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

Valerio Terragni\textsuperscript{1}, Mauro Pezzè\textsuperscript{1,2} and Francesco A. Bianchi\textsuperscript{1}
\textsuperscript{1} USI Università della Svizzera italiana, Switzerland
\textsuperscript{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-safety 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   markpos = pos;
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 }

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 \( c_4 \) can manifest the thread-safety violation.

III. MOTIVATION

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

Figure 2 shows four concurrent tests for the class in Figure 1: \( c_1, c_2, c_3 \) and \( c_4 \). All tests share the same \emph{prefix} (see the top of Figure 2) but have different concurrent \emph{suffixes}. Among these tests only \( c_4 \) exposes the thread-safety violation; when executed with some specific thread interleavings, \( c_4 \) triggers a \emph{NullPointerException} at line 20, which is not thrown by any linearizations of the calls in \( c_4 \). An interleaving that can trigger the exception is: \( \text{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 } c_4: \langle \text{prefix, var1.read(), var1.close()} \rangle \text{ and } \langle \text{prefix, var1.close(), var1.read()} \rangle \text{. 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 } c_4 \text{ 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, \( c_1, c_2 \) and \( c_3 \) 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 \emph{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:

\begin{itemize}
  \item their executions can interleave (\emph{parallel dependent} \( \Leftrightarrow \text{P} \));
  \item they can access (with a write and read) at least one shared-memory location in common (\emph{conflict dependent} \( \Leftrightarrow \text{C} \)).
\end{itemize}

For instance, let us consider the concurrent tests \( c_1, c_2 \) and \( c_3 \) in Figure 2. These tests do not meet the two requirements and thus \emph{DEPCon} can safely avoid generating them: \( c_1 \): available \( \not\Leftrightarrow \text{P} \) available because the method is synchronized; available \( \not\Leftrightarrow \text{C} \) available because the method does not perform write accesses on shared-memory locations. \( c_2 \): read \( \not\Leftrightarrow \text{P} \) mark because both methods are synchronized, and thus their instructions are protected by the same lock (\emph{current-object}); read \( \Leftrightarrow \text{C} \) mark because the method \emph{read} writes and the method \emph{mark} reads the same field \emph{pos}. \( c_3 \): close \( \Leftrightarrow \text{P} \) mark because close is not synchronized; read \( \not\Leftrightarrow \text{C} \) mark because the methods do not access the same shared-memory locations.

\footnote{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 $ct_3$ as the methods in its concurrent suffixes have both parallel and conflict dependencies: close ==P read because the instructions of close are not protected by locks; close ==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 $i$-th instruction in $m$. Each instruction has a type: $\text{type}(inst_i)$ 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: $\text{ callees}(m)$ 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 of instructions accessing shared-memory locations that can be triggered by $m$ and by all of its callees.

$$AS(m) = \{inst_i \in \{m \cup \text{ callees}(m)\} \land (\text{type}(inst_i) = R(x) \lor \text{type}(inst_i) = W(x)), \text{ where } x \text{ is a shared-memory location}\}.$$ 

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 $m$:

$$LS(m) = \{l : (3 \text{ int}_s \in m : \text{type}(\text{inst}_s) = ACQ(l) \land i < k) \land (3 \text{ int}_s \in m : \text{type}(\text{inst}_s) = REL(l) \land j < w), \text{ where } k \text{ and } w \text{ are the indexes of the first and last shared-memory accesses in } m, \text{ respectively}\}.$$ 

For example, the lock summary $LS$(mark) of the method mark in Figure 1 is $\{\text{this}\}$, 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, $m_1 ==P m_2$, if and only if their corresponding lock summaries do not share common locks:

$$m_1 ==P m_2 \text{ iff } LS(m_1) \cap LS(m_2) = \emptyset.$$ 

Definition 5. Methods $m_1$ and $m_2$ are conflict dependent, $m_1 ==C m_2$, if and only if their access summaries contain instructions that read and write the same shared-memory location:

$$m_1 ==C m_2 \text{ iff } (\exists W(x) \in AS(m_1) \land \exists W(y) \in AS(m_2)) \lor (\exists W(x) \in AS(m_1) \land \exists W(y) \in AS(m_2)), \text{ where } x \equiv y.$$ 

Parallel and conflict dependencies are relations between methods. Both relations are symmetric ($m_1 == m_2$ implies $m_2 == m_1$) but neither reflexive (not necessary each method relates with itself) nor transitive ($m_1 == m_2$ and $m_2 == m_3$ does not imply that $m_1 == m_3$).

Our notion of conflict differs from classic race condition [60, 62], which include the case in which two threads 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 \( ct = \langle \text{prefix}, \text{suffix}_x, \text{suffix}_y \rangle \) violates thread-safety (with respect to Definition 1), then the two suffixes contain two methods \( m_1 \in \text{suffix}_x \) and \( m_2 \in \text{suffix}_y \) with parallel and conflict dependencies:

\[
\exists m_1 \in \text{suffix}_x \land \exists m_2 \in \text{suffix}_y : \text{prefix} \Rightarrow P_{m_2} \land \text{prefix} \Rightarrow C_{m_2}.
\]

Proof by contradiction: Let us assume that a concurrent test \( ct \) violates thread-safety and \( \forall (m_1, m_2) : m_1 \in \text{suffix}_x \land m_2 \in \text{suffix}_y \land (m_1 \not\Rightarrow P_{m_2} \lor m_1 \not\Rightarrow C_{m_2}) \). If \( \forall (m_1, m_2) \neq P_{m_2} \), then 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).

As a result, \( ct \) cannot violate thread-safety (contradiction).

C. Preprocessing Phase: Computing Method Summaries

Figure 3 shows the algorithm for computing the method summaries 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 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 bytecode instructions \( \text{inst}_i \) of \( m \) (line 11), and checks the type of each \( \text{inst}_i \) 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 to 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 aload. 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) (for getfield) or W(ref) (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 monitorexit instructions [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 first shared-memory access in m. If AS(m) is empty, the algorithm executes Function getLock(insti) 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 [18]. The algorithm executes Function getLock(insti) to get the lock released by insti (line 24). If AS(m) is empty, the algorithm removes l from LS(m) (line 26), otherwise it adds l to tmp-lock-releases supporting variable. It removes the lock from 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 invokevirtual, invokeinterface, invokespecial, invokevirtual 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 impure (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 monitorexit 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}.

---

3The 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 \[ M_S \leftarrow \text{COMPUTEALLMS}(CUT) \text{ // see Algorithm 1} \]
3 \[ M \leftarrow \text{COMPUTECFP}(CUT) \text{ // all coverage targets} \]
4 \[ M_{pc} \leftarrow \emptyset \text{ // init } M_{pc} \subseteq M \text{ reduced coverage targets} \]
5 for each \((m_1, m_2) \in M\) do
6     if \(m_1 \Rightarrow P m_2 \land m_1 \Rightarrow C m_2\) then
7         add \((m_1, m_2)\) to \(M_{pc}\)
8   while B is not expired do
9       \((m_1, m_2) \leftarrow \text{GETNEXTMETHODPAIR}(M_{pc})\)
10      for i from 0 to 1 do
11          \[ \text{prefix} \leftarrow \text{GETRANDOMCONSTRUCTOR}() \]
12          else
13              \[ \text{prefix} \leftarrow \text{GETSTATECHANGER}(M_S) \]
14              suffix1 \leftarrow \text{GETSUFFIX}(m_1)
15              suffix2 \leftarrow \text{GETSUFFIX}(m_2)
16              ct \leftarrow \text{ASSEMBLETEST}(\text{prefix}, \text{suffix}_1, \text{suffix}_2)
17          for \( i \) from 1 to \\text{numIter} do
18              exception \leftarrow \text{EXECUTE}(ct)
19              if \( \text{exception} \neq \emptyset \) then
20                  ALLPOSSIBLELINEARIZATIONS(ct)
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 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.

While COVCON considers all possible pairs as coverage targets, DEPCON considers only those pairs \(M_{pc} \subseteq M\) that have parallel and conflict dependencies. DEPCON computes the method summaries with Function COMPUTEALLMS (line 2 in Figure 3). Then, DEPCON adds the pair \((m_1, m_2)\) to \(M_{pc}\), if \(m_1 \Rightarrow P m_2 \land m_1 \Rightarrow C m_2\) (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].

\footnote{Choudhary et al. consider \((m_1, m_2)\) and \((m_2, m_1)\) 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 (\(\mathcal{M}_S\)) to further reduce the search space. It forces at least one additional method call in the prefix to be in conflict (\(\Rightarrow C\)) 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 suffix1 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 suffix2 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 \text{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 linearizations 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?
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” assigns 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. Following related work [9, 54, 56] we chose a time-budget of one hour per class (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 B, 0 otherwise.
- **Fault Detection Time (FDT):** time taken by the technique to expose the thread-safety violation, B if the time-budget expires.
- **# Generated Tests (#GT):** number of generated concurrent tests when 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 5 minutes and 48 seconds for C6 (22 minutes and 48 seconds on average). FDT includes all phases of DEPCON: preprocessing (Column “Avg. Time MS millisecond”), 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.
TABLE II
EVALUATION RESULTS

<table>
<thead>
<tr>
<th>ID</th>
<th>(\mathcal{M}_{pc})</th>
<th><strong>DEPCon (this work)</strong></th>
<th><strong>COVCon [9]</strong></th>
<th><strong>Comparison (RQ2)</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>21</td>
<td>970</td>
<td>100%</td>
<td>00:01:21</td>
</tr>
<tr>
<td>C2</td>
<td>66</td>
<td>1,535</td>
<td>40%</td>
<td>00:43:57</td>
</tr>
<tr>
<td>C3</td>
<td>52</td>
<td>1,046</td>
<td>100%</td>
<td>00:11:51</td>
</tr>
<tr>
<td>C4</td>
<td>297</td>
<td>1,584</td>
<td>60%</td>
<td>00:34:13</td>
</tr>
<tr>
<td>C5</td>
<td>22</td>
<td>1,101</td>
<td>100%</td>
<td>00:00:07</td>
</tr>
<tr>
<td>C6</td>
<td>51</td>
<td>1,733</td>
<td>20%</td>
<td>00:51:11</td>
</tr>
<tr>
<td>C7</td>
<td>70</td>
<td>1,448</td>
<td>100%</td>
<td>00:01:24</td>
</tr>
<tr>
<td>C8</td>
<td>292</td>
<td>1,156</td>
<td>40%</td>
<td>00:39:50</td>
</tr>
<tr>
<td>C9</td>
<td>278</td>
<td>1,553</td>
<td>100%</td>
<td>00:03:15</td>
</tr>
<tr>
<td>C10</td>
<td>296</td>
<td>1,335</td>
<td>100%</td>
<td>00:02:19</td>
</tr>
<tr>
<td>C11</td>
<td>844</td>
<td>2,252</td>
<td>40%</td>
<td>00:43:54</td>
</tr>
<tr>
<td>C12</td>
<td>114</td>
<td>1,291</td>
<td>100%</td>
<td>00:00:33</td>
</tr>
<tr>
<td>C13</td>
<td>53</td>
<td>971</td>
<td>20%</td>
<td>00:48:18</td>
</tr>
<tr>
<td>C14</td>
<td>37</td>
<td>1,064</td>
<td>20%</td>
<td>00:48:06</td>
</tr>
<tr>
<td>C15</td>
<td>427</td>
<td>4,032</td>
<td>80%</td>
<td>00:11:44</td>
</tr>
<tr>
<td>Avg</td>
<td>195</td>
<td>1,519</td>
<td>68%</td>
<td>00:22:48</td>
</tr>
</tbody>
</table>

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, 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 subjects 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 MS milli-sec” 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 “\(\mathcal{M}_{pc} \text{ reduction}\)” 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 into 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. presented 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


