sets (lines 1 and 2). Algorithm 1 uses a supporting variable *tmp-lock-releases* to compute *LS(m)* (line 2). Then, Algorithm 1 scans the ordered sequence of bytecode instructions $inst_i$ of *m* (line 3), and checks the type of each instruction $inst_i$ to determine if it belongs to one of the following four instruction types.

Type 1) Shared-memory accesses: W(x) or R(x) (line 5). To expose thread-safety violations of exception type, DEPCON$^+$ generates concurrent tests that share among threads only one shared object under test of type CUT. By construction, DEPCON$^+$ uses the *current-object* (*this* in JAVA) as the SOUT. Therefore, following previous work on concurrency testing [3, 8, 9, 11, 12, 27, 60], only the fields of the *current-object* or fields of static classes can be shared-memory locations. In this context, we do not need an expensive *thread-escape analysis* [61, 62] to identify shared-memory locations. This reduces the complexity of the analysis. As such, in the JAVA bytecode [59] Type 1 instructions are read or write accesses to static or non-static fields: `getfield`, `getstatic` as well as `putfield` and `putstatic` instructions. However, JAVA Type 1 instructions could also be `aload #index` and `astore #index` instructions if the memory location `#index` is an alias to a non-primitive field. DEPCON$^+$ recognizes such cases by performing alias analysis when it encounters `aload #index` and `astore #index` instructions, identifying the aliased memory location loaded and stored by the `getfield` and `putfield` instructions, respectively.

Given the `getfield` and `putfield` instructions, DEPCON$^+$ needs to infer whether these instructions are accessing the *current-object*. Indeed, a method can create and access local objects that do not reference the fields of the *current-object*. Because fields of an object can also be objects of their own, Function GETOUTERMOSTREF traverses the chain of object references to identify the outermost object reference *ref*.

Function ISSHAREDLOC(*ref*) (line 7) checks if *ref* is the *current-object*. This is trivially true when *ref* points to the first register [59], which is popped in the *JVM stack* with the bytecode instruction `aload 0`. Alternatively, ISSHAREDLOC(*ref*) performs an additional backward alias analysis on the *JVM stack* to check whether the alias of *ref* is the *current-object*. If ISSHAREDLOC(*ref*) returns true, the algorithm adds *R(ref.f)* (for `getfield`) or *W(ref.f)* (for `putfield`) to *AS(m)* (line 8), where *f* is field of *ref* that is accessed. Then, the algorithm removes from *LS(m)* all the locks that were released after the lastly added access in *AS(m)* (line 9). This is because *LS(m)* must contain only the locks released after the last shared-memory access in *m* (Definition 6).

To identify lock acquisitions and releases (Type 2 *ACQ(l)* and Type 3 *REL(l)*) instructions, DEPCON$^+$ considers two types of popular JAVA synchronization mechanisms: synchronized blocks and synchronized methods. For synchronized blocks, DEPCON$^+$ maps the *ACQ* and *REL* instructions to bytecode instructions `monitorenter` and `monitoexit`, respectively. In case of synchronized methods, DEPCON$^+$ identifies the presence of *ACQ* and *REL* instructions by checking if the `ACC_SYNCHRONIZED` flag is true. In fact, the JVM acquires and releases locks also when the keyword `synchronized` appears in the method declaration. In this case, the compiler does not generate `monitorenter` and `monitoexit` bytecode instructions, but flags the method declaration to "automatically" acquire and release the lock on method entry and exit.

There is an important consideration regarding monitor instructions. Because we are assuming synchronized blocks and methods, by construction each `monitorenter` instruction is paired to a corresponding `monitoexit` instruction in the same execution path. As such, our path-insensitive alias analysis is adequate in this context. However, in Java, return statements and exceptions prematurely terminate a method execution. In such cases, even with synchronized block and methods, a `monitorenter` instruction is paired with multiple `monitorexit` instructions that are executed immediately before the return statement or exceptions Because the method summaries want to capture an over-approximation of all possible behaviors of the method, we ignore those `monitorexit`

Prepared using stvrauth.cls DOI: 10.1002/stvr

instructions at premature terminations. We only consider `monitorexit` instructions that are executed inside the method body or at the end of the method.

Type 2) Lock acquisition: ACQ(l) (line 11). Recall that *LS(m)* contains only the locks that are acquired before the first shared-memory access in *m* and never released before the last shared-memory access in *m* (Definition 6). As such, the algorithm checks if *AS(m)* is empty to infer if the lock acquire instruction is executed before the first shared-memory access in *m*. We encode locks by their fully qualified names since shared-memory locations (instance or static fields) of the same class have unique identifiers. Function GETLOCK($inst_i$) gets the lock object by performing backward alias analysis on the *JVM stack*, as discussed in Section 5.1.

Type 3) Lock release: REL(l) (line 15). The algorithm executes Function GETLOCK($inst_i$) to get the lock *l* released by $inst_i$ (line 16). If *AS(m)* is empty, the algorithm removes *l* from *LS(m)* (line 18), otherwise it adds *l* to the *tmp-lock-releases* supporting variable. The algorithm will remove the locks in *tmp-lock-releases* from *LS(m)* only if it meets another shared-memory access while scanning subsequent instructions in *m* (line 9).

Type 4) Method invocation: INVOKE(callee) (line 21). JAVA bytecode method invocations correspond to `invokedynamic`, `invokeinterface`, `invokespecial`, `invokestatic` and `invokevirtual` instructions [59]. When the algorithm finds one of such instructions, it gets the fully qualified name of the called method (*callee* line 22), and checks if it is declared in the CUT.

If *callee* is declared in the CUT, the algorithm recursively calls COMPUTEMS(*callee*) to get the access summary of *callee* (*AS(callee)*), and adds it to *AS(m)*. Notably, we do not add the lock summary of *callee LS(callee)* to *LS(m)*, as lock summaries are defined intra-procedurally (see Definition 6).

If *callee* is not declared in the CUT, the algorithm calls Function ISPURE to check if *callee* is a *pure method* [63, 64], that is, if it does not directly or indirectly write on object fields. If *callee* is not a pure method, the algorithm checks if any parameter *p* of the method invocation is aliased to either the *current-object* or to any one of the *current-object* fields (ISSHAREDLOCATION(p) line 29). If *p* is an alias, the algorithm adds a write access (*W(p)*) to *AS(m)* (line 30). The algorithm does not add a read access, because it must have already revealed an instruction that pushes *p* in the *JVM stack*.

Lines 32 to 42 of Algorithm 1 implement DEPCON$^+$'s purity analysis. Function ISPURE scans each instruction of a method *m* as follows. For all instructions that write a memory location, function ISPURE gets the outermost reference (*ref*) in the chain of object references (GETOUTERMOSTREF line 36), and signals the method as *impure* (ISPURE returns false) if *ref* aliases any method parameter, including the object receiver. The function recursively calls ISPURE on the called method when it encounters an instruction of type *INVOKE*. Our purity analysis does not attempt to infer the exact objects that can be modified by the analyzed method. Such a conservative approach has two benefits. First, it guarantees a low computational cost. Second, it guarantees to not miss any impure methods. Missing impure methods might lead to missing real conflict dependencies, inducing DEPCON$^+$ to prune failure-inducing concurrent tests.

5.3. Computing the Double-Lock Summary

Algorithm 2 shows how DEPCON$^+$ computes the double-lock summary of a method *m* (*DS(m)*). The Algorithm starts by initializing *DS(m)* to the empty set (line 43), and by calling Function GETUNROLLEDINSTRUCTIONS (line 44), which returns *unrolled(m)*, the ordered sequence of lock acquisitions and releases that can be triggered by *m* and all of its callees.

Function **GETUNROLLEDINSTRUCTIONS** (lines 61 to 71) scans each instruction of a method *m* and adds all *ACQ(l)* and *REL(l)* instructions to *unrolled(m)*, following the order of occurrence in *m*. As discussed in Section 5.2, DEPCON$^+$ maps *ACQ(l)* and *REL(l)* instructions to the entry and exit points of JAVA synchronized blocks and synchronized methods, respectively. For each *ACQ(l)* and *REL(l)* instructions, Function GETLOCK collects the identifier *l* of the acquired or released lock. In case of aliases, the function performs backward alias analysis on the JVM stack to identify the memory location of *l*.

Prepared using stvrauth.cls DOI: 10.1002/stvr

Algorithm 2: COMPUTEDOUBLELOCKSUMMARY

```
   input  : method (m)
   output : double-lock summary of m (DS(m))
43 DS(M) ← ∅
44 unrolled(m) ← GETUNROLLEDINSTRUCTIONS(m)
45 for inst_i in unrolled(m) do
46     lock-ordered-set ← ∅
47     switch type(inst_i) do
48         case Lock acquire: ACQ(l) do
49             l ← GETLOCK(inst_i)
50             if l ∉ lock-ordered-set then
51                 append l to lock-ordered-set
52                 size ← size of lock-ordered-set
53                 if size ⩾ 2 then
54                     for each i from 0 to (size - 1) do
55                         for each j from (i+1) to (size - 1) do
56                             append ⟨lock-ordered-set[i], lock-ordered-set[j]⟩ to DS(M)
57         case Lock release: REL(l) do
58             l ←GETLOCK(inst_i)
59             remove l from lock-ordered-set
60 return DS(m)
61 function GETUNROLLEDINSTRUCTIONS(m)
62     unrolled(m) ← ∅
63     for each inst_i in callee do
64         switch type(inst_i) do
65             case Lock acquire or release: ACQ(l) or REL(l) do
66                 l ← GETLOCK(inst_i)
67                 add ACQ(l) or REL(l) to unrolled(m)
68             case Method invocation: INVOKE(callee) do
69                 callee ← GETINVOKEDMETHOD(inst_i)
70                 add GETUNROLLEDINSTRUCTIONS(calle) to unrolled(m)
71     return unrolled(m)
```

If the function processes an instruction of type *INVOKE(callee)*, it recursively calls Function GETUNROLLEDINSTRUCTIONS with *callee* as argument, and adds the result of such recursive invocation to *unrolled(m)*.

Notably, before invoking GETUNROLLEDINSTRUCTIONS(callee) at line 70 of Algorithm 2, DEPCON$^+$ performs alias analysis on the method parameters passed to *callee*.

In a nutshell, the function computes *unrolled(m)* recursively by "unrolling" each method invocation in *m* and in all of its callees. As such, *unrolled(m)* is a path-insensitive ordered sequence of lock acquisitions and releases instructions following the program order of each invoked method.

The sequences are path-insensitive because they do not distinguish among different paths that lead to the same program point. Conversely, a path-sensitive analysis would explore paths forked by branch statements independently [58, 65] (it would explore each execution path individually).

Note that *unrolled(m)* is an over-approximation of the possible lock acquires and releases because we consider a path-insensitive abstraction. Thus, we assume that all instructions in *unrolled(m)* are being executed. This can lead to infeasible double-locks because the control flow of m can make some *ACQ* and *REL* instructions mutually exclusive. It is important to clarify that infeasible double-locks would increase the number of spurious double-lock dependent method pairs, but it would not prevent the detection of resource deadlocks. We decide to opt for a path-insensitive abstraction to maintain a low computational cost. In fact, a path-sensitive analysis cannot guarantee a low computational

Prepared using stvrauth.cls DOI: 10.1002/stvr

cost with the presence of many branches. Even though a path-sensitive analysis would lead to less infeasible double-locks, it requires to enumerate and analyze all possible execution paths, which is too expensive.

After computing *unrolled(m)*, Algorithm 2 scans each instruction $inst_i$ in *unrolled(m)* (Line 45). Such scanning simulates the program execution, assuming that all the instructions in *unrolled(m)* are executed in the order prescribed by *m*. At any point while scanning *unrolled(m)*, the *lock-ordered-set* contains the set of locks held by the simulated execution. The *lock-ordered-set* resembles the classic *lockset* [42] with the only difference that *lock-ordered-set* maintains the insertion order of the locks. When the size of *lock-ordered-set* is greater than two, we have found a sequence of two consecutive locks acquisitions. This might lead to a resource deadlock if the analyzed method is concurrently executed with another method that can acquire the same locks in the opposite order.

In detail, Algorithm 2 works as follows. If the analyzed instruction is a lock acquisition (*ACQ*), the algorithm get the lock l of the instruction. Note that, DEPCON$^+$ already computed the lock of all the instructions when populating *unrolled(m)*, so GETLOCK at line 49 simply recovers the already computed unique identifier of the lock l. Line 50 checks if l is already in *lock-ordered-set*, if yes the algorithm skips it. Such a situation occurs when a thread acquires the same lock twice (called reentrant lock in JAVA). In fact, in JAVA a thread can acquire the same locks multiple times (without creating a deadlock with itself). If l is not currently in *lock-ordered-set*, the algorithm appends it at the end of *lock-ordered-set*. Then, the algorithm gets the size of *lock-ordered-set* to check if is greater or equal than two. If yes, we have found an execution point in which a thread could hold more than one lock simultaneously, and thus we have found at least a pair of double locks. Lines 54 and 55 compute all possible pairs of locks that maintain the order of insertion prescribed by *lock-ordered-set*. Line 56 adds the computed pairs in *DS(m)*. For example, let consider that *lock-ordered-set* = $\langle l_1, l_2, l_3 \rangle$. Line 56 adds three pairs of double locks: $\langle l_1, l_2 \rangle$, $\langle l_1, l_3 \rangle$, and $\langle l_2, l_3 \rangle$ in *DS(m)*. Line 57 removes the lock from *lock-ordered-set* in case of *REL* instructions.

We exemplify the computation of the double-lock summary on the fragment of the method `equals` shown in Figure 3 at page 8, method *m* in this example. The algorithm builds *unrolled(m)*, the ordered list of *ACQ* and *REL* instructions that can be executed by *m* and by all of its callees, and uses *unrolled(m)* to build the double-lock summary. The first relevant instruction in *m* is a *ACQ(l)* instruction triggered by the keyword `synchronized` in the method declaration. Function GETLOCK at line 13 infers that the acquired lock is the *current-object*, and appends *ACQ(`this`)* to *unrolled(m)*. The next relevant instructions are two *INVOKE* instructions at line 9 of Figure 3: `t.size()` and `size()`. The algorithm observes that both methods are the synchronized methods, and identifies `t` as aliased to `o`, thus appends *ACQ(`o`)* at the end of *unrolled(m)*, followed by *ACQ(`this`)*, where *this* is the current object. After scanning all instructions in *m*, the algorithm starts building *DS(m)* (line 43 of Algorithm 2). Given *unrolled(m)* = {*ACQ(`this`)*, *ACQ(`o`)*, *ACQ(`this`)*, $\cdots$}, after the algorithm appends *ACQ(`o`)* to *lock-ordered-set* line 51 in Algorithm 2), *lock-ordered-set* has size two. Thus, Algorithm 2 adds the pair $\langle$`this`, `o`$\rangle$ to *DS(m)*. When the algorithm identifies the second *ACQ(`this`)*, it does not add `this` to *lock-ordered-set*, because it is a reentrant lock.

Details implicit in Algorithm 1 and 2 For the sake of readability, we do not explicitly report few details in Algorithm 1 and 2:

- The algorithms cache the computed *method summaries*, *purity results* and *unrolled sequences* to reduce computational time. When the algorithms encounter multiple invocations of the same method, they returned the cached result.

- The algorithms avoid endless recursions of functions COMPUTEMS, ISPURE and GETUNROLLEDINSTRUCTIONS caused by methods that directly or indirectly invoke themselves (such as, with transitive or recursive calls). They achieve this by maintaining a *call stack* of the analyzed methods, and stopping recursion when meeting a method invocation already in the *call stack*.

Prepared using stvrauth.cls DOI: 10.1002/stvr

Path insensitivity The reader should notice that both Algorithm 1 and 2 ignore branch and loop instructions. In fact, the static analysis of DEPCON$^+$ is path-insensitive, as it represents a method as an ordered sequence of instructions regardless of branch and loop instructions. We made this choice to guarantee a low computational cost in the presence of many branches and inter-procedural calls. Indeed, computing the method summaries is pure overhead for the test generation phase.

Ignoring branch and loop instructions is reasonable in our case, for two reasons:

First, we are interested in an over-approximation of all possible behaviors. We are not interested to infer the exact behaviors of methods. Indeed, DEPCON$^+$ assumes that mutually exclusive branches can be taken at the same time, which of course does not represent a possible behavior of the method.

Second, because we are assuming synchronized blocks and methods, by construction each `monitorenter` instruction is always paired to a corresponding `monitoexit` instruction in the same execution path.

6. GENERATING CONCURRENT TESTS

Given a class under test (CUT) designed to be thread-safe, DEPCON$^+$ alternatively generates concurrent tests and explores their interleaving spaces with the goal of exposing thread-safety violations. This section describes how DEPCON$^+$ generates concurrent tests and how it leverages the computed method dependencies to reduce the search space.

Coverage targets DEPCON$^+$ follows the coverage-driven approach of COVCON [8], a state-of-the-art coverage-driven concurrent test generator. COVCON exploits the concept of concurrent method pairs proposed by Deng et al. [66], defined as the set of pairs of methods that can executed concurrently [66]. A thread-safe class should guarantee a correct behavior no matter which methods are executed concurrently. As such, COVCON considers as coverage targets the set $\mathcal{M}$ of all possible pairs of public (non-static) methods of the class under test $\langle m_i \in M, m_j \in M \rangle$ [8], where M is the set of public methods declared in the CUT. Following Choudhary et al., we consider $\langle m_1, m_2 \rangle$ and $\langle m_2, m_1 \rangle$ to be the same method pair because the order is irrelevant for concurrency bug detection [8]. As such, $\mathcal{M}$ is composed of all possible pairs of methods unique with respect to symmetry. More formally, $\mathcal{M} \stackrel{\text{def}}{=} \{M \times M : \forall \langle m_i, m_j \rangle \in \mathcal{M}, \langle m_j, m_i \rangle \notin \mathcal{M}\}$. As such, if $|M|$ is the number of public methods in the *CUT*, there are $|M| \cdot (|M| + 1)/2$ method pairs. To reduce the search space of concurrent test, DEPCON$^+$ prunes from $\mathcal{M}$ all pairs that do not exhibit conflict, parallel and double-lock dependencies. The reader should notice that we are not interested in computing the method dependencies (and thus the method summaries) among private or protected methods as they do not constitute the public interface of the CUT. This implies that concurrent tests (client code) cannot directly invoke them. However, DEPCON$^+$ does consider the shared-memory accesses and locks acquired and released by private or protected methods when such methods are invoked by public methods.

Algorithm 3 and Algorithm 4 show the DEPCON$^+$ test generation approach, by highlighting how the algorithms leverage parallel, conflict, and double-lock dependencies to generate concurrent tests. Algorithms 3 show how DEPCON$^+$ generates concurrent tests to expose thread-safety violations of exception type, while Algorithm 4 of deadlock type. DEPCON$^+$ has two distinct concurrent test generation algorithms because dealing with these two types of thread-safety violations require different approaches.

6.1. Generating Concurrent Tests for Exposing Thread-Safety Violations of Exception Type

Algorithm 3 takes in input (i) the class under test CUT, (ii) the time-budget $\mathcal{B}$, (iii) the number of times to re-execute each test *N-iter*, (iv) the maximum length of the prefixes *Max-len*, (v) all coverage

Prepared using stvrauth.cls DOI: 10.1002/stvr

Algorithm 3: DEPCON-EXCEPTIONMODE

input : class under test (CUT)
time-budget ($\mathcal{B}$)
number of times to re-execute each test (N-iter)
maximum length of the prefixes (Max-len)
all coverage targets ($\mathcal{M}$)
access and lock summaries of each public method m in the CUT (*AS(m)* and *LS(m)*)

output : a concurrent test *ct* that manifests a thread-safety violation of exception type (if any)

```
72 M_pc ← ∅                                              // init reduced coverage targets
73 for each ⟨m1, m2⟩ ∈ M do
74     if m1 ⇌P m2 ∧ m1 ⇌C m2 then
75         add ⟨m1, m2⟩ to M_pc
76 while time-budget B is not expired do
77     ⟨m1, m2⟩ ← GETNEXTMETHODPAIR(M_pc)
78     for each i from 1 to 2 do
79         ⟨sout, prefix⟩ ← GETRANDOMCONSTRUCTOR(CUT)
80         if i % 2 = 0 then
81             prefix ← APPENDSTATECHANGER(sout, prefix, Max-len)
82         suffix1 ← GETSUFFIX(m1, sout)
83         suffix2 ← GETSUFFIX(m2, sout)
84         ct ← ASSEMBLETEST(prefix, suffix1, suffix2)
85         for each iter from 1 to N-iter do
86             exception ← EXECUTE(ct)
87             if exception ≠ ∅ ∧ exception ∉ ALLPOSSIBLELINEARIZATIONS(ct) then
88                 return ct                    /* thread-safety violation (exception type) */
89 return ∅
```

targets $\mathcal{M}$ of the CUT, (vi) the access *AS(m)* and lock *LS(m)* summaries of each public method m in the CUT, computed by Algorithm 1. Algorithm 3 terminates when it detects a thread-safety violation of exception type or the time-budget expires. If DEPCON$^+$ detects a thread-safety violation, it returns the concurrent test ct that manifests it. Such a test is an important starting point to debug the concurrency issue.

While COVCON considers all possible pairs $\mathcal{M}$ as coverage targets, Algorithm 3 considers only those pairs $\mathcal{M}_{pc} \subseteq \mathcal{M}$ that have parallel and conflict dependencies. Starting from an empty $\mathcal{M}_{pc}$, Algorithm 3 adds in $\mathcal{M}_{pc}$ all the pairs $\langle m_1, m_2 \rangle \in \mathcal{M}$ such that $m_1 \rightleftharpoons$P m_2 and $m_1 \rightleftharpoons$C m_2 (lines 73-75 in Algorithm 3). DEPCON$^+$ computes the parallel and conflict dependencies relying on the computed method summaries according to Definition 8 and Definition 7, respectively.

Until the time-budget expires, DEPCON$^+$ generates concurrent tests and it explores their interleaving spaces to check if one of the explored interleavings leads to a thread-safety violation. Function GETNEXTMETHODPAIR (line 77) selects the next pair according to COVCON approach that prioritizes pairs based on the frequency of their concurrent executions, to focus the test generation on infrequently or not at all covered pairs [8].

Given a method pair $\langle m_1, m_2 \rangle \in \mathcal{M}_{pc}$, Algorithm 3 generates two concurrent tests with two different prefixes. The first prefix is composed of a randomly chosen constructor (GETRANDOMCONSTRUCTOR, line 79). The second prefix contains additional method calls after the constructor (GETSTATECHANGER, line 81). The rationale is that some concurrency faults can only be triggered on a freshly instantiated instance, whereas other faults require bringing the instance into a fault-exposing state by invoking a sequence of method calls [3, 8, 9].

Function GETRANDOMCONSTRUCTOR returns the *shared object under test* (*sout*) and the method call sequence that instantiates it (*prefix*). The function randomly selects one of the possible CUT constructors, and randomly generates primitive or non-primitive

Prepared using stvrauth.cls
DOI: 10.1002/stvr

parameters, if needed. For instance, the concurrent test of Figure 2 uses constructor `BufferedInputStream(StringBufferedInputStream)`. Function GETRANDOMCONSTRUCTOR randomly generates `var0` of type `StringBufferedInputStream`, the object that the constructors use as an input.

Function APPENDSTATECHANGER appends at the end of *prefix* a sequence of method calls that use *sout* as a method parameter. The number of such method calls is chosen randomly between 1 and *Max-len*. With these additional method calls, DEPCON$^+$ explores the possible states of the object *sout*, as the (fault triggering) behavior of the suffixes depends on the state of *sout* after the execution of the prefix. Function APPENDSTATECHANGER relies on the computed method summaries to avoid redundant prefixes. It forces at least an additional method call in the prefix to be in conflict ($\rightleftharpoons$C) with either m_1 or m_2. The rationale is that prefixes that do not modify the values of memory locations read by m_1 or m_2 are equivalent to prefixes that only use the constructor. Intuitively, the calls in the sequential prefix must modify at least a memory location read by the concurrent suffixes to affect their behavior, for instance with a new execution path. By avoiding generating concurrent tests with same suffixes but different albeit redundant prefixes, DEPCON$^+$ saves precious time-budget.

DEPCON$^+$ generates the method call *suffix*$_1$ that invokes m_1, by using as input parameter the shared object under test instantiated by the prefix (GETSUFFIX(m_1) at line 82) [8]. Similarly, it generates *suffix*$_2$ with m_2 (GETSUFFIX(m_2) at line 83).

DEPCON$^+$ generates a new concurrent test ct by assembling the obtained method call sequences (line 84). DEPCON$^+$ explores the interleaving space of ct with stress-testing, which repetitively executes ct *N-iter* times (line 86). Stress-testing relies on the non-determinism of the JVM scheduler to explore the interleaving space of a concurrent test ct. The JVM scheduler is likely to induce a different interleaving every time it executes ct [3, 8]. After every execution, if the test thrown an uncaught exception, and the same exception does not manifest when executing each *linerarizations* [3] of ct (line 87), DEPCON$^+$ terminates and reports a thread-safety violation. Otherwise, DEPCON$^+$ iterates generating new concurrent tests until the time-budget expires. It is important to clarify that the test generation of DEPCON$^+$ is decoupled from the interleaving exploration. As such, the interleaving explorer of DEPCON$^+$ could be replaced by any interleaving explorer.

6.2. Generating Concurrent Tests to Expose Thread-Safety Violations of Deadlock Type.

The DEPCON approach, the earlier version of DEPCON$^+$ that we presented at ICST in 2019 [17], is inadequate to expose resource deadlocks in thread-safe classes. The reason is twofold. First, conflict and parallel dependencies do not capture the deadlock condition. Second, DEPCON generates tests with exactly one instance of the CUT shared between the concurrent threads, which is inadequate to reveal most ABBA resource deadlocks.

In this paper, we extend DEPCON with a *"deadlock mode"* shown in Algorithm 4. The Algorithm takes in input (i) the class under test CUT, (ii) the time-budget $\mathcal{B}$, (iii) the number of times to re-execute each test *N-iter*, (iv) the maximum length of the prefixes *Max-len*, (v) all coverage targets $\mathcal{M}$ of the CUT, (vi) the double-lock summary *DL(m)* of each public method m in the CUT, computed by Algorithm 2. Algorithm 4 generates concurrent tests and explores their interleavings to check if one of the explored interleavings leads to a deadlock. Algorithm 4 terminates when it detects a thread-safety violation of deadlock type or the time-budget expires. To reduce the search space of concurrent tests, the algorithm exploits the new double-lock dependency that we propose in Section 4.2, and that constitutes an important novel contribution of this paper.

Algorithm 4 starts by building $\mathcal{M}_d \subseteq \mathcal{M}$ the method pairs that have double-lock dependencies. That is, $\langle m_1, m_2 \rangle \in \mathcal{M}$ such that $m_1 \rightleftharpoons$D m_2 (lines 91-93). DEPCON$^+$ computes the double-lock dependencies relying on the double-lock summary returned by Algorithm 2 (see Section 4.2). Intuitively, to increase the effectiveness of concurrent test generation, DEPCON$^+$ generates concurrent tests considering only the pairs in $\mathcal{M}_d$ as suffixes for the tests. Moreover, Algorithm 4 carefully generates concurrent tests with prefixes and suffixes that properly exercise the deadlock condition captured by the double-lock dependency. In the following we describe Algorithm 4 in detail.

Prepared using stvrauth.cls DOI: 10.1002/stvr

Algorithm 4: DEPCON-DEADLOCKMODE

input : class under test (CUT)
time-budget ($\mathcal{B}$)
number of times to re-execute each test (N-iter)
the maximum length of the prefixes *Max-len*
double-lock summary of each public method m in the CUT (*DL(m)*)

output : a concurrent test *ct* that manifests a thread-safety violation of deadlock type (if any)

```
90  M_d ← ∅                                              // init reduced coverage targets
91  for each ⟨m1, m2⟩ ∈ M do
92      if m1 ⇌D m2 then
93          add ⟨m1, m2⟩ to M_d
94  while time-budget B is not expired do
95      ⟨m1, m2⟩ ← GETNEXTMETHODPAIR(M_d)
96      for each i from 1 to 2 do
97          ⟨sout1, seq1⟩ ← GETRANDOMCONSTRUCTOR(CUT)
98          ⟨sout2, seq2⟩ ← GETRANDOMCONSTRUCTOR(CUT)
99          prefix ← seq1 ∪ seq2
100         if i % 2 = 0 then
101             prefix ← APPENDSTATECHANGERDEADLOCK(sout1,sout2 prefix)
102         suffix1 ← GETSUFFIXDEADLOCK(m1, sout1, sout2)
103         suffix2 ← GETSUFFIXDEADLOCK(m2, sout1, sout2)
104         ct ← ASSEMBLETEST(prefix, suffix1, suffix2)
105         for each iter from 1 to N-iter do
106             deadlock ← EXECUTE(ct)
107             if deadlock ≠ ∅ ∧ deadlock ∉ ALLPOSSIBLELINEARIZATIONS(ct) then
108                 return ct                /* thread-safety violation (deadlock type) */
109 return ∅
```

Given a method pair $\langle m_1, m_2 \rangle \in \mathcal{M}_d$ (line 95), Algorithm 4 generates two concurrent tests. For both concurrent tests, it creates the *prefix* by combining two method call sequences seq_1 and seq_2 (line 99). Both seq_1 and seq_2 instantiate a shared object under test of type CUT, by selecting a random constructor with random input parameters (Function GETRANDOMCONSTRUCTOR). seq_1 instantiates $sout_1$, while seq_2 instantiates $sout_2$. For example, DEPCON$^+$ generates the concurrent test in Figure 8 with a prefix containing two method calls that create two objects of type CUT: `Hashtable sout1 = new Hashtable()` and `Hashtable sout2 = new Hashtable()`.

This is a major difference with the *"exception mode"* of DEPCON$^+$. In line with most concurrent test generators [10], the *"exception mode"* reduces the search space of possible concurrent tests by generating tests with exactly one instance of the CUT (*sout*) shared among threads. This assumption is reasonable for exposing thread-safety violations of exception type [17]. In fact, generating suffixes that concurrently access a single object promotes shared-memory contentions among the concurrent threads, while accessing different objects reduces such a contention (different objects point to different memory locations). However, sharing different objects among concurrent threads is often necessary to expose resource deadlocks. Indeed, shared object instances are often used as locks, and we need at least two distinct locks to expose resource deadlocks [19]. For instance, the failure-inducing concurrent test in Figure 4 needs two thread-shared objects of type CUT (`sout1` and `sout2`) to expose the resource deadlock. Figures 8 and 7 show two additional examples of DEPCON$^+$ generated concurrent tests that manifest thread-safety violations of deadlock type. They also require that the suffixes concurrently access two objects under test.

Similarly to the *"exception mode"*, Algorithm 4 generates two concurrent tests for a given method pair. The first test uses the *prefix* as it is, without adding any additional method call after the two constructor calls, for instance, the prefix of the concurrent test in Figure 7. The second test adds additional method calls to the *prefix* to change the states of $sout_1$ and $sout_2$. For instance, the prefix of the concurrent test in Figure 8. Function APPENDSTATECHANGERDEADLOCK appends to the

Prepared using stvrauth.cls
DOI: 10.1002/stvr

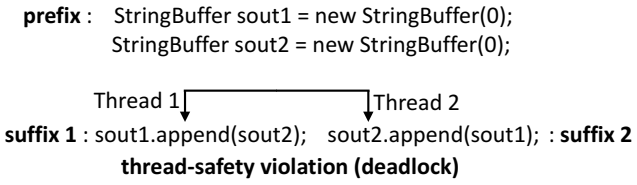

Figure 7. Example of concurrent test that manifests a deadlock in `StringBuffer`.

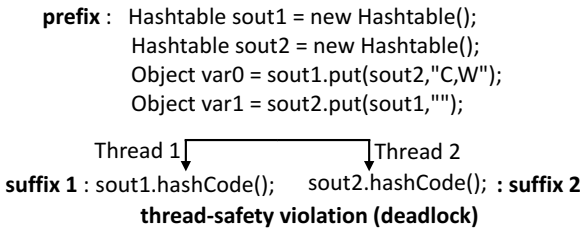

Figure 8. Example of concurrent test that manifest a deadlock in `HashTable`

end of *prefix* a certain number of method calls that use either $sout_1$ and $sout_2$ (or both) as a method parameter. The function chooses this number randomly within the 1, *Max-len* interval. Function APPENDSTATECHANGERDEADLOCK is carefully designed to favor method call sequences that increase the chance of creating the circumstances for deadlocks, as prescribed by the double-lock dependency ($\rightleftharpoons$D). More specifically, the function is biased to use method calls that use both $sout_1$ and $sout_2$ as method parameters. For instance, it selects the method `put(java.lang.Object, java.lang.Object)` for the prefix in Figure 8 with both $sout_1$ and $sout_2$ as parameters: `sout1.put(sout2,"a")`. Both $sout_1$ and $sout_2$ are objects of type `Hashtable`, thus $sout_1$ can be the receiver of method `put(java.lang.Object, java.lang.Object)` and $sout_2$ can be an input parameters because `java.lang.Object` is a superclass of `Hashtable`. Function APPENDSTATECHANGERDEADLOCK ensures that also the symmetric case is added to the prefix: `sout2.put(sout1,"a")`. The rationale of this design choice is that these symmetric method calls may set both the $sout_1$ reference to the fields in $sout_2$ and the $sout_2$ reference to the fields in $sout_1$. If this happens, concurrently invoking two methods in the concurrent suffixes that access $sout_1$ and $sout_2$ may acquire the same locks in opposite order, thus triggering a deadlock. This is because the methods that are used in the suffixes manifest double-lock dependencies.

For instance, consider the deadlock revealing concurrent test in Figure 8. $suffix_1$ invokes the method hashcode using `sout1` as the object receiver. Such an invocation acquires both the lock associated with `sout1` and the lock associated with the key of the hashtable. The call `sout1.put(sout2,"C,W")` in the prefix set `sout2` as the key of the hashtable `sout1`. As such, the invocation `sout1.hashCode()` of Suffix 1 leads to a nested acquisitions of the locks `sout1` and `sout2`. If another threads concurrently invokes `sout2.hashCode()`, a deadlock can occur, because the prefix also sets `sout1` as the key of the hashtable `sout2` (`sout2.put(sout1,"")`). Function GETSUFFIXDEADLOCK is carefully designed to ensure that $suffix_1$ uses `sout1` and $suffix_2$ uses `sout1` as method parameters.

Line 104 of Algorithm 4 generates a new concurrent test ct by assembling the obtained method call sequences. Similarly to the *"exception mode"* , DEPCON$^+$ explores the interleaving space of ct relying on the non-determinism of the JVM scheduler.

To detect if one of the explored interleavings triggers a deadlock, DEPCON$^+$ executes in parallel a runtime deadlock monitor. At constant intervals the monitor checks if any of the executing threads is deadlocked by invoking the `java.lang.ManagementFactory` class, which represents the management interface of the JVM. If `java.lang.ManagementFactory` detects than one or

Prepared using stvrauth.cls DOI: 10.1002/stvr

more threads are deadlocked, DEPCON$^+$ kills the deadlocked threads and reports a thread-safety violation if the same deadlock does not manifest when executing each *linerarizations* of ct. Otherwise, DEPCON$^+$ iterates generating new concurrent tests until the time-budget expires.

6.3. Limitations of DEPCON$^+$

This section discusses the limitations of DEPCON$^+$ and the assumptions it makes on the target programs.

Target concurrency faults As discussed in Section 2.5 and 2.6, DEPCON$^+$ targets those concurrency faults in thread-safe classes that manifest uncaught exceptions or endless hangs. Moreover, DEPCON$^+$ targets one common type of endless hang fault: ABBA resource deadlock. This deadlock manifests when there are exactly two threads that trigger a cyclic deadlock involving exactly two locks.

Synchronization mechanisms Our static analysis can only handle Java programs that use synchronized methods and blocks as synchronization mechanisms. Examples of alternative Java synchronization mechanisms are `CompareAndSwap` (often used by the JAVA classes in the package `java.util.concurrent.locks`) and wait/notify synchronization. Differently from synchronized methods and blocks, both of these alternative mechanisms do not generate monitor instructions and allow that multiple lock acquisitions can be paired with multiple lock releases depending on control flows.

Static analysis limitations The preprocessing phase of DEPCON$^+$ inherits the fundamental limitations of static analysis for object-oriented programs. DEPCON$^+$ cannot guarantee sound and precise results in the presence of Java reflection, dynamic binding and dynamic dispatch. For example, in the presence of polymorphic methods, dynamic dispatch selects at runtime which implementation of the polymorphic method to call. Because super class variables can reference sub class objects, static analysis cannot know which concrete method will be executed. In fact, whether a variable is a super class or sub class object might depend on the calling context of the client code (concurrent test in our case), and thus it can only be known at runtime. DEPCON$^+$ conservatively considers the method implementation of the super class, and this might lead to unsound and imprecise results.

Path-insensitive analysis The alias analysis of DEPCON$^+$ performs a path-insensitive backward analysis on the JVM stack. The analysis might lead to unsound and imprecise results when it encounters branch conditions. For example, if two branches contain an assignment to the same variable, DEPCON$^+$ path-insensitivity will consider only the last assignment that it encounters. This might lead to unsound and imprecise aliases. However, alias analysis remains undecidable for Java programs [67] and DEPCON$^+$ cannot guarantee to compute precise alias information even with a path sensitive alias analysis.

Test generation assumptions The exception mode of DEPCON$^+$ generates concurrent tests that (i) have exactly two concurrent threads, (ii) do not directly invoke static methods in the concurrent suffixes, and (iii) have exactly one thread-shared object under test. The deadlock mode of DEPCON$^+$ maintains the first two assumptions, while it only generates concurrent tests with one or two thread-shared objects under test. These assumptions help DEPCON$^+$ to reduce the search space of possible concurrent tests, while detecting the most common types of concurrency faults. In fact, the concurrency bug characteristic study of Lu et al. shows that most concurrency faults can be detected under these three assumptions [28].

Prepared using stvrauth.cls DOI: 10.1002/stvr

Table I. Subjects description

ID	Code Base	Version	Package	Class Name	LOC	# Methods	# Public Methods	Fault Type
C1	Apache Math	2.4	org.apache.commons.lang.math	IntRange	276	48	26	Atomicity
C2	Apache DBCP	1.4	org.apache.commons.dbcp.datasources	PerUserPoolDataSource	719	84	66	Data race
C3		1.4	org.apache.commons.dbcp.datasources	SharedPoolDataSource	546	68	52	Atomicity
C4	HSQLDB	2.3.3	org.hsqldb.lib	DoubleIntIndex	966	55	34	Atomicity
C5	JDK	1.1	java.io	BufferedInputStream	239	34	10	Atomicity
C6		1.4.2	java.util	Vector	786	80	45	Atomicity
C7	JFreeChart	1.0.13	org.jfree.data.time	Day	267	44	26	Data race
C8		0.9.12	org.jfree.chart.axis	NumberAxis	1,662	154	111	Atomicity
C9		1.0.1	org.jfree.chart.axis	PeriodAxis	1,975	173	126	Data race
C10		0.98	org.jfree.data.time	TimeSeries	359	49	41	Data race
C11		1.0.9	org.jfree.chart.plot	XYPlot	3,080	259	218	Data race
C12		0.9.8	org.jfree.data	XYSeries	200	32	25	Data race
C13	Log4J	1.0	org.apache.log4j	FileAppender	369	37	21	Atomicity
C14		1.0	org.apache.log4j	WriterAppender	317	40	24	Atomicity
C15	XStream	1.4.1	com.thoughtworks.xstream	XStream	926	87	66	Data race
C16	JDK	1.4.1	java.lang	Stringbuffer	1,278	40	33	Deadlock
C17		1.4	java.util	Hashtable	1,058	35	19	Deadlock
C18		1.4	java.util	Hashtable	1,058	35	19	Deadlock
C19		1.4.2	java.util	SynchronizedMap	79	18	15	Deadlock

7. EVALUATION

We empirically evaluated DEPCON$^+$ with a prototype implementation for Java classes. We implemented Algorithm 1 and 2 by relying on the bytecode manipulation framework ASM [68] to scan the bytecode instructions of Java methods. We implemented the test generation Algorithm 3 and 4 on top of the publicly available implementation of COVCON [8].

We evaluated the *"exception mode"* of DEPCON$^+$ with 15 thread-safe classes with known thread-safety violations of exception type. We evaluated the *"deadlock mode"* of DEPCON$^+$ with four thread-safe classes with known thread-safety violations of deadlock type. In both cases we compared DEPCON$^+$ with COVCON, the state-of-the-art in concurrent test generator [8], which targets thread-safety violations of both exception and deadlock types.

We address five research questions:

RQ1 Effectiveness (exception mode) *Does* DEPCON$^+$ *effectively generate concurrent tests that expose thread-safety violations of exception type?*

RQ2 Comparison (exception mode) *Is* DEPCON$^+$ *more effective than state-of-the-art concurrent test generation in exposing thread-safety violations of exception type?*

RQ3 Effectiveness (deadlock mode) *Does* DEPCON$^+$ *effectively generate concurrent tests that expose thread-safety violations of deadlock type?*

RQ4 Comparison (deadlock mode) *Is* DEPCON$^+$ *more effective than state-of-the-art concurrent test generation in exposing thread-safety violations of deadlock type?*

RQ5 Preprocessing Phase *What is the efficiency, completeness and precision of* DEPCON$^+$ *preprocessing phase?*

7.1. Subjects

We selected 19 thread-safe classes with known thread-safety violations that are used in the evaluation of previous work [3, 8, 10, 27, 46]. Table I shows the details of the faulty classes under test (CUT). Column "*ID*" assigns an ID that we use to identify the class in the paper. Column "*Code Base*" reports the subject program that contains the faulty class. Column "*Version*" gives the faulty version of the code base. Columns "*Package*" and "*Class Name*" indicate the package and name of the CUT,

Prepared using stvrauth.cls DOI: 10.1002/stvr

respectively. Column "*LOC*" shows the lines of code of the CUT, which range from 79 to 3,080. Column "*# Methods*" gives the number of public, protected or private methods of the CUT. Column "*# Public Methods*" shows the number of public methods only. We count the number of methods considering the methods declared in the CUT and the non-abstract public methods inherited from the superclasses of the CUT (excluding `java.lang.Object`). Indeed, DEPCON$^+$ considers also inherited public methods when generating concurrent tests. Column "*Fault Type*" indicates the type of concurrency fault for each subject: *Atomicity violation*, *Data race* and *Deadlock*. The concurrency faults in the classes C1 to C15 manifest thread safety violations of exception type, while classes C16 to C19 of ABBA resource deadlock type.

7.2. Evaluation Setup

We ran DEPCON$^+$ in exception mode for the classes with ID from C1 to C15, DEPCON$^+$ in deadlock mode for the classes with ID from C16 to C19, and COVCON for all classes. Following related work [8, 9, 10], we chose a time-budget of one hour per class ($\mathcal{B} = 1$ hr). Each run terminates when the technique either successfully exposes the thread-safety violation or exhausts the time-budget. Both DEPCON$^+$ and COVCON rely on the non-determinism of the default JVM scheduler to explore the interleaving space. The original implementation of COVCON uses one iteration for the interleaving explorer (*N-iter* $= 1$), which we found not adequate to expose the failure-inducing interleaving in many cases. The choice of the number of iterations is important: too few iterations may miss a fault-revealing interleaving even if the concurrent test can exhibit one, while too many waste testing resources. For both DEPCON$^+$ and COVCON we used one hundred iterations (*N-iter* $= 100$), which is a good trade-off between effectiveness and cost. We set the maximum length of the prefixes at ten (*Max-len* $= 10$). We cope with the randomness of the tools, by repeating each experiment five times with different random seeds. To guarantee repeatability of the generated tests, we used the random seeds from 1 to 5, as both COVCON and DEPCON$^+$ generate tests pseudo-deterministically given a random seed. We executed our experiment on a server Ubuntu 16.04.2 with 64 octa-core CPUs Intel® Xeon® E5-4650L @ 2.60GHz and ∼529 GB of RAM

Several of our subjects are shared with the ones used in the evaluation of COVCON [8]. However, we expect similar (but not identical) results, because we used a different *N-iter* value, different random seeds, and different hardware. Also, both COVCON and DEPCON$^+$ rely on the JVM scheduler to explore the interleaving space, which behavior is intrinsically non-deterministic.

We measure the effectiveness of DEPCON$^+$ and COVCON with the following three metrics:

Success Rate (SR): **1** if the technique detects a thread-safety violation within the time-budget $\mathcal{B}$, **0** otherwise.

Fault Detection Time (FDT): time taken by the technique to expose the thread-safety violation, $\mathcal{B}$ if the time-budget expires.

 # Generated Tests (#GT): number of generated concurrent tests when $\mathcal{B}$ expires or when the technique exposes a thread-safety violation.

7.3. RQ1 Effectiveness (Exception Mode)

Columns "DEPCON$^+$ *(RQ1)*" of Table II show the results of RQ1. DEPCON$^+$ has 100% Success Rate (SR) for seven subjects: *C1*, *C3*, *C5*, *C7*, *C9*, *C10* and *C12* (Column "DEPCON$^+$ *(RQ1)* – *Success Rate*"). This means that DEPCON$^+$ exposed the thread-safety violations in all five runs for these subjects. The average DEPCON$^+$ SR is 68%. DEPCON$^+$ SR is always greater than 0%, meaning that DEPCON$^+$ successfully exposed the thread-safety violations in at least one run for all the subjects. This result empirically confirms the validity of Theorem I: pruning the search space via parallel and conflict dependencies does not prevent the generation of failure-inducing tests.

Prepared using stvrauth.cls DOI: 10.1002/stvr

Table II. Evaluation Results of Exception Mode

ID	DepCon+ (RQ1)				CovCon [8]				Comparison (RQ2)			
	$\mathcal{M}_{pc}$ size	Success Rate	Avg. FDT (hh:mm:ss)	Avg. #GT	$\mathcal{M}$ size	Success Rate	Avg. FDT (hh:mm:ss)	Avg. #GT	$\mathcal{M}_{pc}$ reduction	SR improv.	FDT speedup	#GT reduction
C1	21	100%	00:01:21	188	351	40%	00:36:15	6,660	16.71×	+60%	26.72×	35.35×
C2	66	40%	00:43:57	2,810	2,211	40%	00:52:36	9,055	33.50×	-	1.20×	3.22×
C3	52	100%	00:11:51	1,221	1,378	20%	00:56:00	8,947	26.50×	+80%	4.73×	7.33×
C4	297	60%	00:34:13	2,617	595	100%	00:10:38	1,055	2.00×	-40%	0.31×	0.40×
C5	22	100%	00:00:07	18	55	100%	00:00:34	161	2.50×	-	5.21×	8.77×
C6	51	20%	00:51:11	1,174	1,035	20%	00:54:52	5,010	20.29×	-	1.07×	4.27×
C7	70	100%	00:01:24	255	351	100%	00:03:03	640	5.01×	-	2.17×	2.51×
C8	292	40%	00:39:50	5,860	6,216	0%	01:00:00	12,368	21.29×	+40%	1.51×	2.11×
C9	278	100%	00:03:15	438	8,001	100%	00:09:52	1,720	28.78×	-	3.04×	3.93×
C10	296	100%	00:02:19	370	861	100%	00:15:56	2,895	2.91×	-	6.88×	7.82×
C11	844	40%	00:43:54	4,113	23,871	20%	00:57:56	4,320	28.28×	+20%	1.32×	1.05×
C12	114	100%	00:00:33	116	325	100%	00:07:47	1,842	2.85×	-	14.05×	15.88×
C13	53	20%	00:48:18	2,622	231	0%	01:00:00	16,264	4.36×	+20%	1.24×	6.20×
C14	37	20%	00:48:06	2,609	300	0%	01:00:00	15,911	8.11×	+20%	1.25×	6.10×
C15	427	80%	00:11:44	222	2,211	80%	00:26:54	568	5.18×	-	2.29×	2.56×
Avg.	**195**	**68%**	**00:22:48**	**1,642**	**3,199**	**55%**	**00:34:10**	**5,828**	**13.89×**	**+13%**	**4.87×**	**7.1×**

The Average Failure Detection Time (FDT) of DepCon+ ranges between 7 seconds for *C5* to 51 minutes and 11 seconds for *C6*, with an average of 22 minutes and 48 seconds (Column "DepCon+ *(RQ1) – Avg. FDT*"). Note that, FDT includes the execution time of all phases of DepCon+: the computation of the method summaries and of the parallel and conflict dependencies, the generation of concurrent tests, and the interleaving exploration.

DepCon+ generates an average number of concurrent tests that ranges from 18 tests for *C5* to 5,860 tests for *C8*, with an average of 1,642 tests (Column "DepCon+ *(RQ1) – Avg. #GT*"). This indicates that DepCon+ explores relatively few concurrent tests before exposing the thread-safety violations.

7.4. RQ2 Comparison (Exception Mode)

Columns "CovCon [11]" of Table II show the results of CovCon, and columns "*Comparison (RQ2)*" compare DepCon+ and CovCon, thus addressing RQ2.

CovCon has 100% Success Rate (SR) for six subjects: *C4*, *C5*, *C7*, *C9*, *C10* and *C12*. CovCon fails to expose the thread-safety violations in all runs for three subjects: *C8*, *C13* and *C14*. The CovCon average SR is 55%, which is lower than the one of DepCon+ (68%). Column "*Comparion (RQ2) – SR improv.*" shows the improvement in success rate of DepCon+ over CovCon. DepCon+ has a higher success rate than CovCon for 6 out of the 15 subjects. Only for *C4*, DepCon+ has a lower success rate than CovCon. Our manual investigation of the generated tests suggests that the faulty interleaving for Subject *C4* has a low chance of manifestation. As such, the interleaving explorer was not able to expose the faulty interleaving for many generated tests, even if those tests could manifest the faulty interleaving. More sophisticated interleaving explorers might overcome this issue [8].

The average Failure Detection Time (FDT) of CovCon ranges between 3 minutes and 33 seconds for *C7* and 1 hour for the three subjects in which CovCon fails to expose the thread-safety violation before the time-budget expires. The average CovCon FDT across all subjects is 34 minutes and 10 seconds, which is higher than the one of DepCon+ (22 minutes and 48 seconds). Column "*Comparison (RQ2) – FDT speedup*" in Table II indicates the speedup of "Avg. FDT" of DepCon+ over CovCon. The speedup ranges from 0.31× for Subject *C4* to 26.72× for *C1* (4.87× on average). DepCon+ exposes the thread-safety violations faster than CovCon for all subjects but *C4*.

The average number of Generated Tests (#GT) of CovCon ranges from 161 concurrent tests for Subject *C5* and 16,264 for *C13* (5,824 on average). Column "*Comparison (RQ2) – #GT reduction*"

Prepared using stvrauth.cls DOI: 10.1002/stvr

Table III. Evaluation Results of deadlock mode

	DEPCON+ (RQ3)				COVCON [8]				Comparison (RQ4)			
ID	$\mathcal{M}_d$ size	Success Rate	Avg. FDT (hh:mm:ss)	Avg. #GT	$\mathcal{M}$ size	Success Rate	Avg. FDT (hh:mm:ss)	Avg. #GT	$\mathcal{M}_d$ reduction	SR improv.	FDT speedup	#GT reduction
C16	1	100%	00:00:18	22	561	0%	01:00:00	5,792	561.00×	+100%	200.00×	263.27×
C17	3	100%	00:00:04	11	190	0%	01:00:00	2,370	63.33×	+100%	900.00×	215.45×
C18	3	100%	00:00:34	45	190	0%	01:00:00	4,625	63.33×	+100%	105.88×	102.77×
C19	15	100%	00:00:05	34	120	0%	01:00:00	37,827	8×	+100%	720.00×	1,112.56×
Avg.	**5**	**100%**	**00:00:15**	**28**	**265**	**0%**	**01:00:00**	**12,653**	**173.91**×	**+100%**	**481.47**×	**423.51**×

of Table II quantifies the average reduction of the DEPCON+ test suites over the COVCON ones. The smaller sizes of DEPCON+ test suites over COVCON test suites explain the speedup of DEPCON+ over COVCON. Column "*Comparison (RQ2) – #GT reduction* indicates that DEPCON+ generated and explored the interleaving space of 7.1× less tests than COVCON before exposing the thread-safety violations. This result demonstrates the effectiveness of DEPCON+ to reduce the search space and to drive the test generation towards failure-inducing concurrent tests.

7.5. RQ3 Effectiveness (Deadlock Mode)

Columns "DEPCON+ *(RQ3)*" of Table III show the results of RQ3. The Success Rate (SR) of DEPCON+ is 100% for all the deadlock subjects, meaning that DEPCON+ exposes the deadlock in all five runs for each subject. This result empirically confirms that Theorem 2 is empirically sound: pruning the search space via double-lock dependencies does not prevent the generation of failure-inducing tests. Figures 7, 4 and 8 show the concurrent tests that DEPCON+ generates to expose the thread-safety violation for C16, C17 and C18, respectively.

Column "DEPCON+ *(RQ3)* – Avg. FDT" gives the average Failure Detection Time (FDT) of DEPCON+, which ranges from 4 seconds for *C17* to 34 seconds for *C18* (15 seconds on average).

Column "DEPCON+ *(RQ3)* – Avg. #GT" shows the average number of generated tests (#GT) of DEPCON+, which ranges from 11 concurrent tests for Subject *C17* to 45 for *C18* (28 on average). This indicates that DEPCON+ explores just a few concurrent tests before generating a test that exposes the deadlock.

The effectiveness of DEPCON+ is explained by Column "DEPCON+ (RQ3) – $\mathcal{M}_d$ size", which indicates the number of method pairs with double-lock dependencies. Because there are only a few of such method pairs, DEPCON+ explores a small portion of the search space that contain only those tests that invoke such method pairs in the concurrent suffixes. Due to Theorem 2, DEPCON+ guarantees that if a class can manifest a thread-safety violation of ABBA resource deadlock type, only that small portion of the search space contains tests that can expose the deadlock. Moreover, all of these four concurrency faults require symmetric calls in the suffixes and/or prefixes, and DEPCON+ is designed to favor such method calls.

Interestingly, the exception mode of DEPCON+ (DEPCON [17]) does not detect the deadlock in any of the subjects C16-C19. This is because the method pairs that lead to the deadlock do not have both parallel and conflict dependencies. This confirms the complementarity of the exception and deadlock modes of DEPCON+.

7.6. RQ4 Comparison (Deadlock Mode)

Columns "COVCON [11]" and "*Comparison (RQ4)*" of Table III show the results of COVCON and a summary comparison of DEPCON+ and COVCON, respectively.

The Success Rate (SR) of COVCON is 0% for all four deadlock subjects (Column "COVCON [11] – *Success Rate*"), and thus the average Failure Detection Time (FDT) of COVCON is the time budget $\mathcal{B}$ of one hour (Column "COVCON [11] – *Avg. FDT*"). The average number of Generated Tests (#GT) of COVCON ranges from 2,370 concurrent tests for *C17* to 37,827 for *C19* (12,653 on average). Even

Prepared using stvrauth.cls DOI: 10.1002/stvr

Table IV. Evaluation results of the preprocessing phase

ID	# public methods	$\mathcal{M}$ size	$\mathcal{M}_{pc} \cup \mathcal{M}_d$ size	$\mathcal{M}_{pc} \cup \mathcal{M}_d$ reduction	$\mathcal{M}_{pc}$ size	$\mathcal{M}_{pc}$ reduction	$\mathcal{M}_d$ size	$\mathcal{M}_d$ reduction	AVG time (ms)
C1	26	351	21	94%	21	94%	0	100%	1,601
C2	66	2,211	66	97%	66	97%	0	100%	1,898
C3	52	1,378	52	96%	52	96%	0	100%	1,800
C4	34	595	297	50%	297	50%	0	100%	2,119
C5	10	55	24	56%	22	60%	2	96%	1,553
C6	45	1,035	51	95%	51	95%	0	100%	2,040
C7	26	351	70	80%	70	80%	0	100%	2,187
C8	111	6,216	292	95%	292	95%	0	100%	1,892
C9	126	8,001	278	97%	278	97%	0	100%	1,978
C10	41	861	296	66%	296	66%	0	100%	2,022
C11	218	23,871	844	96%	844	96%	0	100%	2,819
C12	25	325	114	65%	114	65%	0	100%	1,932
C13	21	231	54	77%	53	77%	1	100%	1,563
C14	24	300	38	87%	37	88%	1	100%	1,582
C15	66	2,211	427	81%	427	81%	0	100%	4,711
C16	33	561	57	90%	56	90%	1	100%	1,618
C17	19	190	11	94%	8	96%	3	98%	1,599
C18	19	190	10	95%	7	96%	3	98%	1,611
C19	15	120	33	72%	18	85%	15	87%	2,101
AVG	**51**	**2,582**	**160**	**83%**	**158**	**84%**	**1**	**99%**	**2,033**

after generating such a high number of tests, CovCon did not expose the deadlocks. This suggests that CovCon generated and explored the interleaving space of many irrelevant tests. Conversely, DepCon$^+$ reduces the search space of possible concurrent tests and to drive test generation towards those tests that might trigger deadlocks.

Indeed, DepCon$^+$ drastically reduces the number of coverage targets (method pairs) with respect to CovCon, which does not perform any search space pruning. Column "*Comparison (RQ4) – $\mathcal{M}_d$ reduction*" gives the reduction of the coverage targets, which ranges from 8× for C19 to 561× for C16. Column "*Comparison (RQ4) – FDT speedup*" gives the speedup of "*Avg. FDT*" of DepCon$^+$ over CovCon. The speedup ranges from 105.88× for C18 to 900.00× for C17. Column "*Comparison (RQ4) – #GT reduction* indicates that, on average, DepCon$^+$ generated and explored the interleaving space of 423.51× less tests than CovCon before exposing the deadlocks. This result demonstrates the effectiveness of DepCon$^+$ to reduce the search space and to drive the test generation towards those concurrent tests that manifest ABBA resource deadlocks.

7.7. RQ5 Preprocessing Phase

Table IV shows the results of the preprocessing phase for each of the 19 subjects. Column "*# public methods*" shows the number of public methods of the subjects. Column "$\mathcal{M}$ *size*" indicates the number of all possible pairs of public methods (the coverage requirements of CovCon [8]). Column "$\mathcal{M}_{pc} \cup \mathcal{M}_d$" shows the number of method pairs that are either in $\mathcal{M}_{pc}$ or in $\mathcal{M}_d$ (or both), and the reduction with respect to $\mathcal{M}$. The reduction ranges from 50% to 97% (83% on average). Column "$\mathcal{M}_{pc}$" gives the number of method pairs that have both conflict and parallel dependencies (pairs that belong to $\mathcal{M}_{pc}$), and the reduction with respect to $\mathcal{M}$. The reduction ranges from 65% to 97% (84% on average). Column "$\mathcal{M}_d$" reports the number of method pairs that have double-lock dependencies (pairs that belong to $\mathcal{M}_d$), and the reduction with respect to $\mathcal{M}$. The reduction ranges from 87% to 100% (99% on average).

The effectiveness and efficiency of DepCon$^+$ depend on the preprocessing phase that shall be: (i) complete, that is, it identifies all parallel and conflict dependencies; (ii) mostly precise, that is, it identifies as few as possible spurious parallel and conflict dependencies; (iii) efficient, that is, it has a low computational cost. Completeness of the preprocessing phase guarantees that the search space

Prepared using stvrauth.cls DOI: 10.1002/stvr

pruning of DEPCON$^+$ does not miss tests that manifest thread-safety violations. Precision of the preprocessing phase ensures that DEPCON$^+$ explores a relatively small number of concurrent tests to identify the failure-inducing ones, if they exist. Efficiency of the preprocessing phase is essential as a high computational cost would out-weight the reduction in test effort. Indeed, the preprocessing phase is an additional step to the test generation pipeline that should have a negligible cost.

Efficiency Column "*AVG time (ms)*" of Table IV shows the computation cost in milliseconds of the preprocessing phase (Algorithm 1 and 2), which ranges from 1,533 ms for Subject $C5$ to 4,711 ms for $C15$ (2,033 ms on average). We report the average time of five runs of DEPCON$^+$. It includes the time for computing all three method summaries: access, lock and double-lock, and it includes the computation time of the three method dependencies. This result demonstrates the efficiency of our proposed static computation of method summaries. To put it in perspective, the average computation time of the preprocessing step across all subjects is around 2 seconds, while the average Failure Detection Time (FDT) of DEPCON$^+$ exception mode is 22 minutes and 44 seconds, while the average FDT of DEPCON$^+$ deadlock mode is 15 seconds.

Evaluating the precision and completeness of the preprocessing phase is difficult because the ground truth of the method dependencies cannot be easily obtained. However, we can evaluate them indirectly.

Completeness Because DEPCON$^+$ exposes the thread-safety violations in at least one run for each of the 19 subjects, the preprocessing phase is complete. That is, the search space pruning of DEPCON$^+$ did not prevent the detection of the thread-safety violations. Therefore, for the 19 subjects considered, DEPCON$^+$ correctly computes the parallel, conflict and double-lock dependencies of the failure-inducing method pairs.

Precision Table IV shows that DEPCON$^+$ drastically reduces the number of coverage targets. Therefore, the precision of the preprocessing phase is good enough to make the computed parallel and conflict dependencies useful. It is important to clarify that DEPCON$^+$ could hardly achieve a perfect precision due to the intrinsic imprecision and over-approximation of static analysis. DEPCON$^+$ consciously makes conservative choices when computing the method summaries in Algorithm 1 and 2 to avoid missing any parallel, conflict or double-lock dependencies.

The reader should notice that, according to Theorem 1 and 2, the presence of the method dependencies is only a necessary condition (but not sufficient) to expose thread-safety violations. For instance, although the subjects from C1 to C15 manifests thread-safety violations as uncaught exceptions, subjects C5, C13 and C14 have method pairs with double-lock dependencies. However, subjects C5, C13 and C14 do not manifest deadlocks. Similarly, also the subjects C16, C17, C18 and C19 have method pairs with parallel and conflict dependencies without manifesting thread-safety violations of exception type.

7.8. Threats To Validity

A major threat to external validity is whether our results generalize to other subjects. We mitigated this threat by including subjects of seven popular code bases that were used in the evaluation of related work. All the concurrency faults considered in our experiments are known thread-safety issues in previous versions of the considered thread-safe classes.

A threat to internal validity is whether there were implementation errors in our prototype that affected the results. We mitigated this threat by testing the most important parts of our implementation. In addition, we manually validate all reported thread-safety violations of DEPCON$^+$ to make sure that the result was correct: the reported concurrent test indeed exposes a thread-safety violation.

Prepared using stvrauth.cls DOI: 10.1002/stvr

8. RELATED WORK

In their recent survey of testing concurrent systems [7], Bianchi et al. distinguish two complementary sets of techniques: *interleaving explorers* and *test generators*. Interleaving explores navigate the interleaving space of a concurrent test to expose concurrency failures [6, 19, 24, 37, 38, 39, 69, 70]. The fault detection capability of these techniques depends on the capability of the test to trigger faulty interleavings. In other words, interleaving exploration techniques cannot detect concurrency faults if the input tests cannot induce failure-inducing interleavings. To address such critical limitation, researchers proposed test generators for concurrent programs [3, 8, 9, 27, 71] that automatically generate concurrent tests that can be used as inputs for interleaving explorers. DEPCON$^+$ belongs to this latter group of techniques.

We discuss the related work of DEPCON$^+$: test generators for concurrent programs (Section 8.1), test generators for thread-safe classes (which are closely related to DEPCON$^+$, Section 8.2), approaches to reduce the search space during concurrent test generation (Section 8.3), and various static analyses for concurrent object-oriented programs (Section 8.4).

8.1. Test Generators for Concurrent Programs

Dynamic (concolic) Symbolic Execution (DSE) executes a program to automatically generate input values to improve code coverage and expose software faults. Researchers have proposed various DSE techniques for concurrent programs [71, 72, 73, 74]. However, DSE alone cannot generate OO tests [35, 75]. A OO test for a given method is no longer a set of input values but a sequence of method calls [76]. This is because the behavior of a method invocation often depends on the states of its non-primitive parameters, e.g., the object receiver [75]. To ensure valid program states, most test generators for OO classes explore the space of possible object states by generating method call sequences that exercise the program under test through its public interface [35, 75]. Such sequences instantiate non-primitive parameters and bring them to certain (fault-revealing) states [35, 75].

8.2. Test generators for Thread-Safe classes

When dealing with thread-safe classes a concurrent test is a set of method call sequences that exercise from multiple threads the public interface of a thread-safe class under test [3]. Concurrent test generators for thread-safe classes can be divided in *random-based*, *coverage-based* and *sequential-test-based* techniques [10]. We refer interested readers to our recent survey on the effectiveness and challenges of concurrent test generators for thread-safe classes [10].

Random-based techniques [3, 12] generate concurrent tests by combining randomly generated method call sequences with random input parameters. Random-based techniques can efficiently generate concurrent tests because they do not require complex analysis. They can generate thousands of concurrent tests in a few seconds. Random-based techniques has been shown to be less effective in revealing hard-to-find concurrency faults, because randomly generated tests tend to repetitively test similar program behaviors [8, 9, 10]. 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 [3]. For example, CONTEGE [3] requires more than a million tests to expose a single concurrency fault [3]. 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 [8, 9, 12].

Coverage-based techniques address the limitations of random-based techniques by driving the generation of concurrent tests with interleaving coverage criteria [8, 9, 11]. 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. DEPCON$^+$ belongs to this category as it relies on concurrent method pairs [8, 66] as coverage targets.

Prepared using stvrauth.cls DOI: 10.1002/stvr

Sequential-test-based techniques [27, 77, 78, 79] apply the same overall approach to different kinds of concurrency faults: deadlocks [77], data races [27], atomicity violations [78], and assertion violations [79]. They analyze concurrent programs starting from a suite of sequential (single-threaded) tests, which can be either manually-written or generated by existing sequential test generators [35,75]. 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. The effectiveness of sequential-test-based techniques 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 [75].

None of these techniques perform dependency analysis among methods to reduce the search space during test generation. As discussed in our recent empirical study [10], the huge number of possible concurrent tests hinders the effectiveness of these techniques. 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. This motivated us to present DEPCON$^+$ and reduce the search space when generating concurrent tests. We expect that the effectiveness of all previous generators of concurrent tests would improve if they include the search space pruning of DEPCON$^+$ (see RQ2).

8.3. Reducing the Search Space During Concurrent Test Generation

Schimmel et al. proposed in a workshop paper AUTORT [80] that shares a similar goal with DEPCON$^+$. AUTORT proposes the use of parallel and conflict analysis to reduce the number of concurrent tests to be generated. However, the scope and approach of AUTORT and DEPCON$^+$ differ substantially. First, AUTORT delegates to the developers the responsibility to re-run the software under test with method call sequences that cover all parallel program parts [80]. Instead, DEPCON$^+$ automatically generates such method call sequences. In addition, DEPCON$^+$ relies on conflict dependency analysis to generate meaningful prefixes. Second, differently from DEPCON$^+$, AUTORT does not do inter-procedural analysis while statically analyzing a method, and thus it would likely miss conflict dependencies. DEPCON$^+$ provides more accurate results that do not miss dependencies and lead to a better search space reduction. For example, via inter-procedural analysis DEPCON$^+$ checks if transitive method invocations could perform write instructions on shared-memory locations. Third, AUTORT identifies method pairs that do not run in parallel by dynamically executing them on an instrumented version of the program. This solution could miss real parallel dependencies if the methods interleave but AUTORT observes only those executions in which they do not. Conversely, DEPCON$^+$ performs the parallel analysis statically, thus avoiding the cost of generating and running tests and without suffering from the incompleteness of a dynamic approach. Fourth, AUTORT is agnostic to deadlock, while in this paper we proposed a novel dependency analysis to identify the pairs of methods that may lead to deadlock if concurrently executed.

Recently, we presented CONCRASH [46] to generate concurrent tests for reproducing concurrency failures from crash stack traces. CONCRASH performs search space pruning strategies to steer the test generation towards concurrency tests that reproduce the given crash stack trace. Two pruning strategies of CONCRASH, `PS-Interleave` and `PS-Interfere` share similarities with the parallel and conflict dependencies analysis of DEPCON$^+$, respectively. However, to apply the pruning strategies, CONCRASH requires dynamic information obtained by generating and executing concurrent tests. Conversely, DEPCON$^+$ performs conflict and parallel dependency analyses prior to test generation, and thus it avoids the cost of generating and executing those concurrent tests that the two pruning strategies will prune away. Nevertheless, the effectiveness of CONCRASH is expected to improve if the static analysis of DEPCON$^+$ is added in the pipeline of CONCRASH.

Prepared using stvrauth.cls DOI: 10.1002/stvr

8.4. Static Analyses for OO Concurrent Programs

DEPCON$^+$ builds on top of traditional static analyses for concurrent object-oriented programs, such as *object purity* [63, 64], *alias* [81, 82], *may-happen-in-parallel* [53, 54, 55, 56], and *conflict* [51, 83] analyses. Researchers proposed such analyses to resolve problems that are more general than the one considered in this paper. DEPCON$^+$ adapts and combines them in a novel way to resolve the specific problem of reducing the search space during concurrent test generation for thread-safe classes. For instance, *may-happen-in-parallel* analysis is defined at the granularity of statements [55, 56], while we define the parallel analysis of DEPCON$^+$ at the granularity of methods. As another example, differently from the traditional purity analysis (also called side-effect analysis) [63, 64], DEPCON$^+$ purity analysis focuses solely on the side-effects of those methods that could have shared-memory locations as method parameters. Moreover, one needs to define static analysis with a specific trade-off between analysis efficiency and precision of its result [84, 85]. We reasoned about such trade-offs in the context of the problem we wanted to address, favoring efficiency over precision. Classic static analysis techniques often favor precision over efficiency, this can hardly cope with the problem addressed in this paper.

The challenge of how to effectively detect concurrency faults via test input generation applies to only dynamic analysis techniques, but not static ones [20, 84, 86], which do not take test inputs into account. However, static techniques encounter their own challenges in terms of scalability and precision [66], which is partly why most work on concurrency testing has focused on dynamic techniques [3, 66]. Static analysis techniques for concurrent programs are known to lead to many spurious faults due to the imprecision and over-approximation of static analysis [3, 9]. DEPCON$^+$ is a dynamic technique that leverages static analysis to make test generation more effective without dropping the important guarantee of dynamic analysis: reporting only real thread-safety violations.

9. CONCLUSION

In this paper, we presented DEPCON$^+$, a coverage-driven concurrent test generation technique that relies on statically computed dependencies among methods to effectively expose thread-safety violations in thread-safe classes. Our insight is that static and dynamic analyses, with their dual strengths and weaknesses [85, 87, 88], can work well in synergy to generate concurrent tests. Dynamic analysis guarantees to report true thread-safety violations [3], while static analysis mitigates the inherent incompleteness of dynamic analysis by reducing the search space of concurrent tests.

Such search space reduction allows the test generation to focus on relevant areas of the search space, avoiding wasting time and resources in exploring irrelevant tests. DEPCON$^+$ extends our previous work DEPCON [17] that was limited to thread-safety violations of exception type, by proposing a combination of novel techniques that can effectively and efficiently address also thread-safety violations of ABBA resource deadlock type. In this paper, we show the complementarity of DEPCON$^+$ and DEPCON in exposing thread-safety violations of both deadlock and non-deadlock type. More specifically, the parallel and conflict dependencies of DEPCON do not capture the deadlock condition, and thus pruning the search space with parallel and conflict dependencies likely misses deadlock faults. The novel double-lock dependency of DEPCON$^+$ presented in this paper complements the parallel and conflict dependencies by capturing the deadlock condition. The paper presents the results of an experimental evaluation that show that double-lock dependencies are effective in identifying the deadlock condition, as only few method pairs manifest such dependencies. We also present a new way to generate concurrent tests that effectively exercise the interleavings captured by the double-lock dependency.

There are several opportunities for future work that can further improve DEPCON$^+$ effectiveness and applicability. We highlight the most promising ones.

DEPCON$^+$ targets ABBA resource deadlocks only. Targeting communication deadlocks (such as, wait-notify faults) is an important future work. Indeed, as we discussed in our empirical study [10], none of the previous test generators for thread-safe classes handles such deadlock faults.

Prepared using stvrauth.cls DOI: 10.1002/stvr

DEPCON$^+$ targets the JAVA synchronization mechanisms of synchronized methods and blocks, which use monitor instructions. It does not handle other mechanisms, such as `CompareAndSwap`, often used by the JAVA classes in the package `java.util.concurrent.locks`. To improve the applicability of DEPCON$^+$, we plan to modify DEPCON$^+$ to handle more synchronization mechanisms.

ACKNOWLEDGMENTS

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

REFERENCES

1. M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing heisenbugs in concurrent programs," in *Proceedings of the Symposium on Operating Systems Design and Implementation*, ser. OSDI '08. USENIX Association, 2008, pp. 267–280.
2. S. Okur and D. Dig, "How do developers use parallel libraries?" in *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '12. ACM, 2012, pp. 54:1–54:11.
3. M. Pradel and T. R. Gross, "Fully automatic and precise detection of thread safety violations," in *Proceedings of the Conference on Programming Language Design and Implementation*, ser. PLDI '12. ACM, 2012, pp. 521–530.
4. M. Pradel, M. Huggler, and T. R. Gross, "Performance regression testing of concurrent classes," in *Proceedings of the International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. ACM, 2014, pp. 13–25.
5. B. Goetz and T. Peierls, *Java Concurrency in Practice*. Pearson Education, 2006.
6. S. Park, S. Lu, and Y. Zhou, "Ctrigger: Exposing atomicity violation bugs from their hiding places," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '09. ACM, 2009, pp. 25–36.
7. F. A. Bianchi, A. Margara, and M. Pezzè, "A survey of recent trends in testing concurrent software systems," *IEEE Transactions on Software Engineering*, vol. 44, no. 8, pp. 747–783, 2018.
8. A. Choudhary, S. Lu, and M. Pradel, "Efficient detection of thread safety violations via coverage-guided generation of concurrent tests," in *Proceedings of the International Conference on Software Engineering*, ser. ICSE '17. IEEE Computer Society, 2017, pp. 266–277.
9. V. Terragni and S.-C. Cheung, "Coverage-driven test code generation for concurrent classes," in *Proceedings of the International Conference on Software Engineering*, ser. ICSE '16. ACM, 2016, pp. 1121–1132.
10. V. Terragni and M. Pezzè, "Effectiveness and challenges in generating concurrent tests for thread-safe classes," in *Proceedings of the International Conference on Automated Software Engineering*, ser. ASE '18. ACM, 2018.
11. S. Steenbuck and G. Fraser, "Generating unit tests for concurrent classes," in *Proceedings of the International Conference on Software Testing, Verification and Validation*, ser. ICST '13. IEEE Computer Society, 2013, pp. 144–153.
12. 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 *Proceedings of the International Conference on Software Engineering*, ser. ICSE '12. IEEE Computer Society, 2012, pp. 727–737.
13. V. Jagannath, M. Gligoric, D. Jin, Q. Luo, G. Rosu, and D. Marinov, "Improved multithreaded unit testing," in *Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE '11, 2011, pp. 223–233.
14. S. Lu, W. Jiang, and Y. Zhou, "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*, ser. ESEC-FSE companion '07. ACM, 2007, pp. 533–536.
15. S. Hong, M. Staats, J. Ahn, M. Kim, and G. Rothermel, "The impact of concurrent coverage metrics on testing effectiveness," in *Proceedings of the International Conference on Software Testing, Verification and Validation*, ser. ICST '13. IEEE Computer Society, 2013, pp. 232–241.
16. T. Yu, W. Srisa-an, and G. Rothermel, "An empirical comparison of the fault-detection capabilities of internal oracles," in *Proceedings of the International Symposium on Software Reliability Engineering*, ser. ISSRE '13. IEEE Computer Society, 2013, pp. 11–20.
17. V. Terragni, M. Pezzè, and F. A. Bianchi, "Coverage-driven test generation for thread-safe classes via parallel and conflict dependencies," in *Proceedings of the International Conference on Software Testing, Verification and Validation*, ser. ICST '19. IEEE Computer Society, 2019, pp. 264–275.
18. Z. Lin, D. Marinov, H. Zhong, Y. Chen, and J. Zhao, "Jacontebe: A benchmark suite of real-world java concurrency bugs (t)," in *Proceedings of the International Conference on Automated Software Engineering*, ser. ASE '15. IEEE Computer Society, 2015, pp. 178–189.
19. P. Joshi, M. Naik, K. Sen, and D. Gay, "An effective dynamic analysis for detecting generalized deadlocks," in *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '10. ACM, 2010, pp. 327–336.

20. A. Williams, W. Thies, and M. D. Ernst, "Static deadlock detection for java libraries," in *Proceedings of the European Conference on Object-Oriented Programming*, ser. ECOOP '05. Springer, 2005, pp. 602–629.
21. M. Gligoric, L. Zhang, C. Pereira, and G. Pokam, "Selective mutation testing for concurrent code," in *Proceedings of the International Symposium on Software Testing and Analysis*, ser. ISSTA '13. ACM, 2013, pp. 224–234.
22. D. Dig, "A refactoring approach to parallelism," *IEEE Software*, vol. 28, no. 1, pp. 17–22, 2011.
23. C. Flanagan and S. N. Freund, "Fasttrack: Efficient and precise dynamic race detection," in *Proceedings of the Conference on Programming Language Design and Implementation*, ser. PLDI '09. ACM, 2009, pp. 121–133.
24. ——, "Atomizer: A dynamic atomicity checker for multithreaded programs," in *Proceedings of the Symposium on Principles of Programming Languages*, ser. POPL '04. ACM, 2004, pp. 256–267.
25. M. Vaziri, F. Tip, and J. Dolby, "Associating synchronization constraints with data in an object-oriented language," in *Proceedings of the Symposium on Principles of Programming Languages*, ser. POPL '06. ACM, 2006, pp. 334–345.
26. M. Singhal, "Deadlock detection in distributed systems," *IEEE Computer*, vol. 22, no. 11, pp. 37–48, 1989.
27. M. Samak, M. K. Ramanathan, and S. Jagannathan, "Synthesizing racy tests," in *Proceedings of the Conference on Programming Language Design and Implementation*, ser. PLDI '15. ACM, 2015, pp. 175–185.
28. S. Lu, S. Park, E. Seo, and Y. Zhou, "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*, ser. ASPLOS '08. ACM, 2008, pp. 329–339.
29. M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 3, pp. 463–492, 1990.
30. S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan, "Line-up: A complete and automatic linearizability checker," in *Proceedings of the Conference on Programming Language Design and Implementation*, ser. PLDI '10. ACM, 2010, pp. 330–340.
31. M. Pezzè and C. Zhang, "Automated test oracles: A survey," in *Advances in Computers*. Elsevier, 2015, vol. 95, pp. 1–48.
32. "JDK Bug 4728096," https://bugs.java.com/bugdatabase/view_bug.do?bug_id=4728096.
33. "JDK Bug 6582568 ," https://bugs.openjdk.java.net/browse/JDK-6582568.
34. M. Singhal, "Deadlock detection in distributed systems," *Computer*, vol. 22, no. 11, pp. 37–48, 1989.
35. C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proceedings of the International Conference on Software Engineering*, ser. ICSE '07. ACM, 2007, pp. 75–84.
36. G. Fraser and A. Arcuri, "Evosuite: On the challenges of test case generation in the real world," in *Proceedings of the International Conference on Software Testing, Verification and Validation*, ser. ICST '13. IEEE Computer Society, 2013, pp. 362–369.
37. V. Terragni, S.-C. Cheung, and C. Zhang, "Recontest: Effective regression testing of concurrent programs," in *Proceedings of the International Conference on Software Engineering*, ser. ICSE '15. IEEE Computer Society, 2015, pp. 246–256.
38. Z. Lai, S. C. Cheung, and W. K. Chan, "Detecting atomic-set serializability violations in multithreaded programs through active randomized testing," in *Proceedings of the International Conference on Software Engineering*, ser. ICSE '10. ACM, 2010, pp. 235–244.
39. J. Huang and C. Zhang, "Persuasive prediction of concurrency access anomalies," in *Proceedings of the International Symposium on Software Testing and Analysis*, ser. ISSTA '11. ACM, 2011, pp. 144–154.
40. S. Hong, J. Ahn, S. Park, M. Kim, and M. J. Harrold, "Testing concurrent programs to achieve high synchronization coverage," in *Proceedings of the International Symposium on Software Testing and Analysis*, ser. ISSTA '12. ACM, 2012, pp. 210–220.
41. X. Zhang, Z. Yang, Q. Zheng, P. Liu, J. Chang, Y. Hao, and T. Liu, "Automated testing of definition-use data flow for multithreaded programs," in *Proceedings of the International Conference on Software Testing, Verification and Validation*, ser. ICST '17, 2017, pp. 172–183.
42. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Transactions on Computer Systems*, vol. 15, no. 4, pp. 391–411, 1997.
43. S. Hong and M. Kim, "A survey of race bug detection techniques for multithreaded programmes," *Software Testing, Verification and Reliability*, vol. 25, no. 3, pp. 191–217, 2015.
44. J. Huang, P. O. Meredith, and G. Rosu, "Maximal sound predictive race detection with control flow abstraction," in *Proceedings of the Conference on Programming Language Design and Implementation*, ser. PLDI '14. ACM, 2014, pp. 337–348.
45. C. Flanagan, S. N. Freund, and J. Yi, "Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs," in *Proceedings of the Conference on Programming Language Design and Implementation*, ser. PLDI '08. ACM, 2008, pp. 293–303.
46. F. A. Bianchi, M. Pezzè, and V. Terragni, "Reproducing concurrency failures from crash stacks," in *Proceedings of the Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE '17. ACM, 2017, pp. 705–716.
47. T. Yu, W. Srisa-an, and G. Rothermel, "Simrt: An automated framework to support regression testing for data races," in *Proceedings of the International Conference on Software Engineering*, ser. ICSE 2014. ACM, 2014, pp. 48–59.
48. Y. Yu, T. Rodeheffer, and W. Chen, "Racetrack: Efficient detection of data race conditions via adaptive tracking," in *Proceedings of the Symposium on Operating Systems Principles*, ser. SOSP '05. ACM, 2005, pp. 221–234.
49. Y. Cai and L. Cao, "Effective and precise dynamic detection of hidden races for java programs," in *Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE '15. ACM, 2015, pp. 450–461.
50. W. Zhang, C. Sun, and S. Lu, "Conmem: Detecting severe concurrency bugs through an effect-oriented approach," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV. ACM, 2010, pp. 179–192.
51. C. von Praun and T. R. Gross, "Static conflict analysis for multi-threaded object-oriented programs," in *Proceedings of the Conference on Programming Language Design and Implementation*, ser. PLDI '03. ACM, 2003, pp. 115–128.

DOI: 10.1002/stvr

52. S. Park, R. W. Vuduc, and M. J. Harrold, "Falcon: Fault localization in concurrent programs," in *Proceedings of the International Conference on Software Engineering*, ser. ICSE '10. ACM, 2010, pp. 245–254.
53. E. Albert, A. E. Flores-Montoya, and S. Genaim, "Analysis of may-happen-in-parallel in concurrent objects," in *Formal Techniques for Distributed Systems*. Springer, 2012, pp. 35–51.
54. E. Albert, P. Arenas, A. Flores-Montoya, S. Genaim, M. Gomez-Zamalloa, E. Martin-Martin, G. Puebla, and G. Roman-Diez, "Saco: static analyzer for concurrent objects," in *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems*. Springer, 2014, pp. 562–567.
55. G. Naumovich, G. S. Avrunin, and L. A. Clarke, "An efficient algorithm for computing mhp information for concurrent java programs," in *Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESECFSE '99. Springer, 1999, pp. 338–354.
56. G. Naumovich and G. S. Avrunin, "A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel," in *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '98, 1998, pp. 24–34.
57. P. Joshi, C.-S. Park, K. Sen, and M. Naik, "A randomized dynamic program analysis technique for detecting real deadlocks," in *Proceedings of the Conference on Programming Language Design and Implementation*, ser. PLDI '09. ACM, 2009, pp. 110–120.
58. F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis*. Springer Science & Business Media, 2004.
59. J. Gosling, B. Joy, and G. Steele, *The Java language specification*. Addison-Wesley Professional, 2000.
60. Y. Lin and D. Dig, "Check-then-act misuse of java concurrent collections," in *Proceedings of the International Conference on Software Testing, Verification and Validation*, ser. ICST '13, 2013, pp. 164–173.
61. J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff, "Escape Analysis for Java," in *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '99. ACM, 1999, pp. 1–19.
62. J. Huang, "Scalable Thread Sharing Analysis," in *Proceedings of the International Conference on Software Engineering*, ser. ICSE '16. ACM, 2016, pp. 1097–1108.
63. D. Helm, F. Kubler, M. Eichberg, M. Reif, and M. Mezini, "A unified lattice model and framework for purity analyses," in *Proceedings of the International Conference on Automated Software Engineering*, ser. ASE '18, 2018, pp. 340–350.
64. W. Huang and A. Milanova, "Reiminfer: method purity inference for java," in *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '12. ACM, 2012, pp. 38:1–38:4.
65. I. Dillig, T. Dillig, and A. Aiken, "Sound, complete and scalable path-sensitive analysis," in *Proceedings of the Conference on Programming Language Design and Implementation*, ser. PLDI 08, 2008, pp. 270–280.
66. D. Deng, W. Zhang, and S. Lu, "Efficient concurrency-bug detection across inputs," in *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '13. ACM, 2013, pp. 785–802.
67. G. Ramalingam, "The undecidability of aliasing," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 5, pp. 1467–1471, 1994.
68. "Asm 5.0 a java bytecode engineering library," https://asm.ow2.io.
69. A. Farzan, P. Madhusudan, N. Razavi, and F. Sorrentino, "Predicting null-pointer dereferences in concurrent programs," in *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '12. ACM, 2012, pp. 1–11.
70. Y. Cai and W. K. Chan, "Magicfuzzer: Scalable deadlock detection for large-scale applications," in *Proceedings of the International Conference on Software Engineering*, ser. ICSE '12. IEEE Computer Society, 2012, pp. 606–616.
71. A. Farzan, A. Holzer, N. Razavi, and H. Veith, "Con2colic testing," in *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '13. ACM, 2013, pp. 37–47.
72. R. Majumdar and K. Sen, "Hybrid concolic testing," in *Proceedings of the International Conference on Software Engineering*, ser. ICSE '07. IEEE Computer Society, 2007, pp. 416–426.
73. N. Razavi, F. Ivančić, V. Kahlon, and A. Gupta, "Concurrent test generation using concolic multi-trace analysis," in *Asian Symposium on Programming Languages and Systems*, ser. ASPLS '10. Springer, 2012, pp. 239–255.
74. K. Sen and G. Agha, "CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools," in *Proceedings of the International Conference on Computer Aided Verification*, ser. CAV '06. Springer, 2006, pp. 419–423.
75. G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2013.
76. P. Braione, G. Denaro, A. Mattavelli, and M. Pezzè, "Combining symbolic execution and search-based testing for programs with complex heap inputs," in *Proceedings of the International Symposium on Software Testing and Analysis*, ser. ISSTA '17. ACM, 2017, pp. 90–101.
77. M. Samak and M. K. Ramanathan, "Omen+: A precise dynamic deadlock detector for multithreaded java libraries," in *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 735–738.
78. ——, "Synthesizing tests for detecting atomicity violations," in *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '15. ACM, 2015.
79. M. Samak, O. Tripp, and M. K. Ramanathan, "Directed synthesis of failing concurrent executions," in *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '16. ACM, 2016, pp. 430–446.
80. J. Schimmel, K. Molitorisz, A. Jannesari, and W. F. Tichy, "Automatic generation of parallel unit tests," in *Proceedings of the International Workshop on Automation of Software Test*, ser. AST '10. IEEE Computer Society, 2013, pp. 40–46.

DOI: 10.1002/stvr

81. A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to analysis for java," *ACM Transactions on Software Engineering and Methodology*, vol. 14, no. 1, pp. 1–41, 2005.
82. M. Hind, M. Burke, P. Carini, and J.-D. Choi, "Interprocedural pointer alias analysis," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 21, no. 4, pp. 848–894, 1999.
83. C. von Praun and T. R. Gross, "Object race detection," in *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '01. ACM, 2001, pp. 70–82.
84. D. R. Engler and K. Ashcraft, "Racerx: effective, static detection of race conditions and deadlocks," in *Proceedings of the Symposium on Operating Systems Principles*, ser. SOSP '03. ACM, 2003, pp. 237–252.
85. M. D. Ernst, "Static and Dynamic Analysis: Synergy and Duality," in *WODA 2003: ICSE Workshop on Dynamic Analysis*, ser. WODA '03, 2003, pp. 24–27.
86. C. von Praun and T. R. Gross, "Static detection of atomicity violations in object-oriented programs," *Journal of Object Technology*, vol. 3, no. 6, pp. 103–122, 2004.
87. S. Zhang, D. Saff, Y. Bu, and M. D. Ernst, "Combined Static and Dynamic Automated Test Generation," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2011, pp. 353–363.
88. K. Vorobyov and P. Krishnan, "Combining Static Analysis and Constraint Solving for Automatic Test Case Generation," in *Proceedings of the International Conference on Software Testing, Verification and Validation*, 2012, pp. 915–920.

DOI: 10.1002/stvr