Coverage-Driven Test Generation for Thread-Safe Classes via Parallel and Conflict Dependencies

Valerio Terragni[1], Mauro Pezzè[1,2] and Francesco A. Bianchi[1]
[1] USI Università della Svizzera italiana, Switzerland
[2] Università degli Studi di Milano-Bicocca, Italy
{valerio.terragni, mauro.pezze, francesco.bianchi}@usi.ch

***Abstract*—Thread-safe classes are common in concurrent object-oriented programs. Testing such classes is important to ensure the reliability of the concurrent programs that rely on them. Recently, researchers have proposed the automated generation of concurrent (multi-threaded) tests to expose concurrency faults in thread-safe classes (thread-safety violations). However, 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, an approach to effectively reduce the search space of concurrent tests by means of both parallel and conflict dependency analyses. DEPCON is based on the intuition that only methods that can both interleave (parallel dependent) and access the same shared memory locations (conflict dependent) can lead to thread-safety violations when concurrently executed. DEPCON implements an efficient static analysis to compute the parallel and conflict dependencies among the methods of a class and uses the computed dependencies to steer the generation of tests towards concurrent tests that exhibit the computed dependencies. We evaluated DEPCON by experimenting with a prototype implementation for Java programs on a set of thread-safe classes with known concurrency faults. The experimental results show that DEPCON is more effective in exposing concurrency faults than state-of-the-art techniques.**

***Keywords*-test generation; concurrency faults; thread-safety; thread interleavings; shared-memory; static analysis; dynamic analysis**

I. INTRODUCTION

Concurrent programming is pervasive due to the ubiquity of multi-core processors [21]. Developing correct and efficient concurrent programs is difficult because of the complexity of thread synchronization [36, 40, 44]. Developers of concurrent object-oriented programs often delegate such complexity by relying on existing thread-safe classes [40, 43, 44], which already encapsulate synchronization mechanisms that prevent incorrect accesses to the class from multiple threads [17].

Ensuring the correctness of thread-safe classes is important to avoid serious issues (thread-safety violations) in the concurrent programs that rely on them [4, 36, 42]. Thread-safety violations are notoriously difficult to expose at testing time since they often manifest non-deterministically under specific thread interleavings [9, 42, 43, 54]. An effective approach to ensure the correctness of thread-safe classes is *automated test generation* [56]. The basic idea is that a test generator creates concurrent tests and an interleaving explorer checks if the generated tests trigger fault-revealing thread interleavings [9, 39, 43, 53, 54]. A concurrent test for a thread-safe class consists of multiple concurrently executing threads that invoke the public methods that exercise shared instances of the class under test [30, 39, 43]. Figure 2 show examples of concurrent tests for the class in Figure 1.

There exist a myriad of possible concurrent tests for any non-trivial thread-safe class. Often only few concurrent tests that exercise particular combinations of method call invocations can expose thread-safety violations [9, 54, 56]. As such, concurrent test generators need to explore thousands and even millions of concurrent tests before finding one that can trigger a thread-safety violation [9, 43]. For instance, the random-based generator CONTEGE [43] may need to generate 17 million concurrent tests for a given thread-safe class before exposing a thread-safety violation [43]. Generating many tests is an issue for concurrent test generators due to the high computational cost of exploring the interleavings space of each generated test [4, 25, 39, 59, 61]. Within a reasonable time-budget we can only afford to explore the interleaving spaces of few tests due to the large number of possible interleavings [9, 34, 54, 55]. As a result, concurrent test generators often fail to generate failure-inducing concurrent tests within the available time-budget [56].

Recently, researchers have proposed coverage-driven test generators [9, 53, 54] to effectively explore the search space of possible concurrent tests. These techniques steer test generation towards those concurrent tests that lead to new program behaviors (thread interleavings), thus avoiding the high cost of exploring the interleaving spaces of redundant tests [56].

In this paper, we propose an orthogonal approach based on parallel and conflict dependencies that combined with the coverage-driven test generation approach further improves the effectiveness of concurrent test generation. Our intuition is that many concurrent tests that increase interleaving coverage [9, 53, 54] are irrelevant for exposing thread-safety violations. Concurrent tests that increase coverage can trigger a thread-safety violation only if the method invocations that are executed concurrently in the test are (i) *conflict dependent*, that is, they have to read and write at least one common shared-memory location, and (ii) *parallel dependent*, that is, their instructions can interleave, meaning that the synchronization mechanism does not prevent the parallel execution of the concurrent method invocations. Generating and exploring the interleaving spaces of concurrent tests that do not satisfy both requirements waste precious time-budget without increasing the likelihood of exposing thread-safety violations.

Based on this intuition, we present **DEPCON**, *DEPendency-driven CONcurrency Testing*, a technique built on top of the state-of-the-art coverage-driven concurrent test generator COVCON [9]. DEPCON statically analyses the class under test to compute a summary of the public methods of the class (preprocessing phase) that it uses to dynamically generate tests. The summary of each method includes information about both the shared-memory accesses and the synchronization operations of the corresponding method. DEPCON uses the summaries to generate and explore the interleaving spaces of only the concurrent tests that exercise conflict and parallel dependencies.

To make DEPCON useful the preprocessing phase must be: (i) efficient, that is, it has a low computational cost, (ii) complete, that is, it identifies all parallel and conflict dependencies to guarantee that DEPCON does not prevent the generation of failure-inducing concurrent tests, and (iii) mostly precise, that is, it identifies as few as possible spurious parallel and conflict dependencies. Defining an efficient, complete and precise analysis is challenging due to the intrinsic imprecision and high computational cost of static analysis [46]. We address these challenges with a novel combination and adaptation of classic static analysis techniques for concurrent object-oriented programs [1, 19, 46].

We implemented DEPCON in a prototype for the Java language and evaluated it on 15 thread-safe classes with known concurrency faults. Our results show that DEPCON dramatically reduces the search space of concurrent tests to be generated 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 the state-of-the-art technique COVCON [9]. The results also show that the preprocessing phase is efficient (it requires less than two seconds on average), complete (it does not prevent the detection of any thread-safety violation) and mostly precise. In summary, this paper makes the following contributions:

- The first technique to generate concurrent tests for thread-safe classes driven by parallel and conflict dependencies.
- The insight that an efficient static computation of parallel and conflict dependencies can effectively steer dynamic test generation towards thread-safety violations.
- An implementation and an experimental evaluation of DEPCON to show the effectiveness of the proposed technique.

II. PRELIMINARIES: CONCURRENT TEST GENERATION FOR THREAD-SAFE CLASSES

This section presents the preliminaries of concurrent test generation for thread-safe classes.

An object-oriented concurrent program is composed of a set of classes, each defining a set of methods and fields that can be invoked and accessed concurrently by multiple threads, respectively. A class is *thread-safe* if it encapsulates all necessary synchronization that prevents incorrect accesses to the class from multiple threads [17]. Thread-safety guarantees that multiple threads can correctly access the same instance of the class without additional synchronization other than the one implemented in the class [17, 43].

Concurrency faults in thread-safe classes lead to *thread-safety violations*, that is, deviations from the expected behavior of a thread-safe class when concurrently accessed. Such violations often manifest non-deterministically depending on the order of shared-memory accesses across threads. The order of accesses to shared-memory locations is fixed within one thread, but can vary across threads [34]. Such non-deterministic orders of shared-memory accesses are called *interleavings* [34]. Concurrent executions are characterized by many different interleavings, and only some –usually few– of them expose thread-safety violations [42]. An effective approach to effectively expose thread-safety violations is concurrent test generation [56].

A **concurrent test (ct)** of a thread-safe class is composed of three method call sequences $\langle \text{prefix}, \text{suffix}_1, \text{suffix}_2 \rangle$ that exercise from multiple threads the public interface of a class [43]. A *prefix* is a call sequence that is executed in a single thread, and that creates instances of the class under test to be accessed concurrently from multiple threads. A prefix could also 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 created by the prefix and can use them as input parameters for suffix calls. Figure 2 shows examples of concurrent tests for the class in Figure 1. In this paper, we consider tests with exactly two concurrent suffixes. This is in line with state-of-the-art concurrent test generators [56].

In this work, we rely on the thread-safety oracle proposed by Pradel and Gross [43] to infer if an explored interleaving of a concurrent test exposes a thread-safety violation.

Definition 1. ***Thread-safety oracle***. *A concurrent test violates thread-safety if and only if one of its thread interleavings manifests a runtime exception*[1] *that is not manifested in any method-level linearization of the test.*

The method-level linearizations of a concurrent test ct is the set of all permutations of the method calls in ct that maintain the order of calls in each thread [6, 20]. This thread-safefy oracle has two main advantages. First, it is general enough to include a wide range of concurrency faults types, for example, data races [14], atomicity violations [13] and atomic-set serializability violations [57]. Second, it guarantees to report an oracle violation only if the class indeed suffers from a concurrency issue, since it considers as expected behaviors the (sequential) linearizations of the concurrent test [6, 20, 43]. In fact, the manifestation of a runtime exception could be an expected behavior during a linearized execution [6, 9, 20, 43]. For example, calling `firstElement()` on an empty Java Vector triggers a runtime exception (`NoSuchElementException`), which is an expected behavior of the method. The thread-safety oracle reports a violation only if the manifested runtime exceptions are not expected behaviors.

[1] Pradel and Gross consider also deadlocks as violations, we do not. We are currently working on extending DEPCON to handle deadlocks.

```
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 }
```

Fig. 1. Buggy version of the JDK class BufferedInputStream, bug id 4728096. Accesses to the instance fields of the class are in bold/blue.

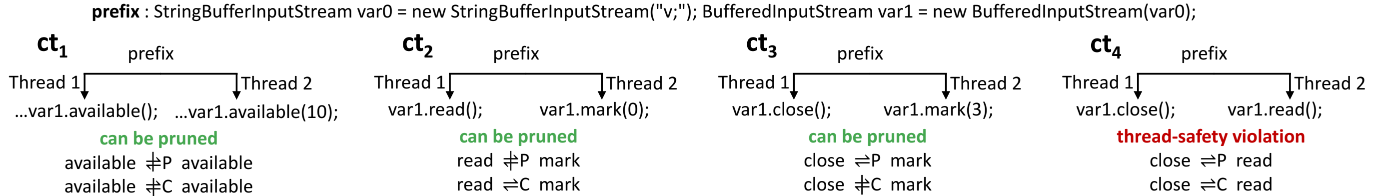

Fig. 2. Examples of concurrent tests for the class in Figure 1. Among the four concurrent tests, only ct_4 can manifest the thread-safety violation.

III. Motivation

Figure 1 shows a portion of the `BufferedInpuStream` class of *JDK 1.4*. The class contains a known concurrency fault [31] that leads to a thread-safety violation when methods `close()` and `read()` are executed concurrently.

Figure 2 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 (see the top of Figure 2) but have different concurrent suffixes. Among these tests only ct_4 exposes the thread-safety violation: when executed with some specific thread interleavings, ct_4 triggers a `NullPointerException` at line 20, which is not thrown by any linearizations of the calls in ct_4. 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`. There are two possible linearizations [6, 20] of ct_4: ⟨*prefix*, `var1.read()`, `var1.close()`⟩ and ⟨*prefix*, `var1.close()`, `var1.read()`⟩. 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 ct_4 exposes a thread-safety violation [20].

As recently discussed in our recent empirical study, a major challenge of automatically generating tests that expose thread-safety violations is the huge search space of possible concurrent tests [56]. 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 maximum call sequence length, and p the maximum number of parameters values for each method call in a test, given a class with w public methods, there are at most $(p \cdot w)^{len}$ method call sequences [16, 41]. Since a concurrent test has three method call sequences (a sequential prefix and two concurrent suffixes), there are $(p \cdot w)^{3 \cdot len}$ possible concurrent tests. The small class in Figure 1 has 10 public methods, including the ones of its superclasses. With relatively small values, $len = 10$ and $p = 5$, there are $\sim 10^{50}$ possible concurrent tests. Since only few concurrent tests manifest thread-safety violations, it is challenging to find them in such a huge search space [43, 56]. There are many concurrent tests for the class in Figure 1 that do not trigger a thread-safety violation (for example, ct_1, ct_2 and ct_3 in Figure 2), and a concurrent test generator could waste all the available time-budget generating and exploring the interleaving spaces of such concurrent tests, thus missing the failure-inducing one [9].

IV. DepCon

We address this challenge by defining DepCon, a technique that leverages static dependency analysis to reduce the search space of concurrent tests without preventing the generation of failure-inducing ones. The core intuition behind DepCon is that only methods that meet the following two requirements can lead to thread-safety violations when concurrently executed:

- their executions can interleave (**parallel dependent** $\rightleftharpoons$**P**);
- they can access (with a write and read) at least one shared-memory location in common[2] (**conflict dependent** $\rightleftharpoons$**C**).

For instance, let us consider the concurrent tests ct_1, ct_2 and ct_3 in Figure 2. These tests do not meet the two requirements and thus DepCon can safely avoid 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 on 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.

[2] In fact, all the problematic access patterns of data races [14], atomicity violations [13] and atomic-set serializability violations [57] have two concurrent threads that access memory location(s) in common.

However, DEPCON would generate the failure-inducing test ct_4 as the methods in its 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 field `buf`.

DEPCON exposes thread-safety violations for a Class Under Test (CUT) as follows: (i) it computes the *method summary* of each public method of the CUT, the summary encapsulates the possible access and synchronization operations of the method, (ii) it computes the parallel and conflict 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 could thus reveal thread-safety violations, (iv) it explores the interleaving space of each generated test with a given interleaving explorer [9, 43].

The remainder of this section is organized as follows. Section IV-A defines the concept of method summary, Section IV-B defines parallel and conflict dependencies relying on method summaries, Section IV-C describes how DEPCON computes the method summaries, Section IV-D presents how DEPCON generates tests leveraging the parallel and conflict dependencies.

A. Method Summaries

The *method summary* of a method *m*, *MS(m)*, contains (i) the *access summary AS(m)*, that is, the set of all possible instructions accessing shared-memory locations that can be triggered by *m* and by all of its callees, and (ii) the *lock summary LS(m)*, that is, the set of locks that protect such instructions. Intuitively, DEPCON relies on the access and lock summaries to identify which pairs of methods have conflict and parallel dependencies, respectively.

We now formally define the access and lock summaries of a method. A method *m* is composed of an ordered sequence of instructions: $inst_i$ denotes the $i\text{-}th$ instruction in *m*. Each instruction has a type: $type(inst_i)$ denotes the type of the instruction $inst_i$. There are four types of instructions needed to define 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*;

A method can invoke other methods: *callees(m)* denotes all the methods that can be directly or indirectly invoked by *m*.

Definition 2. *The* ***access summary AS(m)*** *of a method m, is the set instructions accessing shared-memory locations that can be triggered by m and by all of its callees.*

$AS(m) = \{inst_i \in \{m \cup callees(m)\} \wedge (type(inst_i) = R(x) \vee type(inst_i) = W(x))$, *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* is flow-insensitive as 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 3. *The* ***lock summary LS(m)*** *of a method m is the set of locks that always protect every shared-memory accesses that can be triggered by an invocation of m:*

$LS(m) = \{l : (\exists\ ints_i \in m : type(inst_i) = ACQ(l) \wedge i < k) \wedge (\nexists\ inst_j \in m : type(inst_j) = REL(l) \wedge j < w)$, *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 the method `mark` in Figure 1 is $\{$*this*$\}$, where *this* is the *current-object* of the class. *LS(m)* considers only 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 [23, 28, 32, 65]. Our notion of lock summary slightly differs from the classic notion of *lockset* [51], largely adopted by dynamic and static concurrency bug detectors [13–15, 24, 27, 28, 32]. Classic lockset is defined at the granularity of single instructions, while lock summary at the granularity of a set of instructions (methods). 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, all the shared-memory instructions in a method) can interleave with other blocks [5].

The reader should notice that we define *AS(m)* with also information about the callees of *m*, while we define *LS(m)* without looking at the callees of *m*. This is because any lock acquired with internal method calls of *m* does not protect every shared-memory accesses in *m*, and thus it can be ignored. This reduces the cost of computing lock summaries.

B. Parallel and Conflict Dependency Analyses

Definition 4. *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:*

$m_1 \rightleftharpoons P\ m_2$ *iff* $\mathrm{LS}(m_1) \cap \mathrm{LS}(m_2) = \varnothing$.

Definition 5. *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:*

$m_1 \rightleftharpoons C\ m_2$ *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* $x \equiv y$.

Parallel and conflict dependencies are relations between methods. Both 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 $m_1 \rightleftharpoons m_3$).

Our notion of conflict differs from classic *race condition* [60, 62], which include the case in which two thread writes the same memory location [7, 51]. Instead, DEPCON excludes the following case: $\exists W(x) \in AS(m_1)$ and $\exists W(y) \in AS(m_2)$ ($x \equiv y$). This likely reduces the number of conflict dependencies, yielding a further reduction of the search space. The rationale of this choice is that if m_1 and m_2 write but never read a common memory location their executions cannot interfere [5, 64], and their behavior would be the same of that of a sequential execution [46].

Theorem 1. *If a concurrent test* $ct = \langle prefix, suffix_1, suffix_2\rangle$ *violates thread-safety (with respect to Definition 1), then the two suffixes contain two methods* $m_1 \in suffix_1$ *and* $m_2 \in suffix_2$ *with parallel and conflict dependencies:*

$$\exists m_1 \in suffix_1 \wedge \exists m_2 \in suffix_2 : m_1 \rightleftharpoons P\ m_2 \wedge m_1 \rightleftharpoons C\ m_2.$$

Proof *by contradiction:* Let us assume that a concurrent test ct violates thread-safety and $\forall\langle m_1, m_2\rangle : m_1 \in \text{suffix}_1 \wedge m_2 \in \text{suffix}_2 \wedge (m_1 \not\rightleftharpoons \text{P}\ m_2 \vee m_1 \not\rightleftharpoons \text{C}\ m_2)$. 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, and thus all the concurrent executions of ct are equivalent to the executions of the linearizations of the method calls of ct [20]. Thus ct cannot violate thread-safety (see Definition 1) (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, and thus regardless if their executions interleave or not, the behaviours of all concurrent executions of ct are equivalent to the executions obtained by the linearizations of the method calls of ct [20]. As a result, ct cannot violate thread-safety (contradiction). ■

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. DEPCON relies on such finding to improve the effectiveness of coverage-driven concurrent test generation.

C. Preprocessing Phase: Computing Method Summaries

Figure 3 shows the algorithm for computing the method summaries $\mathbb{MS}$ of a given CUT (COMPUTEALLMS, lines 1 to 6). The algorithm is composed of two main functions: COMPUTEMS (lines 7 to 38) and ISPURE (lines 39 to 49). The function COMPUTEMS examines a given method *m* that is declared in the CUT and performs a fine-grained analysis that incrementally populates the method summary of *m*. COMPUTEMS invokes ISPURE every time it encounters an invocation to a method that is not declared in the CUT but is directly or indirectly invoked from CUT methods. ISPURE analyses the callee methods to infer if they can modify the value of shared-memory locations. The ISPURE analysis is coarse-grained to guarantee a low computational cost, which is not guaranteed in the common case of presence of many invocations to non-CUT methods.

COMPUTEALLMS (lines 1 to 6) returns $\mathbb{MS}$ the set of the method summaries of all the public methods declared in the CUT. The readers should notice that we are not interested in reporting the method summaries of private or protected methods as they do not constitute the public interface of the CUT, which implies that concurrent tests (client code) cannot directly invoke them. DEPCON considers the shared-memory accesses of private or protected methods when such methods are invoked by public methods (line 32).

COMPUTEMS (lines 7 to 38) returns the method summary of a given method *m*. It initializes the access summary *AS(m)*, the lock summary *LS(m)* and *tmp-lock-releases*, which is a supporting variable used for the computation of *LS(m)* (lines 8-10) to empty sets. It then scans the ordered sequence of

```
input  : CUT (class under test)
output : MS method summaries of CUT
1  function COMPUTEALLMS(CUT)
2      MS ← ∅
3      for each public method m in CUT do
4          MS(m) ← COMPUTEMS(m)
5          add MS(m) to MS
6      return MS                      // return method summaries of CUT
7  function COMPUTEMS(m)
8      AS(m) ← ∅                      // init access summary of m
9      LS(m) ← ∅                      // init lock summary of m
10     tmp-lock-releases ← ∅          // temporary lock releases
11     for each inst_i in m do
12         switch type(inst_i) do
13             case Shared-memory accesses: R(x) or W(x) do
14                 ref ← GETOUTERMOSTREF(inst_i)
15                 if ISSHAREDLOCATION(ref) then
16                     add R(ref.f) or W(ref.f) to AS(m)
17                     remove tmp-lock-releases from LS(m)
18                     tmp-lock-releases ← ∅
19             case Lock acquire: ACQ(l) do
20                 if AS(m) = ∅ then
21                     l ← GETLOCK(inst_i)
22                     add l to LS(m)
23             case Lock release: REL(l) do
24                 l ← GETLOCK(inst_i)
25                 if AS(m) = ∅ then
26                     remove l from LS(m)
27                 else
28                     add l to tmp-lock-releases
29             case Method invocation: INVOKE(callee) do
30                 callee ← GETINVOKEDMETHOD(inst_i)
31                 if callee is a method declared in the CUT then
32                     add COMPUTEMS(callee) to MS(m)
33                 else
34                     if ISPURE(calle) == false then
35                         for each parameter p of callee do
36                             if ISSHAREDLOCATION(p) then
37                                 add W(p) to AS(m)
38     return MS(m) ← ⟨AS(m), LS(m)⟩
39 function ISPURE(callee)
40     for each inst_i in callee do
41         switch type(inst_i) do
42             case W(x) do
43                 x ← GETOUTERMOSTREF(inst_i)
44                 if x is a parameter of callee then
45                     return false
46             case INVOKE(callee) do
47                 callee ← GETINVOKEDMETHOD(inst_i)
48                 return ISPURE(callee)
49     return true
```

Fig. 3. Algorithm for the preprocessing phase.

bytecode instructions $inst_i$ of *m* (line 11), and checks the type of each $inst_i$ to determine if it belongs to one of the following four categories.

Category 1) Shared-memory accesses: W(x) or R(x) (line 13). DEPCON generates concurrent tests that share among threads only CUT instances, therefore, only the fields of the *current-object* (the object referred with `this` in Java) or fields of static classes can be shared-memory locations [9, 33, 39, 43, 49, 53, 54]. As such, the instructions belonging to Category 1 are read or write accesses to static or non-static fields: `getfield`,

getstatic as well as putfield and putstatic instructions in the case of Java bytecode [18]. In this context, a costly *thread-escape analysis* [8, 26] is not necessary to identify shared-memory locations. This reduces the complexity of the analysis. We only need to infer whether the getfield and putfield instructions access the *current-object*. Since fields of an object can also be objects of their own, the algorithm needs to traverse the chain of object references until it identifies the outer-most object reference *ref*. Moreover, since a method can create and access local objects that do not reference the fields of the *current-object*, the algorithm performs backward analysis on the *JVM stack* to discover the outer-most object reference in the reference chain [18] (line 14). The Java Virtual Machine (JVM) [18] is a stack-based abstract machine, in which Java bytecode instructions pop their arguments off the stack and push their results on the stack [18]. Function ISSHAREDLOC(*ref*) (line 15) checks if *ref* is the *current-object*. This is trivially true when *x* points to the first register [18], which is popped in the *JVM stack* with the bytecode instruction aload0. Alternatively, ISSHAREDLOC(*ref*) performs an additional backward analysis on the *JVM stack* to discover 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 16), 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 17), since *LS(m)* contains only the locks released after the last shared-memory access in *m* (Definition 3).

Category 2) Lock acquisition: ACQ(l) (line 19). In Java bytecode, lock acquisitions are monitorenter instructions[3] [18]. Since *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 3), the algorithm checks if *AS(m)* is empty to infer if the lock acquire instruction is executed before the the first shared-memory access in *m*. If *AS(m)* is empty, the algorithm executes Function GETLOCK($inst_i$) that performs backward alias analysis to identify the acquired lock l, which can be an object or a class reference (line 21), and adds l to *LS(m)* (line 22).

Category 3) Lock release: REL(l) (line 23). In Java bytecode, lock releases are monitorexit instructions. The algorithm executes Function GETLOCK($inst_i$) to get the lock released by $inst_i$ (line 24). If *AS(m)* is empty, the algorithm removes *l* from *LS(m)* (line 26), otherwise it adds *l* to the *tmp-lock-releases* supporting variable. It removes the locks in *tmp-lock-releases* from *LS(m)* only if it meets another shared-memory access while scanning subsequent instructions in *m*.

Category 4) Method invocation: INVOKE(callee) (line 29). In Java bytecode, method invocations correspond to invokedynamic, invokeinterface, invokespecial, invokestatic and invokevirtual instructions [18]. When the algorithm meets one of such instructions, it gets the fully qualified name of the called method (*callee* line 30), and checks if *callee* 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)*. Otherwise, the algorithm calls Function ISPURE to check if *callee* is a *pure method* [19, 29], that is, if it does not directly or indirectly writes on object fields. If *callee* is not a pure method, the algorithm checks if any parameter of the method invocation *p* is aliased to either the *current-object* or to any one of the *current-object* fields (ISSHAREDLOCATION(p) line 36). If it is an alias, the algorithm adds a write access (*W(p)*) to *AS(m)* (line 37). The algorithm does not need to add a read access, because it must have already seen an instruction that pushes *p* in the *JVM stack*.

ISPURE (lines 39 to 49) scans the instructions of *callee*. If it encounters an instruction that writes a memory location, it gets *ref* the outer-most reference in the chain of object references (GETOUTERMOSTREF line 43). if *ref* aliases a method parameter (including the object receiver), the method is considered inpure (ISPURE returns false). The function recursively calls ISPURE on the called method when it encounters an instruction of type INVOKE.

For the sake of readability, we omitted few details in the algorithm in Figure 3: (i) the algorithm caches the computed method summaries and purity results to save computational time; if it meets multiple invocations of the same method, at line 30 the algorithm simply returns the precomputed result. (ii) the algorithm avoids endless recursions of the recursive functions COMPUTEMS and ISPURE caused by methods that directly or indirectly invoke themselves (such as, with transitive calls) by maintaining a *call stack* of the methods that are being analyzed, and breaking the endless recursion if it encounters an invocation of a method that is already in the *call stack*.

Running example of COMPUTEMS. We exemplify the algorithm on method available in Figure 1, method *m* in this example. The first relevant instruction in *m* is a monitorenter triggered by the keyword synchronized in the method declaration. GETLOCK at line 21 infers that the acquired lock is the *current-object*, and thus DEPCON adds this to *LS(m)* (line 22). The second relevant instruction in *m* is INVOKE(m'), where m' refers to the method ensureOpen. Since m' is a method declared in the CUT the algorithm recursively calls COMPUTEMS(m') at line 32. At line 2 in Figure 1 the class accesses the field in of the *current-object*, and thus the algorithm adds *R(in)* in *AS(*m'*)* at line 16. Note that DEPCON encodes both locks and memory locations with their fully qualified names since fields in the same class must have unique names. COMPUTEMS(ensureOpen) returns *AS(*m'*)*, which is added to *AS(*m'*)*. The algorithm adds *R(count)*, *R(pos)* and *R(in)* to *AS(m)*. Since *AS(m)* already contains *R(in)* it keeps only one. When the algorithm encounters the monitorexit, it adds the released lock to *tmp-lock-releases* at line 28. Since after the monitorexit instruction there are no more instructions that access shared-memory locations, this is never removed from *LS(m)*. As a result, when the algorithm terminates *AS(m)* ={*R(in)*, *R(count)*, *R(pos)*} and *LS(m)* ={*this*}.

[3]The current implementation of DEPCON does not consider low level synchronization mechanisms that uses Unsafe operations like CompareAndSwap.

```
input  : CUT (class under test), time-budget B
output : thread-safety violation
1  function DEPCON(CUT, B)
2    MS ← COMPUTEALLMS(CUT)          // see Algorithm 1
3    M ← COMPUTECFP(CUT)          // all coverage targets
4    M_pc ← ∅   // init M_pc ⊆ M reduced coverage targets
5    for each ⟨m1, m2⟩ ∈ M do
6      if m1 ⇌P m2 ∧ m1 ⇌C m2 then
7        add ⟨m1, m2⟩ to M_pc
8    while B is not expired do
9      ⟨m1, m2⟩ ← GETNEXTMETHODPAIR(M_pc)
10     for i from 0 to 1 do
11       if i % 2 == 0 then
12         prefix ← GETRANDOMCONSTRUCTOR()
13       else
14         prefix ← GETSTATECHANGER(MS)
15       suffix1 ← GETSUFFIX(m1)
16       suffix2 ← GETSUFFIX(m2)
17       ct ← ASSEMBLETEST(prefix, suffix1, suffix2)
18       for from 1 to numIter do
19         exception ← EXECUTE(ct)
20         if exception ≠ ∅ ∧ exception ∉
             ALLPOSSIBLELINEARIZATIONS(ct) then
21           return thread-safety violation
```

Fig. 4. DEPCON algorithm.

D. Generating Tests via Parallel and Conflict Dependencies

Figure 4 summarizes the DEPCON test generation approach, highlighting how it leverages parallel and conflict dependencies while generating concurrent tests. DEPCON takes as an input a (presumed) thread-safe class (CUT) and a time-budget $\mathcal{B}$. DEPCON alternatively generates concurrent tests and explores their interleaving spaces. DEPCON terminates when either it detects a thread-safety violation or the time-budget expires.

DEPCON starts by computing the initial coverage targets $\mathcal{M}$ (COMPUTEALLMS line 3) relying on the coverage-driven approach of COVCON [9], a state-of-the-art coverage-driven concurrent test generation. COVCON exploits the concept of concurrent method pairs proposed by Deng et al. [10], who define concurrent method pairs as the set of pairs of methods that execute concurrently [10]. COVCON considers the set of all possible pairs of public methods of a class $\mathcal{M}$ [9] as coverage targets for a class with *m* public methods[4].

While COVCON considers all possible pairs as coverage targets, DEPCON considers only those pairs $\mathcal{M}_{pc} \subseteq \mathcal{M}$ that have parallel and conflict dependencies. DEPCON computes the method summaries with Function COMPUTEALLMS (line 2 in Figure 3). Then, DEPCON adds the pair $\langle m_1, m_2 \rangle \in \mathcal{M}$ to $\mathcal{M}_{pc}$, if $m_1 \rightleftharpoons\text{P}\ m_2 \wedge m_1 \rightleftharpoons\text{C}\ m_2$ (lines 5-7). DEPCON computes the parallel and conflict dependencies relying on the computed method summaries according to Definition 4 and Definition 5, respectively.

Function GETNEXTMETHODPAIR (line 9) selects the next pair according to COVCON that prioritizes pairs based on the frequency of their concurrent executions, to focus the test generation on infrequently or not at all covered pairs [9].

[4]Choudhary et al. 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 [9]. Thus, there are $m \cdot (m+1)/2$ method pairs.

DEPCON generates two concurrent tests with two different prefixes for a method pair $\langle m_1, m_2 \rangle$, as defined in COVCON. The first prefix is composed of a randomly chosen constructor (GETRANDOMCONSTRUCTOR, line 12), the second prefix contains additional method calls after the constructor (GETSTATECHANGER, line 14) [9]. The rationale is that some concurrency faults can only be triggered after bringing the object into a fault-exposing state by invoking a sequence of method calls, whereas other faults may show only on a freshly instantiated instance [9, 43, 54]. Function GETSTATECHANGER of DEPCON relies on the computed method summaries ($\mathbb{MS}$) to further reduce the search space. It forces at least one additional method call in the prefix to be in conflict ($\rightleftharpoons$C) with either m_1 or m_2. The rationale is that the prefixes that do not modify the values of memory locations read by m_1 or m_2 are equivalent to those prefixes that only use the constructor. Intuitively, to affect (for example, with a new execution path) the behavior of the concurrent suffixes at least a memory location read by them has to be modified by the additional calls in the sequential prefix. Avoiding generating and exploring the interleaving spaces of concurrent tests with same suffixes but different albeit redundant prefixes saves precious time-budget.

DEPCON generates the method call suffix_1 that invokes m_1 using the object under test instantiated by the prefix as input parameter (GETSUFFIX(m_1) at line 15) [9]. Similarly, it generates suffix_2 with m_2 (GETSUFFIX(m_2) at line 16). DEPCON generates a new concurrent test *ct* by assembling the obtained method call sequences (line 17). DEPCON explores the interleaving space of ct relying on the non-determinism of the JVM scheduler that is likely to induce a different interleaving every time it executes ct [9, 43]. DEPCON repetitively executes each generated concurrent test *numIter* times (line 19). After every execution, DEPCON checks if the test thrown an exception, if yes it makes sure that the same exception does not manifest when executing any linerarizations of ct (line 20). In affirmative case, it reports the thread-safety violation. Otherwise, it resumes the generation of concurrent tests and their interleaving space exploration until the time-budget expires.

V. EVALUATION

We empirically evaluate DEPCON with a prototype implementation for Java classes, implementation that we developed on top of COVCON [9]. We implemented the algorithm described in Figure 3 by relying on the bytecode manipulator framework ASM [3] to scan the bytecode instructions of Java methods. We evaluated DEPCON with a set of 15 thread-safe classes and compared DEPCON with COVCON, the most effective state-of-the-art concurrent test generator [9, 56].

We addressed three research questions:

- **RQ1 Effectiveness.** *Can* DEPCON *effectively generate concurrent tests that expose thread-safety violations?*
- **RQ2 Comparison.** *Is* DEPCON *more effective than state-of-the-art concurrent test generation?*
- **RQ3 Preprocessing Phase.** *What is the efficiency, completeness and precision of the preprocessing phase?*

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

A. Subjects

We selected a benchmark of 15 classes with known thread-safety violations that have been used in the evaluation of previous work [5, 9, 43, 49]. Table I shows the details of the subjects. Column "*ID*" assignes a unique identifier that we use to identify the subject program in the paper. Column "*Code Base*" reports the subject program that contains the class under test. In our experiments, we considered classes in seven popular code bases. Column "*Version*" gives information about the version that contains the faulty class. Column "*Package*" and Column "*Class Name*" indicate the package and name of the CUT, respectively. Column "*LOC*" shows the lines of code of the CUT, which range from 200 to 3,080. Column "*# Methods*" gives the number of public, protected or private methods of the CUT, while Column "*# Public Methods*" shows the number of public methods. The number of methods includes both the class itself and its non-abstract superclasses (excluding `java.lang.Object`) because this is the code that DEPCON tests. Column "*Fault Type*" indicates the type of concurrency fault for each subject (*atomicity violation* [13] and *data race* [14]).

B. Evaluation Setup

We ran both DEPCON and COVCON on all 15 Java classes[5]. Following related work [9, 54, 56] we chose a time-budget of one hour per class ($\mathcal{B} = 1$ hr). Each experiment terminated when the technique either successfully exposes the thread-safety violation or exhausts the time-budget. Both DEPCON and COVCON relies on the default scheduler of the JVM as interleaving explorer. The original implementation of COVCON uses one iteration for the interleaving explorer ($numIter = 1$ line 18 in Figure 4), 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 could waste testing resources. For both DEPCON and COVCON we used one hundred iterations ($numIter = 100$), which is a good trade-off between effectiveness and cost. To cope with the randomness of the choices made by the tools while generating tests, we repeated each experiment five times using different random seeds. Both tools generate tests pseudo-deterministically given a random seed. To guarantee repeatability of the experiments we used the seeds from 1 to 5. We measure the effectiveness of each technique 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.

C. RQ1 Effectiveness

Columns 4 to 6 in Table II show the results relative to RQ1.

The Success Rate (SR) of DEPCON is 100% for seven subjects: *C1*, *C3*, *C5*, *C7*, *C9*, *C10* and *C12*; for these subjects DEPCON exposes the thread-safety violations in all five runs. The average SR of all subjects is 68%. SR is always greater than 0%, meaning that DEPCON successfully exposes the thread-safety violations in at least one run for all the subjects. This result suggests that Theorem 1 is empirically sound: pruning the search space via parallel and conflict dependencies does not prevent the generation of failure-inducing tests.

The average Failure Detection Time (FDT) of DEPCON ranges from 1 minute and 21 seconds for subject *C1* to 51 minutes and 11 seconds for *C6* (22 minutes and 48 seconds on average). FDT includes all phases of DEPCON: preprocessing (Column "*Avg. Time* $\mathbb{MS}$ *millisec*"), computation of coverage targets and parallel and conflict dependencies, as well as the time for generating concurrent tests and for exploring their interleaving spaces.

The average number of generated tests (#GT) of DEPCON ranges from 18 concurrent tests for Subject *C5* to 5,860 for *C8* (1,642 on average). This indicates that DEPCON explores a relatively small number of concurrent tests before exposing the thread-safety violations.

[5]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

TABLE II
EVALUATION RESULTS

	DEPCON (this work)					COVCON [9]				Comparison (RQ2)			
ID	$\mathcal{M}_{pc}$	Avg. Time $\mathbb{MS}$ millisec.	Success Rate	Avg. FDT (hh:mm:ss)	Avg. #GT	$\mathcal{M}$	Success Rate	Avg. FDT (hh:mm:ss)	Avg. #GT	$\mathcal{M}_{pc}$ reduction	SR improv.	FDT speedup	#GT reduction
C1	21	970	100%	00:01:21	188	351	40%	00:36:15	6,660	16.71×	+60%	26.72×	35.35×
C2	66	1,535	40%	00:43:57	2,810	2,211	40%	00:52:36	9,055	33.50×	-	1.20×	3.22×
C3	52	1,046	100%	00:11:51	1,221	1,378	20%	00:56:00	8,947	26.50×	+80%	4.73×	7.33×
C4	297	1,584	60%	00:34:13	2,617	595	100%	00:10:38	1,055	2.00×	-40%	0.31×	0.40×
C5	22	1,011	100%	00:00:07	18	55	100%	00:00:34	161	2.50×	-	5.21×	8.77×
C6	51	1,733	20%	00:51:11	1,174	1,035	20%	00:54:52	5,010	20.29×	-	1.07×	4.27×
C7	70	1,448	100%	00:01:24	255	351	100%	00:03:03	640	5.01×	-	2.17×	2.51×
C8	292	1,156	40%	00:39:50	5,860	6,216	0%	01:00:00	12,368	21.29×	+40%	1.51×	2.11×
C9	278	1,353	100%	00:03:15	438	8,001	100%	00:09:52	1,720	28.78×	-	3.04×	3.93×
C10	296	1,335	100%	00:02:19	370	861	100%	00:15:56	2,895	2.91×	-	6.88×	7.82×
C11	844	2,252	40%	00:43:54	4,113	23,871	20%	00:57:56	4,320	28.28×	+20%	1.32×	1.05×
C12	114	1,291	100%	00:00:33	116	325	100%	00:07:47	1,842	2.85×	-	14.05×	15.88×
C13	53	971	20%	00:48:18	2,622	231	0%	01:00:00	16,264	4.36×	+20%	1.24×	6.20×
C14	37	1,064	20%	00:48:06	2,609	300	0%	01:00:00	15,911	8.11×	+20%	1.25×	6.10×
C15	427	4,032	80%	00:11:44	222	2,211	80%	00:26:54	568	5.18×	-	2.29×	2.56×
Avg.	**195**	**1,519**	**68%**	**00:22:48**	**1,642**	**3,199**	**55%**	**00:34:10**	**5,828**	**13.89×**	**13%**	**4.87×**	**7.1×**

D. RQ2 Comparison

Columns 7 to 10 in Table II show the results of the state-of-the-art coverage-driven test generator COVCON and Columns 11 to 14 show the comparison results with DEPCON.

The Success Rate (SR) of COVCON is 100% for Subjects *C4*, *C5*, *C7*, *C7*, *C9*, *C10* and *C12*. COVCON fails to expose the thread-safety violations in all runs for Subjects *C8*, *C13* and *C14*. The average SR of all subjects is 55% that is lower than the one of DEPCON (68%). Column "*SR improv.*" in Table II gives the improvement in success rate of DEPCON over COVCON. For six subjects, DEPCON has a higher success rate. For only Subject *C4*, DEPCON exposes the thread-safety violation in fewer runs than COVCON. A manual investigation of the generated tests suggests that the failure-inducing interleaving for Subject *C4* has a low chance of manifestation. Adopting more sophisticated interleaving explorers might overcome this issue [9].

The average Failure Detection Time (FDT) of COVCON ranges from 3 minutes and 33 seconds for Subject *C7* to 1 hour for the three subjects in which COVCON fails to expose the thread-safety violation before the time-budget expires. The average FDT across all subjects is 34 minutes and 10 seconds, which is lower than the one of DEPCON (22 minutes and 48 seconds). Column "*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 sibjects but *C4*.

The average number of Generated Tests (#GT) of COVCON ranges from 161 concurrent tests for Subject *C5* to 16,264 for *C13* (5,824 on average). For each subject, the #GT of COVCON is always higher than the #GT of DEPCON. This explains the speedup of DEPCON. On average DEPCON needs to generate and explore the interleaving space of 7.1× less concurrent tests before exposing the thread-safety violations (Column "*#GT reduction*"). This result demonstrates the effectiveness of the proposed approach to reduce the search space and to drive test generation towards failure-inducing concurrent tests.

E. RQ3 Preprocessing Phase

Column "*Avg. Time* $\mathbb{MS}$ *millisec*" shows the computation cost of the preprocessing phase (Algorithm in Figure 3), which ranges from 970 ms for Subject $C1$ to 16,264 ms for $C13$. This demonstrates the efficiency of our proposed static analysis. To put it in perspective, the average computation cost of the preprocessing step across all subjects is 1,519 ms, which is only 1.66% of the average FDT (22 minutes and 48 seconds).

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

Since DEPCON exposes the thread-safety violation in at least a run for each subject, the preprocessing phase is complete.

Column "$\mathcal{M}_{pc}$ *reduction*" shows that DEPCON drastically reduces the number of coverage targets by 13.89× on average. 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 the Algorithm in Figure 3 to avoid missing any parallel or conflict dependencies.

F. 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.

Another threat to the validity of the results is the choice of the underline interleaving explorer. DEPCON simply relies on the intrinsic non-determinism of the JVM thread scheduler [43]. More sophisticated interleaving explorers are expected to explore the interleaving space of the generated tests more effectively and thus exposing thread-safety violations faster. We leave the evaluation of DEPCON combined with alternative interleaving explorers as a future work.

VI. Related Work

In this section, we briefly survey existing test generators for thread-safe classes, which are closely related to DepCon, and various parallel and conflict dependencies analysis for concurrent object-oriented programs.

Existing concurrent test generators for thread-safe classes can be divided in *random-based*, *coverage-based* and *sequential-test-based* techniques [56]. Random-based techniques [39, 43] generate concurrent tests by combining randomly generated method call sequences with random input parameters. Coverage-based techniques [9, 53, 54] drive the generation of concurrent tests with interleaving coverage criteria [10, 25, 34]. DepCon belongs to this category as it relies on concurrent method pairs [9, 10] as coverage targets. Sequential-test-based techniques [47–50] analyze a given set of sequential tests in input to identify concurrency faults that may occur when combining multiple sequential tests into concurrent tests. None of these techniques performs parallel and conflict dependency analysis to reduce the search space during test generation. We expect that the effectiveness of these techniques would improve if they include the preprocessing phase of DepCon (see RQ2).

Schimmel et al. proposed in a workshop paper AutoRT [52] that shares a similar goal with DepCon, as it 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 [52]. 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. 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 observed 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.

Recently, we presented ConCrash [5] 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, ConCrash requires dynamic information obtained by generating and executing concurrent tests to apply the pruning strategies. 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.

DepCon builds on top of traditional static analyses for concurrent object-oriented programs, such as *object purity* [19, 29], *alias* [22, 35], *may-happen-in-parallel* [1, 2, 37, 38], and *conflict* [45, 46] 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 [37, 38], 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) [19, 29], 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 [11, 12]. We reasoned about such trade-off 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.

VII. Conclusion and Future Work

In this paper, we presented DepCon, a coverage-driven concurrent test generation technique that relies on conflict and parallel dependencies among methods to effectively expose thread-safety violations in a given thread-safe class. Our insight is that static and dynamic analyses, with their dual strengths and weaknesses [12, 58, 63], can work well in synergy for generating concurrent tests. Dynamic analysis guarantees to report true thread-safety violations [43], while static analysis mitigates the inherent incompleteness of dynamic analysis by reducing the search space of possible concurrent tests.

There are several opportunities for future work that can further improve DepCon effectiveness and applicability. We now discuss the two most promising ones.

The static analysis of DepCon often performs an over-approximation of program behaviors to reduce the analysis cost. As a result, some computed dependencies are spurious. Finding new solutions to improve the precision of the analysis without degrading its efficiency is a possible future work that could improve the effectiveness of DepCon.

Currently, DepCon does not target deadlocks. A possible way to extend DepCon for the deadlock detection problem would be to consider an additional type of dependency: two methods acquire in a reverse order at least two common locks. We leave this as an important and interesting future work.

Acknowledgments

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

References

[1] 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 *TACAS*, pages 562–567, 2014.

[2] E. Albert, A. E. Flores-Montoya, and S. Genaim. Analysis of May-Happen-In-Parallel in Concurrent Objects. In *FORTE*, pages 35–51. Springer, 2012.

[3] Asm 5.0 a java bytecode engineering library. https://asm.ow2.io.

[4] F. A. Bianchi, A. Margara, and M. Pezzè. A Survey of Recent Trends in Testing Concurrent Software Systems. *IEEE TSE*, 44(8):747–783, 2018.

[5] F. A. Bianchi, M. Pezzè, and V. Terragni. Reproducing Concurrency Failures from Crash Stacks. In *ESEC/FSE*, pages 705–716, 2017.

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

[7] Y. Cai and L. Cao. Effective and Precise Dynamic Detection of Hidden Races for Java Programs. In *ESEC/FSE*, pages 450–461, 2015.

[8] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape Analysis for Java. In *OOPSLA*, pages 1–19, 1999.

[9] A. Choudhary, S. Lu, and M. Pradel. Efficient detection of thread safety violations via coverage-guided generation of concurrent tests. In *ICSE*, pages 266–277, 2017.

[10] D. Deng, W. Zhang, and S. Lu. Efficient Concurrency-bug Detection Across Inputs. In *OOPSLA*, pages 785–802, 2013.

[11] D. R. Engler and K. Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *SOSP*, pages 237–252, 2003.

[12] M. D. Ernst. Static and Dynamic Analysis: Synergy and Duality. In *WODA*, pages 24–27, 2003.

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

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

[15] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. In *PLDI*, pages 293–303, 2008.

[16] G. Fraser and A. Arcuri. EvoSuite: On the Challenges of Test Case Generation in the Real World. In *ICST*, pages 362–369, 2013.

[17] B. Goetz and T. Peierls. *Java Concurrency in Practice*. Pearson Education, 2006.

[18] J. Gosling, B. Joy, and G. Steele. *The Java language specification*. Addison-Wesley Professional, 2000.

[19] D. Helm, F. Kubler, M. Eichberg, M. Reif, and M. Mezini. A Unified Lattice Model and Framework for Purity Analyses. In *ASE*, pages 340–350, 2018.

[20] M. P. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM TOPLAS*, 12(3):463–492, 1990.

[21] M. D. Hill and M. R. Marty. Amdahl's Law in the Multicore Era. *Computer*, 41(7), 2008.

[22] M. Hind, M. Burke, P. Carini, and J.-D. Choi. Interprocedural Pointer Alias Analysis. *ACM TOPLAS*, 21(4):848–894, 1999.

[23] S. Hong, J. Ahn, S. Park, M. Kim, and M. J. Harrold. Testing Concurrent Programs to Achieve High Synchronization Coverage. In *ISSTA*, pages 210–220, 2012.

[24] S. Hong and M. Kim. A Survey of Race Bug Detection Techniques for Multithreaded Programmes. *STVR*, 25(3):191–217, 2015.

[25] S. Hong, M. Staats, J. Ahn, M. Kim, and G. Rothermel. The Impact of Concurrent Coverage Metrics on Testing Effectiveness. In *ICST*, pages 232–241, 2013.

[26] J. Huang. Scalable Thread Sharing Analysis. In *ICSE*, pages 1097–1108, 2016.

[27] J. Huang, P. O. Meredith, and G. Rosu. Maximal Sound Predictive Race Detection with Control Flow Abstraction. In *PLDI*, pages 337–348, 2014.

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

[29] W. Huang and A. Milanova. ReImInfer: Method Purity Inference for Java. In *FSE*, 38:1–38:4, 2012.

[30] V. Jagannath, M. Gligoric, D. Jin, Q. Luo, G. Rosu, and D. Marinov. Improved Multithreaded Unit Testing. In *ESEC/FSE*, pages 223–233, 2011.

[31] JDK Bug 4728096. https://bugs.java.com/bugdatabase/view_bug.do?bug_id=4728096.

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

[33] Y. Lin and D. Dig. CHECK-THEN-ACT Misuse of Java Concurrent Collections. In *ICST*, pages 164–173, 2013.

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

[35] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized Object Sensitivity for Points-To Analysis for Java. *ACM TOSEM*, 14(1):1–41, 2005.

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

[37] G. Naumovich and G. S. Avrunin. A Conservative Data Flow Algorithm for Detecting All Pairs of Statements That May Happen in Parallel. In *FSE*, pages 24–34, 1998.

[38] G. Naumovich, G. S. Avrunin, and L. A. Clarke. An Efficient Algorithm for Computing MHP Information for Concurrent Java Programs. In *ESEC/FSE*, pages 338–354, 1999.

[39] A. Nistor, Q. Luo, M. Pradel, T. R. Gross, and D. Marinov. BALLERINA: Automatic Generation and Clustering of Efficient Random Unit Tests for Multithreaded Code. In *ICSE*, pages 727–737, 2012.

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

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

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

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

[44] M. Pradel, M. Huggler, and T. R. Gross. Performance Regression Testing of Concurrent Classes. In *ISSTA*, pages 13–25, 2014.

[45] C. v. Praun and T. R. Gross. Object Race Detection. In *OOPSLA*, pages 70–82, 2001.

[46] C. v. Praun and T. R. Gross. Static Conflict Analysis for Multi-threaded Object-oriented Programs. In *PLDI*, pages 115–128, 2003.

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

[48] M. Samak and M. K. Ramanathan. Synthesizing Tests for Detecting Atomicity Violations. In *FSE*, 2015.

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

[50] M. Samak, O. Tripp, and M. K. Ramanathan. Directed Synthesis of Failing Concurrent Executions. In *OOPSLA*, pages 430–446, 2016.

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

[52] J. Schimmel, K. Molitorisz, A. Jannesari, and W. F. Tichy. Automatic Generation of Parallel Unit Tests. In *AST*, pages 40–46, 2013.

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

[54] V. Terragni and S.-C. Cheung. Coverage-driven Test Code Generation for Concurrent Classes. In *ICSE*, pages 1121–1132, 2016.

[55] V. Terragni, S.-C. Cheung, and C. Zhang. RECONTEST: Effective Regression Testing of Concurrent Programs. In *ICSE*, pages 246–256, 2015.

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

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

[58] K. Vorobyov and P. Krishnan. Combining Static Analysis and Constraint Solving for Automatic Test Case Generation. In *ICST*, pages 915–920, 2012.

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

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

[61] T. Yu, W. Wen, X. Han, and J. H. Hayes. Predicting Testability of Concurrent Programs. In *ICST*, pages 168–179, 2016.

[62] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. In *SOSP*, pages 221–234, 2005.

[63] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst. Combined Static and Dynamic Automated Test Generation. In *ISSTA*, pages 353–363, 2011.

[64] W. Zhang, C. Sun, and S. Lu. ConMem: Detecting Severe Concurrency Bugs Through an Effect-oriented Approach. In *ASPLOS*, pages 179–192, 2010.

[65] 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 *ICST*, pages 172–183, 2017.