Exposing Concurrency Failures

A comprehensive survey of the state of the art and a novel approach to reproduce field failures

Doctoral Dissertation submitted to the Faculty of Informatics of the Università della Svizzera Italiana in partial fulfillment of the requirements for the degree of Doctor of Philosophy

presented by
Francesco Adalberto Bianchi

under the supervision of
Mauro Pezzè

October 2018
I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Francesco Adalberto Bianchi
Lugano, 26 October 2018
To my advisor, for he gave me endless inspiration and guidance,
To my family, for their unconditional love and support,
To my colleagues and train-mates, for they made my PhD an unforgettable journey,
To my beloved, for you make me want to be a better man.
Abstract

With the rapid advance of multi-core and distributed architectures, concurrent systems are becoming more and more popular. Concurrent systems are extremely hard to develop and validate, as their overall behavior depends on the non-deterministic interleaving of the execution flows that comprise the system. Wrong and unexpected interleavings may lead to concurrency faults that are extremely hard to avoid, detect, and fix due to their non-deterministic nature.

This thesis addresses the problem of exposing concurrency failures. Exposing concurrency failures is a crucial activity to locate and fix the related fault and amounts to determine both a test case and an interleaving that trigger the failure. Given the high cost of manually identifying a failure-inducing test case and interleaving among the infinite number of inputs and interleavings of the system, the problem of automatically exposing concurrency failures has been studied by researchers since the late seventies and is still a hot research topic.

This thesis advances the research in exposing concurrency failures by proposing two main contributions. The first contribution is a comprehensive survey and taxonomy of the state-of-the-art techniques for exposing concurrency failures. The taxonomy and survey provide a framework that captures the key features of the existing techniques, identify a set of classification criteria to review and compare them, and highlight their strengths and weaknesses, leading to a thorough assessment of the field and paving the road for future progresses.

The second contribution of this thesis is a technique to automatically expose and reproduce concurrency field failure. One of the main findings of our survey is that automatically reproducing concurrency field failures is still an open problem, as the few techniques that have been proposed rely on information that may be hard to collect, and identify failure-inducing interleavings but do not synthesize failure-inducing test cases. We propose a technique that advances over state-of-the-art approaches by relying on information that is easily obtainable and by automatically identifying both a failure-inducing test case and interleaving. We empirically demonstrate the effectiveness of our approach on a benchmark of real concurrency failures taken from different popular code bases.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Contents</strong></td>
<td>v</td>
</tr>
<tr>
<td><strong>List of Figures</strong></td>
<td>xi</td>
</tr>
<tr>
<td><strong>List of Tables</strong></td>
<td>xiii</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Contributions</td>
<td>4</td>
</tr>
<tr>
<td>1.2 Structure of the Dissertation</td>
<td>4</td>
</tr>
<tr>
<td><strong>2 Concurrency</strong></td>
<td>7</td>
</tr>
<tr>
<td>2.1 Concurrent Systems</td>
<td>7</td>
</tr>
<tr>
<td>2.2 Interleaving of Execution Flows</td>
<td>9</td>
</tr>
<tr>
<td>2.3 Concurrency Faults</td>
<td>11</td>
</tr>
<tr>
<td>2.3.1 Data Races</td>
<td>11</td>
</tr>
<tr>
<td>2.3.2 Atomicity violations</td>
<td>12</td>
</tr>
<tr>
<td>2.3.3 Deadlocks</td>
<td>13</td>
</tr>
<tr>
<td>2.3.4 Order violation</td>
<td>14</td>
</tr>
<tr>
<td><strong>3 A Classification Schema for Techniques for Exposing Concurrency Failures</strong></td>
<td>17</td>
</tr>
<tr>
<td>3.1 Exposing Concurrency Failures</td>
<td>17</td>
</tr>
<tr>
<td>3.2 Towards a Classification Schema</td>
<td>20</td>
</tr>
<tr>
<td>3.2.1 Input</td>
<td>22</td>
</tr>
<tr>
<td>3.2.2 Selection of Interleavings</td>
<td>22</td>
</tr>
<tr>
<td>3.2.3 Property of Interleavings</td>
<td>24</td>
</tr>
<tr>
<td>3.2.4 Output and Oracle</td>
<td>24</td>
</tr>
<tr>
<td>3.2.5 Guarantees</td>
<td>25</td>
</tr>
<tr>
<td>3.2.6 Target System</td>
<td>26</td>
</tr>
<tr>
<td>3.2.7 Technique</td>
<td>26</td>
</tr>
</tbody>
</table>
4 A Taxonomy of Techniques for Exposing Concurrency Failures

4.1 Property Based Techniques

4.1.1 Data Race

4.1.2 Atomicity Violation

4.1.3 Deadlock

4.1.4 Combined

4.1.5 Order Violation

4.2 Space Exploration Techniques

4.2.1 Stress Testing

4.2.2 Exhaustive exploration

4.2.3 Coverage criteria

4.2.4 Heuristics

4.3 Reproduction Techniques

4.3.1 Record-and-replay

4.3.2 Post-processing

5 Reproducing Concurrency Failures From Crash Stack Traces

5.1 Overview

5.2 The CONCRASH Approach

5.3 Test Case Generator

5.3.1 Modeling the Test Cases Search Space

5.3.2 Test Case Minimization

5.3.3 Exploring the Test Cases Search Space

5.3.4 Pruning Strategies

5.4 Interleaving Explorer

5.4.1 Symbolic Trace Collection

5.4.2 Computing Failing Interleavings with Constraint Solving

6 Evaluation

6.1 Research Questions

6.2 Experimental Setting

6.3 Experimental Results

6.3.1 RQ1 - Effectiveness

6.3.2 RQ2 - Pruning Strategies

6.3.3 RQ3 - Comparison with Testing Approaches

6.4 Limitations

6.5 Threats to Validity
7 Conclusion

7.1 Contributions ......................................................... 84
7.2 Open Research Directions ........................................... 85

Bibliography ................................................................. 87
## Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Exposing concurrency failures: how and when.</td>
<td>2</td>
</tr>
<tr>
<td>3.1</td>
<td>A general framework for exposing concurrency failures</td>
<td>18</td>
</tr>
<tr>
<td>3.2</td>
<td>Classification criteria for the techniques to expose concurrency failures</td>
<td>21</td>
</tr>
<tr>
<td>3.3</td>
<td>Classification schema</td>
<td>28</td>
</tr>
<tr>
<td>5.1</td>
<td>Faulty class <code>java.util.logging.Logger</code> of JDK 1.4.1</td>
<td>48</td>
</tr>
<tr>
<td>5.2</td>
<td>Crash stack of class <code>Logger</code> (Bug ID 4779253)</td>
<td>49</td>
</tr>
<tr>
<td>5.3</td>
<td>A concurrent test case that reproduces the crash stack in Figure 5.2</td>
<td>50</td>
</tr>
<tr>
<td>5.4</td>
<td>Conceptual Architecture of CONCRASH</td>
<td>52</td>
</tr>
<tr>
<td>5.5</td>
<td>The CONCRASH algorithm</td>
<td>54</td>
</tr>
<tr>
<td>5.6</td>
<td>A Tree model of class <code>Logger</code></td>
<td>56</td>
</tr>
<tr>
<td>5.7</td>
<td>Definition of Sequential coverage</td>
<td>58</td>
</tr>
<tr>
<td>5.8</td>
<td>Method <code>RemoveHandler</code> of class <code>Logger</code></td>
<td>60</td>
</tr>
<tr>
<td>5.9</td>
<td>Example of lockset history (LH)</td>
<td>62</td>
</tr>
<tr>
<td>6.1</td>
<td>Aggregate comparison of the CONCRASH pruning strategies for the ten subjects</td>
<td>78</td>
</tr>
</tbody>
</table>
### Tables

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Data race detection techniques</td>
<td>31</td>
</tr>
<tr>
<td>4.2</td>
<td>Atomicity violation detection techniques</td>
<td>36</td>
</tr>
<tr>
<td>4.3</td>
<td>Deadlock detection techniques</td>
<td>39</td>
</tr>
<tr>
<td>4.4</td>
<td>Techniques for detecting combined properties violations</td>
<td>39</td>
</tr>
<tr>
<td>4.5</td>
<td>Techniques for detecting order violations</td>
<td>40</td>
</tr>
<tr>
<td>4.6</td>
<td>Techniques for exploring the space of interleavings</td>
<td>42</td>
</tr>
<tr>
<td>4.7</td>
<td>Techniques for reproducing concurrency failures</td>
<td>43</td>
</tr>
<tr>
<td>6.1</td>
<td>CONCRASH Evaluation Subjects</td>
<td>72</td>
</tr>
<tr>
<td>6.2</td>
<td>CONCRASH Evaluation Failures</td>
<td>72</td>
</tr>
<tr>
<td>6.3</td>
<td>RQ1: CONCRASH Effectiveness Results</td>
<td>74</td>
</tr>
<tr>
<td>6.4</td>
<td>RQ2: CONCRASH Pruning Strategies Effectiveness Results</td>
<td>76</td>
</tr>
<tr>
<td>6.5</td>
<td>Comparison with test case generators</td>
<td>79</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Concurrent systems are ubiquitous as multi-core and distributed architectures are the norm: interactive applications exploit the multi-threaded paradigm to decouple the input-output processing from the back-end computation; mobile applications often interact with remote services; Web applications adopt the client-server communication; peer-to-peer applications coordinate a multitude of computing nodes; scientific applications often exploit multiple threads to enable parallel computations on multi-core architectures.

Concurrent systems introduce new challenges in the validation and verification process (V&V). The overall behavior of concurrent systems depends not only on the behavior of the single threads, but also on how they interleave during the execution. As a result, the same input can produce different results depending on the interleaving of threads in the execution. Wrong and unexpected interleavings may lead to concurrency faults that are extremely hard to avoid and detect due to their non-deterministic nature. Fixing concurrency faults has become a key issue since their manifestation as system failure can have severe consequences in daily life. In recent history, concurrency faults have caused several disasters such as the Therac-25 accident [76], the blackout in northeastern America in 2003 [117] and losses for investors for 500 million dollars during Facebook IPO in 2012 [31]. The JaConTeBe benchmark deeply analyses a large set of concurrency faults that led to failures in recent and widely used software systems [1].

This thesis addresses the problem of exposing concurrency functional failures, that is, failures that manifest as unexpected system behavior and lead to incorrect results. Exposing a failure is essential to locate and fix the related fault, and amounts to determine the program conditions that cause the system to fail. In the case of sequential programs, such conditions correspond to a test case, that

\[ \text{http://stap.sjtu.edu.cn/index.php?title=JaConTeBe} \]
is, an input sequence and an expected output (oracle), that triggers the failure. To expose concurrency failures, determining a failure-inducing test case is not sufficient as the behavior of a concurrent system does not depend only on the system input, but also on the threads’ interleaving. As a result, a test case can manifest different system behaviors, and only some - usually few - of them can expose a failure. This means that exposing a concurrency failure involves two activities: generating test case and selecting interleavings for determining a failure-inducing test case and a failure-inducing interleaving, respectively. Test cases and interleavings are generated and selected both before and after delivery, at development and operative time. Test cases and interleaving are generated before the software system has been released during in-house testing to check whether the system under test manifests concurrency failures, i.e., contains concurrency faults, and after the software system has been released during failure reproduction, to help developers diagnose failures that escape in-house testing and sneak in production. Figure 1.1 summarizes the activities needed to expose a concurrency failure and the different time at which they can be performed.

Generating test cases and selecting interleavings are time consuming and expensive activities since developers need to identify the failure-inducing test case and interleaving among the infinite number of possible inputs and interleavings of the system [79]. Automation is a promising research direction to alleviate the developers’ burden of manually performing such activities. Indeed, the problem of automatically exposing concurrency failures has drawn the attention of the research community since the late seventies, and has considerably grown in the last decade. The many testing techniques for exposing concurrency failures has mainly focused on the problem of interleavings selection. They either adopt a systematic approach to explore the entire interleavings space [74][120], or select
and execute those interleavings that manifest specific suspicious patterns, such as data races, atomicity violations, and deadlocks, that increase the probability of revealing concurrency failures [21, 39, 41, 43, 87, 94]. The main techniques for reproducing concurrency failures exploit information collected when a failure occurs in the field. Current reproduction techniques rely on either execution traces [2, 57, 61] or memory core dumps [142, 154].

Despite the large body of research that addresses the problem of exposing concurrency failures, to the best of our knowledge a precise survey and classification of the progresses and the results in the field is still missing. The current lack of a comprehensive classification, analysis and comparison of these techniques limits the understanding of the strengths and weaknesses of each approach and hampers the future advancements in the field. In this thesis, we conduct the first comprehensive survey of the state-of-the-art techniques for exposing concurrency failures. We study the recent literature by systematically browsing the main publishers and scientific search engines, and we trace back the results to the seminal work of the last forty years. We present a general framework that captures the different aspects of the problem of exposing concurrency failures and that we use to identify a set of classification criteria that drive the survey of the different approaches. The survey classifies and compares the state-of-the-art techniques, discusses their advantages and limitations, and indicates open problems and possible research directions in the area of exposing concurrency failures.

One of the main findings of our study is that the problem of exposing concurrency failures has been widely investigated in the context of in-house testing, but little effort has been devoted to the problem of reproducing field concurrency failures.

The few techniques for reproducing concurrency failures that have been proposed so far rely on either execution traces or memory core dumps, which may be expensive and hard to obtain in many practical situations: recording execution traces might introduce a high overhead which may be acceptable in testing but not in production environment [144], memory core dumps are not available on all platforms [17], and they both often contain sensitive information, which introduces privacy concerns [145].

Moreover, current techniques for reproducing concurrency failures focus on identifying failure-inducing interleavings only, leaving largely open the problem of synthesizing the test case that manifests such interleavings. As a result, how to effectively reproduce concurrency failures is still an open problem.

In this thesis, we present CONCRASH, a novel approach that overcomes the above-mentioned limitations. CONCRASH automatically generates test cases that
reproduce concurrency failures from the limited information available in the crash stacks. Differently from execution traces and memory core dumps that are expensive to produce and hard to obtain, crash stack traces are easily obtainable and do not suffer from performance and privacy issues. However, crash stack traces contain only partial and limited information about the failure, thus challenging the problem of automatically reproducing concurrency failures. We demonstrate through an exhaustive empirical validation that ConCrash is both effective and efficient in reproducing concurrency failures. In particular, we show that the technique correctly identifies the failure-inducing test case and interleaving by exploiting the limited information contained in crash stack traces only.

1.1 Contributions

This thesis makes two major contributions:

A survey and a taxonomy of techniques for exposing concurrency failures:
The first contribution of this thesis is a comprehensive survey on the state-of-the-art techniques for exposing concurrency failures. The survey provides a framework to capture the key features of the existing techniques, identifies a set of classification criteria to review and compare them, discusses in details their strengths and weaknesses, and indicates open problems and possible research directions in the area of exposing concurrency failures.

A technique to reproduce concurrency failures from crash stack traces: The second contribution of this thesis is a technique to automatically reproduce concurrency failures. In particular, we define an approach which is able to synthesize both a failure-inducing test case and related interleaving from the limited information contained in the crash stack trace of the failure. We propose a general approach, and a concrete implementation for Java applications. We then evaluate the effectiveness of the technique on a set of case studies and show that the technique is indeed effective in reproducing concurrency failures.

1.2 Structure of the Dissertation

The remainder of this dissertation is structured as follows:
• Chapter 2 introduces the basic concepts, definitions and background of concurrent systems.

• Chapter 3 proposes a classification schema and taxonomy for techniques to expose concurrency failures.

• Chapter 4 surveys, classifies, and compares the state-of-the-art techniques for exposing concurrency failures.

• Chapter 5 defines a technique to automatically reproduce concurrency failures from crash stack traces.

• Chapter 6 presents the results of the empirical evaluation that we performed to show the effectiveness and efficiency of ConCrash.

• Chapter 7 summarizes the contributions of this dissertation, and discusses future directions.
1.2 Structure of the Dissertation
Chapter 2

Concurrency

Many modern software systems are composed of multiple execution flows that run simultaneously, spanning from applications designed to exploit the power of modern multi-core architectures to distributed systems consisting of multiple components deployed on different physical nodes. We collectively refer to such systems as concurrent systems.

This chapter introduces the fundamental concepts, definitions and background of concurrent systems. Section 2.1 introduces concurrent systems and their main communication models. Section 2.2 presents the fundamental concept of interleaving of execution flows and the concurrent synchronization mechanisms. Section 2.3 describes the main classes of concurrency faults, namely data races, atomicity violations, deadlocks, and order violations.

2.1 Concurrent Systems

In this thesis, we follow the definition of concurrent system proposed by Andrews and Schneider in their popular survey and book [3, 4] that represent classic references and accommodate the wide range of heterogeneous techniques and tools presented in the literature.

A system is concurrent if it includes a number of execution flows\(^1\) that can progress simultaneously, and that interact with each other. This definition encompasses both flows that execute in overlapping time frames, like concurrent programs executed on multi-core, multi-processor parallel and multi-node distributed architectures, and flows that execute only in non-overlapping frames,

---

\(^1\)Although we adopt the terminology of Andrews and Schneider, we prefer the term execution flow over process to avoid biases towards a specific technology.
like concurrent programs executed on single-core architectures. Depending on
the specific architecture and programming paradigm, execution flows can be
concretely implemented as processes in distributed architectures, or threads in
single-core and multi-core architectures, as common in modern programming
languages such as C++, Java, C# and Erlang.

We distinguish two classes of concurrent systems based on the mechanism
they adopt to enable the interaction between execution flows, shared memory
and message passing systems. In shared memory systems, execution flows inter-
act by accessing a common memory. In message passing systems, execution flows
interact by exchanging messages. Message passing can be used either by execu-
tion flows hosted on the same physical node or on different physical nodes (dis-
tributed systems). Conversely, shared memory mechanisms are usually adopted
by execution flows located on the same node (as in multi-threaded systems).

We model a shared memory as a repository of one or more data items. A data
item has an associated value and type. The type of a data item determines the set
of values it is allowed to assume. We model the interaction of an execution flow
f with the repository using two primitive operations: write operations \( w_x(v) \),
meaning that \( f \) updates the value of the data item \( x \) to \( v \), and read operations
\( r_x(v) \), meaning that \( f \) reads the value \( v \) of \( x \). Operations are composed of one or
more instructions. Instructions are atomic, meaning that their execution cannot
be interleaved with other instructions, while operations are in general not atomic.
This model captures both operations on simple data, like primitive variables in
C, and operations on complex data structures like Java objects, where types are
classes, data items are objects and operations are methods that can operate only
on some of the fields of the objects.

We model message passing systems using two primitive operations: send op-
erations \( s_f(m) \) that send a message \( m \) to the execution flow \( f \), and receive op-
erations \( r_f(m) \) that receive a message \( m \) from the execution flow \( f \). Message
passing can be either synchronous or asynchronous. An execution flow \( f \) that
sends a synchronous message \( s_f(m) \) to an execution flow \( f' \) must wait for \( f' \) to
receive the message \( m \) before continuing, while an execution flow \( f \) that sends an
asynchronous message \( s_f(m) \) to an execution flow \( f' \) can progress immediately
without waiting for \( m \) to be received by \( f' \).

The message passing paradigm can be mapped to the shared memory paradigm
by modeling a send primitive as a write operation on a shared queue and a re-
ceive primitive as a read operation on the same shared queue. Thus, without loss
of generality, we refer to shared memory systems in most of the definitions and
examples presented in this thesis.
2.2 Interleaving of Execution Flows

The behavior of a concurrent system depends both on the input parameters and the sequences of instructions of the individual flows, and on the interleaving of instructions from the execution flows that comprise the system.

We introduce the main concepts of concurrency under the assumption of a sequentially consistent model \[73\]. This model guarantees that all the execution flows in a concurrent system observe the same order of instructions, and that this order preserves the order of instructions defined in the individual execution flows. We discuss the implications of relaxing this assumption at the end of this section.

Under the assumption of sequential consistency, we can model the interleaving of instructions of multiple execution flows in a concrete program execution with a history, which is an ordered sequence of instructions of the different execution flows. In a shared memory system, histories include sequences of invocations of read and write operations on data items. Since in general the operations on shared data items are not atomic, we model the invocation and the termination of an operation \(op\) as two distinct instructions. The execution of an operation \(o'\) overlaps the execution of another operation \(o\) if the invocation of \(o'\) occurs between the invocation and the termination of \(o\). In a message passing system, histories include sequence of atomic send and receive operations.

Given a concrete execution \(ex\) of a concurrent system \(S\), a history \(H_{ex}\) is a sequence of instructions that (i) contains the union of all and only the instructions of the individual execution flows that comprise \(ex\), and (ii) preserves the order of the individual execution flows: for all instructions \(o_i\) and \(o_j\) that occur in \(ex\) and belong to the same execution flow \(f\), if \(o_i\) occurs before \(o_j\) in \(f\), then \(o_i\) occurs before \(o_j\) in \(H_{ex}\).

We use the term interleaving of an execution \(ex\) to indicate the order of instructions defined in the history \(H_{ex}\).

Listing 2.1. A non-deterministic concurrent system

```plaintext
1  begin f_1
2    if (x==1) {
3      print OK
4    }
5  end f_1

1  begin f_2
2    x=1
3  end f_2
```

We refer to listing 2.1 to exemplify the impact of the interleaving of instruc-
tions from multiple execution flows on the result of an execution. In Listing 2.1 both execution flows $f_1$ and $f_2$ access a shared data item $x$ with initial value 0. $f_2$ writes 1 to $x$, while $f_1$ prints OK if it reads 1 for $x$. Multiple interleavings are possible. If the write operation $x = 1$ of $f_2$ occurs before the read operation $x == 1$ of $f_1$ in the history, then $f_1$ reads 1 and prints OK, otherwise $f_1$ reads 0 and does not print anything.

Systems that do not assume sequential consistency refer to relaxed memory models. Examples of systems that refer to relaxed models are shared memory systems where different execution flows may observe different orders of operations due to a lazy synchronization between the caches of multiple cores in a multi-core architecture. Several popular programming languages refer to relaxed models by allowing the compiler to re-order the operations within a single execution flow to improve the performance. For example, this is allowed in the memory models of both Java \[86, 108\] and C++ \[15\].

Concurrent programming languages offer various synchronization mechanisms that constrain the order of instructions to prevent erroneous behaviors, thus limiting the space of possible interleavings. The synchronization mechanisms depend on the concurrency paradigm, the granularity of the synchronization structures and the constraints imposed on the system architecture.

The Java programming language offers synchronized blocks and atomic instructions to ensure that program blocks are executed without the interleaving of instructions of other execution flows \[55\]. Other programming languages like C and C++ offer locks, mutexes and semaphores to constrain the concurrent execution of code regions. Yet other concurrent programming environments, like Posix threads and OpenMP, offer barrier synchronization to constrain the access to code regions executed concurrently by multiple execution flows: barriers and phasers introduce program points that all the execution flows in a group must reach before any of them is allowed to proceed \[125\].

In the context of message passing, programming languages and libraries that implement the actor-based paradigm ensure that individual messages are processed atomically and in isolation \[5\]. Synchronous message passing ensures that the sender of a message can progress only after the message has been successfully delivered to the recipient \[146\].

\footnote{In this example, we assume that read and write of $x$ are atomic. Relaxing the atomicity assumption would produce even more interleavings.
2.3 Concurrency Faults

In this thesis we focus on the problem of exposing concurrency failures, which are failures caused by unexpected interleavings of instructions of otherwise correct execution flows. Concurrency failures can be extremely hard to reveal and reproduce, since they manifest only in the presence of specific interleavings that may be rarely executed. In this section we introduce the main types of concurrency faults, namely data races, atomicity violations, deadlocks, and order violations.

2.3.1 Data Races

A data race occurs when two operations from different execution flows access the same data item \( d \) concurrently, at least one is a write operation, and no synchronization mechanism is used to control the (order of) accesses to \( d \). A system is data race free if no data races can occur during its execution.

Listing 2.2. An example of data race

<table>
<thead>
<tr>
<th>1</th>
<th>begin f_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>x++</td>
</tr>
<tr>
<td>3</td>
<td>end f_1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>begin f_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>x++</td>
</tr>
<tr>
<td>3</td>
<td>end f_2</td>
</tr>
</tbody>
</table>

We refer to listing 2.2 to exemplify data races. Both execution flows \( f_1 \) and \( f_2 \) increment the value of the integer variable \( x \) by one. If the interpreter splits the increment of \( x \) into two low-level operations that read the value of \( x \) and update the value of \( x \), respectively, when \( f_1 \) and \( f_2 \) are executed concurrently, the two operations can increment \( x \) of 1 only, and not 2, as expected depending on the order of the elementary atomic operations.

Data races are violations of atomicity assumptions on the execution of individual operations. Such violations break the serializability of the system behavior. The concept of serializability was originally defined in the context of database systems as a guarantee for the correctness of transactions [96]. In our context a history is serializable if it is equivalent to a serial history, which is a history in which all the atomicity assumptions are satisfied. Two histories are equivalent if they produce the same values for all the data items.

A data race implies an uncontrolled access to a data item, which may or may

\[\text{In literature such property is referred to as view serializability [51]}\]
not be an error. In the example of Listing 2.2, the value \( x = 'AAAABBBB' \) may be valid or not depending on the application logic.

Data races can be classified as low level and high level data races [139]. Low level data races involve accesses to individual memory locations. High level data races involve concurrent operations on shared complex data structures, like public methods of an object or a library in an object oriented program.

Data races can occur also in programs that implement the message passing paradigm, when the code fragments that process two messages access a common data item, and at least one fragment modifies that data item.

### 2.3.2 Atomicity violations

*Atomicity violations* extend the concept of atomicity to sequences of operations. An atomicity violation occurs when a sequence of operations of an execution flow that is assumed to be executed atomically is interleaved with conflicting operations from other flows [6].

Listing 2.3. An example of atomicity violation

```
begin f_1
  if (x>0)
    x = x-1
  end f_1

begin f_2
  x = 0
end f_2
```

We refer to Listing 2.3 as an example of atomicity violation. In the example, we assume that \( x \) initially holds a non-negative value, and we expect its value to always remain non-negative. The property is satisfied under the assumption that \( f_1 \) executes atomically, that is, the sequence of operations in \( f_1 \) is executed atomically without interleaved operations of \( f_2 \). However, if the atomicity of \( f_1 \) is not properly enforced through synchronization mechanisms, the operation \( x=0 \) in \( f_2 \) can occur between the two operations of \( f_1 \), causing the value of \( x \) to become negative (-1).

We distinguish two main classes of atomicity violations, namely *code centric* and *data centric* atomicity violations. Code centric atomicity violations involve code blocks that should be executed atomically according to some specification of the system. Data centric atomicity violations were first studied in 2006 by Vaziri et al. who introduced the concept of *atomic-set serializability* as a programming abstraction to ensure data consistency [136]. Atomic-set serializability builds on the concepts of *atomic sets* that represent sets of data items that are correlated by
some consistency constraints, and units of work that are the blocks of code used to update variables in an atomic set. In the atomic-set programming model, developers only need to specify atomic sets and units of work, and the compiler infers synchronization mechanisms to avoid potentially dangerous interleavings.

Atomicity violations can occur also in programs that implement the message passing paradigm, when the code fragments that process two or more messages that shall be executed atomically are interleaved by the processing of some other messages.

Atomicity violations can include one or more data races. Listing 2.3 is an example of atomicity violation that includes two data races, since all the operations performed by both $f_1$ and $f_2$ do no use any synchronization mechanisms. Listing 2.4 shows an example of atomicity violation that does not include any data race since each operation is protected by a lock, but the two operations performed by $f_1$ are not executed atomically thus leading to an atomicity violation.

Listing 2.4. An example of atomicity violation without data races

```
1 | begin f_1
2 | acquire(L1)
3 | temp = x
4 | release(L1)
5 | if (temp > 0)
6 | acquire(L1)
7 | x = x - 1
8 | release(L1)
9 | end f_1
```

2.3.3 Deadlocks

Deadlocks occur when the synchronization mechanisms indefinitely prevent some execution flows from continuing their execution. This happens in the presence of circular waits, where each execution flow of a given set of flows is waiting for another execution flow from the same set to progress, and thus cannot continue its own execution.
2.3 Concurrency Faults

Listing 2.5. An example of deadlock

We refer to Listing 2.5 as a simple example of deadlock due to an incorrect use of locks. Locks provide two atomic primitives, acquire(L) and release(L). When an execution flow f acquires a lock L (acquire(L)), no other execution flow f’ can acquire L until f releases the lock (release(L)). Locks are used to implement mutual exclusion: for instance, to guarantee the atomicity of a set O of operations, each execution flow shall acquire a lock L before accessing an operation in O, and release the lock L upon terminating the access to the resource to prevent other flows to execute an operation in O concurrently.

In the example of Listing 2.5, the execution flow f_1 acquires first lock L1 and then lock L2 before releasing L1, while the execution flow f_2 acquires lock L2 before releasing L1. If f_1 acquires L1 and f_2 acquires L2 before a progress of f_1, then f_1 is blocked waiting for f_2 to release L2 and f_2 is blocked waiting for f_1 to release L1, thus resulting in a deadlock.

Deadlocks can be classified in resource deadlocks and communication deadlocks \[126\]. Resource deadlocks occur when a set of execution flows try to access some common resources and each execution flow in the set requests a resource held by another execution flow in the set. The code in Listing 2.5 is an example of resource deadlock. Communication deadlocks occur in message passing systems when a set of execution flows exchange messages and each of them waits for a message from another execution flow in the same set.

Deadlocks do not relate to any other concurrency fault (data races, atomicity violations, order violations) as they do not involve memory access events.

2.3.4 Order violation

Data races, atomicity violations and deadlocks are classic and well studied classes of concurrency fault. Another common but less studied type of concurrency faults is known as order violation. An order violation occurs when the
desired order between two (groups of) memory accesses is flipped [81].

Listing 2.6. An example of order violation

1: begin f_1
2: acquire(L1)
3: pool.push(obj)
4: release(L1)
5: end f_1

1: begin f_2
2: acquire(L1)
3: pool = null
4: release(L1)
5: end f_2

We refer to Listing 2.6 as a simple example of order violation. Execution flow f_1 accesses item pool by assuming that it has been previously initialized, while execution flow f_2 sets the value of the item to null. The execution of the two execution flows may result in a failure depending on the order of execution of f_1 and f_2. In particular, the interleaving in which f_2 is executed before f_1 results in a system failure with a NULL-pointer dereference. The readers should notice that the failure occurs even if the two execution flows properly uses synchronization mechanisms to access the shared data item.

Order violations can occur also in programs that implement the message passing paradigm, when the desired order of delivery of two messages is flipped.

Order violations represent the most general class of concurrency faults. Order violations include both data races and atomicity violations, as data races and atomicity violations lead to unexpected and wrong behaviors due to an undesired order between two (groups of) memory accesses. Nevertheless, many order violations are neither data races nor atomicity violations, like the example in Listing 2.6.
Chapter 3

A Classification Schema for Techniques for Exposing Concurrency Failures

The research on exposing concurrency failures has emerged overbearing in the last fifteen years fostered by the rapid spread of multi-core technologies, distributed, Web and mobile architectures and novel concurrent paradigms. Despite the remarkable and increasing interest of the research community, to the best of our knowledge a precise classification and analysis of the many techniques to expose concurrency failures is still missing. In this chapter, we introduce a classification schema for techniques to expose concurrency failures that we use to survey the main techniques developed so far. Our classification schema is inspired by a systematic review of the literature that we conducted by browsing the main publishers and scientific search engines.

The remainder of this chapter is organized as follows: Section 3.1 presents a general framework that captures the key features of the available techniques to expose concurrency failures. Section 3.2 introduces a set of classification criteria we identified. Section 3.3 describes the classification schema that we use in Chapter 4 to survey the state-of-the-art techniques for exposing concurrency.

3.1 Exposing Concurrency Failures

In this thesis we focus on the problem of exposing concurrency failures. Concurrency failures can be extremely hard to reveal and reproduce, since they manifest only in the presence of specific interleavings that may be rarely executed. Exposing concurrency failures amounts to sample not only a potentially infinite input space, but also the space of possible interleavings, which can grow exponentially with the number of execution flows and the number of instructions that
Figure 3.1. A general framework for exposing concurrency failures

comprise the flows.

The many approaches for exposing concurrency failures that have been proposed so far address different aspects of the problem. Our detailed analysis of the literature led to a simple conceptual framework that captures the different aspects of the problem and relates the many approaches for exposing concurrency failures. Figure 3.1 presents the conceptual framework that we define to provide a comprehensive view of the problem and to organize our survey.

Approaches for exposing concurrency failures deal with specific types of target systems and address one or more of the three main aspects of the problem visualized with rectangles in Figure 3.1: generating test cases, selecting interleavings and comparing the results with oracles. Generating test cases amounts to sample the program input space and produce a finite set of test cases to exercise the target system. Selecting interleavings amounts to augment the test cases with different interleavings of the execution flows to exercise the operations that process the same input data in different order. Comparing the results with oracles amounts to check the behavior of the target system with respect to some oracles. The approaches that we found in the literature focus on either generating test cases or selecting interleavings, sometimes dealing with comparing with oracles as well.

Figure 3.1 presents a conceptual framework for the techniques, without prescribing a specific process. Some approaches may first generate a set of test cases and a set of relevant interleavings and then compare the execution results with oracles, while other approaches may alternate the selection of interleavings and the comparison with oracle by executing each interleaving as soon as identified.
Approaches for generating test cases sample the input space to produce a finite set of test cases by considering the target system. They optionally also consider a target property of interleaving, a system model that provides additional information about the target system, or failure information that has been collected when a failure occurred in the field, for instance execution traces or memory core dumps.

Approaches for selecting interleavings identify a subset of relevant interleavings to be executed, and target (i) the interleaving space as a whole, (ii) some specific properties of interleaving, or (iii) some specific failure to reproduce.

Techniques that target the interleaving space as a whole, hereafter space exploration techniques, explore the space of interleavings randomly, exhaustively or driven by some coverage criteria or heuristics. Two relevant classes of space exploration techniques are stress testing and bounded search techniques.

As we discuss in detail in Section 3.2, properties of interleaving typically identify patterns of interactions across execution flows that are likely to expose concurrency failures. Approaches that target some interleaving properties, hereafter property based techniques, aim to identify the interleavings that are most likely to expose such patterns. Some property based techniques use the property of interleavings not only to select a relevant subset of interleavings, but also to generate a set of test cases that can execute the identified interleavings. Such techniques steer the generation towards test cases that can manifest interleavings that expose the property of interest. Most property based techniques rely on some dynamic or hybrid analysis to build an abstract model of the system and capture the order relations between the program instructions, and then exploit the model to identify a set of interleavings that expose the property of interest. Some property based techniques, hereafter detection techniques, use the model to simply detect the presence of the pattern of interest in the analyzed trace. Other property based techniques, hereafter prediction techniques, also look for alternative interleavings that may expose the property of interest, usually relying on model checking or SAT/SMT solvers.

Techniques that target some specific failure to reproduce, hereafter reproduction techniques, identify the interleavings that reproduce a given concurrency failure. The two relevant classes of reproduction techniques are record-and-replay techniques, which identify the failure-inducing interleaving relying on information continuously collected at runtime (e.g. execution traces), and post-processing techniques, which rely on information collected only at the time of the failure (e.g. memory core dumps).

Approaches that address also the problem of comparing the results with oracles execute the system in a controlled environment that forces the selected
interleavings and compare the results of the execution with the given oracle.

3.2 Towards a Classification Schema

Our analysis of the literature led to the definition of the framework of Figure 3.1 that we use to identify seven distinct classification criteria as reported in Figure 3.2. The seven classification criteria distinguish techniques based on input, selection of interleavings, property of interleavings, output and oracle, guarantees and target systems.

Input. Most approaches assume the availability of a set of input test cases, few approaches work on some models of the system under test, reproduction approaches require failure information. Other approaches require both test cases and models.

Selection of interleavings. Many approaches refer to some properties of the interleavings to select a relevant subset (property based), other approaches either exhaustively explore the interleaving space or exploit some coverage criteria or heuristics (space exploration), yet other approaches identify the interleavings that reproduce a given concurrency failure (reproduction).

Property of interleavings. Many approaches select interleavings referring to some specific concurrency properties: data race, atomicity violation, deadlock, combined or order violation properties.

Output and oracle. Some techniques simply report whether the explored interleavings satisfy the property of interest (property satisfying interleavings), while others identify failing executions according to some oracles, in the form of system crashes, deadlocks or violated assertions.

Guarantees. Different techniques offer various levels of assurance in terms of precision and correctness of their results.

Target systems. Most techniques target some specific kinds of systems that depend on the communication model (shared memory, message passing or general, that is, independent from the communication paradigm), the programming paradigm (either general or specific, such as object oriented, actor based or event based paradigms) and the consistency model (either sequential or relaxed).

Technique. The techniques differ in terms of the type of analysis, which can be dynamic or hybrid, the granularity (unit, integration or system), and the architecture used to implement the technique, which can be either centralized or
Figure 3.2. Classification criteria for the techniques to expose concurrency failures
distributed.

In the remainder of this section, we discuss the classification criteria in detail.

3.2.1 Input

All techniques take in input the system under test, which we keep implicit in our classification. Many techniques require also a set of test cases, while others generate test cases automatically, thus implementing the generating test cases feature of Figure 3.1.

Some techniques require also a system model that specifies either some relevant properties or the expected behavior of the system. For instance, some techniques rely on code annotations to identify either code blocks that are intended to be atomic or sets of data items that are assumed to be updated atomically [41]. For some technique, the presence of a system model is optional: they work independently from an initial system model, but can benefit from an optional model to improve the accuracy of the approach.

Reproduction techniques take in input failure information. Some techniques rely on information continuously collected during the execution of the system (execution traces), yet other techniques exploit information collected only at the time of the failure (memory core dumps).

3.2.2 Selection of Interleavings

Selecting interleavings is the primary objective of many approaches. Some techniques exploit properties of interest to select interleavings: we refer to them as property based. Other techniques explore the space of interleavings exhaustively or randomly, possibly exploiting heuristics or coverage criteria: we refer to them as space exploration techniques. Other techniques select the interleavings that reproduce a given failure observed in the field: we refer to them as reproduction techniques.

Property based techniques

Property based techniques select interleavings according to one or more properties of interest. They typically apply some form of analysis to an execution trace to identify relevant synchronization constraints between instructions. For instance, lockset analysis focuses on lock based synchronization and looks for
accesses to shared data items that are not protected with locks \cite{116}. Happens-before analysis extends this approach by capturing general order constraints. Different types of happens-before analysis apply to different synchronization mechanisms \cite{101}, and present various cost-accuracy trade offs \cite{59, 128}.

Property based techniques exploit the order information identified with the analysis to either only understand whether the analyzed trace exposes the property of interest (detection techniques) or also identify alternative interleavings that can expose the property of interest (prediction techniques).

Space exploration techniques

Space exploration techniques explore the space of interleavings without referring to a specific property, and include stress testing, exhaustive exploration, coverage criteria and heuristics.

**Stress testing.** Stress testing approaches execute the test suite several times, aiming to observe different interleavings. They do not offer any guarantee of observing a given portion of the interleaving space, and do not introduce any mechanism to improve the probability of executing new interleavings.

**Exhaustive exploration.** Exhaustive exploration approaches aim to execute all possible interleavings. Since in general the space of interleavings can be huge, these techniques either limit both the number of instructions and the execution flows of the input test cases, or introduce bounds to the exploration space \cite{88}. They also often adopt reduction techniques such as dynamic partial order reduction \cite{49} to avoid executing equivalent interleavings.

**Coverage criteria.** Coverage criteria identify the interleavings to exercise in terms of depth of the explored space. For example, in shared memory systems, a coverage criterion might require that for each pair of instructions $i_1$ and $i_2$ that belong to different execution flows and operate on the same data item $d$ there should exist at least a test case that exercises the interleaving in which $i_1$ occurs before $i_2$ and one that exercises the interleaving in which $i_2$ occurs before $i_1$.

**Heuristics.** Heuristics guide the exploration of the interleaving space. For instance, some techniques prioritize interleavings by their diversity with respect to the executed ones. Similarly, some other techniques prioritize interleavings that can be obtained by introducing a bounded number of scheduling constraints.
Reproduction techniques

Reproduction techniques select interleavings that reproduce a specific concurrency failure that has been observed in the field, and include record-and-replay and post-processing approaches.

Record-and-replay. Record-and-replay approaches rely on information continuously collected at runtime. Since recording the whole execution would introduce a performance overhead which is not acceptable in production environment, these techniques aim to reduce the amount of information that needs to be recorded to enable failure reproduction. They either collect a sample of the executed instructions \[2\], the accesses to shared variables \[57\], or the local control-flow choices of each thread \[61\].

Post-processing. Post-processing approaches rely on information that is collected only at the time of the failure, avoiding the performance overhead that record-and-replay approaches introduce. The post-processing approaches presented so far rely on memory core dumps and usually combine static analysis and symbolic execution to identify the interleaving that reproduces the given failure \[142, 154\].

3.2.3 Property of Interleavings

Property based techniques target specific properties of interleavings, which are patterns of interactions between execution flows that are likely to violate the developers’ assumptions on the order of execution of instructions. Some property based techniques target the classical properties of interleavings: data races, atomicity violations, deadlocks, and order violations (see Chapter 2). Other techniques combine multiple classical properties.

3.2.4 Output and Oracle

Techniques for exposing concurrency failures produce two types of outputs, some simply produce property satisfying interleavings, that is, interleavings that expose a property of interest, while others compare the results produced by executing an interleaving with an oracle, and return the failing executions, thus implementing the comparing with oracle feature of Figure 3.1.

In general, not all the interleavings that exhibit a property of interest lead to a failure. Thus, techniques that output property satisfying interleavings may result in false positives. For instance, some techniques that detect data races may
signal many benign data races.

Oracles define criteria to discriminate between acceptable and failing executions, and can be system crash, deadlock or violated assertion oracles. System crash and deadlock oracles, also known as implicit oracles, identify executions that lead to system crashes and deadlocks, respectively. Violated assertion oracles exploit assertions about the correct behavior of the system, based either on explicit specifications or implicit assumptions about the programming paradigm.

3.2.5 Guarantees

Different techniques for selecting interleavings guarantee various levels of validity of the results.

A technique guarantees the feasibility of the results if it produces only interleavings that can be observed in some concrete executions. Not all techniques guarantee the feasibility of interleavings, for instance, some prediction techniques that analyze traces and produce alternative interleavings of the observed operations may miss some program constraints, and thus return interleavings that do not correspond to any feasible execution.

A technique guarantees soundness of the results if it produces only interleavings that lead to an oracle violation. A technique guarantees the completeness of the results if it produces all the interleavings that can be observed in some concrete executions and that lead to an oracle violation. A technique guarantees property soundness if it identifies only feasible interleavings, which exhibit the property of interest for the considered test cases. A technique guarantees property completeness if it identifies all feasible interleavings that exhibit the property of interest for the considered test cases.

The concepts of property soundness and property completeness only apply to property based techniques. Several authors of property based techniques present their approach as sound and/or complete with respect to the property of interleavings they consider. However, their claims often rely on the assumption that only some specific synchronization mechanisms are used. In the general case, the type of order relations and analysis adopted in most property based techniques, such as happens-before analysis [101] or causally-precedes relations [128], can introduce approximations that hamper both soundness and completeness [59].

Soundness and completeness describe the accuracy of a technique in exposing concurrency failures. Property soundness and property completeness describe the accuracy of a technique in detecting interleavings that expose a given property.
3.2.6 Target System

We identify three elements that characterize the type of target concurrent systems, the communication model, the programming paradigm and the consistency model.

The communication model specifies how the execution flows interact with each other, and includes shared memory and message passing models. General techniques target both types of systems.

Many techniques target a specific programming paradigm, sometimes identified by the target synchronization mechanisms. For instance, some techniques exploit specific properties of the object oriented paradigm, such as encapsulation of state or subtype substitutability. Similarly, other techniques build on the assumptions provided in actor based systems. Some techniques only consider faults that arise from the use of specific synchronization mechanisms, such as deadlocks that derive from the incorrect use of lock based synchronization. Yet other techniques do not make any assumption on the programming paradigm adopted and work with general systems.

Finally, some techniques assume a sequential consistency model, while other techniques can be applied to relaxed consistency models.

3.2.7 Technique

We characterize the many testing techniques proposed so far along three axes, the type of analysis they implement, the granularity of the technique, and the type of architecture they adopt.

Analysis

Techniques for exposing concurrency failures implement either dynamic or hybrid analysis. Dynamic analysis techniques use only information derived from executions of the system under test. Hybrid analysis techniques use both static and dynamic information. Techniques that rely on static information fall outside the scope of this survey.

Granularity

Different techniques work at various granularity levels that span from unit to integration and system. Unit techniques target individual units in isolation. For instance, in object oriented programs, unit techniques consider classes in isolation, without taking into account their interactions with other components of
the system. Integration techniques focus on the interactions between units, and check that the communication interface between the units works as expected. System techniques target the system as a whole, and verify that it meets its requirements.

Architecture

Techniques for exposing concurrency failures refer to different architectures that characterize the concrete infrastructure used to exercise the system under test. Such infrastructures are composed of one or more drivers, which produce input data to one or more execution flows of the system under test, and that observe the outputs produced by the system under test. For instance, to stimulate a client-server distributed system, a driver can initialize a client, submit some requests to the server, wait for replies from the server, and evaluate the received replies with respect to an oracle.

Architectures can be either centralized, when a single driver interacts with the system, or distributed, when more than one driver interacts with (different) execution flows of the system. The above scenario of a client-server distributed system exemplifies a centralized architecture. A framework for stimulating a peer-to-peer system in a distributed environment with a driver for each peer is a simple example of a distributed architecture.

We distinguish between synchronized and independent distributed architecture. In synchronized architectures, the drivers coordinate each other by exchanging messages, while in independent architectures the drivers execute independently and do not exchange messages to coordinate their actions. In distributed architectures, the driver synchronization is used to overcome controllability and observability problems. These problems occur if a driver cannot determine when to produce a particular input, or whether a particular output is generated in response to a specific input or not, respectively [24].

3.3 Classification Schema

Figure 3.3 shows the classification schema that we define and use to classify the main approaches for exposing concurrency failures according to the criteria discussed in Section 3.2. We use the selection of interleaveings as the main classification criterion since (i) the vast majority of the techniques we survey addresses the problem of selecting interleaveings; (ii) among the main aspects of the problem reported in Figure 3.1, the selection of interleaveings is an activity specific for
3.3 Classification Schema

concurrent systems and more characterizes the problem, as generating test cases and comparing results with oracle are activities required and performed also in the case of sequential programs.

We distinguish between property based, space exploration, and reproduction techniques, that we discuss in Sections 4.1, 4.2, and 4.3, respectively. We classify property based approaches according to the target property of interleaving as data race (Section 4.1.1), atomicity violation (Section 4.1.2), deadlock (Section 4.1.3), combined (Section 4.1.4), and order violation (Section 4.1.5). We classify space exploration techniques as stress testing (Section 4.2.1), exhaustive exploration (Section 4.2.2), coverage criteria (Section 4.2.3), and heuristics (Section 4.2.4). We classify reproduction techniques as record-and-replay (Section 4.3.1), and post-processing (Section 4.3.2).

This organization groups together approaches that implement related classes of methodologies and algorithms to expose failures. This is the case of property based techniques that exploit the same property of interleavings, which typically target the same type of concurrency faults, as well as space exploration and reproduction approaches, which typically target generic types of faults.
A Taxonomy of Techniques for Exposing Concurrency Failures

In this chapter, we propose a taxonomy of the state-of-the-art techniques for exposing concurrency failures. The taxonomy classifies the approaches according to the classification schema presented in Chapter 3, which distinguishes between property based (Section 4.1), space exploration (Section 4.2), and reproduction (Section 4.3) techniques. To provide a comprehensive survey of the emerging trends in exposing concurrency failures, we systematically reviewed the literature from 2000 to 2017: we (i) searched the online repositories of the main scientific publishers, including IEEE Explore, ACM Digital Library, Springer Online Library and Elsevier Online Library, and more generally the Web through the popular online search engines such as Google Scholar and Microsoft Academic Search; we collected papers that are published from year 2000 and that present one of the following set of keywords in their title or abstract: “testing + concurrent”, “testing + multi-thread”, “testing + parallel”, “testing + distributed”, “reproducing + concurrent”, “reproducing + multi-thread”, “reproducing + parallel”, “reproducing + distributed”, (ii) considered all publications that are cited or citing the papers in our repository, and that match the same criteria, (iii) manually analyzed the proceedings of the conferences and the journals where the papers in our repository appear, (iv) filtered out the papers outside the scope of our analysis, for example papers on hardware testing and papers on theoretical aspects, and (v) filtered out workshop papers and preliminary work that has been later subsumed by conference or journal publications.

We summarize the classification of the techniques in seven tables according to the criteria. Table 4.1, Table 4.2, Table 4.3, Table 4.4 and Table 4.5 overview property based techniques. Table 4.6 overviews space exploration techniques.
Table 4.7 overviews reproduction techniques. The tables share the same structure. The rows indicate the techniques with their name, when available, or with the name of the authors of the paper that proposed the technique. Rows are grouped by subcategory when applicable, and report the approaches sorted by main contribution within the same subcategory. The contribution of most approaches is multi-faced: we list the approaches according to what we identify as their core novelty, mentioning other elements when particularly relevant. The columns correspond to the criteria identified in Figure 3.2.

In the tables we do not report explicitly the testing architecture, since all the techniques we analyze implement a centralized architecture. In the tables we also do not report failure information, since either it does not apply (property-based and space exploration techniques) or it is a fix requirement in input (reproduction techniques). In Tables 4.6 and 4.7 we also omit columns prediction, property completeness and property soundness, since they apply only to property based techniques.

4.1 Property Based Techniques

4.1.1 Data Race

We classify the large number of techniques designed to detect data races in shared memory programs according to both their granularity and the type of analysis they perform. We consider techniques that target low level and high level data races, and we further classify low level techniques as lockset, happens-before and hybrid analysis techniques.

Low level data race detection techniques. Low level data race detection techniques target data races that occur at the level of individual memory locations. They rely on some form of analysis to track the order relations between memory instructions in a given execution trace, and either detect the occurrence of a data race or predict if a data race is possible in alternative interleavings.

These techniques rely either on lockset analysis, which simply identifies concurrent memory accesses not protected by locks, or on happens-before analysis, which detects some order relations among concurrent memory accesses. Testing techniques that rely on happens-before analysis often claim to be property complete, meaning that they can detect all the data races that can be generated with alternative schedules of an input execution trace. However, happens-before analysis is in general conservative: for instance, when it observes the release of a lock followed by an acquisition of the same lock in an execution trace, it
## Table 4.1. Data race detection techniques

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Low level — Lockset</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shacham et al. [122]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
<td>S</td>
</tr>
<tr>
<td>Racez [123]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
<td>S</td>
</tr>
<tr>
<td>Praun and Gross [139]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>SM</td>
<td>OO</td>
<td>Seq</td>
<td>S</td>
</tr>
<tr>
<td>ACCORD [87]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
<td>Fork-join</td>
<td>Seq</td>
<td>S</td>
</tr>
<tr>
<td><strong>Low level — Happens-before</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FastTrack [42]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
<td>S</td>
</tr>
<tr>
<td>LiteRace [87]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
<td>U</td>
</tr>
<tr>
<td>Pacer [16]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
<td>S</td>
</tr>
<tr>
<td>SOS [77]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
<td>S</td>
</tr>
<tr>
<td>Carisma [155]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
<td>S</td>
</tr>
<tr>
<td>Reactor [109]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
<td>S</td>
</tr>
<tr>
<td>Narayanasamy et al. [93]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
<td>S</td>
</tr>
<tr>
<td>Frost [137]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
<td>S</td>
</tr>
<tr>
<td>Portend [68]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
<td>S</td>
</tr>
<tr>
<td>Tian et al. [134]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
<td>S</td>
</tr>
<tr>
<td>RDFT [109]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
<td>S</td>
</tr>
<tr>
<td>Smaragdakis et al. [128]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
<td>S</td>
</tr>
<tr>
<td>DrFinder [20]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
<td>S</td>
</tr>
<tr>
<td>RVPredict [59]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
<td>S</td>
</tr>
<tr>
<td>WebRacer [102]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
<td>Web platforms</td>
<td>Seq</td>
<td>S</td>
</tr>
<tr>
<td>EventRacer [110]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
<td>Event-based</td>
<td>Seq</td>
<td>S</td>
</tr>
<tr>
<td>DroidRacer [85]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
<td>Android apps</td>
<td>Seq</td>
<td>S</td>
</tr>
<tr>
<td>Java RaceFinder [70]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
<td>General</td>
<td>Rel</td>
<td>S</td>
</tr>
<tr>
<td>Relaxer [19]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
<td>General</td>
<td>Rel</td>
<td>S</td>
</tr>
<tr>
<td><strong>Low level — Hybrid</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Choi et al. [29]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
<td>S</td>
</tr>
<tr>
<td>Wester et al. [143]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
<td>S</td>
</tr>
<tr>
<td>RaceMob [69]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
<td>S</td>
</tr>
<tr>
<td>RaceFuzzer [119]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
<td>S</td>
</tr>
<tr>
<td>RaceTrack [142]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
<td>S</td>
</tr>
<tr>
<td>Goldilocks [87]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
<td>S</td>
</tr>
<tr>
<td>MultiRace [106]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
<td>General</td>
<td>Rel</td>
<td>S</td>
</tr>
<tr>
<td>SimRT [150]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
<td>S</td>
</tr>
<tr>
<td>Racageddon [38]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
<td>S</td>
</tr>
<tr>
<td>Narada [114]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
<td>S</td>
</tr>
<tr>
<td><strong>High level</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Colt [121]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
<td>OO</td>
<td>Seq</td>
<td>U</td>
</tr>
<tr>
<td>Dimitrov et al. [84]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
<td>OO</td>
<td>Seq</td>
<td>U</td>
</tr>
<tr>
<td>Simian [12]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>SM</td>
<td>Web Platforms</td>
<td>Seq</td>
<td>U</td>
</tr>
</tbody>
</table>

Legend (common to all tables):
- SM shared memory
- Seq sequential consistency
- S system testing
- H hybrid analysis
- MP message passing
- Rel relaxed consistency
- U unit testing
- D dynamic analysis
interprets the two operations as totally ordered, while they could appear in a
different order in other interleavings of the same trace. Because of this, tradi-
tional happens-before analysis may miss some possible interleavings, and may
thus miss some faults [128].

The property completeness problem has been addressed by either implement-
ing variants of the happens-before relation that capture the order of events more
accurately, and thus reduce the possibility of missing some faults [59, 128], or by
exploiting model checking to explore re-orderings of instructions that are not al-
lowed according to the over-restrictive happens-before analysis, but still possible
in practice. Some hybrid solutions combine the advantages of the less accurate
but inexpensive lockset analysis with the more accurate but expensive happens-
before analysis [94].

High level data race detection techniques. High level data race detection
 techniques target complex data structures, such as objects in object oriented
 programs, and look for interleavings that lead to results not compatible with any
 serial execution. The naïve approach to detect high level data races consists in
 comparing the results of an interleaving with all possible serial executions. The
 intuitive scalability issues of the exhaustive analysis of all possible serial execu-
tions is tackled by many testing techniques that exploit some form of specification
 provided by the developer, which indicates relevant order characteristics among
 operations, such as commutativity.

Low level — Lockset

Lockset analysis has been proposed for data race detection by Savage et al.
in the late nineties based on the theory of reduction that Lipton introduced in
the seventies [78]. Savage et al.’s Eraser approach [116] addresses both the con-
servative limitations of static data race analysis and the performance problems
of happens-before analysis, the two popular classes of approaches for detecting
data races that were investigated at that time. Indeed in the nineties, happens-
before analysis was considered too expensive, since it requires information for
each execution flow about the concurrent accesses to each shared data item.
Lockset analysis reveals possible data races by dynamically computing the set of
common locks held when accessing shared data items, and identifying execution
flows that access the same shared data items without sharing any lock. By tak-
ing into account only lock based synchronization, lockset analysis improves the
efficiency with respect to happens-before analysis since it only needs to store in-
formation about the set of locks held by the execution flows, but looses accuracy,
since it ignores additional order relations determined by other synchronization
mechanisms. Thus, the techniques based on lockset analysis are not property sound.

In the last fifteen years, research on data race detection focused mostly on reducing the amount of false positives, moving the attention back to happens-before analysis and forward to hybrid approaches. The few recent techniques based exclusively on lockset analysis aim to either improve precision and efficiency or to extend to new programming paradigms. Shacham et al. [122] improve the precision by combining dynamic lockset analysis with model checking. Racez [123] improves the efficiency with a sampling approach that captures only a subset of the synchronization operations and the memory accesses performed by the target system. ACCORD [67] extends lockset analysis to object oriented and array based concurrent programs.

Low level — Happens-before

Happens-before analysis was introduced by Leslie Lamport in the late seventies [72] and has been adopted in several early techniques to detect data races in concurrent programs [36, 91].

Happens-before analysis is more precise than lockset analysis, since it can deal with any kind of synchronization mechanism beyond lock based synchronization, and thus can avoid many false positives that are inevitable in lockset analysis at the price of additional computational costs. Happens-before analysis has been widely studied in the last fifteen years in the context of exposing data races with focus on (i) improving performance, (ii) reducing false positives, (iii) improving completeness, (iv) tailoring the analysis to specific programming paradigms or languages, and (v) extending the analysis to relaxed (non sequential) memory models.

Different approaches improve the performance of happens-before analysis. FastTrack [42] proposes a lightweight representation of the happens-before analysis with constant time and space complexity that records only the information about the last write operation on each data item. LiteRace [87], Pacer [16], SOS [77] and Carisma [155] reduce the cost of the expensive monitoring activities through sampling: LiteRace instruments only code elements that are less frequently accessed, based on the assumption that frequently accessed code elements have fewer probabilities to be involved in data races; Pacer estimates the probability of finding data races within code regions based on the sampling rate; SOS excludes from the analysis stationary objects, which are objects that are only read after being written during the initialization period; Carisma exploits the similarity between multiple accesses to the same data structures to estimate
and balance the sample budget. ReEnact [107] improves the performance of happens-before analysis by proposing an hardware implementation.

Some approaches focus on reducing the amount of false positives that happens-before analysis can produce. Narayanasamy et al. [90], Frost [137] and Portend [68] reduce the false positive rate by automatically classifying the detected races as either benign or harmful: for a given data race, the approaches replay the execution for the different orders among the memory operations involved in the data race, and classify the race as harmful only if the executions result in different program states. Tian et al. [134] improve the accuracy of happens-before analysis by inferring program specific synchronization mechanisms, such as flags, test-and-set locks and barriers. RDIT [109] reduces the amount of false positives of happens-before analysis by considering synchronization events generated from external libraries.

Some approaches improve the completeness of happens-before analysis reducing the number of false negatives it produces. Smaragdakis et al. [128] propose the causally-precedes relation, which relaxes the happens-before relation with respect to lock releases and acquisitions by inferring an order between two lock-protected blocks if and only if they contain conflicting statements. Cat et al. propose DrFinder [20] to deal with hidden data races that consists of pairs of accesses to the same shared memory locations that are in a happens-before relation only for some subsets of all possible interleavings. RVIPredict [59] improves completeness by integrating happens-before analysis with control flow information.

Some techniques improve the precision of the analysis by reducing its scope to specific types of applications, and exploiting the domain semantics to infer order constraints more precisely: WebRacer [102] and EventRacer [110] adapt the analysis to Web and event-based applications, respectively; DroidRacer [85] applies the analysis to Android applications.

RaceFinder [70] and Relaxer [19] do not assume a sequentially consistent memory model and detect data races in a relaxed memory model. RaceFinder augments the analysis with model checking to capture order relations in the relaxed Java memory model. Relaxer examines a sequentially-consistent execution and dynamically detects potential data races, which are used to predict possible violations of sequential consistency under alternate executions on a relaxed memory model.
Low level — Hybrid

Hybrid techniques combine lockset and happens-before analyses to benefit from the accuracy of happens-before analysis with the efficiency of lockset analysis. Following the seminal work of Choi et al. [28], hybrid techniques limit the scope of expensive happens-before analysis to code fragments that either static or dynamic lockset analysis efficiently isolates as possibly affected by data races.

The approaches differ from their focus that can be on (i) improving performance by means of specific execution frameworks, (ii) reducing the amount of false positives, (iii) targeting specific synchronization mechanisms, (iv) covering relaxed memory models, and (v) completing the identified interleavings with test cases.

Wester et al. [143] improve performance by pipelining the analysis on a parallel infrastructure that exploits multiple cores to speed up lockset and happens-before analyses. RaceMob [69] optimizes performance of the analyses by crowdsourcing distribute data race detection across several executions.

RaceFuzzer [119] reduces the amount of false positives by randomly generating executions that expose data races and observing their effects. RaceFuzzer executes the different interleavings determined by a data race and uses either program crashes or assertions to discriminate between faulty and benign races.

RaceTrack [152] and Goldilocks [37] extend the analysis to specific synchronization mechanisms: RaceTrack targets both lock-based and fork-join synchronization primitives, while Goldilocks targets the synchronization mechanisms of software transactions.

MultiRace [103] extends the analysis to the relaxed C++ memory model and aims to detect data races in production mode by introducing analysis optimizations that reduce the number of checks to memory accesses thus reducing the overhead.

SimRT [150], Racageddon [38], and Narada [114] complement interleavings with test case prioritization and generation: SimRT selects and prioritizes regression test cases according to the probability of exposing newly introduced data races after program changes by identifying the variables that are impacted by a program change; Racageddon generates a test input together with an interleaving that lead to executing a target data race by combining a hybrid data race detection technique with symbolic execution; Narada analyzes sequential test cases to synthesize concurrent test cases that can expose data races.
Table 4.2. Atomicity violation detection techniques

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Code centric</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Atomizer [41]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
<td>S</td>
<td>D</td>
<td>-</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SVD [147]</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
<td>S</td>
<td>D</td>
<td>-</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AVIO [42]</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
<td>S</td>
<td>D</td>
<td>-</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Velodrome [43]</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
<td>S</td>
<td>D</td>
<td>-</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AtomFuzzer [27]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
<td>S</td>
<td>D</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HAVE [27]</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
<td>S</td>
<td>H</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Penelope [129]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
<td>U</td>
<td>D</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wang and Stoller [141]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
<td>U</td>
<td>D</td>
<td>✓</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CTriigger [98]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
<td>S</td>
<td>D</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Falcon [99]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
<td>U</td>
<td>D</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DoubleChecker [14]</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
<td>U</td>
<td>D</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intruder [113]</td>
<td>✓</td>
<td>n.a.n.a.n.a.n.a.</td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
<td>U</td>
<td>D</td>
<td>n.a.n.a.n.a.n.a.n.a.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AutoConTest [112]</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
<td>U</td>
<td>D</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data centric</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Muvi [80]</td>
<td>✓</td>
<td>n.a.n.a.n.a.n.a.</td>
<td>n.a.</td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
<td>S</td>
<td>D</td>
<td>n.a.n.a.n.a.n.a.n.a.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hammer et al. [51]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
<td>S</td>
<td>H</td>
<td>-</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AssetFuzzer [71]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
<td>S</td>
<td>D</td>
<td>-</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ReConTest [133]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
<td>S</td>
<td>D</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

High level

Few approaches move from the analysis of direct shared memory accesses to the analysis of complex data structures. Colt [121] detects non-linearizable sequences of operations on Java objects by analyzing the trace of a single execution flow and randomly generating an adversary concurrent execution flow. Dimitrov et al. [34] present a technique to detect commutativity races, which are pairs of operations on the same object that are not ordered according to the happens-before relation and that do not commute. Simian [113] targets multi-client web applications, where multiple clients concurrently work on a shared resource, such as a text document, a spreadsheet, or source code.

4.1.2 Atomicity Violation

We classify atomicity violation techniques based on the considered atomicity constraints as code centric, if they target code regions, and data centric, if they...
target data items. Similarly to techniques to detect data races, techniques to
detect atomicity violations dynamically analyze the target system to investigate
the order relations imposed by the synchronization mechanisms, by exploiting
some form of lockset or happens-before analysis, and suffer from the limitations
of the ground analyses.

Code centric

Detecting code centric atomicity violations involves (i) identifying code re-
gions that are intended to be atomic, and (ii) detecting interleavings that violate
the atomicity of the identified code regions.

Different techniques identify code regions that are intended to be atomic by
relying on (i) system specifications or models, (ii) assumptions or heuristics on
atomic blocks, or (iii) static or dynamic analyses that infer atomic blocks.

In general, detecting if an interleaving violates atomicity corresponds to check-
ing if the interleaving is serializable, that is, the results of its execution are equiv-
alent to the results of any serial execution. Checking serializability is impractical
due to the large number of serial executions that are present even in small pro-
grams. Code centric approaches reduce the problem of verifying the atomicity
of interleavings by searching for memory-access patterns that encode sufficient
but not necessary conditions for non-serializability, thus trading completeness
for performance. Current approaches exploit some form of dynamic analysis,
usually happens-before analysis.

The two-phase code centric approach to detect atomicity violations originates
from Atomizer, the seminal work of Flanagan and Freund [41]. Atomizer ex-
loits lockset analysis to identify non serializable interleavings, relying on code
annotations that specify atomic code blocks. The recent code centric approaches
improve the original Atomizer approach with techniques to: (i) infer atomic re-
gions, (ii) reduce the amount of false positives, (iii) improve performance, and
(iv) augment the selection of interleavings with test case generation.

SVD [147] and AVIO [82] automatically infer atomic code regions: SVD infers
atomic regions by exploiting data and control dependencies; AVIO models atomic
regions in terms of interleaving invariants that it iteratively infers from system
executions.

Some approaches reduce the amount of false positives generated by the analy-
ysis. Velodrome [43] uses serializability to distinguish between benign and harm-
ful atomicity violations. AtomFuzzer [97] and Penelope [129] actively control
the thread scheduler to trigger atomicity violations and report only those atom-
icity violations that lead to system crashes. HAVE [27] implements a hybrid
Property Based Techniques

analysis that speculatively approximates the results that could have occurred in branches that have not been observed in the executed traces yet.

The approach by Wang and Stoller [141], CTrigger [98], Falcon [99], Best [47] and DoubleChecker [14] improve performance, in many ways. Wang and Stoller exploit object oriented properties to optimize the Atomizer approach; CTrigger reduces the time to reveal atomicity violations by prioritizing the interleavings according to the probability they will occur; Falcon limits the amount of information stored during the analysis, thus trading accuracy for performance; Best aggregates interleavings into equivalence classes and examines only one representative interleaving per class; DoubleChecker combines an imprecise but efficient analysis with a precise but more expensive analysis on demand.

Intruder [113] completes the interleaving by generating test cases that expose atomicity violations. It combines sequential test cases available in a given test suite to generate concurrent test cases that have a high probability to expose atomicity violations. AutoConTest [132] complements the interleavings exploration with test case generation driven by a coverage metric computed during sequential test case generation.

Data centric

Detecting data centric atomicity violations involves finding violations of atomic set serializability, a property of interleavings that Vaziri et al. introduced in 2006 to capture data consistency properties that bind multiple data items in concurrent programs [136]. The recent data centric approaches focus on (i) analyzing the correlation between data items (Muvi [80]), (ii) revealing data access patterns that characterize some form of atomicity violation (Hammer et al. and AssetFuzzer [51, 71]) or (iii) producing regression tests that take advantage of the differences among system versions (ReConTest [133]).

4.1.3 Deadlock

Most testing approaches for detecting deadlocks build on the seminal work of Goodlock [52] and DeadlockFuzzer [64]. Goodlock dynamically records the locking pattern of each program execution as a lock tree, and compares the trees of the threads pairwise to detect circular dependencies that can lead to possible resource deadlocks. DeadlockFuzzer elaborates the potential deadlocks identified with Goodlock to find feasible executions that deadlock.

The main recent techniques build on top of Goodlock and DeadlockFuzzer to: (i) optimize performance, (ii) complement selection of interleavings with test
Table 4.3. Deadlock detection techniques

<table>
<thead>
<tr>
<th>Test Case Model</th>
<th>Output / Oracle</th>
<th>Select System</th>
<th>Testing Tech.</th>
<th>Guarantees</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goodlock [52]</td>
<td>✓</td>
<td>✓</td>
<td>SM Lock sync</td>
<td>S D</td>
</tr>
<tr>
<td>DeadlockFuzzer [64]</td>
<td>✓</td>
<td>✓</td>
<td>SM Lock sync</td>
<td>S D</td>
</tr>
<tr>
<td>MagicFuzzer [21]</td>
<td>✓</td>
<td>✓</td>
<td>SM Lock sync</td>
<td>S D</td>
</tr>
<tr>
<td>Wolf [112]</td>
<td>✓</td>
<td>✓</td>
<td>SM Lock sync</td>
<td>S D</td>
</tr>
<tr>
<td>ConLock [22]</td>
<td>✓</td>
<td>✓</td>
<td>SM Lock sync</td>
<td>S D</td>
</tr>
<tr>
<td>Sherlock [39]</td>
<td>✓</td>
<td>✓</td>
<td>SM Lock sync</td>
<td>S D</td>
</tr>
<tr>
<td>OMEN [111]</td>
<td>✓</td>
<td>✓</td>
<td>SM Lock sync</td>
<td>S D</td>
</tr>
<tr>
<td>Armus [30]</td>
<td>✓</td>
<td>✓</td>
<td>SM Barrier sync</td>
<td>S D</td>
</tr>
</tbody>
</table>

Table 4.4. Techniques for detecting combined properties violations

<table>
<thead>
<tr>
<th>Test Case Model</th>
<th>Output / Oracle</th>
<th>Select System</th>
<th>Testing Tech.</th>
<th>Guarantees</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agarwal et al. [1]</td>
<td>✓</td>
<td>✓</td>
<td>SM General</td>
<td>Seq S H</td>
</tr>
<tr>
<td>Chen and MacDonal [25]</td>
<td>✓</td>
<td>✓</td>
<td>SM General</td>
<td>Seq S H</td>
</tr>
<tr>
<td>Kahlon and Wang [66]</td>
<td>✓</td>
<td>✓</td>
<td>SM General</td>
<td>Seq S D</td>
</tr>
<tr>
<td>PECAN [60]</td>
<td>✓</td>
<td>✓</td>
<td>SM General</td>
<td>Seq S D</td>
</tr>
</tbody>
</table>

case generation, (iii) consider synchronization mechanisms other than locks.

MagicFuzzer [21], Wolf [112], and ConLock [22] improve the performance of the Goodlock analysis. MagicFuzzer and Wolf prune the lock tree by removing nodes that cannot lead to deadlocks and candidate deadlocks that are infeasible due to order relations among events, respectively. ConLock improves the randomized scheduler of DeadlockFuzzer to avoid artificially-generated deadlocks.

Sherlock [39] and OMEN [111] augments interleavings selection with test case generation. Sherlock exploits symbolic execution to identify relevant inputs that can lead to deadlock. OMEN analyzes sequential test cases to synthesize concurrent test cases that can expose deadlocks.

4.1.4 Combined


4.1.5 Order Violation

Several approaches address order violations that cannot be reduced to any of the classic properties of interleavings discussed in the previous sections (data races, atomicity violations and deadlocks). These approaches look for violations of properties that derive from (i) specific types of faults, (ii) program specifications or (iii) semantics of the programming models.

PRETEX [65] and 2ndStrike [48] address typestate faults, that is, faults that involve violations of the high level semantics of the data structures. ConMem [157], ConSeq [156] and ExceptioNULL [40] target violations of faults that impact on the program behavior: ConMem selects interleavings that can lead to null pointer dereference, read of an uninitialized data item and access to invalid memory locations; ConSeq extends ConMem by considering also assertion violations, infinite

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PRETEX [65]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM OO Seq</td>
<td>U H</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>2ndStrike [48]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM OO Seq</td>
<td>U D</td>
<td>✓ ✓ ✓</td>
<td></td>
</tr>
<tr>
<td>ConMem [157]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM General Seq</td>
<td>S D</td>
<td>✓ ✓ ✓</td>
<td></td>
</tr>
<tr>
<td>ConSeq [156]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM General Seq</td>
<td>S H</td>
<td>✓ ✓ ✓</td>
<td></td>
</tr>
<tr>
<td>ExceptioNULL [40]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM General Seq</td>
<td>S D</td>
<td>✓ ✓ ✓</td>
<td></td>
</tr>
<tr>
<td>Maple [148]</td>
<td>✓</td>
<td>✓ ✓</td>
<td>✓</td>
<td>SM General Seq</td>
<td>S D</td>
<td>✓ ✓ ✓</td>
<td></td>
</tr>
<tr>
<td>Sinha and Wang [122]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM General Seq</td>
<td>S D</td>
<td>✓ ✓ ✓</td>
<td></td>
</tr>
<tr>
<td>DefUse [124]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM General Seq</td>
<td>S D</td>
<td>✓ ✓ ✓</td>
<td></td>
</tr>
<tr>
<td>GPredict [85]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM General Seq</td>
<td>S D</td>
<td>✓ ✓ ✓</td>
<td></td>
</tr>
<tr>
<td>SimRacer [149]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>SM Process level Seq</td>
<td>S D</td>
<td>✓ ✓ ✓</td>
<td></td>
</tr>
<tr>
<td>Cafa [59]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>MP Javascript app Seq</td>
<td>S D</td>
<td>✓ ✓ ✓</td>
<td></td>
</tr>
<tr>
<td>Mutlu et al. [89]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>MP Javascript app Seq</td>
<td>S D</td>
<td>✓ ✓ ✓</td>
<td></td>
</tr>
</tbody>
</table>
loops, and calls to error message procedures; ExceptioNULL targets null pointer exceptions. Maple [148] focuses on patterns of shared variable accesses. Differently from the other approaches that target fault types, Maple targets critical patterns that may or may not lead to concurrency faults.

jPredictor [23], Sinha and Wang [127] and GPredict [58] look for concurrent behaviors that violate user-defined program specifications. DefUse [124] targets both sequential and concurrent faults that violate automatically generated data-flow invariants.

SimRacer [149] targets process level violations, Cafa targets Android programs [56], Mutlu et al. target JavaScript programs [89].

4.2 Space Exploration Techniques

Some testing techniques explore the space of interleavings not driven by specific properties: stress testing, exhaustive exploration, coverage criteria and heuristic exploration of the interleaving space.

4.2.1 Stress Testing

Classic stress testing approaches execute test suites that exercise the target system under increasingly heavy and up to extreme load conditions. In the context of testing concurrent systems, stress testing approaches execute the same test suite many times without explicitly controlling the scheduling. Pike [44] checks the linearizability of a concurrent execution by comparing the output and the internal state of that execution with the output and the internal state of the many possible linearizations. SpeedGun [106] targets performance regression testing by executing a test case several times on different versions of the target system and reporting relevant performance differences between the two versions. CovCon [29] generates test cases guided by a coverage criterion that focuses the test generation on infrequently or not at all covered pairs of methods.

4.2.2 Exhaustive exploration

Exhaustive exploration approaches analyze all possible interleavings for a given test input. Ballerina [92] and its extensions [104, 105] limit the exploration to test cases composed of only two execution flows. Sen and Agha [120] analyze all possible interleavings of MPI programs by exploiting symbolic execution. Basset [74] studies exhaustive exploration in the context of actor based
Table 4.6. Techniques for exploring the space of interleavings

<table>
<thead>
<tr>
<th>Input</th>
<th>Output / Oracle</th>
<th>Target System</th>
<th>Testing Tech.</th>
<th>Guarantees</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case</td>
<td>Model</td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
</tr>
<tr>
<td>Stress testing</td>
<td>SpeedGun [106]</td>
<td>SM</td>
<td>General</td>
<td>Seq</td>
</tr>
<tr>
<td>Exhaustive exploration</td>
<td>Pike [44]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Sen and Agha [120]</td>
<td>✓</td>
<td>✓</td>
<td>MP</td>
</tr>
<tr>
<td></td>
<td>Bassett [74]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>CheckMate [63]</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
</tr>
<tr>
<td></td>
<td>CPPMem [9]</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
</tr>
<tr>
<td></td>
<td>Vip [35]</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
</tr>
<tr>
<td></td>
<td>CDSChecker [93]</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
</tr>
<tr>
<td>Coverage criteria</td>
<td>Wang et al. [140]</td>
<td>✓</td>
<td>✓</td>
<td>SM</td>
</tr>
<tr>
<td></td>
<td>Hong et al. [54]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Bita [131]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Heuristics</td>
<td>Rapos [118]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>PCT [18]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Gambit [32]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

programs. CheckMate [63] and Vip [35] exploit model checking to identify deadlocks and violations of user-defined properties expressed in past-time Linear Temporal Logic, respectively. CPPMem [9] and CDSChecker extend the analysis to the C and C++ relaxed memory models, respectively.

4.2.3 Coverage criteria

Coverage criteria identify relevant subsets of the program space to be explored. Wang et al. [140] introduce HaPSet, a coverage criterion based on data flow relations with a reasonable cost and the capability of detecting subtle bugs manifested only by rare interleavings. Hong et al. [54] define the Synchronisation-Pair coverage criterion, which requires to execute all atomic block pairs in different orders. Bita [131] targets actor based programs and introduces three coverage criteria which require covering different sequences of receive events.
4.3 Reproduction Techniques

Table 4.7. Techniques for reproducing concurrency failures

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Model</th>
<th>Input</th>
<th>Output / Oracle</th>
<th>Target System</th>
<th>Testing Tech.</th>
<th>Guarantees</th>
</tr>
</thead>
<tbody>
<tr>
<td>Record-and-replay</td>
<td>ODR [2]</td>
<td>✓</td>
<td>SM General</td>
<td>Seq</td>
<td>S D</td>
<td>✓ ✓</td>
</tr>
<tr>
<td></td>
<td>Pres [100]</td>
<td>✓</td>
<td>SM General</td>
<td>Seq</td>
<td>S D</td>
<td>✓ ✓</td>
</tr>
<tr>
<td></td>
<td>LEAP [57]</td>
<td>✓</td>
<td>SM General</td>
<td>Seq</td>
<td>S H</td>
<td>✓ ✓</td>
</tr>
<tr>
<td></td>
<td>Stride [61]</td>
<td>✓</td>
<td>SM General</td>
<td>Seq</td>
<td>S D</td>
<td>✓ ✓</td>
</tr>
<tr>
<td></td>
<td>Chimera [75]</td>
<td>✓</td>
<td>SM General</td>
<td>Seq</td>
<td>S H</td>
<td>✓ ✓</td>
</tr>
<tr>
<td></td>
<td>CLAP [61]</td>
<td>✓</td>
<td>SM General</td>
<td>Seq</td>
<td>S D</td>
<td>✓ ✓</td>
</tr>
<tr>
<td></td>
<td>RecBuLC [153]</td>
<td>✓</td>
<td>SM General</td>
<td>Seq</td>
<td>S D</td>
<td>✓ ✓</td>
</tr>
<tr>
<td></td>
<td>DESCRY [153]</td>
<td>✓</td>
<td>SM General</td>
<td>Seq</td>
<td>S D</td>
<td>✓ ✓</td>
</tr>
<tr>
<td>Post-processing</td>
<td>ESD [154]</td>
<td>✓</td>
<td>SM General</td>
<td>Seq</td>
<td>S H</td>
<td>✓ ✓</td>
</tr>
<tr>
<td></td>
<td>Weratunge et al [142]</td>
<td>✓</td>
<td>SM General</td>
<td>Seq</td>
<td>S H</td>
<td>✓ ✓</td>
</tr>
</tbody>
</table>

4.2.4 Heuristics

Heuristic approaches bound the space of interleavings to be explored, and prioritize their execution: Rapos [118] exploits partial-order reduction to identify equivalent interleavings; PCT [18] bounds the number of scheduling constraints; Gambit [32] prioritizes interleavings by their diversity with respect to the executed ones.

4.3 Reproduction Techniques

Reproduction techniques explore the space of interleavings to reproduce a specific concurrency failure that has been observed in the field. We classify reproduction techniques according to the type of information they collect to reproduce the failure: Record-and-replay approaches, approaches that rely on information continuously collected at runtime, and Post-processing approaches, approaches that rely on information collected at the time of the failure only.

4.3.1 Record-and-replay

Record-and-replay approaches collect information at runtime and analyze the recorded information to reproduce the observed concurrency failure. The approaches differ from the type and amount of information they collect.
4.3 Reproduction Techniques

ODR [2] records the synchronization operations and a sample of the executed instructions to reproduce an interleaving that lead to the same result of the observed interleaving. PRES [100] collects only synchronization operations. LEAP [57] identifies shared variables through static analysis and records accesses to such variables only. Stride [158] introduces bounded-linkages, which record the relevant relations among synchronization actions without recording the global order of executed instructions. Chimera [75] statically detects data races, protects racy accesses with weak locks and collects all the synchronization operations. CLAP [61] reproduces concurrency failures by collecting the local control-flow choices of each thread. ReCBuLC [153] records hardware per-core local clocks. DESCRY [151] reproduces system-level concurrency failures by leveraging default logs and combining static and dynamic analysis with symbolic execution.

4.3.2 Post-processing

Post-processing approaches collect information only at the time of the failure and do not require runtime tracing or monitoring, thus incurring no runtime overhead.

The two approaches presented so far rely on memory core dumps. ESD [154] combines symbolic execution with inter- and intra-procedural static analysis to reproduce the failure. Weeratunge et al. [142] propose an approach that reproduces a failure by comparing the memory core dumps of a failing execution with the memory core dumps of passing execution.
Chapter 5

Reproducing Concurrency Failures
From Crash Stack Traces

In the previous chapters we presented an extensive and comprehensive classification, analysis and comparison of the state-of-the-art techniques for exposing concurrency failures.

The study indicates the availability of only few preliminary approaches for reproducing concurrency failures. The main characteristics and limitations of the techniques proposed so far are that they (i) rely on either traces that are expensive to produce or memory core dumps that are hard to obtain, and (ii) focus on failure-inducing interleavings, ignoring failure-inducing concurrent test cases.

In this chapter we propose CONCRASH, a technique that synthesizes both failure-inducing concurrent test cases and related interleavings and relies on crash stacks, which are easily obtainable and do not suffer from performance and privacy issues. CONCRASH (i) efficiently explores the huge space of possible test cases to identify a failure-inducing test case, by using a suitable combination of search pruning strategies, and (ii) integrates and adapts existing techniques for exploring interleavings, to automatically reproduce a concurrency failure, thus identifying both a failure-inducing test case and corresponding interleaving.

In the remainder of this chapter we present the algorithmic aspects of CONCRASH, and discuss in detail the strategies it adopts to automatically reproduce concurrency failures from crash stack traces. The experimental evaluation that we present in the next chapter shows the effectiveness of CONCRASH.
5.1 Overview

ConCrash synthesizes concurrent test cases 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 [50]. In essence, thread-safe classes guarantee that each thread access an instance of the class as if no other threads are using the same instance concurrently, without requiring introducing any synchronization mechanism. Thread-safe classes encapsulate concurrency related-challenges, and are commonly used in concurrent software design. Incorrect implementations of synchronization mechanisms that violate thread-safeness lead to wrong concurrent accesses to the class and consequence concurrency failures. Thread-safety violations represent a significant fraction of real-world concurrency failures, due to the wide use of thread-safe classes.

ConCrash automatically synthesizes concurrent test cases that reproduce concurrency failures of classes that violate thread-safety by requiring only the source code of the class-under-test and the standard crash stack of the failure. ConCrash addresses concurrency failures that manifest as runtime exceptions and generate a crash stack. Recent studies on concurrency failures show that 56-70% of the examined failures manifest as system crashes or hangs [81], and produce a crash stack. Crash stacks contain only partial information about the state of the system, thus challenging the reproduction of concurrency failures [145]. In particular, crash stacks provide little information about the state of the objects and the values of the input parameters of the methods involved in the failure, and, more importantly, provide only information about the failing thread, with no information about the other threads that comprise the system and that contribute trigger to failure. This is a key difference with respect to reproducing sequential failures from crash stack traces, which has been previously studied in recent years [8, 26].

ConCrash identifies a failing execution that reproduces a crash stack by efficiently exploring the huge space of interleavings, by alternately generating test cases and exploring thread interleavings. ConCrash iteratively generates concurrent test cases by implementing pruning strategies that exclude both redundant and irrelevant test cases to optimize the exploration of the interleaving space. Test cases are redundant if they induce the same interleaving space of previously investigated test cases and thus would not reproduce the failure. Test cases are irrelevant if ConCrash can infer the impossibility of reproducing the failure from the crash stack trace and the single-threaded execution of the call sequences that comprise the test case. ConCrash pruning strategies
are cost-effective as they analyze single-threaded executions of the method call sequences rather than exploring the full interleaving space of concurrent executions. CONCRASH privileges short test cases to improve the efficiency of exploring interleavings, localizing, and fixing the fault.

Motivating Example

Figure 5.1 shows the code snippet of a concurrency fault in the (supposedly) thread-safe class java.util.logging.Logger of the JDK library. The fault has been introduced in Version 1.4.1 of the library, and has been fixed only in Version 1.7, and has affected the correct usage of the class for many years. The fault witness the difficulty to detect and reproduce treat-safety violations.

Method log (line 28) accesses field filter at lines 33 and 34 within a synchronized block that locks the object instance. The method checks whether the field is initialized (line 33) before dereferencing it (line 34). As the two accesses are enclosed within a synchronized block, the method expects to execute the two statements with an exclusive access to the object instance, thus expecting that no other thread can access the object instance and modify it. However, method setFilter (line 54) can potentially interleave as it accesses and modifies the same field filter (line 58) without locking the object instance. As a result, in a concurrent execution of methods log and setFilter, one thread can execute line 58 between the executions of lines 33 and 34 performed by the other thread, thus setting the reference to null and violating the intended atomicity of method log. If both threads access the same object instance, this thread interleaving triggers a system crash caused by a NullPointerException at line 34 (Figure 5.2). Figure 5.3 shows a concurrent test case that induces such interleaving and reproduces the failure.

Crash Stack Trace

CONCRASH generates concurrent test cases that reproduce concurrency failures from crash stack traces. A crash stack trace (or simply crash stack) reports the sequence of functions that were on the call stack at the time of the failure [62]. Figure 5.2 shows an example of crash stack trace produced by the class java.util.logging.Logger when executing the test case in Figure 5.3 and the specific failure-inducing interleaving described above.

A crash stack trace reports the ordered sequence of functions from the bottom to the top and terminates the sequence with the exception that results from the failure (NullPointerException at line 1 in Figure 5.2). Each entry (frame) in the
public class Logger {
    public void info(String msg) {
        if (Level.INFO.intValue() < levelValue) {
            return;
        }
        log(Level.INFO, msg);
    }

    public void log(Level level, String msg) {
        if (level.intValue() < levelValue || levelValue == offValue) {
            return;
        }
        LogRecord lr = new LogRecord(level, msg);
        doLog(lr);
    }

    private void doLog(LogRecord lr) {
        lr.setLoggerName(name);
        String ebname = getEffectiveResourceBundleName();
        if (ebname != null) {
            lr.setResourceBundleName(ebname);
            lr.setResourceBundle(findResourceBundle(ebname));
        }
        log(lr);
    }

    public void log(LogRecord record) {
        if (record.getLevel().intValue() < levelValue || levelValue == offValue) {
            return;
        }
        synchronized (this) {
            if (filter != null) {
                if (!filter.isLoggable(record)) {
                    return;
                }
            }
            Logger logger = this;
            while (logger != null) {
                Handler targets[] = logger.getHandlers();
                if (targets != null) {
                    for (int i = 0; i < targets.length; i++) {
                        targets[i].publish(record);
                    }
                }
                if (!logger.getUseParentHandlers()) {
                    break;
                }
                logger = logger.getParent();
            }
        }
    }

    public void setFilter(Filter newFilter) throws SecurityException {
        if (!anonymous) {
            manager.checkAccess();
        }
        filter = newFilter;
    }
}

Figure 5.1. Faulty class java.util.logging.Logger of JDK 1.4.1
java.lang.NullPointerException
at Logger.log(Logger.java:34)
at Logger.doLog(Logger.java:25)
at Logger.log(Logger.java:15)
at Logger.info(Logger.java:7)
at LoggerTest$1.runTest(LoggerTest.java:11)
at java.lang.Thread.run(Thread.java:662)

Figure 5.2. Crash stack of class Logger (Bug ID 4779253)

A crash stack trace indicates a function and a code location. The code location of each entry identifies either the location of the call to the next function or the location of the Point Of Failure (POF) in the top entry, which is the static line of code that triggered the failure (line 2 in Figure 5.2).

**CONCRASH** prunes the calls performed by the client code from the crash stack in input. The client code corresponds to those function in the call stack that do not involve the thread-safe class. For example, in Figure 5.2, lines 6 and 7 correspond to calls performed by the client code to start a thread, and do not involve the thread-safe class Logger. Since **CONCRASH** targets concurrency failures of thread-safe classes, the calls performed by the client code are not relevant for reproducing the failure. The crash stacks commonly available in bug reports contain only frames related to the class that leads to the failure with no information about the client code.

We denote the first non-client method in a crash stack as **crashing method** and the class of such method as **Class Under Test (CUT)**. The crashing method corresponds to the method of the thread-safe classes invoked by the client code, and which leads to the failure reported in the crash stack trace. In our running example the CUT is the class Logger and the crashing method is method info as inferred from the crash stack in Figure 5.2.

**Concurrent Test Case**

**CONCRASH** synthesizes failure-inducing concurrent test cases of thread-safe classes. Concurrency failures of (assumed) thread-safe classes can be reproduced with multi-threaded executions of concurrent test cases. A concurrent test case 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 [92, 104, 130, 132]. A call sequence is an ordered sequence of method calls \( \delta = \langle m_1, \ldots, m_n \rangle \) that are executed in a thread. The methods in the sequence have a possible empty set of input parameters, which can be either of primitive type or references to objects created in previous method
Figure 5.3. A concurrent test case that reproduces the crash stack in Figure 5.2 calls. We treat the object receiver of an instance method as the first parameter of the method [95, 132].

A test case 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 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. We consider test cases 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 [81], and in line with work on concurrent test case generation [92, 104, 130, 132].

Intuitively, for reproducing the concurrency failure one suffix shall invoke the crashing method reported in the crash stack trace, 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.
5.2 The ConCrash Approach

Problem Definition and Challenges

The problem that we address with ConCrash can be formulated as follows: Given a crash stack trace, the corresponding Class Under Test (CUT), a set of auxiliary classes, which CUT depends on, and a time-budget, generate a concurrent test case and a failure-inducing interleaving that produce the crash stack trace in input.

When addressing this problem, we face two main challenges: (i) the limited information available in the crash stack and (ii) the high cost of exploring the large interleaving space of a test case. The crash stack does not provide enough information to infer the methods and the input parameter values that comprise the test case, and we need to define an efficient way 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 case.

For instance, to reproduce the concurrency failure of the Logger example in Figure 5.3, ConCrash needs to identify the sequential prefix \(<\text{Logger sout = Logger.getAnonymousLogger(); MyFilter myFilter0=new MyFilter(); sout.setFilter(myFilter0)}\>\), the interfering method \(\text{sout.setFilter(null)}\) and the crashing method \(\text{sout.info("")}\). With different sequential prefixes, for example \(\langle\text{Logger sout=Logger.getAnonymousLogger(); sout.setFilter(null)}\rangle\), the test case does not reproduce the failure for any interleavings, since the if condition at line 33 would evaluate false. The cost of exploring the interleaving space of a test case is inflated by the large amount of possible interleavings for a concurrent test case. A simple random exploration of test cases is not effective, since we may need to randomly generate thousands test cases for triggering a concurrency failure \[92, 104\]. ConCrash introduces an effective strategy for generating few concurrent test cases likely to reproduce the failure.

5.2 The ConCrash Approach

Figure 5.4 illustrates the overall architecture schema of ConCrash. Given as input (i) the CUT source code, (ii) a crash stack trace, and (iii) the source code of the auxiliary classes the CUT depends on, ConCrash synthesizes a failure-inducing test case and interleaving that reproduce the failure that produces the input crash stack trace. As illustrated in Figure 5.4, ConCrash is articulated into two main steps that are executed iteratively until generating a test case and interleaving that reproduce the failure: the Test Case Generator and the Interleaving Explorer steps. At each iteration, the Test Case Generator synthesizes a new test
5.2 The ConCrash Approach

The ConCrash Approach

The Test Case generator is composed of two main components: the call sequence generator and the pruning strategies. The call sequence generator generates candidate test cases to be explored, and is based on AutoConTest, a state-of-the-art concurrent test case generator recently proposed by Terragni and Cheung [132]. CONCRASH tunes the original implementation in order to only generate test cases with the crashing method as first parallel suffix and to adopt a Breadth-First Search exploration strategy (as described in Section 5.3).

The Test Case Generator exploits a set of pruning strategies to steer the test case generation towards test cases that are likely to reproduce the failure. By pruning the test case space before exploring the interleaving space, CONCRASH limits the expensive exploration of the interleaving space to the interleavings that correspond to test cases that most likely expose the concurrency failures. The pruning strategies trim both test cases that are redundant with respect to previously generated test cases and test cases that are irrelevant with respect to the concurrency failure in input. Intuitively, a test case is redundant if it manifests the same interleavings of previously explored test cases, and thus would not reproduce the failure. A test case is irrelevant if it cannot manifest a failure-inducing interleaving. For this reason, exploring the interleaving spaces of redundant or irrelevant test cases is fruitless.

The pruning strategies rely on runtime information collected by executing sequentially and in isolation the method call sequences that comprise the candidate concurrent test case. The sequential execution of a call sequence $\delta$ can effectively approximate the behavior of $\delta$ when executed concurrently with other method

Legend:
$\delta_p$ == sequential prefix, $\delta_{s1}, \delta_{s2}$ == concurrent suffixes

Figure 5.4. Conceptual Architecture of ConCrash
call sequences [113, 114, 115, 130, 132]. Analyzing sequential executions is less expensive than exploring all the possible interleavings of concurrent executions. While state-of-the-art techniques leverage sequential executions for concurrency testing purposes [113, 114, 115, 130, 132], the key intuition of CONCRASH is to use this information together with crash stacks to effectively synthesize test cases that reproduce a concurrency failure.

The Interleaving Explorer checks if the interleaving space of a test case that the Test Case Generator synthesizes contains at least one interleaving that reproduces the failure. The Interleaving Explorer is based on the approach recently proposed by Machado et al. to determine the existence of an interleaving of a given test case that violates a program assertion that encodes the concurrency failure [84]. CONCRASH iteratively executes the Test Case Generator and the Interleaving Explorer until producing a test case and an interleaving that reproduce the failure or until the time budget expires. The next sections describe in detail the Test Case Generator and Interleaving Explorer components.

5.3 Test Case Generator

Figure 5.5 shows the test case generation algorithm. As discussed in Section 5.1, a test case is composed of a sequential prefix, denoted as $\delta_p$, and two concurrent suffixes $\delta_s_1$ and $\delta_s_2$ that are executed concurrently after $\delta_p$. The prefix $\delta_p$ creates a shared object under test (SOUT) of type CUT, and invokes methods that bring the SOUT in a failure-inducing state. The suffixes $\delta_s_1$ and $\delta_s_2$ access the SOUT concurrently trying to manifest a failure-inducing interleaving.

The algorithm explores a search space that is modeled with a tree, which is explored to minimize the generated test cases, 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 13-24) and the elaboration of the new combination, function PRUNING (lines 25-45), which includes the collection of runtime information (lines 26-28) and the pruning strategies (lines 29-45).

Below we describe in details the Tree model of the search space (Section 5.3.1), the minimization of the test cases (Section 5.3.2), and the exploration of the search space (Section 5.3.3), which is composed of the initialization, the exploration of new combinations, the collection of runtime information and the pruning strategies.
1 function CONCRASH
2     /*
3     Initialization
4     */
5     $\mathcal{S} \leftarrow \emptyset$  // state repository
6     $\mathcal{C} \leftarrow \emptyset$  // coverage repository
7     pendingSeqs[...] \leftarrow \emptyset  // sequences to be extended
8     level \leftarrow 0  // current level of sequence exploration
9     pool \leftarrow CREATE-PARAMETER-VALUES(classes)
10    cm \leftarrow EXTRACT-CRASHING-METHOD(CST)
11
12    for each constructor m($\tau_1,...,\tau_n$) of CUT do
13        for each value ($v_1,...,v_n$) of type ($\tau_1,...,\tau_n$) in pool do
14            $\delta_{sout} \leftarrow m(v_1,...,v_n)$
15            if $\delta_{sout}$ does not throw an exception then
16                add $\delta_{sout}$ to end of pendingSeqs[0]
17
18    /*
19       Exploration of new combinations $\delta_p, \delta_{s1}, \delta_{s2}$
20    */
21    while timeBudget is not expired do
22        while pendingSeqs[level] \neq \emptyset do
23            $\delta_p \leftarrow$ get and delete the first $\delta$ in pendingSeqs[level]
24            for each value ($v_1,...,v_n$) of type cm($\tau_1,...,\tau_n$) in pool do
25                $\delta_{s1} \leftarrow cm(v_1,...,v_n)$
26                for each public method m($\tau_1,...,\tau_n$) of CUT do
27                    for each value ($v_1,...,v_n$) of type m($\tau_1,...,\tau_n$) in pool do
28                        $\delta_{s2} \leftarrow m(v_1,...,v_n)$
29                        result $\leftarrow$ PRUNING($\delta_p, \delta_{s1}, \delta_{s2}$)
30                        if result \neq null then
31                            return result
32
33    level++
34
35    function PRUNING($\delta_p, \delta_{s1}, \delta_{s2}$)
36    $\delta_{p,s1} \leftarrow$ append $\delta_{s1}$ to $\delta_p$
37    $\delta_{p,s2} \leftarrow$ append $\delta_{s2}$ to $\delta_p$
38    execute $\delta_{p,s1}$ and $\delta_{p,s2}$  // Collection of runtime information
39    if $\delta_{p,s1}$ and $\delta_{p,s2}$ do not throw exceptions (PS-Exception)
40        \& $\exists e \in \overline{M}(\delta_{cm})$ : stack(e) $=$ CST
41            (PS-Stack)
42        \& $(\overline{M}(\delta_{p,s1}), \overline{M}(\delta_{p,s2})) \notin \mathcal{C}$
43            (PS-Redundant)
44        \& $\overline{M}(\delta_{p,s1})$ interferes with $\overline{M}(\delta_{p,s1})$
45            (PS-Interfere)
46        $\overline{M}(\delta_{p,s1}) \parallel \overline{M}(\delta_{p,s2})$
47            (PS-Interleave)
48        then
49            add $(\overline{M}(\delta_{p,s1}), \overline{M}(\delta_{p,s2}))$ to $\mathcal{C}$
50            $t \leftarrow$ ASSEMBLE-TEST-CODE($\delta_p, \delta_{s1}, \delta_{s2}$)
51            isFailure $\leftarrow$ INTERLEAVING-EXPLORER($t$)
52            if isFailure $=$ true then
53                return $t$  // failure-inducing test code
54    if $\delta(\delta_{s2}) \notin \mathcal{S}$ and $\delta_{p,s2}$ does not throw exception then
55        add $\delta(\delta_{s2})$ to $\mathcal{S}$
56        if $\overline{M}(\delta_{p,s1})$ interferes with $\overline{M}(\delta_{p,s2})$ then
57            add $\delta_{p,s2}$ begin of pendingSeqs[level+1]
58        else
59            add $\delta_{p,s2}$ end of pendingSeqs[level+1]
60    return null

Figure 5.5. The ConCrash algorithm
5.3.1 Modeling the Test Cases Search Space

CONCrash finds a combination of $\delta_p$, $\delta_{s1}$, and $\delta_{s2}$ that altogether constitute a failure-inducing test case, by exploring the space of possible call sequences. Following Terragni and Cheung's approach [132], CONCrash models the test cases search space as a tree whose root node is a call sequence that instantiates the object under test SOUT of type CUT.

Figure 5.6 shows an excerpt of a tree model of the class Logger of our running example. The edges of the tree represent method call sequences. The root represents the initialization sequence (Logger sout = Logger.getAnonymousLogger();), and 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 in Figure 5.6, Node $\delta_0$ represents the sequence $\langle$ Logger sout = Logger.getAnonymousLogger(); Filter f = new Filter(); sout.setFilter(f); sout.info(""); $\rangle$ that is obtained by traversing the tree from the root to the node through edges $\delta_5$ and $\delta_1$.

CONCrash incrementally builds the Tree model starting from the root. The basic operator for building the Tree model is the node traversal operator, which creates a new child node [132]. Given a method $m$ and a node representing a sequence $\delta$, the node traversal operator produces a child node that represents a new sequence obtained from $\delta$ 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 5.6, Edge $\delta_1$ corresponds to a single method call, while Edge $\delta_3$ 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 [92]. The values of the input parameters that are not bounded to SOUT, if any, are chosen from a pool of representative values. The sequence extension operator selects a parameter value depending on the type of $\tau_i$. If $\tau_i$ is of type CUT and is the receiver method, CONCrash binds it to the object reference of SOUT, otherwise ($\tau_i$ is either a primitive type or a non-primitive type), CONCrash chooses a value from a pool of representative values, where the pools of primitive and non-primitive parameter values are built pseudo-deterministically using the same random seed at each iteration. In particular, CONCrash adopts
5.3 Test Case Generator

the input generation random strategy adopted by Randoop [95], a popular test case generator for sequential programs.

5.3.2 Test Case Minimization

The same concurrency failure can be triggered with many test cases composed of different method calls, some of which may be irrelevant with respect to the failure. Short test cases 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 of each thread [79]. Long sequential prefixes increase the computational costs and are more difficult to understand and investigate than short ones [46]. Therefore, the CONCrASH Test Case Generator aims to generate short test cases.

The concurrent suffixes of the test cases correspond to single edges in the tree. Each edge corresponds to a method call that accesses the SOUT object. Limiting concurrent suffixes to single edges does not impact on the failure reproduction
capabilities, since multiple calls within the same concurrent suffix do not expose any thread-safety violation that cannot be exposed with a single method call \cite{50, 53, 132}. In fact, the general form of thread-safety \cite{50} does not guarantee the atomic execution of multiple calls to a shared object of a thread-safe class in the same thread. \textsc{ConCrash} reduces the length of sequential prefixes by adopting a \textit{Breadth-First Search} exploration strategy: it explores all sequences with \( n \) edges before exploring sequences with \( n + 1 \) edges.

### 5.3.3 Exploring the Test Cases Search Space

**Initialization (lines 2-12)**. \textsc{ConCrash} starts by initializing to empty the state repository \( S \), the coverage repository \( C \), and the list of pending sequences to be extended \( \text{pendingSeqs} \) (lines 2, 3 and 4, respectively). \textsc{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 \cite{95} (line 6). Subsequently, it determines the crashing method (line 7) by parsing the crash stack in input (see Section 5.1). 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 (line 9). \textsc{ConCrash} creates a new root \( \delta_{\text{sout}} \) for each of such methods and for each combination of parameter values (lines 8, 9, 10). The algorithm checks whether the execution of \( \delta_{\text{sout}} \) throws an exception (line 11) and, if this is not the case, adds \( \delta_{\text{sout}} \) to the list \( \text{pendingSeqs}[0] \) (line 12). The algorithm does not further elaborate the sequences \( \delta_{\text{sout}} \) 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 13-24)**. \textsc{ConCrash} explores new combinations iteratively until either it reproduces the failure or the time budget expires (line 13). At each iteration, \textsc{ConCrash} removes a sequence \( \delta_p \) from \( \text{pendingSeqs}[\text{level}] \) (line 15), and generates the children of the leaf of the Tree that corresponds to \( \delta_p \), 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 \( \delta_{s1} \) (line 17). \textsc{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 \( \delta_{s2} \) (line 20). Every combination of \( \delta_p, \delta_{s1} \) and \( \delta_{s2} \) corresponds to a candidate concurrent test case. Function \textsc{pruning} analyzes each combination of \( \delta_p, \delta_{s1} \) and \( \delta_{s2} \) to determine if it should be pruned or not (line 21). \textsc{ConCrash} considers
Let the trace \( E = \langle e_1, e_2, \ldots, e_k \rangle \) 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 \( ACQ(l) \) and lock release \( REL(l) \) events;
- method enter \( ENTER(m) \) and exit \( EXIT(m) \) events.

Given a call sequence \( \delta = \langle m_1, \ldots, m_n \rangle \), 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 \( m_i \). Given a call sequence \( \delta \), its sequential coverage \( M(\delta) \) is defined as the partition \( \{ E_1, E_2, \ldots, E_n \} \) of \( E \).

Figure 5.7. Definition of Sequential coverage

all public methods in the CUT to obtain \( \delta_{s_2} \) (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 25-45).

Function **PRUNING** prunes the search space (lines 29-45) relying on the runtime information obtained by executing the input call sequences (lines 26-28).

**Collecting runtime information** (lines 26-28). Let \( \delta_{p,s_1} \) and \( \delta_{p,s_2} \) be the sequences that extend \( \delta_p \) with edges \( \delta_{s_1} \) and \( \delta_{s_2} \), respectively (lines 26 and 27), CONCRASH executes \( \delta_{p,s_1} \) and \( \delta_{p,s_2} \) in isolation (single-threaded execution) (line 28), and collects the following runtime information for each sequence \( \delta \in \{ \delta_{p,s_1}, \delta_{p,s_2} \} \):

- whether (i) \( \delta \) throws an uncaught exception,
- the sequential coverage of the last method call in \( \delta \) [132],
- the state of the object SOUT after executing \( \delta \), which is obtained by serializing SOUT in a deep copy semantic.

**Sequential coverage** is a metric recently presented by Terragni and Cheung, that is defined on the sequential execution of call sequences [132], and is used to infer the possibility of a concurrent test case to induce new interleavings with respect to the previously generated test cases. CONCRASH exploits sequential coverage to identify and avoid both redundant and irrelevant test cases. Figure 5.7 reports the definition of sequential coverage [132]. Since all the CONCRASH concurrent test cases are composed of concurrent suffixes with only the last method accessing SOUT, we are only interested in the sequential coverage of such methods. We denote the last method call trace \( E_n \) in \( M(\delta) \) as \( \overline{M(\delta)} \).

**Pruning strategies** (lines 29-45). CONCRASH prunes the combination \( \langle \delta_p, \delta_{s_1}, \delta_{s_2} \rangle \) according to different strategies. If the test case is neither redundant (line 31) nor irrelevant (lines 29-32) CONCRASH updates the coverage repository \( C \) (line 34), assembles a new concurrent test case \( t \) (line 35) and in-
vokes the Interleaving Explorer component to determine if the interleaving space of $t$ contains at least one interleaving that can reproduce the failure (line 36). If this is the case (line 37), CONCrASH produces the output $t$ and its failure-inducing interleaving and terminates (line 38).

If CONCrASH does not terminate, it checks if $\delta_{p,s_2}$ should be added to the pendingSeqs list for further extensions (line 39), that is, CONCrASH checks whether the state $S(\delta_{s_2})$ produced by executing $\delta_{p,s_2}$ either throws an exception or has been already explored. If not, CONCrASH adds $S(\delta_{s_2})$ to $S$ and inserts $\delta_{p,s_2}$ in the pendingSeqs of the next level (line 40). Following previous work \[95, 104, 132\], CONCrASH does not extend sequences that throw exceptions when executed sequentially, as all of their extensions throw the same exception at the same point \[95\]. CONCrASH adds the current $\delta_{p,s_2}$ either at the beginning or at the end of pendingSeqs[$level + 1$] depending on the priority of the sequence (lines 42 and 44, respectively). Sequences at the beginning of the list have higher priority, as they will be consumed earlier by CONCrASH (line 15).

5.3.4 Pruning Strategies

We now describe in details both the pruning strategies and the decision procedure that determines whether $\delta_{p,s_2}$ should be added at the beginning or at the end of pendingSeqs[$level + 1$].

PS-Exception

CONCrASH prunes a combination $\langle \delta_p, \delta_{s_1}, \delta_{s_2} \rangle$ if either $\delta_{p,s_1}$ or $\delta_{p,s_2}$ throw an exception when executed sequentially, even if the exception matches the crash stack trace in input.

CONCrASH prunes method call sequences that throw an exception when executed sequentially since CONCrASH targets failures that can only be reproduced during concurrent executions. The occurrence of an exception likely indicates that CONCrASH has generated an illegal method call sequence, which are more likely to be generated than legal ones \[7\]. It is a standard practice for concurrent test case generation to prune method call sequences that generate an exception when executed sequentially \[92, 104, 130, 132\]. For example in Figure 5.6 CONCrASH prunes the combinations with $\delta_{s_2} = \delta_C$ by applying PS-Exception:

\footnote{Similarly with previous work we are assuming that 1) sequential executions are deterministic given the same inputs \[104, 132\]. 2) the CUT does not spawn new threads \[130, 132\], which is generally the case for concurrent libraries \[104\].}
public synchronized void removeHandler(Handler handler) {
    if (!anonymous) {
        manager.checkAccess();
    }
    if (handler == null) {
        throw new NullPointerException();
    }
    if (handlers == null) {
        return;
    }
    handlers.remove(handler);
}

Figure 5.8. Method RemoveHandler of class Logger

the method removeHandler is invoked by passing null as parameter, which
leads to a NullPointerException in the sequential execution of the method
call sequence (line 6 in Figure 5.8, which reports the implementation of the
removeHandler method).

PS-Stack

CONCRASH prunes a combination \( \langle \delta_p, \delta_{s1}, \delta_{s2} \rangle \) if \( \exists e \in \mathcal{M}(\delta_{p,s1}) : \text{stack}(e) = \text{CST} \) (Crash Stack Trace), where \( \text{stack}(e) \) is the call stack trace of \( e \), obtained by
analyzing the method entry and exit points in \( \mathcal{M}(\delta_{p,s1}) \).

A necessary condition of a test case to reproduce a failure is to reach the point
of failure (POF) with the same calling context of the considered crash stack [62].
CONCRASH prunes the call sequences \( \delta_{s1} \) that when executed sequentially do not
reach the POF with the same call stack of CST. For example in Figure 5.6, CON-
CRASH prunes the combinations with \( \delta_{s1} = \delta_B \) since the sequential execution of
\( \delta_B \) does not reach the POF (line 34 in Figure 5.1). Indeed, when executing \( \delta_B \)
the filter field is not initialized, thus the condition at line 33 in Figure 5.1 evalu-
ates false and the POF is not executed. On the contrary, the combinations with
\( \delta_{s1} = \delta_O \) will not be pruned: by invoking method setFilter with a reference to
an instance of Filter class, the condition at line 33 in Figure 5.1 evaluates true
and the sequential execution of \( \delta_{p,s1} \) will always reach the POF with the same
call stack of CST.

PS-Redundant

CONCRASH prunes a combination \( \langle \delta_p, \delta_{s1}, \delta_{s2} \rangle \) if \( (\mathcal{M}(\delta_{p,s1}), \mathcal{M}(\delta_{p,s2})) \in \mathcal{C} \).
CONCRAFT prunes the combinations whose concurrent suffixes induce an already observed pair of sequential coverages $\overline{M}(\delta_{p,s_1})$ and $\overline{M}(\delta_{p,s_2})$, as inferred from the coverage repository $C$. CONCRAFT prunes redundant pairs of sequential coverage since the resulting concurrent test case would lead to an interleaving space already explored with a previously generated test case [132]. For example in Figure 5.6, CONCRAFT prunes the combination with $\delta_{s_1} = \delta_B$ and $\delta_{s_2} = \delta_E$ because the combination leads to the same sequential coverage of the previously explored combination with $\delta_{s_1} = \delta_B$ and $\delta_{s_2} = \delta_E$: invoking method setFilter by passing null ($\delta_E$) or by passing a reference to an instance of class Filter ($\delta_F$) leads to the same sequential execution of the method, and thus to the same sequential coverage.

PS-Interfere

CONCRAFT prunes a combination $\{\delta_p, \delta_{s_1}, \delta_{s_2}\}$ if $\exists e_1, e_2 \in \overline{M}(\delta_{p,s_1}) \times \overline{M}(\delta_{p,s_2}) : e_1 = R(f), e_2 = W(f)$

CONCRAFT prunes the combinations in which the two concurrent suffixes do not access the same variables or the interfering method $\delta_{p,s_2}$ does not write any variable read by the crashing method $\delta_{p,s_1}$. The intuition behind this pruning strategy is that for triggering a concurrency failure the interfering method must interact with the crashing method by writing a shared variable that is accessed by the crashing method itself. For example, in Figure 5.6, CONCRAFT prunes the combination with $\delta_{s_1} = \delta_B$ and $\delta_{s_2} = \delta_P$ because the sequential execution of $\delta_P$ does not write any variable read by $\delta_B$. On the contrary, PS-Interleave does not prune the combination with $\delta_{s_1} = \delta_B$ and $\delta_{s_2} = \delta_F$ as the crashing method info reads variable filter, and the candidate interfering method setFilter writes the same field. CONCRAFT considers the object fields not only of the CUT but also of the auxiliary classes, since the object fields of the non-primitive fields of SOUT are also shared across threads.

PS-Interleave

CONCRAFT prunes a combination $\{\delta_p, \delta_{s_1}, \delta_{s_2}\}$ if $\forall e_1, e_2 \in \overline{M}(\delta_{p,s_1}) \times \overline{M}(\delta_{p,s_2}) : e_1 = RW, e_2 = RW, LH(e_1) \cap LH(e_2) = \emptyset$, where $LH$ denotes the lock history of an event $e_x \in \overline{M}(\delta)$, defined as the set of locks that are acquired but never released in $\overline{M}(\delta)$ before triggering the event $e_x$, and $RW$ denotes either a read or write memory access. More formally, $LH(e_x) = \{ l \mid \exists l_j = ACQ(l), e_k = REL(l) \in \overline{M}(\delta), w < k < x \}$, where $w$ is the index of the first event in $\overline{M}(\delta)$. 
CONCRASH prunes a combination if the two concurrent suffixes cannot interleave when assembled in a concurrent test case, and thus their concurrent execution cannot lead to a concurrency failure [50]. Differently from traditional lockset analysis (LS) [116], 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, but cannot infer if two executions can interleave.

For instance in the sequential coverage of the threads reported in Figure 5.9, 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 $e_2$, $e_5$, 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 $LH(e_5) \cap LH(e_8) = \emptyset$.

CONCRASH relies on PS-Interfere to decide how to prioritize $\delta_{p,s_2}$ in the list (lines 42 and 44). If $M(\delta_{p,s_2})$ interferes with $M(\delta_{p,s_1})$ (see Condition PS-Interfere), CONCRASH adds $\delta_{p,s_2}$ at the beginning of the list, otherwise at the end, since if $M(\delta_{p,s_2})$ writes the same variables accessed by the crashing method, $\delta_{p,s_2}$ is more likely to lead to a program state that could make the crashing method behave differently if executed in this new state. In such situation, $\delta_{p,m}$ is added at the beginning of the list, otherwise at the end. This is only a prioritization, not a pruning strategy.

5.4 Interleaving Explorer

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

We investigated current techniques to identify an approach that can be effectively exploited in the CONCRASH Interleaving Explorer. Current techniques to explore the interleaving space of a given test case examine the space either
exhaustively or selectively. Techniques that exhaustively explore all possible interleavings \cite{35} can be very expensive due to enormous size of interleaving spaces \cite{140}. Techniques that explore interleaving spaces selectively, based on particular classes of concurrency faults, like data races \cite{87, 94}, atomicity violations \cite{41, 43, 97, 148}, and deadlocks \cite{21, 22, 39} can be efficient, but may miss failure-inducing interleavings that do not belong to the considered class of the concurrency faults.

We selected CORTEX, a technique for reproducing concurrency failures \cite{84}, which is more efficient than exhaustive exploration of interleavings spaces, and does not make any assumptions on the type of concurrency fault. In a nutshell, the CONCRASH Interleaving Explorer executes the given test case, collects an execution trace in which the shared variables are treated as symbols, and builds and solves an SMT formula \cite{33} the solutions of which, 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 the stack trace indicates both the point of failure (POF) and the type of runtime exception. Below we describe in more details how the CONCRASH Interleaving Explorer works, by discussing how the Interleaving Explorer collects the symbolic trace \textit{5.4.1} and explores the interleavings through constraint solving \textit{5.4.2}. For a detailed description of CORTEX, the interested readers can refer to the seminal work \cite{61} and its extensions \cite{83, 84}.

\section{5.4.1 Symbolic Trace Collection}

CONCRASH symbolically collects traces in three main steps: \textit{static program analysis}, \textit{concrete trace collection}, and \textit{symbolic trace generation}.

\textit{Static Program Analysis.}

The Interleaving Explorer statically analyzes the class-under-test CUT and the test case in input. The static analysis instruments the program to identify interleaving-dependent variables. It instruments the program by inserting probe code at the beginning of each basic block of the program to trace the thread path profiles. The path profile is the sequence of basic blocks executed by each thread during concrete executions, and encodes the control-flow outcomes of each thread during the execution.

CONCRASH identifies the interleaving-dependent variables by determining all the program variables whose value can change according to the specific executed interleaving. These variables correspond to both non-private variables of the program and local variables that are data-dependent to non-private variables.
5.4 Interleaving Explorer

The variables identified by the analysis will be considered as symbolic variables in the subsequent symbolic trace generation step.

**Concrete Trace Collection.** The Interleaving Explorer executes the input test case on the instrumented version of the program produced with the static analysis. CONCrASH executes the test case with the default scheduler, thus with random interleaving. The execution of the instrumented program produces a concrete trace with the path-profile of each thread.

**Symbolic Trace Generation.** The Interleaving Explorer symbolically executes the program to generate a symbolic trace. It executes the program symbolically driven by both the concrete trace generated in the previous step and the interleaving-dependent variables identified with static analysis. When symbolically executing the program, the Interleaving Explorer (i) manipulates symbolic expressions rather than concrete values when accessing the variables identified as interleaving-dependent, (ii) drives the symbolic execution of each thread with the recorded path-profiles thus symbolically executing only the corresponding control-flow paths, and (iii) records each synchronization operation (lock, join, fork) and access to symbolic variables into a symbolic trace.

5.4.2 Computing Failing Interleavings with Constraint Solving

The Interleaving Explorer formulates the problem of checking if the interleaving space of a given test case contains an interleaving leading to the violation of an assertion as an SMT (Satisfiability Modulo Theories) problem \[33\]. Starting from the symbolic trace, the Interleaving Explorer builds an SMT formula whose solutions (if any) identify interleavings that violate the assertion and reproduce the crash stack trace in input.

The SMT formula contains both order and value variables. Order variables encode the relative order among the operations in the symbolic trace, while Value variables encode the values that each read operation can return depending on the interleaving. The SMT formula models the possible interleavings of the symbolic trace, by constraining the value of the order variables, the possible values that each read operation can read depending on the executed interleaving, and the bug condition, which is the condition to trigger the failure to reproduce.

The SMT formula generated by the Interleaving Explorer is a conjunction of five sub-formulas:

\[
\Phi = \phi_{\text{path}} \land \phi_{\text{mo}} \land \phi_{\text{po}} \land \phi_{\text{lock}} \land \phi_{\text{rw}} \land \phi_{\text{fault}}
\]
Path Constraints ($\phi_{\text{path}}$) model the path conditions collected during the symbolic execution. Path constraints predicate on value variables, and constrain the value returned by read operations according to the condition of the branch taken by the thread during symbolic execution.

Memory Order Constraints ($\phi_{\text{mo}}$) involve order variables, and model the total order among operations within a thread according to the considered memory model. CONCRASH assumes a sequential consistent memory model [73] which ensures that operations within a thread follow a sequential order.

Partial Order Constraints ($\phi_{\text{po}}$) encode the order among operations enforced by the fork and join synchronization mechanisms. In particular, a fork operation always happens before the start operation of the forked thread, and a join operation always happens after the exit operation of the joined thread.

Locking Constraints ($\phi_{\text{lock}}$) model the mutual exclusion among critical sections protected by the same lock. For a given critical section and the corresponding lock and unlock operations, they model that between the two operations there cannot be other lock and unlock operations on the same object.

Read-Write Constraints ($\phi_{\text{rw}}$) encode the possible values that each read operation can return depending on the executed interleaving. Each read operation returns the value written by the last write operation on the variable, but the order of the write operations can change from one execution to another. The read-write constraints map to each value variable the values it can return, and for each value the order among the write operations that must hold.

Fault Constraint ($\phi_{\text{fault}}$) encodes the condition that, if satisfied, leads to the reproduction of the fault and corresponds to the negation of the assertion to violate. For instance, in the case of a NullPointerException, the fault constraint forces the value variable corresponding to the read operation at the Point of Failure (POF) to return the null value.

CONCRASH determines the satisfiability of the SMT formulas with an SMT solver, to determine if the test cases leads to a failure, and to infer a sequence of interleaving that led to the failure. If there exists an assignment of variables that makes the formula satisfiable, the failure can be reproduced and any solution of the formula represents an interleaving which reproduces the failure. If the formula is satisfiable, CONCRASH returns the test case and the interleaving as final result. Otherwise, CONCRASH iterates the Test Case Generator for producing a new test case to be explored. The process terminates when either the failure is successfully reproduced or the time budget expires.
Chapter 6

Evaluation

In this chapter we report the results of a set of experiments that we conducted to assess the effectiveness of CONCRASH in reproducing concurrency failures from the limited information contained in crash stack traces.

Our experiments consist in running a prototype implementation of CONCRASH on a set of ten subject programs with known thread safety violations, and calculating the effectiveness of our approach in terms of success rate, time to reproduce the failure, number of test cases generated before reproducing the failure, and the size of the generated failure-inducing test cases.

To evaluate the individual contribution of the pruning strategies of CONCRASH, we repeat the experiments by using a version of our approach with a single pruning strategy enabled, for each pruning strategy. We then compare these measurements with the results obtained by running the original implementation of CONCRASH with all pruning strategies enabled.

Finally, we compare the effectiveness of CONCRASH with state-of-the-art approaches that generate concurrent test cases for testing concurrent programs to assess the ability of CONCRASH to drive the generation of test cases towards a specific failure by exploiting the information contained in crash stack traces.

In this chapter we (i) present the research questions addressed by our experimental evaluation, describe the experimental setting, and in particular the prototype implementation of our approach, the subject programs used for the evaluation and the setup of the experiments, (ii) discuss the results of our experiments with reference to each research question, and (iii) identify the threats to the validity of our results and indicate the countermeasures that we adopted to mitigate their impact.
6.1 Research Questions

Our experimental evaluation addresses the following research question:

**RQ\textsubscript{1}** How effective is CONCRASH in reproducing concurrency failures?

**RQ\textsubscript{2}** What is the contribution of each pruning strategy in reducing the search space?

**RQ\textsubscript{3}** Is CONCRASH more effective than competing state-of-the-art testing approaches?

**RQ\textsubscript{1}** investigates the effectiveness of CONCRASH in reproducing concurrency failures from crash stack traces. To answer this research question, we collect different metrics: success rate, failure reproduction time and size of the failure-inducing test cases. We measure the success rate of our approach as the ability of CONCRASH to successfully reproduce a given failure within a given time budget. We measure the time required by the technique to reproduce the failure. Since the time required by CONCRASH to reproduce a failure depends on the specific machine used for the experiments and on the time required by the interleaving explorer to explore the interleaving space of the generated test cases, we also measure how many test cases CONCRASH generates before reproducing the failure. We measure the size of the failure-inducing test cases generated by CONCRASH: since smaller a test case is, the easier localizing and understanding the failure is, we expect CONCRASH to generate small test cases.

**RQ\textsubscript{2}** questions the contribution of each pruning strategy in reducing the search space of possible test cases that CONCRASH needs to explore. We answer this research question by measuring the effectiveness of CONCRASH with a single pruning strategy enabled, for each pruning strategy. We then compare these measurements with a version of CONCRASH with no pruning strategy enabled and the original implementation of CONCRASH, with all pruning strategies enabled. Our experiments aim to confirm that, on one hand, each pruning strategy in isolation effectively reduce the search space, on the other hand, that the effectiveness of CONCRASH is given by the synergetic combination of all pruning strategies.

**RQ\textsubscript{3}** investigates the effectiveness of CONCRASH with respect to state-of-the-art test case generation approaches for concurrent systems. Testing techniques are failure-oblivious, that is, they aim to find concurrency failures rather than reproducing a given one. As CONCRASH is driven by failure information, we expect a large saving in the time required to identify the failure-inducing test case that reproduces the given failure. To answer this research question, we compare the effectiveness of CONCRASH with the effectiveness of two state-of-the-art testing techniques for concurrent programs.
6.2 Experimental Setting

Prototype

We experimented with a prototype Java implementation of ConCrash that implements the algorithm presented in Figure 5.5. The prototype requires as input the source code of the thread-safe class under test and the crash stack trace of the failure to reproduce. Once successfully reproduced the failure, the prototype generates as output the failure-inducing test case, in the form of an executable Java class, and the failure-inducing interleaving, which is encoded in a textual file reporting the ordered sequence of instructions to execute to reproduce the failure. The output produced by ConCrash allows to easily debug and localize the concurrency fault that causes the given failure.

We built the prototype test case generator on top of AutoConTest [132], a concurrent test case generator developed by Terragni and Cheung based on the sequential coverage metric. We configure AutoConTest to implement the ConCrash algorithm: (i) we tune the test case generation step to generate only test cases with the crashing method as first parallel suffix, as described in Section 5.3; (ii) we apply ConCrash pruning strategies after generating each candidate test case, to determine whether the test case is either redundant or irrelevant, as described in Section 5.3; (iii) we invoke the ConCrash interleaving explorer described in Section 5.4 for each test case that is not retained by the ConCrash pruning strategies.

The prototype interleaving explorer relies on Cortex, an interleaving exploration tool developed by Machado et al. [84]. Cortex leverages the Soot instrumentation framework [135] to perform static program analysis, Java PathFinder (JPF) [138] for symbolic execution, and Z3 [33] as constraint solver. We extended Java PathFinder to support the bytecode instructions that check for equality and inequality between references. These instructions are not supported by the publicly available version of the tool and are required to reproduce some of the failures we considered in our evaluation.

Overall, the ConCrash prototype includes around 40 thousands lines of code. The Test Case Generator, which includes the tuned version of AutoConTest with our pruning strategies, is composed of 300 Java classes and amounts to around 27 thousands lines of code. The Interleaving Explorer, which relies on the original implementation of Cortex, is implemented by combining C++ and Java code. In particular, the Interleaving Explorer is composed of 14 C++ and 75 Java classes, for a total amount of around 13 thousands lines of code.

To answer RQ2, which investigates the effectiveness of the pruning strate-
ties adopted by CONCRASH, the prototype can be easily configured to enable or disable the different pruning strategies. 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 it is not our contribution, thus we are not interested in evaluating its effectiveness in isolation.

Competing approaches

We compared CONCRASH with CONTeGe [104] and AUTOCONTEST [132], two representative state-of-the-art approaches that generate concurrent test cases for testing concurrent programs. We use the publicly available versions of AUTOCONTEST and CONTeGe for our experiments.

CONTeGe randomly generates sequential call sequences and randomly combines them in concurrent test cases. It adopts a feedback-directed approach [95], which consists in executing each generated call sequence and discarding those that throw exceptions. CONTeGe relies on stress testing for exploring interleavings, which amounts to execute a test case several times, aiming to observe different interleavings. Stress testing does not offer any guarantee of observing a given portion of the interleaving space, and does not introduce any mechanism to improve the probability of executing new interleavings. CONTeGe uses linearizability [53] as a test oracle. A linearizability oracle reports a violation whenever a thread interleaving produces a suspicious behavior that cannot be produced by any sequential test execution where all methods are executed atomically [92].

AUTOCONTEST is a coverage-based test case generation technique. In line with other coverage-based techniques for concurrent systems, AUTOCONTEST relies on interleaving coverage criteria which identify properties that an interleaving must satisfy to be considered as a coverage requirement. More specifically, AUTOCONTEST relies on the sequential coverage (see Section 5.2). It explores the interleaving space of each generated test case with a dynamic detector of atomicity violations and outputs the generated test cases and interleavings that lead to a system crash.

Both CONTeGe and AUTOCONTEST are failure-oblivious, that is, they are designed to detect and find concurrency failures rather than to reproduce a given one. In absence of techniques that generate test cases for reproducing a given failure, we use CONTeGe and AUTOCONTEST as baseline to assess the ability of CONCRASH to drive the generation of test cases 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 \cite{92, 104, 115, 132}. 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 case documented in related work \cite{104, 132}. We added the program assertions encoding the failures, as required by the Interleaving Explorer (CORTEX \cite{84}). We easily inserted such assertions relying on the POF and the type of the thrown exception reported in the crash stack.

Tables 6.1 and 6.2 provide details about the subject programs and failures, respectively: column \textit{ID} introduces a unique identifier that we use in the paper to identify the subject program; column \textit{Class Under Test (CUT)} is the class under test of the subject program; columns \textit{Version} and \textit{Code Base} report the version and the code base that contains the faulty class, respectively; columns \textit{Total SLOC} and \textit{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 \textit{# Methods} indicates the number of public methods of the CUT; columns \textit{Issue ID}, \textit{Fault Type}, \textit{Type of Exception} and \textit{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.

The subjects are taken from five Java code bases. Apache Commons DBCP is a popular library to create and manage pools of database connections. Apache Commons Math is a library that provides a series of self-contained mathematics and statistics components. Java JDK is the development environment of Java, subjects with ID 4 and 6 belong to the java.io package, while the subject with ID 5 belongs to the java.util.logging package. JFreeChart is a popular library for displaying various types of charts. Log4j is one of the most popular Java-based logging utility.

Experimental Setup

We run each technique on all the subjects. Each experiment terminates either when the technique successfully reproduces the failure or exhausts 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
### Table 6.1. ConCrash Evaluation Subjects

<table>
<thead>
<tr>
<th>ID</th>
<th>Class Under Test (CUT)</th>
<th>Version</th>
<th>Code Base</th>
<th>Total SLOC</th>
<th>CUT SLOC</th>
<th># Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PerUserPoolDataSource</td>
<td>1.4</td>
<td>Commons DBCP</td>
<td>9,451</td>
<td>719</td>
<td>68</td>
</tr>
<tr>
<td>2</td>
<td>SharedPoolDataSource</td>
<td>1.4</td>
<td>Commons DBCP</td>
<td>9,451</td>
<td>546</td>
<td>44</td>
</tr>
<tr>
<td>3</td>
<td>IntRange</td>
<td>2.4</td>
<td>Commons Math</td>
<td>18,016</td>
<td>278</td>
<td>44</td>
</tr>
<tr>
<td>4</td>
<td>BufferedInputStream</td>
<td>1.1</td>
<td>Java JDK</td>
<td>3,791</td>
<td>304</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>Logger</td>
<td>1.4.1</td>
<td>Java JDK</td>
<td>2,193</td>
<td>528</td>
<td>45</td>
</tr>
<tr>
<td>6</td>
<td>PushbackReader</td>
<td>1.8</td>
<td>Java JDK</td>
<td>11,562</td>
<td>143</td>
<td>13</td>
</tr>
<tr>
<td>7</td>
<td>NumberAxis</td>
<td>0.9.12</td>
<td>JFreeChart</td>
<td>64,713</td>
<td>1,662</td>
<td>119</td>
</tr>
<tr>
<td>8</td>
<td>XYSeries</td>
<td>0.9.8</td>
<td>JFreeChart</td>
<td>51,614</td>
<td>200</td>
<td>28</td>
</tr>
<tr>
<td>9</td>
<td>Category</td>
<td>1.1</td>
<td>Log4j</td>
<td>10,773</td>
<td>387</td>
<td>43</td>
</tr>
<tr>
<td>10</td>
<td>FileAppender</td>
<td>1.2</td>
<td>Log4j</td>
<td>10,273</td>
<td>185</td>
<td>13</td>
</tr>
</tbody>
</table>

### Table 6.2. ConCrash Evaluation Failures

<table>
<thead>
<tr>
<th>ID</th>
<th>Issue ID</th>
<th>Fault Type</th>
<th>Type of Exception</th>
<th>Crash Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>369</td>
<td>Race</td>
<td>ConcurrentModificationException</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>369</td>
<td>Race</td>
<td>ConcurrentModificationException</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>481</td>
<td>Atom.</td>
<td>AssertionError</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4225348</td>
<td>Atom.</td>
<td>NullPointerException</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>4779253</td>
<td>Atom.</td>
<td>NullPointerException</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>8143394</td>
<td>Atom.</td>
<td>NullPointerException</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>806667</td>
<td>Atom.</td>
<td>IllegalArgumentException</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>187</td>
<td>Race</td>
<td>ConcurrentModificationException</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>1507</td>
<td>Atom.</td>
<td>NullPointerException</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>509</td>
<td>Atom.</td>
<td>NullPointerException</td>
<td>2</td>
</tr>
</tbody>
</table>
space containing failure inducing test case could be explored before or after, de-
pending 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 cases, we repeat each experiment five times using
different random seeds, in line with the experimental setup of related work \cite{29,104,132}. 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 $B$, 0 otherwise.

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

**Failure-inducing Test ID (FTID)** that is the ID of the first failure-inducing test
case. The IDs are assigned in ascending order. FTID = $n$ indicates that
CONCRASH explored the interleaving space of $n$ test cases before reproduc-
ing the failure. A low value of FTID indicates that the technique is effective
in generating a failure-inducing test case. If the failure is not reproduced
within $B$ ($FR = 0$), we underapproximate FTID to the, ID of the last gen-
erated test case.

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

### 6.3 Experimental Results

In this section, we report the results of our experimental evaluation to answer
the research questions presented in Section 6.1

#### 6.3.1 RQ1 - Effectiveness

Our first research question investigates the effectiveness of CONCRASH to re-
produce concurrency failures from crash stack traces. To answer this research
Table 6.3. RQ1: ConCrash Effectiveness Results

<table>
<thead>
<tr>
<th>ID</th>
<th>FR</th>
<th>FRT (sec.)</th>
<th>FTID</th>
<th>FTS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>%</td>
<td>min</td>
<td>max</td>
<td>avg</td>
</tr>
<tr>
<td>1</td>
<td>100%</td>
<td>27</td>
<td>99</td>
<td>63</td>
</tr>
<tr>
<td>2</td>
<td>100%</td>
<td>22</td>
<td>85</td>
<td>42</td>
</tr>
<tr>
<td>3</td>
<td>100%</td>
<td>10</td>
<td>16</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>100%</td>
<td>10</td>
<td>21</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>100%</td>
<td>39</td>
<td>84</td>
<td>70</td>
</tr>
<tr>
<td>6</td>
<td>100%</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>100%</td>
<td>27</td>
<td>31</td>
<td>30</td>
</tr>
<tr>
<td>8</td>
<td>100%</td>
<td>26</td>
<td>185</td>
<td>107</td>
</tr>
<tr>
<td>9</td>
<td>100%</td>
<td>17</td>
<td>33</td>
<td>25</td>
</tr>
<tr>
<td>10</td>
<td>100%</td>
<td>23</td>
<td>183</td>
<td>92</td>
</tr>
<tr>
<td>AVG</td>
<td>100%</td>
<td>21</td>
<td>74</td>
<td>46</td>
</tr>
</tbody>
</table>

question, we ran the ConCrash prototype on our benchmark and collected a set of metrics as described in Section 6.2.

Table 6.3 reports the experimental results about the effectiveness of ConCrash in reproducing concurrency failures. The table reports the aggregated results of the five runs for each subject. For the Failure Reproduction Time (FRT) and Failure-inducing Test ID (FTID) metrics, the table shows the minimum, the average, and the maximum value that the prototype achieves on each subject in the five runs. Moreover, we report the standard deviation and the 95% confidence interval.

Column FR, Failure Reproduction, indicates that ConCrash reproduces all the concurrency failures in all the five runs. We inspected every failure-inducing test case and interleaving generated by ConCrash to check whether they lead to the same crash stack trace given as input. In particular, we executed the failure-inducing test cases augmented with the specified failure-inducing interleaving. The execution of the failure-inducing interleaving was achieved by manually inserting sleep statements in the test cases. Overall, all the executions led to the same failure and crash stack trace given as input. Thus, ConCrash correctly reproduces the given failures and reports to developers the required information to debug and fix the fault.

Columns FRT, Failure Reproduction Time, indicate a time for reproducing
the failures that ranges from seven seconds in the best case to 185 seconds in the worst case, with an average of 46 seconds. We also measured the time for generating test cases and exploring interleavings with CONCRASH, and we notice that CONCRASH spends 1% of the time for generating test cases and 99% of the time for exploring interleavings, on average on all fifty runs. This result indicates the effectiveness of CONCRASH test case generator in identifying failure-inducing test cases.

Columns FTID, Failure Inducing Test ID, indicate that CONCRASH explored the interleaving space of a minimum of 1 test case and a maximum of 15 test cases, with an average of 3 test cases. We check that such low value of explored test cases is indeed due to the effectiveness of CONCRASH pruning strategies, by repeating the measurements on a version of CONCRASH with all the pruning strategies disabled, and observing an average FTID of 147 test cases (the results are reported in Table 6.4). This means that CONCRASH pruning strategies prune 144 test cases on average, thus significantly saving time for reproducing a failure.

Column FTS, Failure-inducing Test Size, reports the size of the generated test cases that ranges from 3 to 10 outermost method calls, on average. As described in Section 5.3, short test cases are preferable since are easier to understand and debug. The low value of FTS in our experiments is achieved with the Breadth-First Search exploration strategy.

The results indicate the effectiveness of CONCRASH. The approach reproduces all the considered failures on all the fifty runs within a reasonable time, with test cases 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 case generation through a failure-inducing test case is the main efficiency factor. CONCRASH generates three test cases on average and at most 15 test cases in the worst cases to identify a failure-inducing test case.

6.3.2 RQ2 - Pruning Strategies

Our second research question investigates the effectiveness of the pruning strategies that CONCRASH implements to reduce the space of test cases that CONCRASH needs to further explore. To answer this research question, we compare the effectiveness of the CONCRASH prototype with all the pruning strategies disabled with the effectiveness of CONCRASH with a single pruning strategy enabled, for each pruning strategy.

The rightmost columns of Table 6.4 report the experimental results about the effectiveness of the different pruning strategies. The table reports the Failure
Table 6.4. RQ2: ConCrash Pruning Strategies Effectiveness Results

<table>
<thead>
<tr>
<th>ID</th>
<th>NO-Pruning</th>
<th>PS-Stack</th>
<th>PS-Redundant</th>
<th>PS-Interfere</th>
<th>PS-Interleave</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>%</td>
<td>avg</td>
<td>%</td>
<td>avg</td>
<td>%</td>
</tr>
<tr>
<td>1</td>
<td>20%</td>
<td>15,456</td>
<td>376</td>
<td>100%</td>
<td>526</td>
</tr>
<tr>
<td>2</td>
<td>80%</td>
<td>9,240</td>
<td>250</td>
<td>100%</td>
<td>362</td>
</tr>
<tr>
<td>3</td>
<td>100%</td>
<td>204</td>
<td>13</td>
<td>100%</td>
<td>157</td>
</tr>
<tr>
<td>4</td>
<td>100%</td>
<td>77</td>
<td>7</td>
<td>100%</td>
<td>64</td>
</tr>
<tr>
<td>5</td>
<td>100%</td>
<td>6,520</td>
<td>491</td>
<td>100%</td>
<td>2,576</td>
</tr>
<tr>
<td>6</td>
<td>100%</td>
<td>33</td>
<td>3</td>
<td>100%</td>
<td>20</td>
</tr>
<tr>
<td>7</td>
<td>100%</td>
<td>508</td>
<td>11</td>
<td>100%</td>
<td>294</td>
</tr>
<tr>
<td>8</td>
<td>100%</td>
<td>2,758</td>
<td>269</td>
<td>100%</td>
<td>166</td>
</tr>
<tr>
<td>9</td>
<td>100%</td>
<td>348</td>
<td>28</td>
<td>100%</td>
<td>267</td>
</tr>
<tr>
<td>10</td>
<td>100%</td>
<td>540</td>
<td>22</td>
<td>100%</td>
<td>487</td>
</tr>
<tr>
<td>AVG</td>
<td>90%</td>
<td>3,569</td>
<td>147</td>
<td>100%</td>
<td>492</td>
</tr>
</tbody>
</table>

Reproduction (columns FR), the Failure Reproduction Time (columns FRT) and the Failure-inducing Test ID (columns FTID) for the different pruning strategies. We do not report the Failure-inducing Test Size, as we did not record significant modifications 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 cases. 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 cases 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 cases 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 cases. 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 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 the effectiveness of CoNCrash derives from the synergetic combination of all pruning strategies.

The diagrams in Figure 6.1 show that CoNCrash outperforms each pruning strategy in isolation. The diagram in Figure 6.1(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 6.1(b) plots the success rate of the first TOP-N test cases generated with CoNCrash and with the different pruning strategies. CoNCrash achieves more than 90% of failure success rate with only 10 test cases, and 100% within the first 25 ones. PS-Stack achieves 100% of failure success rate within the first 100 test cases, while all the other pruning strategies never achieves 100% within 100 test cases.
Figure 6.1. Aggregate comparison of the ConCrash pruning strategies for the ten subjects
6.3.3 RQ3 - Comparison with Testing Approaches

Our third research question compares the effectiveness of CONCRAsh with the state-of-the-art approaches for generating test cases for concurrent programs. To answer this research question we ran CONTeGe and AUTOConTest on our benchmark.

Table 6.5 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 of Table 6.3 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 cases, 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 in-
terleaving (atomicity violation) with inputs that do not trigger the failure. The effectiveness of AutoContTest is comparable with ConCrash in the cases it succeeds, but AutoContTest generates much larger test cases than ConCrash.

Our results suggest that testing techniques that are designed as failure-obllivious are not effective in reproducing a specific concurrency failure, as they generate many test cases that are irrelevant for the specific failure. On the other hand, ConCrash leverages crash stacks and novel pruning strategies to effectively drive the test case generation towards a failure-inducing test case.

We conclude the comparison observing that in many experiments ConTeGE and AutoContTest exposed failures with crash stacks different from the one we considered as input, and these failures may or may not be triggered by the same concurrency fault that triggers the failure considered in the experiments. These experiments aim to compare the effectiveness of the different approaches in reproducing a given concurrency failure, and not the fault-detection effectiveness of the approaches. This result confirms that state-of-the-art testing techniques can be in general effective in detecting concurrency failures, but they are not suitable for reproducing a specific concurrency failure.

6.4 Limitations

In this section, we discuss the current limitation of ConCrash, highlighting their impact and nature.

ConCrash automatically reproduces concurrency failures that manifest as runtime exceptions and produce a crash stack trace. Thus ConCrash does not reproduce deadlocks, which do not generate a crash stack trace when they occur. ConCrash can be extended to reproduce also deadlocks by integrating dedicated runtime monitors that generate a crash stack trace when a deadlock is detected.

ConCrash generates concurrent test cases composed of exactly two concurrent suffixes, following a recent empirical study that shows that most of the concurrency failures can be exposed by interleaving two execution flows only [81]. This means that ConCrash cannot reproduce failures that can be exposed only with the interleaving of more than two threads. ConCrash can be easily extended to overcome such limitation, albeit with some performance issues.

ConCrash could fail in generating a test case that reproduces a failure for failures that can be exposed only with very specific input parameters of the methods comprising the test case. ConCrash chooses the input parameters from a pool of randomly generated values, and this pool will unlikely contain the specific value required to trigger the failure.
Finally, *CONCrASH* may not be very effective when a failure can be triggered only with a complex state of the class-under-test, which requires a sequential prefix composed of several method calls to be reached. Since *CONCrASH* explores the space of possible test cases by using a breadth-first search strategy, *CONCrASH* requires much time to generate and explore test cases with long sequential prefixes.

### 6.5 Threats to Validity

In this section we briefly discuss the threats that could affect the validity of our empirical evaluation and the countermeasures that we adopted to mitigate such threats.

One important threat to the validity of the results presented in this dissertation concerns the correctness of the prototype implementation and the results it produces. To mitigate this threat, we extensively tested the prototype and we manually checked that each failure-inducing test case and interleaving generated by *CONCrASH* were actually able to reproduce the input failure by executing them in a controlled environment. We released the prototype and evaluation results to help replicating our empirical results \[45\].

The results of our empirical evaluation may also be biased by the experimental setting, and in particular by the order in which *CONCrASH* explores the space of possible test cases. Although *CONCrASH* systematically explores such space, the order of exploration is arbitrary, and different runs of *CONCrASH* can lead to different results as portions of the search space containing failure-inducing test cases could be explored before or after, depending on such order. In the *CONCrASH* prototype we implemented the order of exploration is pseudodeterministic, since it depends on an input seed. To mitigate threats that may derive from the order of exploration of the space of possible test cases, in our empirical evaluation we repeated each experiment five times using different random seeds, simulating different orders of exploration. We did not reveal significant differences in the results produced by *CONCrASH*.

Another threat to the validity of our empirical evaluation is that our results may not generalize on other subjects, as the number of subject classes is rather limited. We mitigated this threat by selecting real world concurrency failures, which have been confirmed and fixed by developers. The considered concurrency failures are taken from different five different popular code bases. We included subjects with a wide range of LOC, depth of the crash stack trace, and type of runtime exception they throw.
6.5 Threats to Validity
Chapter 7

Conclusion

This thesis addresses the problem of automatically exposing concurrency failures, that is, the highly relevant and widely investigated problem of determining the program conditions that lead to system failures. In the case of concurrency failures, such conditions can be defined as pairs of a test case and an interleaving that jointly trigger the failure, since the behavior of concurrent systems depends on both the behavior of the single threads and their interleaving.

This thesis contributes to the problem of exposing concurrency failures both with a comprehensive survey and taxonomy of the state-of-the-art techniques for exposing concurrency failures and with a technique to expose and reproduce concurrency field failures.

The first contribution of the thesis is the first comprehensive survey, analysis, and classification of the state-of-art techniques for exposing concurrency failures. We present a set of classification criteria that characterize the techniques for exposing concurrency failures, and introduce a classification schema that we use to classify and survey the different approaches. The survey reveals many interesting properties regarding the state-of-the-art techniques, and highlights strengths and weaknesses. Research on exposing concurrency failures can benefit from this study. The survey appeared on IEEE Transactions on Software Engineering [10], and has been selected as Journal- First paper for presentation at the Joint Meeting on Foundations of Software Engineering (FSE) in 2017.

The second contribution of this thesis is a novel approach to expose and reproduce concurrency field failures. Our survey reveals that the problem of automatically reproducing concurrency field failures has not received enough attention, and the few techniques that have been proposed present two main limitations: They rely on execution traces or memory core dumps that may be expensive and hard to collect, respectively, and identify failure-inducing interleavings but
do not synthesize failure-inducing test cases. We propose a novel technique that
overcomes such limitations by relying only on crash stack traces, which are easily
obtainable and commonly available, and automatically identifying both a failure-
inducing test case and interleaving. We implemented a prototype of the approach
and we conducted an empirical evaluation demonstrating its effectiveness. The
approach described in this dissertation and the results of the empirical evalua-
tion appeared in the proceedings of the Joint Meeting on Foundations of Software

7.1 Contributions

The main contributions of this thesis are the first comprehensive survey of
the state-of-the-art techniques for exposing concurrency failures and a technique
to automatically expose and reproduce concurrency field failures. In details,
this thesis contributes to the state of the art in exposing concurrency failures by
proposing:

A classification schema for techniques for exposing concurrency failures. We
propose a set of criteria for classifying techniques to expose concurrency
failures, and propose a classification schema that we use to survey the
state-of-the-art techniques. The classification schema uses the selection
of interleavings as main classification criterion, and distinguishes between
property-based, space exploration, and reproduction techniques.

A taxonomy of techniques for exposing concurrency failures. We use the pro-
posed classification schema to provide a taxonomy of the state-of-the-art
techniques for exposing concurrency failures. The taxonomy classifies and
compares the state-of-the-art techniques, discusses their advantages and
limitations, and indicates open problems and possible research directions.

A novel approach to reproduce concurrency field failures We propose CON-
CRASH, the first automated technique that reproduces concurrency failures
from the limited information available in crash stack traces. CONCRASH dif-
fers from the approaches proposed in the literature in that it requires com-
monly available information that does not introduce either overhead or
privacy issues, and synthesizes both failure-inducing concurrent test cases
and related interleavings. CONCRASH adopts a suitable set of search prun-
ing strategies to efficiently explore the huge space of possible inputs and
interleavings of the system. The pruning strategies trim both test cases
that are redundant with respect to previously generated test cases and test cases that are irrelevant with respect to the concurrency failure in input.

**Prototype implementation.** We developed a prototype Java implementation of our CONCRASH approach. The prototype requires as input the crash stack trace of the failure to reproduce and synthesizes a failure-inducing test case in the form of both an executable Java class and a failure-inducing interleaving, which is encoded in a textual file reporting the ordered sequence of instructions to execute to reproduce the failure. We used this prototype to experimentally evaluate the effectiveness of the approach presented in this dissertation.

**Evaluation of the technique.** We evaluated CONCRASH on a benchmark of ten concurrency failures taken from different popular code bases. Our results show that CONCRASH successfully reproduces all the considered failures within a reasonable time and that its effectiveness is given by the synergetic combination of the pruning strategies it adopts. Moreover, our results show that CONCRASH outperforms state-of-the-art testing approaches by a large margin, both in terms of failures successfully reproduced and of its performance.

### 7.2 Open Research Directions

The results of this thesis open new research directions towards automatically exposing concurrency failures.

**Exposing concurrency failures in message passing systems.** One of the main findings of our survey is that the vast majority of approaches for exposing concurrency failures target shared memory systems. Validation and verification of message passing as well as event-driven systems has exploited mostly static analysis and model checking approaches, leaving the important area of exposing concurrency failures in message passing and event-driven systems open for future research.

**Generating concurrent test cases.** Our survey reveals that most of the techniques for exposing concurrency failures target the problem of selecting relevant interleavings, and few techniques focus on test case generation. Current approaches rely on assumptions that might be invalid in many cases: for instance, they assume that concurrency failures can be exposed
by interleaving two threads only and by accessing a single shared object instance. Furthermore, they do not support the wait-notify mechanism that is largely adopted in Object-Oriented programs, and whose misuse can lead concurrent systems to deadlock. Relaxing the restrictive assumptions of current approaches and supporting the wait-notify mechanism is an open research topic.

**Generating test oracles.** Our survey highlights that the state-of-the-art techniques for exposing concurrency failures rely solely on implicit oracles to detect failures. Implicit oracles deem as erroneous any system crash and unhandled exception, and are effective in identifying generally wrong behaviors, but cannot identify semantic failures, that is, incorrect results produced by the system. How to automatically generate test oracles that detect semantic failures taking into account concurrency is yet another important open research direction.

**Extending CONCrASH to reproduce deadlocks.** CONCrASH automatically reproduces concurrency failures that manifest as runtime exceptions and generate a crash stack trace. Thus, CONCrASH cannot reproduce deadlocks, which usually do not generate a crash stack trace when they occur. Enhancing CONCrASH to support also the reproduction of deadlock failures is a possible research direction, which requires to extend CONCrASH with a dedicated runtime monitor that generates a crash stack trace when a deadlock is detected and with an interleaving explorer specific for deadlocks.
Bibliography


URL: https://www.google.com/patents/US20020087950


