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  
mauro.pezzè@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.

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
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.

Figure 1: Faulty class java.util.logging.Logger of JDK 1.4.1
\[
\begin{align*}
1 & \text{java.lang.NullPointerException} \\
2 & \text{at java.util.logging.Logger.log(Logger.java:421)} \\
3 & \text{at java.util.logging.Logger.log(Logger.java:458)} \\
4 & \text{at java.util.logging.Logger.log(Logger.java:482)} \\
5 & \text{at java.util.logging.Logger.info(Logger.java:996)} \\
6 & \text{at test.TestCode1.runTest(TestCode.java:10)} \\
7 & \text{at java.lang.Thread.run(Thread.java:662)}
\end{align*}
\]

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

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
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 execute 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 δ = (m₁, . . . , mₙ) 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 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 myFilter=new MyFilter(); sout.setFilter((Filter) myFilter0)), the interfering method interfacing 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 code sequences that are most likely to expose the concurrency failures. The pruning strategies trim both test code sequences that are redundant with respect to previously generated test code sequences and test code sequences 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 into 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_{s1} in Figure 6 represents the sequence (Logger sout = Logger.getAnonymousLogger(); Filter f = new Filter(); sout.setFilter(f); sout.info("")) obtained traversing the tree from the root to the node (δ_{s1}, δ_{s2}). 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 creating a new child node [47].

...
method (cm) exploring different values for the input parameters. We denote each of the edges resulting from the extensions as $\delta_1$ (line 17). $\text{ConCrash}$ explores all the children of $\delta_p$, obtained by extending $\delta_p$ 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 $\delta_2$ (line 20). Every combination of $\delta_p$, $\delta_1$ and $\delta_2$ corresponds to a candidate concurrent test code. Function $\text{PRUNING}$ analyses each combination of $\delta_p$, $\delta_1$ and $\delta_2$ to determine if it should be pruned or not (line 21). $\text{ConCrash}$ considers all public methods in the CUT to obtain $\delta_2$ (line 18) because the crash stack does not contain information about the interfering method, and thus $\text{ConCrash}$ needs to explore all the possible candidates to identify the right one.

$\text{PRUNING}$ (lines 23-42). Function $\text{PRUNING}$ 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 $\delta_{p,1}$ and $\delta_{p,2}$ be the sequences that extend $\delta_p$ with the edges $\delta_1$ and $\delta_2$, respectively (lines 24 and 25), $\text{ConCrash}$ executes $\delta_{p,1}$ and $\delta_{p,2}$ in isolation (single-threaded execution) (line 26), and collects the following runtime information for each sequence $\delta \in \{\delta_{p,1}, \delta_{p,2}\}$:

(i) whether $\delta$ throws an uncaught exception, (ii) the sequential coverage of the last method call in $\delta$ [47], and (iii) the state of the object SOUT after executing $\delta$, 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 interleadings with respect to the previously generated test codes. $\text{ConCrash}$ exploits sequential coverage to identify and avoid both redundant and irrelevant test codes. Let the trace $E = (e_1, \ldots, e_k)$ of a call sequence $\delta$ be the ordered sequence of events exhibited by a sequential (single-threaded) execution of $\delta$. An event can be one of the following:

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

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

Since all the test codes generated by $\text{ConCrash}$ 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 $\mathcal{M}(\delta)$.

Pruning strategies (lines 27-42). $\text{ConCrash}$ prunes the combination $(\delta_p, \delta_1, \delta_2)$ according to different strategies. If the code is neither redundant (line 29) nor irrelevant (lines 27, 28, 30, 31) $\text{ConCrash}$ updates the coverage repository $C$ (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), $\text{ConCrash}$ outputs $t$ and its failure-inducing interleaving and terminates (line 36).
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 $\delta_s$ produced by executing $\delta_{p,s2}$ either throws an exception or has been already explored. If not, ConCrash adds $S(\delta_s)$ to $\mathcal{S}$ 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].

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 $(\delta_p, \delta_1, \delta_2)$ (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:** $\exists e \in \mathcal{M}(\delta_{p,s1}) : \text{stack}(e) = \text{CST}$ (Crash Stack Trace), where $\text{stack}(e)$ is the call stack trace of $e$, obtained by analysing the method entry and exit points in $\mathcal{M}(\delta_{p,s1})$. 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 $\delta_1$ 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 $\delta_1 = \delta_B$ since the sequential execution of $\delta_B$ does not reach the POF (line 421).

**PS-Redundant:** $(\mathcal{M}(\delta_{p,s1}), \mathcal{M}(\delta_{p,s2})) \in \mathcal{C}$. ConCrash prunes the combinations whose concurrent suffixes induce an already observed pair of sequential coverages $\mathcal{M}(\delta_{p,s1})$ and $\mathcal{M}(\delta_{p,s2})$, as inferred from the coverage repository $\mathcal{C}$. 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-Interference:** $\exists e_1, e_2 \in \mathcal{M}(\delta_{p,s1}) \times \mathcal{M}(\delta_{p,s2}) : e_1 = W(t), e_2 = W(f)$. 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

\[ \delta_p \delta_1 \delta_2 \text{ Is the test generated?} \]

\begin{itemize}
  \item $\delta_p \delta_1 \delta_2$ No, $\delta_p$ throws exception
  \item $\delta_p \delta_1 \delta_2$ No, pruned by PS-Interference
  \item $\delta_p \delta_1 \delta_2$ Yes, but not failure inducing
  \item $\delta_p \delta_1 \delta_2$ No, pruned by PS-Redundant
  \item $\delta_p \delta_1 \delta_2$ Yes, failure-inducing
\end{itemize}

Figure 7: Example of Lockset History (LH)
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>
<thead>
<tr>
<th>Subjects</th>
<th>Concurrency Failures</th>
<th>Crash Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>Fault</td>
<td>Type of Exception</td>
</tr>
</tbody>
</table>
| 1 PerUserPoolDataSource 1.4 Commons DBCP 9,451 719 68 369 Race ConcurrentModificationException 4
| 2 SharedPoolDataSource 1.4 Commons DBCP 9,451 546 44 369 Race ConcurrentModificationException 4
| 3 IntRange 2.4 Commons Math 18,016 278 44 481 Atom. AssertionException 1
| 4 BufferInputStream 1.1 Java JDK 3,991 304 42 4225348 Atom. NullPointerException 2
| 5 Logger 1.4.1 Java JDK 2,193 528 45 4779253 Atom. NullPointException 4
| 6 PushbackReader 1.8 Java 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 Log4j 51,614 200 28 187 Race ConcurrentModificationException 4
| 9 Category 1.4 Log4j 10,773 387 43 1907 Atom. NullPointerException 1
| 10 FileAppender 1.2 Log4j 10,273 185 13 509 Atom. NullPointException 2

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
### 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 FTS, Failure Inducing Test Size, 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 FID, Failure-inducing Test ID, 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
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 strategies 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 are written by only few 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 achieve the highest speedup for every subject. 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 FTIS) 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...
(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. Weeratunga 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


