Reproducing Concurrency Failures from Crash Stacks

Francesco A. Bianchi
USI Università della Svizzera italiana, Switzerland
francesco.bianchi@usi.ch

Mauro Pezzè
USI Università della Svizzera italiana, Switzerland
University of Milano-Bicocca, Italy
mauro.pezze@usi.ch

Valerio Terragni
USI Università della Svizzera italiana, Switzerland
valerio.terragni@usi.ch

ABSTRACT

Reproducing field failures is the first essential step for understanding, localizing and removing faults. Reproducing concurrency field failures is hard due to the need of synthesizing a test code jointly with a thread interleaving that induce the failure in the presence of limited information from the field. Current techniques for reproducing concurrency failures focus on identifying failure-inducing interleavings, leaving largely open the problem of synthesizing the test code that manifests such interleavings.

In this paper, we present ConCrash, a technique to automatically generate test codes that reproduce concurrency failures that violate thread-safety from crash stacks, which commonly summarize the conditions of field failures. ConCrash efficiently explores the huge space of possible test codes to identify a failure-inducing one by using a suitable set of search pruning strategies. Combined with existing techniques for exploring interleavings, ConCrash automatically reproduces a given concurrency failure that violates the thread-safety of a class by identifying both a failure-inducing test code and corresponding interleaving. In the paper, we define the ConCrash approach, present a prototype implementation of ConCrash, and discuss the experimental results that we obtained on a known set of ten field failures that witness the effectiveness of the approach.

CCS CONCEPTS

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

KEYWORDS

Software Crashes, Debugging, Concurrency, Test Generation

ACM Reference Format:
Francesco A. Bianchi, Mauro Pezzè, and Valerio Terragni. 2017. Reproducing Concurrency Failures from Crash Stacks. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 04-08, 2017 (ESEC/FSE'17),* 12 pages.
https://doi.org/10.1145/3106237.3106292

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE'17, September 04-08, 2017, Paderborn, Germany
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3106292

1 INTRODUCTION

Concurrent systems are increasingly popular due to the spread of multi-core architectures. These systems are prone to concurrency faults, which are extremely hard to avoid due to the complexity of the thread synchronization and the huge size of the interleaving space. Concurrency faults often remain undetected during the testing process, and manifest in production runs, leading to failure that are difficult to reproduce because they often occur only in the presence of specific thread interleavings [37]. Reproducing failures is the first essential step towards understanding, localizing and removing the related faults [4]. Reproducing concurrency failures from field reports is a non trivial task, since it requires identifying both a test code and thread interleaving of the test code that induce the failure from the limited information available in the reports, where a test code is a runnable piece of code that exercises the system under test, and an interleaving is a temporal order of a set of shared memory accesses.

The main techniques to reproduce concurrency failures rely on information collected at runtime either continuously (*Record-and-replay* approaches [1, 21, 23, 29]) or only at the time of the failures (*Post-processing* approaches [52, 57]). Both classes of approaches require information that may be expensive and hard to obtain in many practical situations, and identify the failure-inducing interleaving but not the failure-inducing test code. Record-and-replay techniques instrument the program for recording executions, with a runtime overhead ranging from 10% up to 4,000% in some worst cases [23], which may be acceptable in testing but not in production environments [54]. Post-processing techniques rely on memory core-dumps that provide full information of the program state at the time of the failure [52, 57]. Memory core-dumps are expensive to collect and not available on all platforms [6]. Moreover, both the recorded executions and memory core-dumps often contain sensitive information, which introduces privacy concerns [53]. Both Record-and-replay and Post-processing techniques produce failure-inducing conditions on the program input and the state [52, 57], the failure-inducing interleaving [1, 21, 23, 29] or both, but do not synthesize a fully executable failure-inducing test code, as the one presented in Figure 3 that we discuss in the next section.

In this paper, we present ConCrash (*CONcurrency CRASHes reproduction*), the first automated technique that synthesizes both failure-inducing test codes and related interleavings with neither overhead nor privacy issues. ConCrash targets concurrency failures that violate the thread-safety of a class. Thread-safe classes encapsulate efficient synchronization mechanisms that guarantee a correct behavior of the class when invoked concurrently from multiple threads, and are largely adopted in modern concurrent systems as they avoid the difficulty of writing such synchronization

```
992: void info(String s) {
993:   . . .
996:   log(Level.INFO, msg);
997: }

476: void log(Level l, String s) {
477:   . . .
480:   LogRecord r =
481:     new LogRecord(l, s);
482:   doLog(r);
483: }

451: void doLog(LogRecord r) {
452:   . . .
458:   log(r);
453: }
```

```
414: void log(LogRecord r) {
413:   . . .
418:   synchronized(this) {
419:     if(filter != null) {
420:     // Filter can be set to null
421:       if(!filter.isLoggable(r)) {
422:          return;
423:       }
424:     }
425:   }
426: }

386: void setFilter(Filter f) {
387:   . . .
391:   this.filter = f;
392: }
```

Figure 1: Faulty class `java.util.logging.Logger` of JDK 1.4.1

```
1 | java.lang.NullPointerException
2 |  at java.util.logging.Logger.log(Logger.java:421)             POF
3 |  at java.util.logging.Logger.doLog(Logger.java:458)
4 |  at java.util.logging.Logger.log(Logger.java:482)
5 |  at java.util.logging.Logger.info(Logger.java:996) Crashing Method
6 |  at test.TestCode$1.runTest(TestCode.java:10)
7 |  at java.lang.Thread.run(Thread.java:662)
```

Figure 2: Crash stack of class `Logger` (Bug ID 4779253)

mechanisms [19]. ConCrash requires in input only the class that violates thread-safety and the standard crash stack of the failure. Differently from execution traces and memory core-dumps that are expensive to produce and hard to obtain, crash stacks are easily obtainable and do not suffer from performance and privacy issues [53]. Recent studies show that 56-70% of concurrency failures lead to crashes or hangs that usually generate a crash stack [27, 46].

Crash stacks contain only partial information about the state of the system, thus challenging the reproduction of concurrency failures [55]. In particular, crash stacks provide only information about the failing thread and give little data about the state of the objects and the values of the input parameters of the methods involved in the failure. To identify a failing execution that reproduces a crash stack, ConCrash must explore a huge space of possible test codes and thread interleavings. ConCrash explores efficiently the huge space by alternately generating test codes and exploring thread interleavings. It iteratively generates test codes by implementing pruning strategies that exclude both redundant and irrelevant test codes to optimize the exploration of the interleaving space. Test codes are redundant if they induce the same interleaving space of previously investigated test codes and thus would not reproduce the failure. Test codes are irrelevant if ConCrash can infer the impossibility of reproducing the failure from the crash stack and the single-threaded execution of the call sequences that comprise the test code. ConCrash pruning strategies are cost-effective as they analyze single-threaded executions of the call sequences rather than exploring the full interleaving space of concurrent executions. ConCrash privileges short test codes to improve the efficiency of exploring interleavings, localizing, and fixing the fault.

We evaluated ConCrash by experimenting with a prototype implementation for the Java language on ten real-world concurrency failures. The experiments indicate that ConCrash takes on average 46 seconds to reproduce a concurrency failure (including the time for both generating test codes and exploring their interleavings).

The contributions of this paper include: (i) the first automatic technique to synthesize failure-inducing concurrent test codes from crash stacks to reproduce concurrency failures in thread-safe classes, (ii) a publicly available implementation of the technique, ConCrash [49], (iii) an experimental evaluation of ConCrash showing the effectiveness of the proposed technique.

```
 private void runTest() throws Throwable {
   // Sequential Prefix
   Logger sout = Logger.getAnonymousLogger();
   MyFilter myFilter0 = new MyFilter();
   sout.setFilter((Filter)myFilter0);

   // Suffix 1
   Thread t1 = new Thread(new Runnable() {
     public void run() {
       sout.info("");        // Crashing Method
     }});
    // Suffix 2
   Thread t2 = new Thread(new Runnable() {
     public void run() {
       sout.setFilter(null);   // Interfering Method
     }});
   t1.start();t2.start();
 }
```

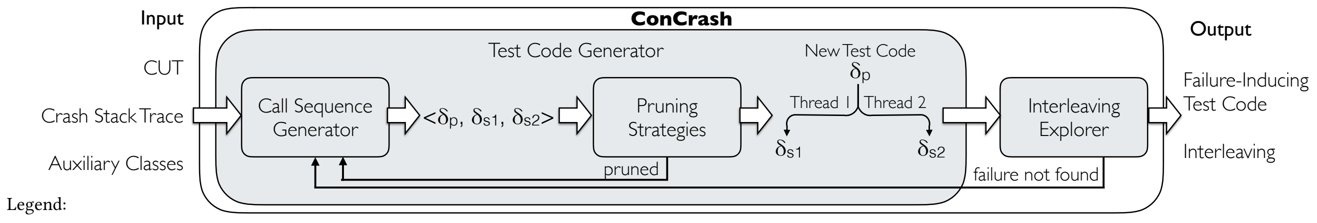
Figure 3: A concurrent test code that reproduces the crash stack in Figure 2

2 REPRODUCING CONCURRENCY FAILURES

In this paper, we address the problem of synthesizing concurrent test codes that reproduce concurrency failures of classes that violate thread-safety. A class is *thread-safe* if it encapsulates synchronization mechanisms that prevent incorrect accesses to the class from multiple threads [19]. Incorrect synchronization mechanisms are *concurrency faults* that manifest at runtime as *concurrency failures*, that is, deviations from the expected behaviour of a concurrent usage of the class, and expose a thread-safety violation. In this work, we address the relevant class of concurrency failures that manifest as runtime exceptions. A key characteristic of concurrency failures is that they manifest non-deterministically, due to the non-determinism of the scheduler that decides the threads order of multi-threaded executions. The order of accesses to shared memory locations is fixed within one thread, but can vary across threads. An *interleaving* is a total order relation of shared memory accesses among threads [26]. Concurrent executions can manifest many different interleavings, and only some –usually few– of them trigger concurrency failures [37].

Motivating example. Figure 1 shows the code snippet of a known concurrency fault in class `java.util.logging.Logger` of the JDK 1.4.1 library. Method `log` accesses field `filter` at lines 419 and 421 within a synchronized block that locks the object instance. The method checks whether the field is initialized (line 419) before dereferencing it (line 421). Method `setFilter` (line 386) accesses and modifies the same field without locking the object instance. As a result another thread can execute line 391 between the executions of lines 419 and 421 while a thread is executing method `log` and set the reference to `null`, thus violating the intended atomicity of method `log`. If both threads access the same object instance, this thread interleaving triggers a `NullPointerException` at line 421 (Figure 2). Figure 3 shows a concurrent test code that can induce such failure-inducing interleaving.

Crash stack trace. ConCrash generates concurrent tests code that reproduce concurrency failures from crash stacks. Figure 2 presents an example of crash stack produced when executing the test code in Figure 3. A *crash stack trace* (or simply crash stack) reports the ordered sequences of functions on the call stack at the time of the failure and terminates the sequence with the exception that results from the failure (`NullPointerException` at line 1 in Figure 2) [25]. Each entry (frame) in the crash stack reports a function



δ_p == sequential prefix, δ_{s1}, δ_{s2} == concurrent suffixes

Figure 4: Logical architecture of ConCrash

and a code location. The code location of each entry identifies either the location of the call to the next function or, in the case of the top entry (e.g., line 2 in Figure 2), the location of the *Point Of Failure* (POF), which is the static line of code that triggered the failure. Given a crash stack, developers can easily identify both the class responsible for the concurrency failure, which we denote as Class Under Test (CUT), and the CUT method whose invocation led to the failure, which we denote as *crashing method*. Such method corresponds to the outermost CUT method in the crash stack. In our running example the CUT is the JDK class `Logger` and the crashing method is method `info` of class `Logger` as inferred from the frame at line 5 in Figure 2.

Concurrent test code. Concurrency failures of (supposedly) thread-safe classes can be reproduced with multi-threaded executions of concurrent test codes. In this paper, a *concurrent test code* is a set of method call sequences that exercise the public interface of the CUT from multiple threads without additional synchronization mechanisms other than the one implemented in the CUT [32, 38, 45, 47]. A *call sequence* is an ordered sequence of method calls $\delta = \langle m_1, \dots, m_n \rangle$ that are executed in a single thread. The methods in the sequence have a possible empty set of input parameters, which can be either primitive values, for instance of type floats, integers and booleans, or references to objects created in previous method calls. We treat the object receiver of an instance method as the first parameter of the method [35, 47]. The methods in a call sequence can be either methods of the public interface of the CUT or methods of *auxiliary classes* required to instantiate non-primitive parameters of the CUT methods.

A test code is composed of a *sequential prefix* and a set of *concurrent suffixes*. The sequential prefix is a call sequence that invokes (i) a constructor to create an instance of the CUT that we call *Shared Object Under Test* (SOUT) and (ii) a sequence of method calls that modifies the SOUT state in order to enable the execution of the concurrent suffixes to trigger the concurrency failure. A concurrent suffix is a call sequence that is executed concurrently with other concurrent suffixes after the sequential prefix. The concurrent suffixes invoke methods that access the SOUT concurrently. We consider test codes with exactly two concurrent suffixes, following the results that show that 96% of concurrency faults manifest by enforcing a certain partial order between two threads only [27], and in line with most studies on concurrent test code generation [32, 38, 45, 47].

Intuitively for reproducing the concurrency failure, one suffix shall invoke the crashing method, while the other suffix shall invoke a method whose execution can lead to an unexpected interleaving that *interferes* with the crashing method. We call such method *interfering method*. The method `setFilter` is an example of interfering method of the `Logger` running example.

Problem definition and challenges. The problem addressed in this paper can be formulated as follows:

Problem Definition. *Given a crash stack trace, the corresponding CUT, a set of auxiliary classes and a time-budget, generate a concurrent test code that reproduces the crash stack trace in input, annotated with a failure-inducing interleaving within the time-budget.*

When addressing this problem, we are challenged by (i) the limited information in the crash stack and (ii) the high cost of exploring the interleaving space of a test code. The crash stack only gives limited information about how to construct a failure-inducing test code. It does not provide enough information to infer the methods and the input parameter values that comprise the test code. ConCrash needs to explore the huge space of different combinations of sequential prefixes, interfering methods and input parameter values to identify a specific combination of method calls and parameters that comprise a failure-inducing test code. For instance, to reproduce the concurrency failure of the `Logger` example in Figure 3, ConCrash needs to identify the sequential prefix ⟨`Logger sout=Logger.getAnonymousLogger(); MyFilter myFilter0=new MyFilter(); sout.setFilter((Filter) myFilter0)`⟩, the interfering method `sout.setFilter(null)` and the crashing method `sout.info("")`. With different sequential prefixes, for example ⟨`Logger sout=Logger.getAnonymousLogger();sout.setFilter(null)`⟩, the test code does not reproduce the failure for any interleaving, since the if condition at line 419 would be evaluated to `false`.

The cost of exploring the interleaving space of a test code is inflated by the large amount of possible interleavings. With a time budget that allows to explore the interleaving space of only few test codes, a random exploration of test codes would not be effective, since thousands of randomly generated test codes are needed for triggering a concurrency failure [32, 38]. ConCrash introduces an effective strategy for exploring the huge space of test codes and generating few concurrent test codes likely to reproduce the failure.

3 CONCRASH

As depicted in Figure 4, ConCrash iteratively executes two main components, the *Test Code Generator* and the *Interleaving Explorer* until generating a failure-inducing test code and interleaving. At each iteration, the Test Code Generator synthesizes a new test code, and the Interleaving Explorer looks for a thread interleaving of the test code that reproduces the concurrency failure.

The Test Code Generator exploits a set of pruning strategies to *steer the test code generation towards test codes that are likely to reproduce the failure*. By pruning test code space before exploring the interleaving space, ConCrash limits the expensive exploration of the interleaving space to the interleavings that correspond to test codes that are most likely to expose the concurrency failures. The pruning strategies trim both test codes that are *redundant* with respect to previously generated test codes and test codes that are *irrelevant* with respect to the concurrency failure in input. Intuitively, a test code is redundant if it manifests the same interleavings of previously explored test codes, and irrelevant if it cannot manifest a failure-inducing interleaving. Exploring the interleaving space of such test codes is fruitless.

The pruning strategies rely on runtime information collected by executing sequentially and in isolation the method call sequences that comprise a candidate concurrent test code. The sequential execution of a call sequence can effectively approximate the behavior of the call sequence when executed concurrently with other method call sequences [47]. Analyzing sequential executions is less expensive than exploring all the possible interleavings of concurrent executions. While existing concurrent test code generators leverage sequential executions for concurrency testing purposes [41–43, 45, 47], the key intuition of ConCrash is to use this information together with crash stacks to effectively synthesize test codes that reproduce a concurrency failure.

The Interleaving Explorer checks if the interleaving space of a test code synthesized by the Test Code Generator contains at least one interleaving that reproduces the failure. The Interleaving Explorer is not a contribution of this paper, but is based on the approach recently proposed by Machado et al. to determine the existence of an interleaving of a given test code that violates a program assertion that encodes the concurrency failure [29]. ConCrash iteratively executes the Test Code Generator and the Interleaving Explorer until producing a test code and an interleaving that reproduce the failure or until the time budget expires.

3.1 Test Code Generator

Figure 5 shows the test code generation algorithm. As discussed in Section 2, a test code is composed of a sequential prefix, denoted as δ_p, and two concurrent suffixes δ_{s1} and δ_{s2} that are executed concurrently after δ_p. The prefix δ_p creates a shared object under test (SOUT) of type CUT, and invokes the methods that bring the SOUT in a failure-inducing state. The suffixes δ_{s1} and δ_{s2} access the SOUT concurrently trying to manifest a failure-inducing interleaving.

The algorithm explores a search space modeled with a tree, and is composed of an initialization step (lines 2-12) and two main steps: the exploration of a new combination of method call sequences (lines 13-22) and the elaboration of the new combination, function `PRUNING` (lines 23-42), which includes the collection of runtime information (lines 24-26) and the pruning strategies (lines 27-42). Below we describe in details the Tree model, the minimization of the test codes, the initialization, the exploration of new combinations, the collection of runtime information and the pruning strategies.

Tree model. ConCrash finds a combination of δ_p, δ_{s1} and δ_{s2} that constitutes a failure-inducing test code, by exploring the space of possible call sequences. Following Terragni's and Cheung's approach [47] we represent the search space as a rooted, directed and potentially infinite tree whose root node is a call sequence that instantiates the shared object under test SOUT of type CUT. Figure 6 shows an excerpt of a tree model of class `Logger`. The edges represent method call sequences. Starting from the root that represents the initialization sequence, the nodes represent concatenations of call sequences (edges) that correspond to the ordered sequence of the method calls along the path from the root to the node. For instance, the node δ_O in Figure 6 represents the sequence ⟨ `Logger sout = Logger.getAnanymousLogger(); Filter f = new Filter(); sout.setFilter(f); sout.info("");` ⟩ obtained traversing the tree from the root to the node ($\delta_A, \delta_5, \delta_1$). ConCrash incrementally builds the Tree model starting from the root. The basic operator for building the Tree model is the *node traversal operator* that creates a new child node [47]. Given a method m and a node representing a sequence δ, the node traversal operator produces a child node that represents a new sequence obtained from δ by appending a sequence of method calls (an edge), with m being the last method call. The node traversal operator may add other method calls before m to create the non-primitive parameters of m, if any. For instance, in Figure 6 the edge δ_1 corresponds to a single method call, while the edge δ_3 corresponds to two method calls, where the first call creates the input parameter h of the method `removeHandler`. ConCrash always extends nodes with methods of the CUT that have at least an input parameter (including the method receiver) of type CUT, and binds exactly one of them to the object SOUT. Therefore by construction, each edge in the tree accesses the same object SOUT. Referring always to SOUT is crucial because the suffixes trigger concurrency failures by accessing the same shared object [32]. The values of the input parameters that are not bounded to SOUT, if any, are chosen from a pool of representative values.

Test code minimization. The same concurrency failure can be triggered with many test codes made of different method calls, some of which may be irrelevant with respect to the failure. Short test codes are preferable, because long concurrent suffixes increase the cost of exploring the interleavings, as the number of interleavings grows factorially with respect the number of statements executed in each concurrent thread [26]. Long sequential prefixes increase the computational costs and are more difficult to understand and investigate than short ones [18]. Therefore, ConCrash aims at generating short test codes.

The concurrent suffixes of the test codes correspond to single edges in the tree. Each edge contains a method call that accesses the SOUT object. Limiting concurrent suffixes to single edges does not impact on the failure reproduction capabilities, since interleavings of shared memory accesses from multiple calls within the same concurrent suffix do not expose any thread-safety violation that cannot be exposed with a single method call [19, 20, 47]. ConCrash reduces the length of sequential prefixes by adopting a *Breadth-First Search* exploration strategy: it explores all sequences with n edges before exploring sequences with $n + 1$ edges.

Initialization (lines 2-12). ConCrash starts by initializing to empty the state repository $\mathbb{S}$, the coverage repository $\mathbb{C}$, and the list of sequences that are pending for being extended `pendingSeqs` (lines 2, 3 and 4, respectively). ConCrash then initializes the current level of exploration of the tree to zero (line 5). The algorithm randomly generates the pool of primitive and non-primitive parameters values using Randoop [35] (line 6). Subsequently, it determines the crashing method (line 7) by parsing the crash stack in input (see Section 2). It then creates the root nodes of the Trees, one for each constructor in the class, by instantiating sequences that invoke the methods with an object of type CUT as return type, for instance constructors (line 8). ConCrash creates a new root δ_{sout}

```
input  : CUT (class under test), classes (auxiliary classes), CST (crash
         stack trace), $\mathcal{B}$ (timeBudget)
output : t failure-inducing test code
1  function ConCrash
     /*                     Initialization                          */
2    $\mathbb{S} \leftarrow \varnothing$                                  // state repository
3    $\mathbb{C} \leftarrow \varnothing$                               // coverage repository
4    pendingSeqs[...] $\leftarrow \varnothing$                   // sequences to be extended
5    level $\leftarrow$ 0          // current level of sequence exploration
6    pool $\leftarrow$ CREATE-PARAMETER-VALUES(classes)
7    cm $\leftarrow$ EXTRACT-CRASHING-METHOD(CST)
8    for each construcor $m(\tau_1, \dots \tau_n)$ of CUT do
9      for each value $(v_1 \dots v_n)$ of type $(\tau_1 \dots \tau_n)$ in pool do
10       $\delta_{sout} \leftarrow m(v_1, \dots v_n)$
11       if $\delta_{sout}$ does not throw an exception then
12         add $\delta_{sout}$ to end of pendingSeqs[0]
     /*        Exploration of new combinations $\delta_p, \delta_{s1}, \delta_{s2}$         */
13   while timeBudget is not expired do
14     while pendingSeqs[level] $\neq \varnothing$ do
15       $\delta_p \leftarrow$ get and delete the first $\delta$ in pendingSeqs[level]
16       for each value $(v_1 \dots v_n)$ of type $cm(\tau_1 \dots \tau_n)$ in pool do
17         $\delta_{s1} \leftarrow cm(v_1 \dots v_n)$
18         for each public method $m(\tau_1, \dots \tau_n)$ of CUT do
19           for each value $(v_1 \dots v_n)$ of type $m(\tau_1 \dots \tau_n)$ in
             pool do
20             $\delta_{s2} \leftarrow m(v_1 \dots v_n)$
21             PRUNING($\delta_p$, $\delta_{s1}$, $\delta_{s2}$)
22     level++
23 function PRUNING($\delta_p$, $\delta_{s1}$, $\delta_{s2}$)
24   $\delta_{p,s1} \leftarrow$ append $\delta_{s1}$ to $\delta_p$
25   $\delta_{p,s2} \leftarrow$ append $\delta_{s2}$ to $\delta_p$
26   execute $\delta_{p,s1}$ and $\delta_{p,s2}$   // Collection of runtime information
27   if $\delta_{p,s1}$ and $\delta_{p,s2}$ do not throw exceptions (PS-Exception)
28   $\wedge\ \exists\ e \in \overline{\mathcal{M}}(\delta_{cm})$ : stack(e)= CST          (PS-Stack)
29   $\wedge\ \langle \overline{\mathcal{M}}(\delta_{p,s1}), \overline{\mathcal{M}}(\delta_{p,s2}) \rangle \notin \mathbb{C}$          (PS-Redundant)
30   $\wedge\ \overline{\mathcal{M}}(\delta_{p,s2})$ interferes with $\overline{\mathcal{M}}(\delta_{p,s1})$      (PS-Interfere)
31   $\wedge\ \overline{\mathcal{M}}(\delta_{p,s1})\ ||\ \overline{\mathcal{M}}(\delta_{p,s2})$              (PS-Interleave) then
32     add $\langle \overline{\mathcal{M}}(\delta_{p,s1}), \overline{\mathcal{M}}(\delta_{p,s2}) \rangle$ to $\mathbb{C}$
33     t $\leftarrow$ ASSEMBLE-TEST-CODE($\delta_p$, $\delta_{s1}$, $\delta_{s2}$)
34     isFailure $\leftarrow$ INTERLEAVING-EXPLORER(t)
35     if isFailure = true then
36       return t                    // failure-inducing test code
37   if $\mathcal{S}(\delta_{s2}) \notin \mathbb{S}$ and $\delta_{p,s2}$ does not throw exception then
38     add $\mathcal{S}(\delta_{s2})$ to $\mathbb{S}$
39     if $\overline{\mathcal{M}}(\delta_{p,s1})$ interferes with $\overline{\mathcal{M}}(\delta_{p,s2})$ then
40       add $\delta_{p,s2}$ begin of pendingSeqs[level+1]
41     else
42       add $\delta_{p,s2}$ end of pendingSeqs[level+1]
```

Figure 5: The ConCrash algorithm

for each of such methods and for each combination of parameter values (lines 8, 9, 10). The algorithm checks whether the execution of δ_{sout} throws an exception (line 11) and, if this is not the case, adds δ_{sout} to the list `pendingSeqs[0]` (line 12). The algorithm does not further elaborate the sequences δ_{sout} that throw an exception since they do not successfully create the object.

Exploration of new combinations $\langle \delta_p, \delta_{s1}, \delta_{s2} \rangle$ (lines 13-22). ConCrash explores new combinations iteratively until either the failure is reproduced or the time budget expires (line 13). At each iteration, ConCrash removes a sequence δ_p from `pendingSeqs[`*level*`]` (line 15), and generates the children of the leaf of the Tree that corresponds to δ_p, starting from the children related to the crashing method (*cm*) exploring different values for the input parameters. We denote each of the edges resulting from the extensions as δ_{s1} (line 17). ConCrash explores all the children of δ_p, obtained by extending δ_p with all public methods in CUT (line 18) with each combination of the input parameters in the pool (line 19). We denote the edges resulting from the extensions as δ_{s2} (line 20). Every combination of δ_p, δ_{s1} and δ_{s2} corresponds to a candidate concurrent test code. Function `PRUNING` analyses each combination of δ_p, δ_{s1} and δ_{s2} to determine if it should be pruned or not (line 21). ConCrash considers all public methods in the CUT to obtain δ_{s2} (line 18) because the crash stack does not contain information about the interfering method, and thus ConCrash needs to explore all the possible candidates to identify the right one.

`PRUNING` (lines 23-42). Function `PRUNING` prunes the search space (lines 27-42) relying on the runtime information obtained by executing the input call sequences (lines 24-26).

Collecting runtime information (lines 24-26). Let $\delta_{p,s1}$ and $\delta_{p,s2}$ be the sequences that extend δ_p with the edges δ_{s1} and δ_{s2}, respectively (lines 24 and 25), ConCrash executes $\delta_{p,s1}$ and $\delta_{p,s2}$ in isolation (single-threaded execution) (line 26), and collects the following runtime information for each sequence $\delta \in \{\delta_{p,s1}, \delta_{p,s2}\}$: (i) whether δ throws an uncaught exception, (ii) the *sequential coverage* of the last method call in δ [47], and (iii) the state of the object SOUT after executing δ, which is obtained by serializing SOUT in a deep copy semantic.

The *sequential coverage* is a metric recently presented by Terragni and Cheung, that is defined on the sequential execution of call sequences [47], and is used to infer the possibility of a concurrent test code to induce new interleavings with respect to the previously generated test codes. ConCrash exploits *sequential coverage* to identify and avoid both redundant and irrelevant test codes. Let the trace $E = \langle e_1, \dots, e_k \rangle$ of a call sequence δ be the ordered sequence of events exhibited by a sequential (single-threaded) execution of δ. An event can be one of the following:

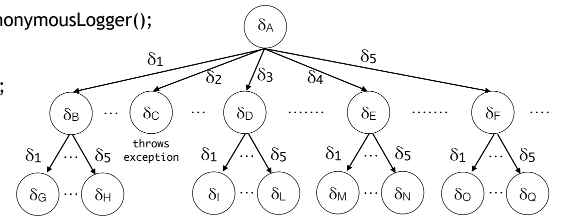
- write *W(f)* and read *R(f)* accesses to an object field *f*;
- lock acquire *ACQ(l)* and lock release *REL(l)* events;
- method enter *ENTER(m)* and exit *EXIT(m)* events.

Given a call sequence $\delta = \langle m_1, \dots, m_n \rangle$, the *trace of a method call* $m_i \in \delta$ is the non-empty segment E_i of E such that E_i contains only the events triggered directly or indirectly by the invocation of m_i [47]. Given a call sequence δ, its *sequential coverage* $\mathcal{M}(\delta)$ is defined as the partition $\{E_1, \dots, E_n\}$ of E, that is the unordered set composed of the *n* method call traces of *E* [47].

Since all the test codes generated by ConCrash are composed of concurrent suffixes with only the last method call accessing SOUT, we are only interested in the sequential coverage of such method calls. We denote the last method call trace E_n in $\mathcal{M}(\delta)$ as $\overline{\mathcal{M}}(\delta)$.

Pruning strategies (lines 27-42). ConCrash prunes the combination $\langle \delta_p, \delta_{s1}, \delta_{s2} \rangle$ according to different strategies. If the code is neither redundant (line 29) nor irrelevant (lines 27, 28, 30, 31) ConCrash updates the coverage repository $\mathbb{C}$ (line 32), assembles a new concurrent test code *t* (line 33) and invokes the Interleaving Explorer component to determine if the interleaving space of *t* contains at least one interleaving that can reproduce the failure (line 34). If this is the case (line 35), ConCrash outputs *t* and its failure-inducing interleaving and terminates (line 36).

```
δA: Logger sout = Logger.getAnonymousLogger();
δ1: sout.info("");
δ2: sout.removeHandler(null);
δ3: Handler h = new Handler();
    sout.removeHandler(h);
δ4: sout.setFilter(null);
δ5: Filter f = new Filter();
    sout.setFilter(f);
```



δ_p δ_{s1} δ_{s2}	***Is the test generated ?***
<δA, δ1, δ2>	No, δC throws exception
<δA, δ1, δ3>	No, pruned by PS-Interfere
<δA, δ1, δ4>	Yes, but not failure inducing
<δA, δ1, δ5>	No, pruned by PS-Redundant
<δF, δ1, δ4>	**Yes, failure-inducing**

Figure 6: A Tree model of class Logger

If ConCrash does not terminate, it checks if $\delta_{p,s2}$ should be added to the `pendingSeqs` list for further extensions (line 37), that is, ConCrash checks whether the state δ_{s2} produced by executing $\delta_{p,s2}$ either throws an exception or has been already explored. If not, ConCrash adds $\mathcal{S}(\delta_{s2})$ to $\mathbb{S}$ and inserts $\delta_{p,s2}$ in the `pendingSeqs` of the next level (line 38). Following previous work [35, 38, 47], ConCrash does not extend sequences that throw exceptions when executed sequentially, as all of their extensions throw the same exception at the same point [35][1]. ConCrash decides to add the current $\delta_{p,s2}$ either at the begging or at the end of `pendingSeqs`$[level + 1]$ depending on the priority of the sequence (lines 40 and 42, respectively). Sequences at the beginning of the list have higher priority, as they will be consumed earlier by ConCrash (line 15).

We now describe in detail the pruning strategies and the decision procedure that determines whether $\delta_{p,s2}$ should be added at the beginning or at the end of `pendingSeqs`$[level + 1]$.

ConCrash prunes a combination $\langle\delta_p, \delta_{s1}, \delta_{s2}\rangle$ (lines 27-31) if:

***PS-Exception**: $\delta_{p,s1}$ or $\delta_{p,s2}$ throw an exception when executed sequentially* (line 27), even if the exception matches the crash stack trace in input. This is because our focus is on failures that can only be reproduced during concurrent executions. This pruning strategy is a standard practice for concurrent test code generation [32, 38, 45, 47].

***PS-Stack**: $\nexists\ e \in \overline{\mathcal{M}}(\delta_{p,s1})\ :\ stack(e) = CST$* (Crash Stack Trace), where $stack(e)$ is the call stack trace of e, obtained by analysing the method entry and exit points in $\overline{\mathcal{M}}(\delta_{p,s1})$. A necessary condition of a test code for reproducing a failure is to reach the point of failure (POF) with the same calling context of the considered crash stack [25]. ConCrash prunes the call sequences δ_{s1} that when executed sequentially do not reach the POF with the same call stack of CST. For example in Figure 6, ConCrash prunes the combinations with $\delta_{s1} = \delta_B$ since the sequential execution of δ_B does not reach the POF (line 421).

***PS-Redundant:** $\langle\overline{\mathcal{M}}(\delta_{p,s1}), \overline{\mathcal{M}}(\delta_{p,s2})\rangle \in \mathbb{C}$.* ConCrash prunes the combinations whose concurrent suffixes induce an already observed pair of sequential coverages $\overline{\mathcal{M}}(\delta_{p,s1})$ and $\overline{\mathcal{M}}(\delta_{p,s2})$, as inferred from the coverage repository $\mathbb{C}$. ConCrash prunes redundant pairs of sequential coverage since the resulting concurrent test code would lead to an interleaving space identical to a previously generated test code [47].

***PS-Interfere:** $\nexists\, e_1, e_2 \in \overline{\mathcal{M}}(\delta_{p,s1}) \times \overline{\mathcal{M}}(\delta_{p,s2}) : e_1 = R(f), e_2 = W(f)$.* ConCrash prunes the combination if the two concurrent suffixes do not access the same variables or the interfering method $\delta_{p,s2}$ only reads the variable accessed in common. The intuition behind this pruning strategy is that for triggering a concurrency failure the interfering method has to interfere, by writing a shared variable that is accessed by the crashing method. For example, in Figure 6, ConCrash prunes the combination with $\delta_{s1} = \delta_B$ and $\delta_{s2} = \delta_D$ because the sequential execution of δ_D does not write any variable read by δ_B. ConCrash considers the object fields not only of the CUT but also of the auxiliary classes. This is because the object fields of the non-primitive fields of SOUT are also shared across threads.

$\overline{\mathcal{M}}(\delta_1)$: e1: ACQ(l), e2: W(f), e3: REL(l), e4: ACQ(l), e5: W(f), e6: REL(l)

$\overline{\mathcal{M}}(\delta_2)$: e7: ACQ(l), e8: W(f), e9: REL(l)

	LS	LH
e2	{l}	{l}
e5	{l}	∅
e8	{l}	{l}

Figure 7: Example of Lockset History (LH)

***PS-Interleave:** $\nexists\, e_1, e_2 \in \overline{\mathcal{M}}(\delta_{p,s1}) \times \overline{\mathcal{M}}(\delta_{p,s2})\ :\ e_1 = RW, e_2 = RW, LH(e_1) \cap \underline{LH}(e_2) = \varnothing$*, where LH denotes the lock history of an event $e_x \in \overline{\mathcal{M}}(\delta)$, defined as the set of locks that are acquired but never released in $\overline{\mathcal{M}}(\delta)$ before triggering the event e_x, and RW denotes either a read or write memory access. More formally, $LH(e_x) = \{l \mid \exists e_j = ACQ(l) \in \overline{\mathcal{M}}(\delta), \nexists e_k = REL(l) \in \overline{\mathcal{M}}(\delta), w < k < x\}$, where w is the index of the first event in $\overline{\mathcal{M}}(\delta)$. ConCrash prunes a combination if the two concurrent suffixes cannot *interleave* when assembled in a concurrent test code, and thus their concurrent execution cannot lead to a concurrency failure [19]. Differently from the traditional lockset (LS) [44], LH takes into account the history of the release events. In fact, the traditional lockset can determine if a pair of events of two threads are protected by the same lock (e.g., to detect data races [44]), but cannot infer if two executions can interleave. For instance in the sequential coverage of the threads reported in Figure 7, the two executions can clearly interleave when executed concurrently, and in particular, the events e_7, e_8 and e_9 can interleave between the events e_3 and e_4. However, the lockset LS cannot infer such property, since the events e_2, e_4 and e_8 have the same lockset, and lockset cannot determine if the lock held by e_2 and e_4 has been released between the two events. On the contrary, LH can determine that the two executions can interleave, since the $LH(e_5) \cap LH(e_8) = \varnothing$.

ConCrash relies on PS-Interfere to decide how to prioritize $\delta_{p,s2}$ in the list (lines 40 and 42). If $\overline{\mathcal{M}}(\delta_{p,s2})$ interferes with $\overline{\mathcal{M}}(\delta_{p,s1})$ (see Condition PS-Interfere), ConCrash adds $\delta_{p,s2}$ at the beginning or the list, otherwise at the end, since if $\overline{\mathcal{M}}(\delta_{p,s2})$ writes the same variables accessed by the crashing method, $\delta_{p,s2}$ is more likely to lead to a program state that could make the crashing method behave differently if executed in this new state. This is only a prioritization, not a pruning strategy.

Our pruning strategies could potentially rely on static, rather than dynamic information. For example, PS-Interfere could identify

[1]Similarly with previous work we are assuming that 1) sequential executions are deterministic given the same inputs [38, 47]. Classes whose behaviour depend on system time or network communication are excluded. 2) the CUT does not spawn new threads [45, 47], which is generally the case for concurrent libraries [38]

Table 1: Subjects and concurrency failures

Subjects							Concurrency Failures			
ID	**Class Under Test (CUT)**	**Version**	**Code Base**	**Total SLOC**	**CUT SLOC**	**# Methods**	**Issue ID**	**Fault Type**	**Type of Exception**	**Crash Depth**
1	PerUserPoolDataSource	1.4	Commons DBCP	9,451	719	68	369	Race	`ConcurrentModificationException`	4
2	SharedPoolDataSource	1.4		9,451	546	44	369	Race	`ConcurrentModificationException`	4
3	IntRange	2.4	Commons Math	18,016	278	44	481	Atom.	`AssertionError`	1
4	BufferedInputStream	1.1	Java JDK	3,791	304	12	4225348	Atom.	`NullPointerException`	2
5	Logger	1.4.1		2,193	528	45	4779253	Atom.	`NullPointerException`	4
6	PushbackReader	1.8		11,562	143	13	8143394	Atom.	`NullPointerException`	1
7	NumberAxis	0.9.12	JFreeChart	64,713	1,662	119	806667	Atom.	`IllegalArgumentException`	2
8	XYSeries	0.9.8		51,614	200	28	187	Race	`ConcurrentModificationException`	4
9	Category	1.1	Log4j	10,773	387	43	1507	Atom.	`NullPointerException`	1
10	FileAppender	1.2		10,273	185	13	509	Atom.	`NullPointerException`	2

the candidate interfering methods by statically collecting all the possible accesses to object fields, without requiring program execution. However, by relying on dynamic information PS-Interfere can be more effective since only a subset of accesses could be executed under a specific control flow. Moreover, ConCrash already executes each generated call sequences to identify those that throw exceptions or lead to redundant states, thus the additional overhead of collecting dynamic information is minimal.

3.2 Interleaving Explorer

ConCrash explores the interleaving space of a generated test code to infer if the test code is failure-inducing, i.e., it can manifest an interleaving that reproduces the concurrency failure in input.

Current techniques to explore the interleaving space of a given test code examine the space either exhaustively or selectively [5]. Techniques that exhaustively explore all possible interleavings can be very expensive due to enormous size of interleaving spaces [13, 51]. Techniques that explore interleaving spaces selectively, based on particular classes of concurrency faults, like data races [31, 33], atomicity violations [16, 17, 36, 48, 56], order violations [15, 22, 58, 59] and deadlocks [7, 8, 14] can be efficient, but may miss the failure-inducing interleaving if it does not belong to the particular class of the concurrency fault considered.

The ConCrash Interleaving Explorer relies on Cortex, a technique for reproducing concurrency failures proposed by Machado et al. [29], which is more efficient than exhaustive exploration of interleavings spaces and does not make any assumptions on the type of concurrency fault.

The ConCrash Interleaving Explorer executes the given test code and collects an execution trace in which the shared variables and the local variables that are data-dependent from shared variables are treated as symbolic. Starting from the execution trace, the ConCrash Interleaving Explorer builds a SMT formula [12] whose solutions (if any) identify the interleavings that violate an assertion that encodes the concurrency failure. A program failure can be easily encoded in form of an assertion from a crash stack trace, since it gives the point of failure (POF) and the type of runtime exception. For a detailed description of Cortex, the interested readers can refer to the seminal work [23] and its extensions [28, 29].

4 EVALUATION

To experimentally evaluate ConCrash, we developed a prototype implementation, and we experimented with a set of ten known concurrency failures reported in five popular Java code bases.

We addressed three research questions:

RQ1 *How effective is ConCrash in reproducing concurrency failures?*

RQ2 *What is the contribution of each pruning strategy in reducing the search space?*

RQ3 *Is ConCrash more effective than competing state-of-the-art testing approaches?*

4.1 Experimental Setup

We experimented with a prototype implementation of ConCrash that implements the algorithm presented in Figure 5. The prototype uses AutoConTest [47], a concurrent test code generator developed by Terragni and Cheung based on the sequential coverage metric, and Cortex, an interleaving exploration tool developed by Machado et al. [29], which leverages Java PathFinder (JPF) [50] for symbolic execution and Z3 [12] as constraint solver. The ConCrash prototype implements the pruning strategies described in Section 3.1. We denote the versions of the ConCrash prototype with the different pruning strategies as `PS-Stack`, `PS-Redundant`, `PS-Interfere`, `PS-Interleave`, respectively, and the version of ConCrash without pruning strategies as `NO-Pruning`. All six ConCrash prototypes have `PS-Exception` enabled since is not our contribution.

We compared ConCrash with ConTeGe [38] and AutoConTest [47], two representative state-of-the-art approaches that generate concurrent test code for testing concurrent programs. ConTeGe randomly generates test codes and explores the interleavings through stress testing. AutoConTest generates test codes guided by sequential coverage (see Section 3) and explores the interleaving space of each generated test code with a dynamic detector of atomicity violations. Both ConTeGe and AutoConTest are *failure-oblivious*, that is, they are not designed to reproduce a given failure. In absence of techniques that generate test codes for reproducing a given failure, we use ConTeGe and AutoConTest as baseline to assess the ability of ConCrash to drive the generation of test codes towards a specific failure.

Subjects. We selected a benchmark of ten classes with known thread safety violations that have been used in the evaluation of previous work [32, 38, 43, 47]. We considered the subjects used in the related papers, and selected the subjects that (i) produce a crash stack, (ii) have been confirmed to be failures, and (iii) can be analysed with JPF without compatibility issues. For each subject we obtained a single crash stack either from the bug report, when available, or by executing a failure-inducing test code documented in related work [38, 47]. We added the program assertions encoding

Table 2: Experimental results. The cell background indicates the degrees of speedup with respect to `NO-Pruning`: LOW (>1.0x and <2.0x), MEDIUM (≥ 2.0 and < 10.0), or HIGH (≥ 10.0)

	RQ1								RQ2														
	ConCrash								NO-Pruning			PS-Stack			PS-Redundant			PS-Interfere			PS-Interleave		
ID	FR	FRT (sec.)			FTID			FTS	FR	FRT	FTID	FR	FRT	FTID	FR	FRT	FTID	FR	FRT	FTID	FR	FRT	FTID
	%	min	max	avg	min	max	avg	avg	%	avg	avg	%	avg	avg	%	avg	avg	%	avg	avg	%	avg	avg
1	100%	27	99	63	1	4	2	4	20%	15,456	376	100%	526	23	40%	14,749	258	100%	727	18	20%	15,395	430
2	100%	22	85	42	1	4	2	4	80%	9,240	250	100%	362	17	80%	7,294	128	100%	390	10	80%	9,128	195
3	100%	10	16	13	1	1	1	4	100%	204	13	100%	157	13	100%	138	8	100%	17	1	100%	196	13
4	100%	10	21	15	1	2	2	5	100%	77	7	100%	64	7	100%	63	5	100%	43	4	100%	26	2
5	100%	39	84	70	2	4	3	5	100%	6,520	491	100%	2,576	36	100%	3,200	232	100%	543	32	100%	3,369	185
6	100%	7	7	7	1	1	1	4	100%	33	3	100%	20	3	100%	34	3	100%	11	1	100%	31	3
7	100%	27	31	30	1	1	1	3	100%	508	11	100%	294	11	100%	463	9	100%	52	1	100%	491	11
8	100%	26	185	107	1	15	8	6	100%	2,758	269	100%	166	14	100%	2,752	269	100%	1,292	126	100%	2,751	269
9	100%	17	33	25	1	1	1	5	100%	348	28	100%	267	27	100%	336	28	100%	60	3	100%	345	28
10	100%	23	183	92	1	9	5	10	100%	540	22	100%	487	22	100%	342	12	100%	122	5	100%	523	23
AVG	**100%**	**21**	**74**	**46**	**1**	**4**	**3**	**5**	**90%**	**3,569**	**147**	**100%**	**492**	**17**	**92%**	**2,937**	**95**	**100%**	**326**	**20**	**90%**	**3,226**	**116**

the failures, as required by the Interleaving Explorer (Cortex [29]). We easily inserted such assertions relying on the POF and the type of the thrown exception reported in the crash stack.

Table 1 provides details about the subject programs: column *ID* introduces a unique identifier that we use in the paper to identify the subject program; column *Class Under Test (CUT)* is the class under test of the subject program; columns *Version* and *Code Base* report the version and the code base that contains the faulty class, respectively; columns *Total SLOC* and *CUT SLOC* report the total number of Source Lines of Code of both the code base and of the CUT, including non-abstract super classes, if any, respectively; column *# Methods* indicates the number of public methods of the CUT; columns *Issue ID*, *Fault Type*, *Type of Exception* and *Crash Depth* report the identifier of the issue in the corresponding bug repository, the type of the known fault, the type of the resulting exception and the number of frames in the crash stack, respectively.

Setup. We ran each technique on all the subjects. Each experiment terminated either when the technique successfully reproduces the failure or exhaustes a time budget of five hours. Although ConCrash systematically explores method call sequences and input parameters, the order of exploration is arbitrary, and thus, different runs of ConCrash can lead to different results as portions of the search space containing failure inducing test code could be explored before or after, depending on such order. Such order is set pseudo-deterministically with an input random seed. To mitigate threats that may derive from the non-deterministic choices while generating test codes, we repeated each experiment five times using different random seeds. We measure the effectiveness of each technique with the following metrics:

Failure Reproduction (FR) that is 1 if the failure is reproduced within the given time budget $\mathcal{B}$, 0 otherwise.

Failure Reproduction Time (FRT) that is the overall elapsed time in seconds for identifying the first test code and failure inducing interleaving. This time includes the cost of both generating test codes and exploring their interleavings. If the failure is not reproduced within $\mathcal{B}$, that is, $FR = 0$, we optimistically underapproximate FRT to $\mathcal{B}$.

Failure-inducing Test ID (FTID) that is the ID of the first failure-inducing test code. The IDs are assigned in ascending order. FTID = n indicates that ConCrash explored the interleaving space of n test codes before reproducing the failure. A low value of FTID indicates that the technique is effective in generating a failure-inducing test code. If the failure is not reproduced within $\mathcal{B}$ ($FR = 0$), we underapproximated FTID to the ID of the last generated test code.

Failure-inducing Test Size (FTS) that is the size of the failure-inducing test code measured as the total number of outermost method calls in δ_p, δ_{s1}, and δ_{s2}. The lower the value of FTS is, the easier localizing and understanding the failure is. If the failure is not reproduced within $\mathcal{B}$, that is, $FR = 0$, we set FTS = N/A.

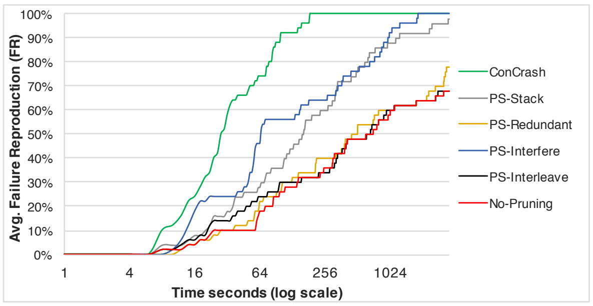
4.2 RQ1 - Effectiveness

The leftmost columns of Table 2 report the experimental results about the effectiveness of ConCrash in reproducing concurrency failures (*RQ1*). The table reports the aggregated results of the five runs for each subject. Column *FR*, Failure Reproduction, indicates that ConCrash reproduces all the concurrency failures. Columns *FRT*, Failure Reproduction Time, indicate a time for reproducing the failures that ranges from 7 to 185 seconds, with an average of 46 seconds. On average on all fifty runs, ConCrash spends 1% of the time for generating test codes and 99% of the time for exploring interleavings. Column *FTID*, Failure Inducing Test ID, indicates that ConCrash explored the interleaving space of a minimum of 1 test codes and a maximum of 15 test codes, with an average of 3 test codes. Column *FTS*, Failure-inducing Test Size, reports a size of the generated test codes from 3 to 10 outermost method calls.

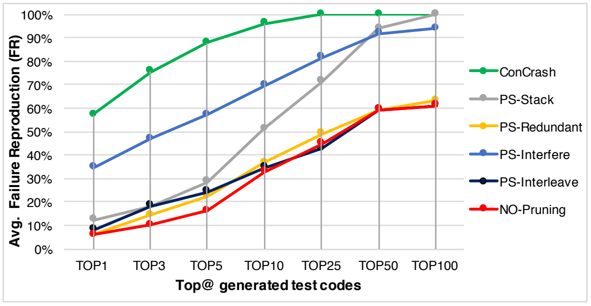
These results witness the effectiveness of ConCrash. The approach reproduces all the considered failures on all the fifty runs within a reasonable time, with test codes of reasonable size. The results also confirm that the cost of reproducing a concurrency failure mostly depends on the cost of exploring interleavings. The ability of ConCrash to effectively steer the test code generation through a failure-inducing test code is the main efficiency factor. ConCrash generates three test codes on average and at most 15 test codes in the worst cases to identify a failure-inducing test code.

4.3 RQ2 - Pruning Strategies

The rightmost columns of Table 2 report the experimental results about the effectiveness of the different pruning strategies (*RQ2*). The table reports the Failure Reproduction (columns *FR*), the Failure Reproduction Time (columns *FRT*) and the Failure-inducing Test



(a) **Time vs Average Failure Reproduction Rate**



(b) **# of Generated Test Codes vs Average Failure Reproduction Rate**

Figure 8: Aggregate comparison of the ConCrash pruning strategies for the ten subjects

ID (columns *FTID*) for the different pruning strategies. We did not record significant modifications of the Failure-inducing Test Size with respect to the experiment discussed above that has been carried on with the main ConCrash approach.

PS-Stack and PS-Interfere reproduce the concurrency failures for all runs (columns *FR*=100%), while No-Pruning, PS-Redundant and PS-Interleave reproduce the failures only in some runs (rows ID 1 and ID 2), leading to an average failure reproduction rate (columns *FR*) of 90%, 92%, and 90%, respectively.

The average failure reproduction time (columns *FRT*) ranges from 326 seconds for PS-Interfere to 3,569 seconds for NO-Pruning, while the average failure-inducing test ID (columns *FTID*) ranges from 1 to 430 test codes. The cell background highlights the contribution of each pruning strategy by indicating the degree of speedup with respect to NO-Pruning: LOW (>1.0x and <2.0x), MEDIUM (≥ 2.0 and < 10.0), or HIGH (≥ 10.0). Both PS-Stack and PS-Interfere strategies lead to a speedup for all subjects, with an average medium and high speedup, respectively (bottom row *AVG*). On the contrary, PS-Redundant and PS-Interleave strategies lead to a speedup for five and three subjects, respectively, and they both achieve a low overall average speedup (bottom row *AVG*). The results indicate that the effectiveness of the various pruning strategy can vary across subjects.

PS-Stack is particularly effective when only few executions reach the POF under the calling context specified in the crash stack trace (CST), as it prunes all those test codes that fail to do so. In fact, PS-Stack is more effective for the subjects with the highest CST depth (four) (ID 1, 2, 5, and 8), while it is less effective for those subjects with depth one (ID 3, 6, and 9). Intuitively, the deeper the CST is, the harder is to reach the POF with the right calling context. PS-Redundant is particularly effective when the execution space of the CUT methods is characterized by few execution paths. In such situation, the invocation of the same method with different parameters leads to a redundant sequential coverage, thus increasing the effectiveness of PS-Redundant.

PS-Interfere is particularly effective when the object fields read by the crashing method are written by only few methods in the CUT, as it prunes the test codes in which the interfering methods do not write such fields. For instance, in the subject IntRange the crashing method reads object fields that are written by only one of the 18 methods in the CUT, and PS-Interfere drastically reduces the search space to only one pair of crashing and interfering methods.

PS-Interleave is particularly effective when a CUT largely adopts synchronization mechanisms to access its object fields. This is the case, for instance, of a Java class that declares most of its methods as synchronized. In such case, PS-Interleave prunes many irrelevant test codes. For instance, PS-Interleave is very effective with BufferedInputStream (ID 4), as the crashing methods and the majority of the CUT methods are declared as synchronized.

The results indicate that PS-Interfere is in general the most effective pruning strategy, followed by PS-Stack, PS-Redundant and PS-Interleave. However, PS-Interfere does not achive the highest speedup for every subjects. For instance, PS-Stack and PS-Interleave are more effective than PS-Interfere for two subjects (ID 4 and ID 8, respectively). This result suggests that ConCrash effectiveness is given by the synergetic combination of all pruning strategies. The diagrams in Figure 8 show that ConCrash outperforms each pruning strategy in isolation. The diagram in Figure 8 (a) plots the average *FR* of all the ten subjects for ConCrash and for the different pruning strategies with respect to the time (in log scale), and indicates that ConCrash achieves a failure rate of 100% much faster than any individual pruning strategy. The diagram in Figure 8 (b) plots the success rate of the first TOP-N test codes generated with ConCrash and with the different pruning strategies. ConCrash achieves more than 90% of failure success rate with only 10 test codes, and 100% within the first 25 ones. PS-Stack achieves 100% of failure success rate within the first 100 test codes, while all the other pruning strategies never achieves 100% within 100 test codes.

4.4 RQ3 - Comparison with Testing Approaches

Table 3 reports the failure reproduction (columns *FR*), the average values for the failure reproduction time (columns *FRT*), the failure-inducing test ID (columns *FTID*), and the failure-inducing test size (columns *FTS*) for ConTeGe and AutoConTest. The results are directly comparable with the corresponding columns *RQ1* of Table 2 that report the data for ConCrash on the same benchmark. The results indicate that ConCrash outperforms both ConTeGe and AutoConTest. ConTeGe presents an average failure reproduction rate of 18%, since it reproduces the crash stacks in 9 out of 50 runs, while generating more than 20,000 test codes, on average. AutoConTest also achieves a low average failure reproduction rate, with an average of 28%. AutoConTest focuses on atomicity violations only, is ineffective in the presence of data race failures (subjects ID 1, 2, 8), suffers from compatibility problems with subjects ID 6, 9, and 10 ("-" in the table), and does not reproduce the failure of class Logger, since it covers the failure-inducing interleaving

Table 3: Comparison with test code generators

ID	ConTeGe [38]				AutoConTest [47]			
	FR %	**FRT avg**	**FTID avg**	**FTS avg**	**FR %**	**FRT avg**	**FTID avg**	**FTS avg**
1	0%	18,000	14,177	N/A	0%	18,000	N/A	N/A
2	0%	18,000	7,736	N/A	0%	18,000	N/A	N/A
3	0%	18,000	25,712	N/A	100%	23	1	56
4	80%	4,487	7,465	12	0%	18,000	6	N/A
5	0%	18,000	1,491	N/A	0%	18,000	6	N/A
6	20%	14,510	5,796	10	-	-	-	-
7	0%	18,000	34,960	N/A	100%	93	1	124
8	40%	12,387	25,215	10	0%	18,000	N/A	N/A
9	40%	14,410	41,461	15	-	-	-	-
10	0%	18,000	39,912	N/A	-	-	-	-
AVG	**18%**	**15,379**	**20,392**	**12**	**28%**	**12,874**	**4**	**90**

(atomicity violation) with inputs that do not trigger the failure. The effectiveness of AutoConTest is comparable with ConCrash in the cases it succeeds, but AutoConTest generates much larger test codes than ConCrash.

Our results suggest that testing techniques that are designed as failure-oblivious are not effective in reproducing a given concurrency failure, as they generate many test codes that are irrelevant for the given failure. While ConCrash that leverages crash stacks and novel pruning strategies effectively drives the test code generation towards a failure-inducing test code.

5 RELATED WORK

Reproducing sequential failures. There are two classes of approaches for reproducing failures in sequential programs. *Record-replay* approaches [2, 3, 25, 34] rely on information collected *before* the failure occurred, for example, execution traces, while *Post-processing* approaches [9, 10, 30, 39] rely on information collected *after* the failure occurred, for example, crash stacks and memory core-dumps. Differently from ConCrash, these techniques do not explicitly target concurrency failures and do not address the challenges introduced by multi-threaded executions and concurrency, which are addressed by ConCrash.

Reproducing concurrency failures. To reproduce failure-inducing thread interleavings, Record-replay approaches for concurrency failures record thread-sensitive information during multi-threaded executions. ODR [1] requires to record the input of the program, the total ordering of lock instructions, and a sampling of the executed instructions; LEAP [21] dynamically records all the accesses to shared variables; STRIDE [60] and CARE [24] improve LEAP by reducing the runtime overhead and the amount of recorded information; CLAP [23] uses a lightweight instrumentation which records only the local control-flow choices of each thread; Symbiosis [28] and Cortex [29] extend CLAP by isolating the root cause of the failure [28] and by reproducing failures that are control-flow dependent [29]. Post-processing approaches for reproducing concurrency failures rely on core-dump information. ESD [57] combines core-dump information analysis and symbolic execution to determine the failure-inducing inputs and interleavings. Weeratunge et al. [52] present a technique that generates a failure inducing interleaving by comparing the failing core-dump with the core-dump of passing runs. The main limitation of all these techniques is that they require either a trace of a failing execution or the core-dump of the failure, which may be hard to obtain [6]. Instead, ConCrash only requires the crash stack of the failure, which is easily obtainable. Furthermore, all of these techniques, differently from ConCrash, do not aim at generating failure-inducing concurrent test codes. ConCrash complements existing approaches, as the test codes generated by ConCrash can be given in input to each of these techniques. In fact, ConCrash relies on Cortex to identify the failure-inducing interleavings.

Generating concurrent test codes. Recent techniques that generate concurrent test codes for exposing thread-safety violations [11, 32, 38, 40–42, 45, 47] do not aim to reproduce a given concurrency failure but rather to test the many concurrent behaviors of a class under test. As such, in the context of failure reproduction, they generate many test codes that are irrelevant for the given failure (RQ3). Instead, ConCrash effectively drives the test code generation towards test codes that are likely to reproduce the failure in input.

6 CONCLUSION AND FUTURE WORK

This paper presented ConCrash, the first technique to generate concurrent test codes for reproducing concurrency failures of classes that violate thread-safety from crash stacks. The key contribution of ConCrash is a set of pruning strategies that analyze the executions of call sequences, and infer whether the test code obtained by combing such sequences can reproduce the concurrency failure. Our experimental results demonstrated the effectiveness of ConCrash. Both ConCrash and the benchmark are publicly available to ease future work in this area [49].

For efficiency reasons, the pruning strategies analyze the sequential executions of call sequences, which only approximate their behaviors in a concurrent execution. Due to this approximation, our pruning strategies cannot guarantee to never discard test codes that can reproduce the failure. For example, even if a call sequence δ does not reach the POF when executed sequentially, it is unsafe to prune it away (PS-Stack) because δ could reach the POF due to the interference of a concurrent thread [58]. However, in the subjects considered such situation never occurred. Future studies are needed to evaluate how safe the pruning strategies are in practice. ConCrash is the first attempt to address the problem of generating test codes for reproducing concurrency failures.

There are several opportunities for future work, and we now discuss the two most promising. First, ConCrash can be extended to reproduce failures caused by deadlocks, by leveraging dedicated runtime monitors to produce the crash stacks that ConCrash requires, adopting an interleaving explorer specific for deadlocks, for instance MagicFuzzer [7], and disabling the PS-Interfere pruning strategy, which is inadequate for deadlocks. Second, ConCrash explores the space of possible method call sequences by considering a set of fixed primitive values. As a result, ConCrash can miss failures which depend on values not contained in this set. Combining ConCrash with symbolic execution could mitigate such limitation.

ACKNOWLEDGMENTS

This work is partially supported by the Swiss SNF project *ASysT: Automatic System Testing* (162409). We would like to thank the FSE 2017 reviewers for their suggestions that helped improve this work.

REFERENCES

[1] Gautam Altekar and Ion Stoica. 2009. ODR: Output-Deterministic Replay for Multicore Debugging. In *Proceedings of the Symposium on Operating Systems Principles (SOSP '09)*. ACM, 193–206.

[2] Shay Artzi, Sunghun Kim, and Michael D. Ernst. 2008. ReCrash: Making Software Failures Reproducible by Preserving Object States. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '08)*. Springer, 542–565.

[3] Jonathan Bell, Nikhil Sarda, and Gail Kaiser. 2013. Chronicler: Lightweight Recording to Reproduce Field Failures. In *Proceedings of the International Conference on Software Engineering (ICSE '13)*. IEEE Computer Society, 362–371.

[4] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. 2008. What Makes a Good Bug Report?. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '16)*. ACM, 308–318.

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

[6] D. Brodeur and T. Fors. 2002. Capturing Snapshots of a Debuggee's State during a Debug Session. (July 4 2002). https://www.google.com/patents/US20020087950 US Patent App. 09/963,085.

[7] Yan Cai and W. K. Chan. 2012. MagicFuzzer: Scalable Deadlock Detection for Large-Scale Applications. In *Proceedings of the International Conference on Software Engineering (ICSE '12)*. IEEE Computer Society, 606–616.

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

[9] Satish Chandra, Stephen J. Fink, and Manu Sridharan. 2009. Snugglebug: A Powerful Approach to Weakest Preconditions. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, 363–374.

[10] Ning Chen and Sunghun Kim. 2015. STAR: Stack Trace Based Automatic Crash Reproduction via Symbolic Execution. *IEEE Transactions on Software Engineering* 41, 2 (2015), 198–220.

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

[12] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS/ETAPS '08)*. Springer, 337–340.

[13] Juergen Dingel and Hongzhi Liang. 2004. Automating Comprehensive Safety Analysis of Concurrent Programs Using Verisoft and TXL. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '12)*. ACM, 13–22.

[14] Mahdi Eslamimehr and Jens Palsberg. 2014. Sherlock: Scalable Deadlock Detection for Concurrent Programs. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '14)*. ACM, 353–365.

[15] Azadeh Farzan, P. Madhusudan, Niloofar Razavi, and Francesco Sorrentino. 2012. Predicting Null-pointer Dereferences in Concurrent Programs. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '12)*. ACM, 1–11.

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

[17] Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. 2008. Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, 293–303.

[18] G. Fraser and A. Gargantini. 2009. Experiments on the Test Case Length in Specification Based Test Case Generation. In *2009 ICSE Workshop on Automation of Software Test (AST '13)*. 18–26.

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

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

[21] Jeff Huang, Peng Liu, and Charles Zhang. 2010. LEAP: Lightweight Deterministic Multi-processor Replay of Concurrent Java Programs. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10)*. ACM, 207–216.

[22] Jeff Huang, Qingzhou Luo, and Grigore Rosu. 2015. GPredict: Generic Predictive Concurrency Analysis. In *Proceedings of the International Conference on Software Engineering (ICSE '15)*. ACM, 847–857.

[23] Jeff Huang, Charles Zhang, and Julian Dolby. 2013. CLAP: Recording Local Executions to Reproduce Concurrency Failures. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, 141–152.

[24] Yanyan Jiang, Tianxiao Gu, Chang Xu, Xiaoxing Ma, and Jian Lu. 2014. CARE: Cache Guided Deterministic Replay for Concurrent Java Programs. In *Proceedings of the International Conference on Software Engineering (ICSE '14)*. ACM, 457–467.

[25] Wei Jin and Alessandro Orso. 2012. BugRedux: Reproducing Field Failures for In-house Debugging. In *Proceedings of the International Conference on Software Engineering (ICSE '12)*. IEEE Computer Society, 474–484.

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

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

[28] Nuno Machado, Brandon Lucia, and Luís Rodrigues. 2015. Concurrency Debugging with Differential Schedule Projections. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, 586–595.

[29] Nuno Machado, Brandon Lucia, and Luís Rodrigues. 2016. Production-guided Concurrency Debugging. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP '16)*. ACM, 29:1–29:12.

[30] Roman Manevich, Manu Sridharan, Stephen Adams, Manuvir Das, and Zhe Yang. 2004. PSE: Explaining Program Failures via Postmortem Static Analysis. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '12)*. ACM, 63–72.

[31] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. 2009. LiteRace: Effective Sampling for Lightweight Data-race Detection. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, 134–143.

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

[33] Robert O'Callahan and Jong-Deok Choi. 2003. Hybrid Dynamic Data Race Detection. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPOPP '03)*. ACM, 167–178.

[34] Peter Ohmann and Ben Liblit. 2013. Lightweight Control-Flow Instrumentation and Postmortem Analysis in Support of Debugging. In *Proceedings of the International Conference on Automated Software Engineering (ASE '13)*. IEEE Computer Society, 378–388.

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

[36] Chang-Seo Park and Koushik Sen. 2008. Randomized Active Atomicity Violation Detection in Concurrent Programs. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '08)*. ACM, 135–145.

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

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

[39] Jeremias Rössler, Andreas Zeller, Gordon Fraser, Cristian Zamfir, and George Candea. 2013. Reconstructing Core Dumps. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST '13)*. IEEE Computer Society, 114–123.

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

[41] Malavika Samak and Murali Krishna Ramanathan. 2015. Synthesizing Tests for Detecting Atomicity Violations. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '15)*. ACM, 131–142.

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

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

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

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

[46] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. 2014. Bug Characteristics in Open Source Software. 19, 6 (Dec. 2014), 1665–1705.

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

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

[49] USI Università della Svizzera italiana. 2017. ConCrash. (2017). http://star.inf.usi.ch/star/software/concrash/

[50] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. 2004. Test Input Generation with Java PathFinder. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '04)*. ACM, 97–107.

[51] Chao Wang, Mahmoud Said, and Aarti Gupta. 2011. Coverage Guided Systematic Concurrency Testing. In *Proceedings of the International Conference on Software Engineering (ICSE '11)*. ACM, 221–230.

[52] Dasarath Weeratunge, Xiangyu Zhang, and Suresh Jagannathan. 2010. Analyzing Multicore Dumps to Facilitate Concurrency Bug Reproduction. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '10)*. ACM, 155–166.

[53] Rongxin Wu. 2014. Diagnose Crashing Faults on Production Software. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '14)*. ACM, 771–774.

[54] Rongxin Wu, Xiao Xiao, Shing-Chi Cheung, Hongyu Zhang, and Charles Zhang. 2016. Casper: An Efficient Approach to Call Trace Collection. In *Proceedings of the Symposium on Principles of Programming Languages (POPL '16)*. ACM, 678–690.

[55] Rongxin Wu, Hongyu Zhang, Shing-Chi Cheung, and Sunghun Kim. 2014. CrashLocator: Locating Crashing Faults Based on Crash Stacks. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '14)*. ACM, 204–214.

[56] Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam. 2012. Maple: A Coverage-driven Testing Tool for Multithreaded Programs. In *Proceedings of the Conference on Object-Oriented Programming Systems and Applications (OOPSLA '12)*. ACM, 485–502.

[57] Cristian Zamfir and George Candea. 2010. Execution Synthesis: A Technique for Automated Software Debugging. In *Proceedings of the ACM SIGOPS EuroSys European Conference on Computer Systems (EuroSys '10)*. ACM, 321–334.

[58] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas Reps. 2011. ConSeq: Detecting Concurrency Bugs Through Sequential Errors. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*. ACM, 251–264.

[59] Wei Zhang, Chong Sun, and Shan Lu. 2010. ConMem: Detecting Severe Concurrency Bugs Through an Effect-oriented Approach. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '10)*. ACM, 179–192.

[60] Jinguo Zhou, Xiao Xiao, and Charles Zhang. 2012. Stride: Search-based Deterministic Replay in Polynomial Time via Bounded Linkage. In *Proceedings of the International Conference on Software Engineering (ICSE '12)*. IEEE Computer Society, 892–902.