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

Dissertation Committee

Matthias Hauswirth	Università della Svizzera italiana, Switzerland
Michele Lanza	Università della Svizzera italiana, Switzerland
Antonia Bertolino	Consiglio Nazionale delle Ricerche, Italy
Michael Pradel	Technische Universität Darmstadt, Germany

Dissertation accepted on 26 October 2018

Research Advisor	PhD Program Director
Mauro Pezzè	**Prof. Walter Binder, Prof. Olaf Schenk**

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

Figures

Tables

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

[1] http://stap.sjtu.edu.cn/index.php?title=JaConTeBe

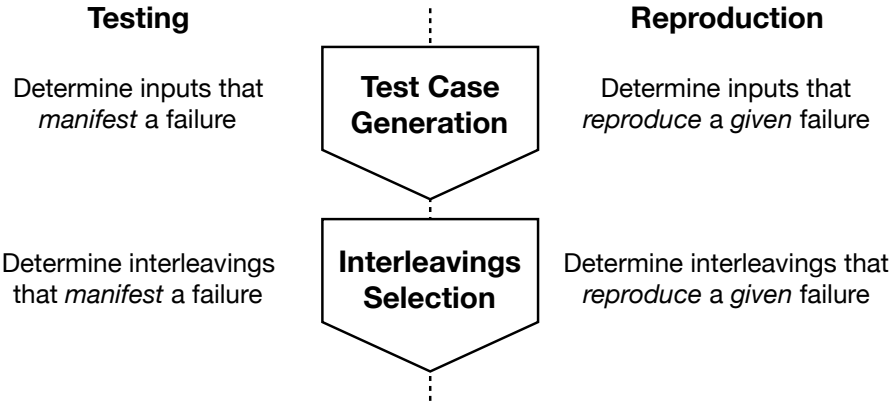


Figure 1.1. Exposing concurrency failures: how and when.

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 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.

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 interact by accessing a common memory. In message passing systems, execution flows interact by exchanging messages. Message passing can be used either by execution flows hosted on the same physical node or on different physical nodes (distributed 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 operations $s_f(m)$ that send a message m to the execution flow f, and receive operations $r_f(m)$ 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 receive 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

```
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[2].

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].

[2] 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

```
1 begin f_1
2 x++
3 end f_1
```

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

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 intepreter 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.[3]

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

[3] 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

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

```
1  begin f_2
2  x = 0
3  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
```

```
1  begin f_2
2  acquire(L1)
3  x = 0
4  acquire(L1)
5  end f_2
```

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.

Listing 2.5. An example of deadlock

```
1  begin f_1
2  acquire(L1)
3    acquire(L2)
4    ...
5    release(L2)
6  release(L1)
7  end f_1
```

```
1  begin f_2
2  acquire(L2)
3    acquire(L1)
4    ...
5    release(L1)
6  release(L2)
7  end f_2
```

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

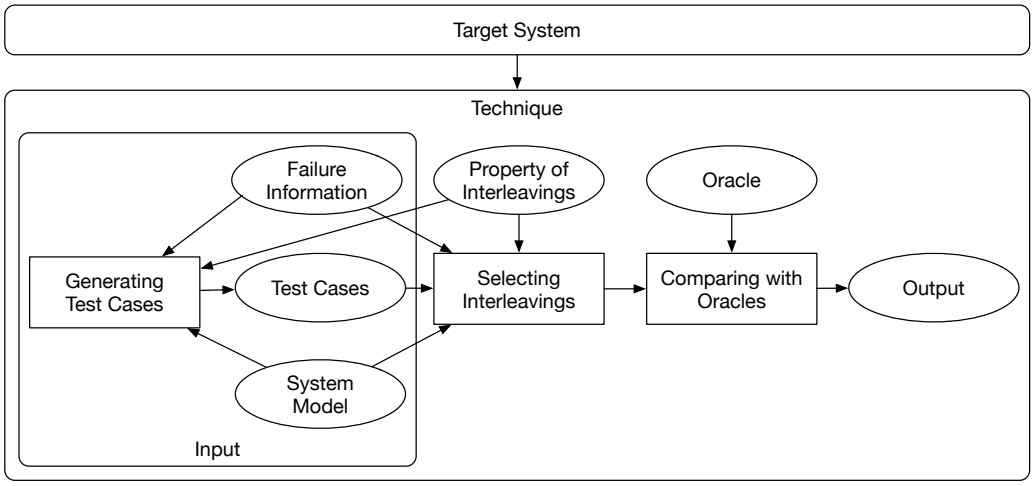


Figure 3.1. A general framework for exposing concurrency failures

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

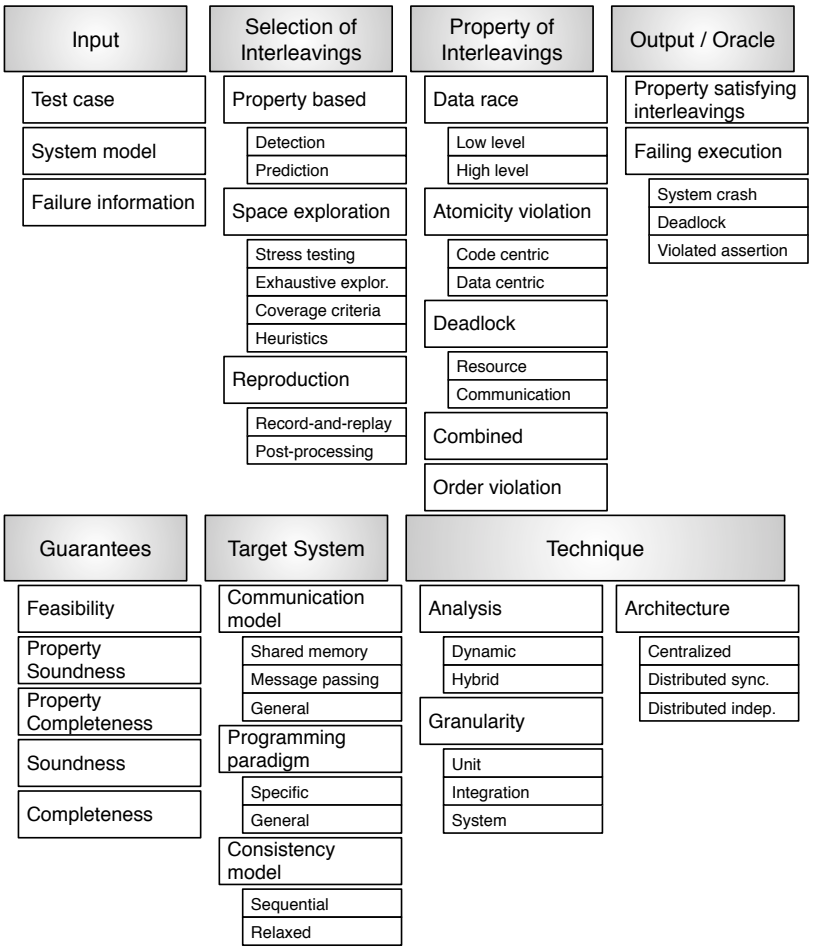


Figure 3.2. Classification criteria for the techniques to expose concurrency failures

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 [116]. Happens-before analysis extends this approach by capturing general order constraints. Different types of happens-before analysis apply to different synchronization mechanisms [101], and present various cost-accuracy trade offs [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 [88]. They also often adopt reduction techniques such as dynamic partial order reduction [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 interleavings* as the main classification criterion since (i) the vast majority of the techniques we survey addresses the problem of selecting interleavings; (ii) among the main aspects of the problem reported in Figure 3.1, the selection of interleavings is an activity specific for

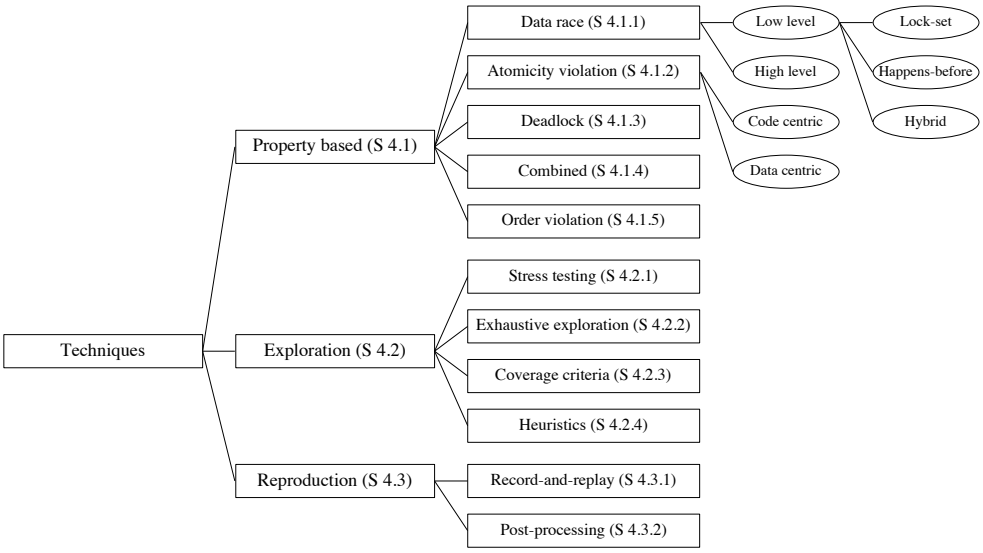


Figure 3.3. Classification schema

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.

Chapter 4

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

	Input		Output / Oracle				Select. Interl.	Target System			Testing Tech.		Guarantees				
	Test Case	Model	Sat. Interl.	Crash	Deadlock	Assert. Viol.	Predict.	Commun.	Paradigm	Consist.	Granularity	Analysis	Prop. Complet.	Prop. Sound.	Complet.	Soundness	Feasibility
Low level — Lockset																	
Shacham et al. [122]	✓			✓	✓	✓	✓	SM	General	Seq	S	D	✓		✓	✓	✓
Racez [123]	✓		✓					SM	General	Seq	S	D			-	-	✓
Praun and Gross [139]	✓		✓					SM	OO	Seq	S	H			-	-	✓
ACCORD [67]	✓	✓	✓				✓	SM	Fork-join	Seq	S	D			-	-	
Low level — Happens-before																	
FastTrack [42]	✓		✓					SM	General	Seq	S	D			-	-	✓
LiteRace [87]	✓		✓					SM	General	Seq	U	D			-	-	✓
Pacer [16]	✓		✓					SM	General	Seq	S	D			-	-	✓
SOS [77]	✓		✓					SM	General	Seq	S	D			-	-	✓
Carisma [155]	✓		✓					SM	General	Seq	S	D			-	-	✓
ReEnact [107]	✓	✓	✓					SM	General	Seq	S	D			-	-	✓
Narayanasamy et al. [90]	✓		✓				✓	SM	General	Seq	S	D		✓	-	-	✓
Frost [137]	✓			✓	✓	✓	✓	SM	General	Seq	S	D					✓
Portend [68]	✓		✓				✓	SM	General	Seq	S	D		✓	-	-	✓
Tian et al. [134]	✓		✓					SM	General	Seq	S	D			-	-	✓
RDIT [109]	✓		✓				✓	SM	General	Seq	S	D			-	-	
Smaragdakis et al. [128]	✓		✓				✓	SM	General	Seq	S	D			-	-	
DrFinder [20]	✓		✓				✓	SM	General	Seq	S	D			-	-	✓
RVPredict [59]	✓		✓				✓	SM	General	Seq	S	D			-	-	
WebRacer [102]	✓		✓					SM	Web platforms	Seq	S	D			-	-	✓
EventRacer [110]	✓		✓					SM	Event-based	Seq	S	D			-	-	✓
DroidRacer [85]	✓		✓					SM	Android apps	Seq	S	D			-	-	✓
Java RaceFinder [70]	✓		✓				✓	SM	General	Rel	S	D	✓		-	-	✓
Relaxer [19]	✓		✓				✓	SM	General	Rel	S	D			-	-	✓
Low level — Hybrid																	
Choi et al. [28]	✓		✓					SM	General	Seq	S	H			-	-	✓
Wester et al [143]	✓		✓					SM	General	Seq	S	D			-	-	✓
RaceMob [69]	✓		✓					SM	General	Seq	S	H		✓	-	-	✓
RaceFuzzer [119]	✓			✓			✓	SM	General	Seq	S	D				✓	✓
RaceTrack [152]	✓		✓					SM	General	Seq	S	D			-	-	✓
Goldilocks [37]	✓		✓					SM	General	Seq	S	H			-	-	✓
MultiRace [103]	✓		✓					SM	General	Rel	S	D			-	-	✓
SimRT [150]	✓			✓			✓	SM	General	Seq	S	D				✓	✓
Racageddon [38]			✓				✓	SM	General	Seq	S	D			-	-	✓
Narada [114]	✓		✓				✓	SM	General	Seq	S	D				✓	✓
High level																	
Colt [121]	✓	✓	✓				✓	SM	OO	Seq	U	D			-	-	✓
Dimitrov et al. [34]	✓	✓	✓					SM	OO	Seq	U	D			-	-	✓
Simian [12]						✓		SM	Web Platforms	Seq	U	D			-	-	✓

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, traditional happens-before analysis may miss some possible interleavings, and may thus miss some faults [128].

The property completeness problem has been addressed by either implementing 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 allowed 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 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 executions 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 conservative 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 taking into account only lock based synchronization, lockset analysis improves the efficiency with respect to happens-before analysis since it only needs to store information 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. RVPredict [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 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

	Input		Output / Oracle				Select. Interl.	Target System			Testing Tech.		Guarantees			
	Test Case	Model	Sat. Interl.	Crash	Deadlock	Assert. Viol.	Predict.	Commun.	Paradigm	Consist.	Granularity	Analysis	Prop. Complet.	Prop. Sound.	Complet.	Soundness	Feasibility
Code centric																	
Atomizer [41]	✓	✓	✓					SM	General	Seq	S	D			-	-	✓
SVD [147]	✓		✓					SM	General	Seq	S	D			-	-	✓
AVIO [82]	✓		✓					SM	General	Seq	S	D			-	-	✓
Velodrome [43]	✓		✓					SM	General	Seq	S	D			-	-	✓
AtomFuzzer [97]	✓			✓	✓		✓	SM	General	Seq	S	D				✓	✓
HAVE [27]	✓		✓				✓	SM	General	Seq	S	H			-	-	
Penelope [129]	✓			✓		✓	✓	SM	General	Seq	U	D				✓	✓
Wang and Stoller [141]	✓		✓				✓	SM	OO	Seq	S	D	✓		-	-	
CTrigger [98]	✓			✓	✓	✓	✓	SM	General	Seq	S	D				✓	✓
Falcon [99]	✓		✓					SM	General	Seq	U	D			-	-	✓
Best [47]	✓		✓				✓	SM	General	Seq	S	H			-	-	✓
DoubleChecker [14]	✓	✓	✓					SM	General	Seq	U	D			-	-	✓
Intruder [113]	✓		n.a.	n.a.	n.a.	n.a.	n.a.	SM	General	Seq	U	D	n.a.	n.a.	n.a.	n.a.	n.a.
AutoConTest [132]				✓		✓		SM	General	Seq	U	D		✓		✓	✓
Data centric																	
Muvi [80]	✓		n.a.	n.a.	n.a.	n.a.	n.a.	SM	General	Seq	S	D	n.a.	n.a.	n.a.	n.a.	n.a.
Hammer et al. [51]	✓		✓				✓	SM	General	Seq	S	H			-	-	✓
AssetFuzzer [71]	✓	✓	✓				✓	SM	General	Seq	S	D			-	-	✓
ReConTest [133]	✓			✓	✓	✓	✓	SM	General	Seq	S	D	✓		✓	✓	✓

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 [13] 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 regions 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 checking if the interleaving is serializable, that is, the results of its execution are equivalent 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 programs. 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 exploits 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 regions, (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 analysis. Velodrome [43] uses serializability to distinguish between benign and harmful atomicity violations. AtomFuzzer [97] and Penelope [129] actively control the thread scheduler to trigger atomicity violations and report only those atomicity violations that lead to system crashes. HAVE [27] implements a hybrid

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

	Input		Output / Oracle				Select. Interl.	Target System			Testing Tech.		Guarantees				
	Test Case	Model	Sat. Interl.	Crash	Deadlock	Assert. Viol.	Predict.	Commun.	Paradigm	Consist.	Granularity	Analysis	Prop. Complet.	Prop. Sound.	Complet.	Soundness	Feasibility
Goodlock [52]	✓		✓				✓	SM	Lock sync	Seq	S	D			-	-	
DeadlockFuzzer [64]	✓				✓		✓	SM	Lock sync	Seq	S	D		✓		✓	✓
MagicFuzzer [21]	✓				✓		✓	SM	Lock sync	Seq	S	D		✓		✓	✓
Wolf [112]	✓				✓		✓	SM	Lock sync	Seq	S	D		✓		✓	✓
ConLock [22]	✓				✓		✓	SM	Lock sync	Seq	S	D		✓		✓	✓
Sherlock [39]	✓				✓		✓	SM	Lock sync	Seq	S	D		✓		✓	✓
OMEN [111]	✓				✓		✓	SM	Lock sync	Seq	S	D		✓		✓	✓
Armus [30]	✓				✓			SM	Barrier sync	Seq	S	D		✓		✓	✓

Table 4.4. Techniques for detecting combined properties violations

	Input		Output / Oracle				Select. Interl.	Target System			Testing Tech.		Guarantees				
	Test Case	Model	Sat. Interl.	Crash	Deadlock	Assert. Viol.	Predict.	Commun.	Paradigm	Consist.	Granularity	Analysis	Prop. Complet.	Prop. Sound.	Complet.	Soundness	Feasibility
Agarwal et al. [1]	✓		✓					SM	General	Seq	S	H			-	-	✓
Chen and MacDonal [25]	✓			✓	✓	✓	✓	SM	General	Seq	S	H		✓		✓	✓
Kahlon and Wang [66]	✓		✓				✓	SM	General	Seq	S	D			-	-	
PECAN[60]	✓	✓	✓				✓	SM	General	Seq	S	D			-	-	✓

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.

Armus [30] targets deadlocks caused by barrier synchronization primitives.

Table 4.5. Techniques for detecting order violations

	Input		Output / Oracle				Select. Interl.	Target System			Testing Tech.		Guarantees			
	Test Case	Model	Sat. Interl.	Crash	Deadlock	Assert. Viol.	Predict.	Commun.	Paradigm	Consist.	Granularity	Analysis	Prop. Complet.	Prop. Sound.	Complet.	Soundness	Feasibility
PRETEX [65]	✓		✓				✓	SM	OO	Seq	U	H			-	-	
2ndStrike [48]	✓	✓		✓			✓	SM	OO	Seq	U	D		✓		✓	✓
ConMem [157]	✓			✓			✓	SM	General	Seq	S	D		✓		✓	✓
ConSeq [156]	✓			✓		✓	✓	SM	General	Seq	S	H		✓		✓	✓
ExceptioNULL [40]	✓			✓			✓	SM	General	Seq	S	D		✓		✓	✓
Maple [148]	✓			✓	✓	✓	✓	SM	General	Seq	S	D		✓		✓	✓
jPredictor [23]	✓	✓	✓				✓	SM	General	Seq	S	H		✓		✓	✓
Sinha and Wang [127]	✓		✓				✓	SM	General	Seq	S	D			-	-	
DefUse [124]	✓		✓					SM	General	Seq	S	D			-	-	✓
GPredict [58]	✓	✓	✓				✓	SM	General	Seq	S	D			-	-	
SimRacer [149]	✓			✓		✓	✓	SM	Process level	Seq	S	D		✓		✓	✓
Cafa [56]	✓		✓				✓	MP	Event driven	Seq	S	D			-	-	
Mutlu et al. [89]	✓		✓				✓	MP	Javascript app	Seq	S	D			-	-	

4.1.4 Combined

Few techniques target both atomicity violations and data races, and rely on happens-before analysis. Agarwal et al. [1] focus on performance by complementing happens-before analysis with static type checking. Chen and MacDonald [25] focus on accuracy, by considering control flow information. Kahlon and Wang [66] and PECAN [60] focus on generality, by providing developers with languages to express undesired patterns of memory accesses.

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

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

	Input		Output / Oracle				Target System			Testing Tech.		Guarantees		
	Test Case	Model	Sat. Interl.	Crash	Deadlock	Assert. Viol.	Commun.	Paradigm	Consist.	Granularity	Analysis	Complet.	Soundness	Feasibility
Stress testing														
Pike [44]	✓	✓	✓				SM	General	Seq	S	D		✓	✓
SpeedGun [106]						✓	SM	General	Seq	n.a.	n.a.	✓		
CovCon [29]				✓		✓	SM	General	Seq	S	D		✓	✓
Exhaustive exploration														
Ballerina [92]			✓				SM	OO	Seq	U	D		✓	✓
Sen and Agha [120]				✓	✓		MP	General	Seq	S	D		✓	✓
Basset [74]	✓			✓	✓	✓	MP	Actors	Seq	S	D	✓	✓	✓
CheckMate [63]	✓		✓				SM	General	Seq	S	D			
CPPMem [9]	✓		✓				SM	General	Rel	S	D			
ViP [35]	✓		✓				SM	General	Seq	S	D	✓		
CDSChecker [93]	✓		✓				SM	General	Rel	S	D			
Coverage criteria														
Wang et al. [140]	✓			✓			SM	General	Seq	U	D		✓	✓
Hong et al. [54]	✓			✓		✓	SM	General	Seq	S	D		✓	✓
Bita [131]	✓			✓		✓	MP	Actors	Seq	S	D		✓	✓
Heuristics														
Rapos [118]	✓			✓	✓	✓	SM	General	Seq	S	D		✓	✓
PCT [18]	✓			✓		✓	SM	General	Seq	S	D		✓	✓
Gambit [32]	✓			✓	✓	✓	SM	General	Seq	S	D		✓	✓

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 *Synchronization-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.

Table 4.7. Techniques for reproducing concurrency failures

	Input		Output / Oracle				Target System			Testing Tech.		Guarantees		
	Test Case	Model	Sat. Interl.	Crash	Deadlock	Assert. Viol.	Commun.	Paradigm	Consist.	Granularity	Analysis	Complet.	Soundness	Feasibility
Record-and-replay														
ODR [2]			✓				SM	General	Seq	S	D			
Pres [100]			✓				SM	General	Seq	S	D		✓	✓
LEAP [57]			✓				SM	General	Seq	S	H		✓	✓
Stride [61]			✓				SM	General	Seq	S	D			
Chimera [75]			✓				SM	General	Seq	S	H		✓	✓
CLAP [61]			✓				SM	General	Seq	S	D			
ReCBuLC [153]			✓				SM	General	Seq	S	D		✓	✓
DESCRY [151]			✓				SM	General	Seq	S	D		✓	✓
Post-processing														
ESD [154]			✓				SM	General	Seq	S	H		✓	✓
Weratunge et al [142]			✓				SM	General	Seq	S	H		✓	✓

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.

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

```
1  public class Logger {
2
3    public void info(String msg) {
4      if (Level.INFO.intValue() < levelValue) {
5        return;
6      }
7      log(Level.INFO, msg);
8    }
9
10   public void log(Level level, String msg) {
11     if (level.intValue() < levelValue  levelValue == offValue) {
12       return;
13     }
14     LogRecord lr = new LogRecord(level, msg);
15     doLog(lr);
16   }
17
18   private void doLog(LogRecord lr) {
19     lr.setLoggerName(name);
20     String ebname = getEffectiveResourceBundleName();
21     if (ebname != null) {
22       lr.setResourceBundleName(ebname);
23       lr.setResourceBundle(findResourceBundle(ebname));
24     }
25     log(lr);
26   }
27
28   public void log(LogRecord record) {
29     if (record.getLevel().intValue() < levelValue  levelValue == offValue) {
30       return;
31     }
32     synchronized (this) {
33       if (filter != null) {
34         if (!filter.isLoggable(record)) {
35           return;
36         }
37       }
38     }
39     Logger logger = this;
40     while (logger != null) {
41       Handler targets[] = logger.getHandlers();
42       if (targets != null) {
43         for (int i = 0; i < targets.length; i++) {
44           targets[i].publish(record);
45         }
46       }
47       if (!logger.getUseParentHandlers()) {
48         break;
49       }
50       logger = logger.getParent();
51     }
52   }
53
54   public void setFilter(Filter newFilter) throws SecurityException {
55     if (!anonymous) {
56       manager.checkAccess();
57     }
58     filter = newFilter;
59   }
60 }
```

Figure 5.1. Faulty class `java.util.logging.Logger` of JDK 1.4.1

```
1  java.lang.NullPointerException
2  at Logger.log(Logger.java:34)
3  at Logger.doLog(Logger.java:25)
4  at Logger.log(Logger.java:15)
5  at Logger.info(Logger.java:7)
6  at LoggerTest$1.runTest(LoggerTest.java:11)
7  at java.lang.Thread.run(Thread.java:662)
```

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

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

```
1  public class LoggerTest {
2
3   public static void main(String[] args) throws Throwable {
4    java141.util.logging.Logger logger0;
5    logger0 = Logger.getAnonymousLogger();
6
7    MyFilter myFilter0 = new MyFilter();
8    logger0.setFilter((Filter)myFilter0);
9
10   Thread t1 = new Thread(new Runnable() {
11    public void run() {
12     logger0.info("");
13    }
14   });
15
16   Thread t2 = new Thread(new Runnable() {
17    public void run() {
18     logger0.setFilter(null);
19    }
20   });
21
22   t1.start();
23   t2.start();
24   t1.join();
25   t2.join();
26 }
```

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.

Problem Definition and Challenges

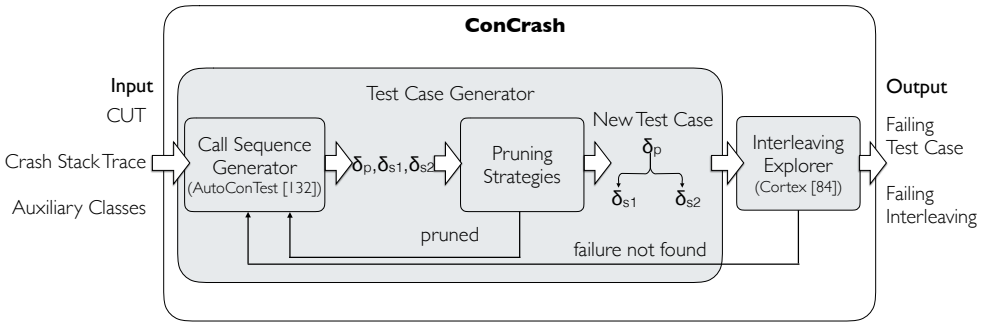
The problem that we address with CONCRASH can be formulated as follows: *Given a crash stack trace, the corresponding Class Under Test (CUT), a set of auxiliary classes, which CUT depends on, and a time-budget, generate a concurrent test case and a failure-inducing interleaving that produce the crash 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 ⟨`Logger sout = Logger.getAnonymousLogger(); MyFilter myFilter0=new MyFilter(); sout.setFilter(myFilter0)`⟩, the interfering method `sout.setFilter(null)` and the crashing method `sout.info("")`. With different sequential prefixes, for example ⟨`Logger sout=Logger.getAnonymousLogger(); sout.setFilter(null)`⟩, the test 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



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

Figure 5.4. Conceptual Architecture of ConCrash

case, and the Interleaving Explorer looks for a thread interleaving of the newly generated test case that reproduces the concurrency failure.

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 δ can effectively approximate the behavior of δ when executed concurrently with other method

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 δ_p, and two concurrent suffixes δ_{s1} and δ_{s2} that are executed concurrently after δ_p. The prefix δ_p creates a shared object under test (SOUT) of type CUT, and invokes methods that bring the SOUT in a failure-inducing state. The suffixes δ_{s1} and δ_{s2} access the SOUT concurrently trying to manifest a failure-inducing interleaving.

The algorithm explores a search space 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
       /*                  Initialization                                  */
2      𝕊 ← ∅                                                  // state repository
3      ℂ ← ∅                                                  // coverage repository
4      pendingSeqs[...] ← ∅                                   // sequences to be extended
5      level ← 0                                              // current level of sequence exploration
6      pool ← CREATE-PARAMETER-VALUES(classes)
7      cm ← EXTRACT-CRASHING-METHOD(CST)
8      for each construcor m(τ1,...τn) of CUT do
9          for each value (v1...vn) of type (τ1...τn) in pool do
10             δsout ← m(v1,...vn)
11             if δsout does not throw an exception then
12                 add δsout to end of pendingSeqs[0]
       /*        Exploration of new combinations δp, δs1, δs2              */
13     while timeBudget is not expired do
14         while pendingSeqs[level] ≠ ∅ do
15             δp ← get and delete the first δ in pendingSeqs[level]
16             for each value (v1...vn) of type cm(τ1...τn) in pool do
17                 δs1 ← cm(v1...vn)
18                 for each public method m(τ1,...τn) of CUT do
19                     for each value (v1...vn) of type m(τ1...τn) in pool do
20                         δs2 ← m(v1...vn)
21                         result ← PRUNING(δp, δs1, δs2)
22                         if result ≠ null then
23                             return result
24         level++
25 function PRUNING(δp, δs1, δs2)
26     δp,s1 ← append δs1 to δp
27     δp,s2 ← append δs2 to δp
28     execute δp,s1 and δp,s2                          // Collection of runtime information
29     if δp,s1 and δp,s2 do not throw exceptions (PS-Exception)
30     ∧ ∃ e ∈ M̄(δcm) : stack(e)= CST                                  (PS-Stack)
31     ∧ ⟨M̄(δp,s1), M̄(δp,s2)⟩ ∉ ℂ                                     (PS-Redundant)
32     ∧ M̄(δp,s2) interferes with M̄(δp,s1)                            (PS-Interfere)
33     ∧ M̄(δp,s1) || M̄(δp,s2)                                         (PS-Interleave)
       then
34         add ⟨M̄(δp,s1), M̄(δp,s2)⟩ to ℂ
35         t ← ASSEMBLE-TEST-CODE(δp, δs1, δs2)
36         isFailure ← INTERLEAVING-EXPLORER(t)
37         if isFailure = true then
38             return t                                   // failure-inducing test code
39     if 𝕊(δs2) ∉ 𝕊 and δp,s2 does not throw exception then
40         add 𝕊(δs2) to 𝕊
41         if M̄(δp,s1) interferes with M̄(δp,s2) then
42             add δp,s2 begin of pendingSeqs[level+1]
43         else
44             add δp,s2 end of pendingSeqs[level+1]
45     return null
```

Figure 5.5. The ConCrash algorithm

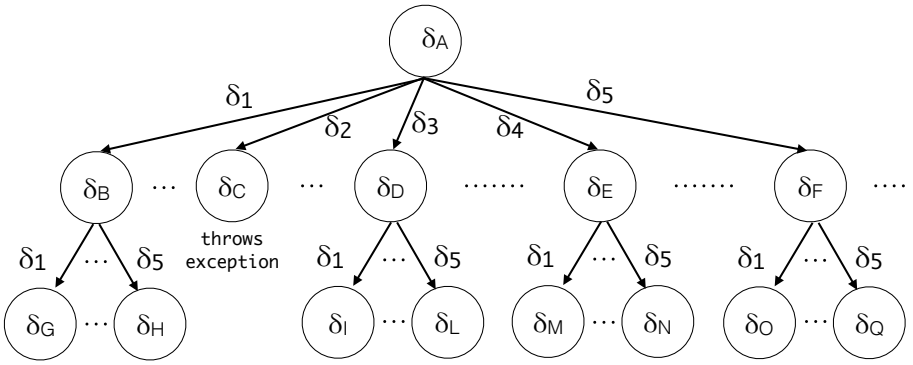
5.3.1 Modeling the Test Cases Search Space

CONCRASH finds a combination of δ_p, δ_{s1} and δ_{s2} that alltogheter 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.getAnanymousLogger();`), 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 δ_O represents the sequence ⟨ `Logger sout=Logger.getAnanymousLogger(); Filter f=new Filter(); sout.setFilter(f); sout.info("");` ⟩ that is obtained by traversing the tree from the root to the node through edges δ_5 and δ_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 δ, the node traversal operator produces a child node that represents a new sequence obtained from δ by appending a sequence of method calls (an edge), with m being the last method call. The node traversal operator may add other method calls before m to create the non-primitive parameters of m, if any. For instance in Figure 5.6, Edge δ_1 corresponds to a single method call, while Edge δ_3 corresponds to two method calls, where the first call creates the input parameter h of the method `removeHandler`.

CONCRASH always extends nodes with methods of the CUT that have at least an input parameter (including the method receiver) of type CUT, and binds exactly one of them to the object SOUT. Therefore by construction, each edge in the tree accesses the same object SOUT. Referring always to SOUT is crucial because the suffixes trigger concurrency failures by accessing the same shared object [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 τ_i. If τ_i is of type CUT and is the receiver method, CONCRASH binds it to the object reference of SOUT, otherwise (τ_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



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

δ_p	δ_{s1}	δ_{s2}	*Is the test generated ?*
$\langle\delta_A, \delta_1, \delta_2\rangle$			No, δ_C throws exception
$\langle\delta_A, \delta_1, \delta_3\rangle$			No, pruned by PS-Interfere
$\langle\delta_A, \delta_1, \delta_4\rangle$			Yes, but not failure inducing
$\langle\delta_A, \delta_1, \delta_5\rangle$			No, pruned by PS-Redundant
$\langle\delta_F, \delta_1, \delta_4\rangle$			**Yes, failure-inducing**

Figure 5.6. A Tree model of class `Logger`

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 [50, 53, 132]. In fact, the general form of thread-safety [50] does not guarantee the atomic execution of multiple calls to a shared object of a thread-safe class in the same thread. CONCRASH reduces the length of sequential prefixes by adopting a *Breadth-First Search* exploration strategy: it explores all sequences with n edges before exploring sequences with $n+1$ edges.

5.3.3 Exploring the Test Cases Search Space

Initialization (lines 2-12).
CONCRASH starts by initializing to empty the state repository $\mathbb{S}$, the coverage repository $\mathbb{C}$, and the list of pending sequences to be extended `pendingSeqs` (lines 2, 3 and 4, respectively). CONCRASH then initializes the current level of exploration of the tree to zero (line 5). The algorithm randomly generates the pool of primitive and non-primitive parameters values [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). CONCRASH creates a new root δ_{sout} for each of such methods and for each combination of parameter values (lines 8, 9, 10). The algorithm checks whether the execution of δ_{sout} throws an exception (line 11) and, if this is not the case, adds δ_{sout} to the list `pendingSeqs[0]` (line 12). The algorithm does not further elaborate the sequences δ_{sout} that throw an exception since they do not successfully create the object.

Exploration of new combinations $\langle \delta_p, \delta_{s1}, \delta_{s2} \rangle$ (lines 13-24).
CONCRASH explores new combinations iteratively until either it reproduces the failure or the time budget expires (line 13). At each iteration, CONCRASH removes a sequence δ_p from `pendingSeqs`[*level*] (line 15), and generates the children of the leaf of the Tree that corresponds to δ_p, starting from the children related to the crashing method (*cm*) exploring different values for the input parameters. We denote each of the edges resulting from the extensions as δ_{s1} (line 17). CONCRASH explores all the children of δ_p, obtained by extending δ_p with all public methods in CUT (line 18) with each combination of the input parameters in the pool (line 19). We denote the edges resulting from the extensions as δ_{s2} (line 20). Every combination of δ_p, δ_{s1} and δ_{s2} corresponds to a candidate concurrent test case. Function `PRUNING` analyzes each combination of δ_p, δ_{s1} and δ_{s2} to determine if it should be pruned or not (line 21). CONCRASH considers

Let the trace $E = \langle e_1, e_2 \dots e_k \rangle$ of a call sequence δ be the ordered sequence of events exhibited by a sequential (single-threaded) execution of δ. An event can be one of the following:

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

Given a call sequence $\delta = \langle m_1, \dots, m_n \rangle$, the *trace of a method call* $m_i \in \delta$ is the non-empty segment E_i of E such that E_i contains only the events triggered directly or indirectly by the invocation of m_i [132]. Given a call sequence δ, its *sequential coverage* $\mathcal{M}(\delta)$ is defined as the partition $\{E_1, E_2, \dots, E_n\}$ of E, that is the unordered set composed of the n method call traces of E [132].

Figure 5.7. Definition of Sequential coverage

all public methods in the CUT to obtain δ_{s2} (line 18) because the crash stack does not contain information about the interfering method, and thus ConCrash needs to explore all the possible candidates to identify the right one.

`PRUNING` (lines 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 $\delta_{p,s1}$ and $\delta_{p,s2}$ be the sequences that extend δ_p with edges δ_{s1} and δ_{s2}, respectively (lines 26 and 27), ConCrash executes $\delta_{p,s1}$ and $\delta_{p,s2}$ in isolation (single-threaded execution) (line 28), and collects the following runtime information for each sequence $\delta \in \{\delta_{p,s1}, \delta_{p,s2}\}$: whether (i) δ throws an uncaught exception, (ii) the *sequential coverage* of the last method call in δ [132], and (iii) the state of the object SOUT after executing δ, 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 $\mathcal{M}(\delta)$ as $\overline{\mathcal{M}}(\delta)$.

Pruning strategies (lines 29-45). ConCrash prunes the combination $\langle \delta_p, \delta_{s1}, \delta_{s2} \rangle$ according to different strategies. If the test case is neither redundant (line 31) nor irrelevant (lines 29, 30, 32, 33) ConCrash updates the coverage repository $\mathbb{C}$ (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 $\delta_{p,s2}$ should be added to the `pendingSeqs` list for further extensions (line 39), that is, ConCrash checks whether the state $\mathcal{S}(\delta_{s2})$ produced by executing $\delta_{p,s2}$ either throws an exception or has been already explored. If not, ConCrash adds $\mathcal{S}(\delta_{s2})$ to $\mathbb{S}$ and inserts $\delta_{p,s2}$ in the `pendingSeqs` of the next level (line 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][1]. ConCrash adds the current $\delta_{p,s2}$ 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 $\delta_{p,s2}$ should be added at the beginning or at the end of `pendingSeqs`$[level+1]$.

PS-Exception

ConCrash *prunes a combination* $\langle \delta_p, \delta_{s1}, \delta_{s2} \rangle$ *if either* $\delta_{p,s1}$ *or* $\delta_{p,s2}$ *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 $\delta_{s2} = \delta_C$ by applying PS-Exception:

[1] 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]

```
1  public synchronized void removeHandler(Handler handler) {
2    if (!anonymous) {
3       manager.checkAccess();
4    }
5    if (handler == null) {
6       throw new NullPointerException();
7    }
8    if (handlers == null) {
9       return;
10   }
11   handlers.remove(handler);
12 }
```

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* $\nexists\ e \in \overline{\mathcal{M}}(\delta_{p,s1}) : stack(e) = CST$ *(Crash Stack Trace), where* $stack(e)$ *is the call stack trace of e, obtained by analyzing the method entry and exit points in* $\overline{\mathcal{M}}(\delta_{p,s1})$.

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 δ_{s1} that when executed sequentially do not reach the POF with the same call stack of CST. For example in Figure 5.6, ConCrash prunes the combinations with $\delta_{s1} = \delta_B$ since the sequential execution of δ_B does not reach the POF (line 34 in Figure 5.1). Indeed, when executing δ_B the `filter` field is not initialized, thus the condition at line 33 in Figure 5.1 evaluates 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* $\langle \overline{\mathcal{M}}(\delta_{p,s1}), \overline{\mathcal{M}}(\delta_{p,s2}) \rangle \in \mathbb{C}$.

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

PS-Interfere

CONCRASH *prunes a combination* $\langle \delta_p, \delta_{s1}, \delta_{s2} \rangle$ *if* $\nexists\, e_1, e_2 \in \overline{\mathcal{M}}(\delta_{p,s1}) \times \overline{\mathcal{M}}(\delta_{p,s2})$*:* $e_1 = R(f), e_2 = W(f)$

CONCRASH prunes the combinations in which the two concurrent suffixes do not access the same variables or the interfering method $\delta_{p,s2}$ does not write any variable read by the crashing method $\delta_{p,s1}$. 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, CONCRASH prunes the combination with $\delta_{s1} = \delta_B$ and $\delta_{s2} = \delta_D$ because the sequential execution of δ_D does not write any variable read by δ_B. On the contrary, PS-Interfere does not prune the combination with $\delta_{s1} = \delta_B$ and $\delta_{s2} = \delta_F$ as the crashing method `info` reads variable `filter`, and the candidate interfering method `setFilter` writes the same field. CONCRASH 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

CONCRASH *prunes a combination* $\langle \delta_p, \delta_{s1}, \delta_{s2} \rangle$ *if* $\nexists\, e_1, e_2 \in \overline{\mathcal{M}}(\delta_{p,s1}) \times \overline{\mathcal{M}}(\delta_{p,s2})$: $e_1 = RW, e_2 = RW, LH(e_1) \cap LH(e_2) = \emptyset$*, where LH denotes the lock history of an event* $e_x \in \overline{\mathcal{M}}(\delta)$*, defined as the set of locks that are acquired but never released in* $\overline{\mathcal{M}}(\delta)$ *before triggering the event* e_x*, and RW denotes either a read or write memory access. More formally,* $LH(e_x) = \{l \mid \exists e_j = ACQ(l) \in \overline{\mathcal{M}}(\delta), \nexists e_k = REL(l) \in \overline{\mathcal{M}}(\delta), w < k < x\}$*, where w is the index of the first event in* $\overline{\mathcal{M}}(\delta)$*.*

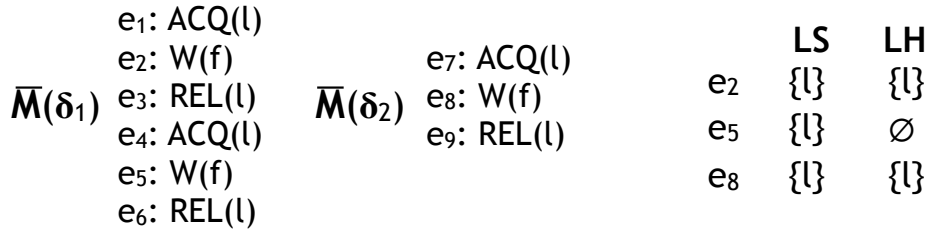


Figure 5.9. Example of lockset history (LH)

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 $LH(e_5) \cap LH(e_8) = \varnothing$.

CONCRASH relies on `PS-Interfere` to decide how to prioritize $\delta_{p,s2}$ in the list (lines 42 and 44). If $\overline{\mathcal{M}}(\delta_{p,s2})$ interferes with $\overline{\mathcal{M}}(\delta_{p,s1})$ (see Condition PS-Interfere), CONCRASH adds $\delta_{p,s2}$ at the beginning of the list, otherwise at the end, since if $\overline{\mathcal{M}}(\delta_{p,s2})$ writes the same variables accessed by the crashing method, $\delta_{p,s2}$ is more likely to lead to a program state that could make the crashing method behave differently if executed in this new state. 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 [35] can be very expensive due to enormous size of interleaving spaces [140]. Techniques that explore interleaving spaces selectively, based on particular classes of concurrency faults, like data races [87, 94], atomicity violations [41, 43, 97, 148], and deadlocks [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 [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 [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 5.4.1 and explores the interleavings through constraint solving 5.4.2. For a detailed description of CORTEX, the interested readers can refer to the seminal work [61] and its extensions [83, 84].

5.4.1 Symbolic Trace Collection

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

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.

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:

$$\Phi = \phi_{path} \wedge \phi_{mo} \wedge \phi_{po} \wedge \phi_{lock} \wedge \phi_{rw} \wedge \phi_{fault}$$

Path Constraints (ϕ_{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 (ϕ_{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 (ϕ_{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 (ϕ_{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 (ϕ_{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 (ϕ_{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$_1$ *How effective is* CONCRASH *in reproducing concurrency failures?*

RQ$_2$ *What is the contribution of each pruning strategy in reducing the search space?*

RQ$_3$ *Is* CONCRASH *more effective than competing state-of-the-art testing approaches?*

RQ$_1$ 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$_2$ 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$_3$ 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 RQ_2 which investigates the effectiveness of the pruning strate-

gies 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 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 [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 [104, 132]. We added the program assertions encoding the failures, as required by the Interleaving Explorer (CORTEX [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 *ID* introduces a unique identifier that we use in the paper to identify the subject program; column *Class Under Test (CUT)* is the class under test of the subject program; columns *Version* and *Code Base* report the version and the code base that contains the faulty class, respectively; columns *Total SLOC* and *CUT SLOC* report the total number of Source Lines of Code of both the code base and of the CUT, including non-abstract super classes, if any, respectively; column *# Methods* indicates the number of public methods of the CUT; columns *Issue ID*, *Fault Type*, *Type of Exception* and *Crash Depth* report the identifier of the issue in the corresponding bug repository, the type of the known fault, the type of the resulting exception and the number of frames in the crash stack, respectively.

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

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

Table 6.2. ConCrash Evaluation Failures

Concurrency Failures				
ID	**Issue ID**	**Fault Type**	**Type of Exception**	**Crash Depth**
1	369	Race	ConcurrentModificationException	4
2	369	Race	ConcurrentModificationException	4
3	481	Atom.	AssertionError	1
4	4225348	Atom.	NullPointerException	2
5	4779253	Atom.	NullPointerException	4
6	8143394	Atom.	NullPointerException	1
7	806667	Atom.	IllegalArgumentException	2
8	187	Race	ConcurrentModificationException	4
9	1507	Atom.	NullPointerException	1
10	509	Atom.	NullPointerException	2

space containing failure inducing test case could be explored before or after, depending on such order. Such order is set pseudo-deterministically with an input random seed. To mitigate threats that may derive from the non-deterministic choices while generating test cases, we repeat each experiment five times using different random seeds, in line with the experimental setup of related work [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 $\mathcal{B}$, 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 $\mathcal{B}$, that is, $FR = 0$, we optimistically underapproximate FRT to $\mathcal{B}$.

Failure-inducing Test ID (FTID) that is the ID of the first failure-inducing test case. The IDs are assigned in ascending order. FTID $= n$ indicates that ConCrash explored the interleaving space of n test cases before reproducing 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 $\mathcal{B}$ ($FR = 0$), we underapproximate FTID to the, ID of the last generated 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 δ_p, δ_{s1}, and δ_{s2}. The lower the value of FTS is, the easier localizing and understanding the failure is. If the failure is not reproduced within $\mathcal{B}$, that is, $FR = 0$, we set FTS $=$ N/A.

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 reproduce concurrency failures from crash stack traces. To answer this research

Table 6.3. RQ1: ConCrash Effectiveness Results

ID	FR	FRT (sec.)					FTID					FTS
	%	min	max	avg	sd	ci	min	max	avg	sd	ci	avg
1	100%	27	99	63	34	30	1	4	2	2	1	4
2	100%	22	85	42	27	24	1	4	2	1	1	4
3	100%	10	16	13	2	2	1	1	1	0	0	4
4	100%	10	21	15	5	4	1	2	2	1	0	5
5	100%	39	84	70	18	16	2	4	3	1	1	5
6	100%	7	7	7	0	0	1	1	1	0	0	4
7	100%	27	31	30	2	1	1	1	1	0	0	3
8	100%	26	185	107	64	56	1	15	8	6	5	6
9	100%	17	33	25	7	6	1	1	1	0	0	5
10	100%	23	183	92	68	60	1	9	5	3	3	10
AVG	**100%**	**21**	**74**	**46**	**23**	**20**	**1**	**4**	**3**	**1**	**1**	**5**

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

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

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 ($\geq$ 2.0 and < 10.0), or HIGH ($\geq$ 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.

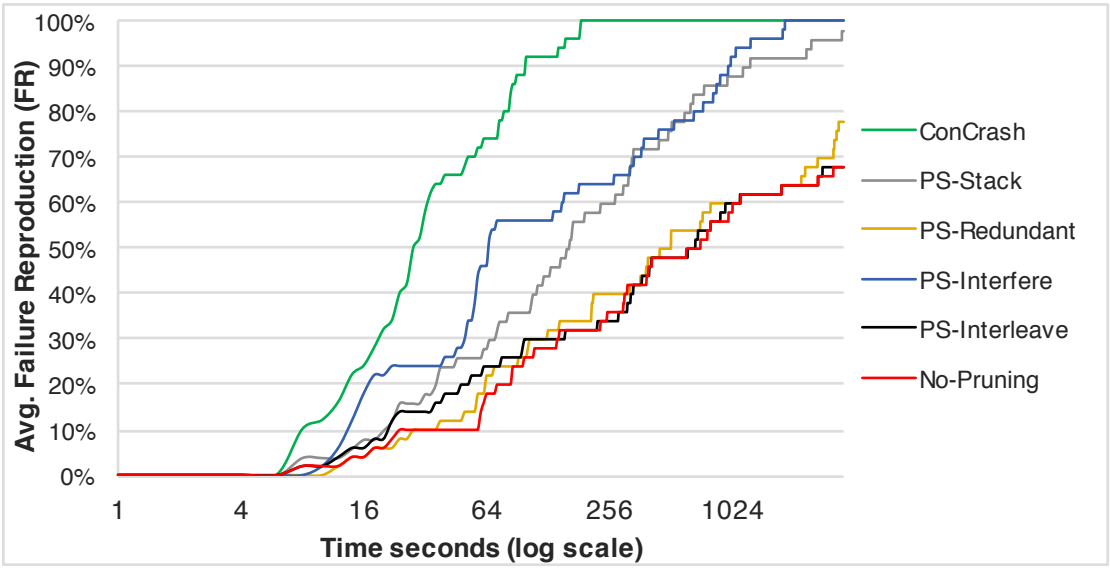
`PS-Redundant` is particularly effective when the execution space of the CUT methods is characterized by few execution paths. In such situation, the invocation of the same method with different parameters leads to a redundant sequential coverage, thus increasing the effectiveness of `PS-Redundant`.

`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.

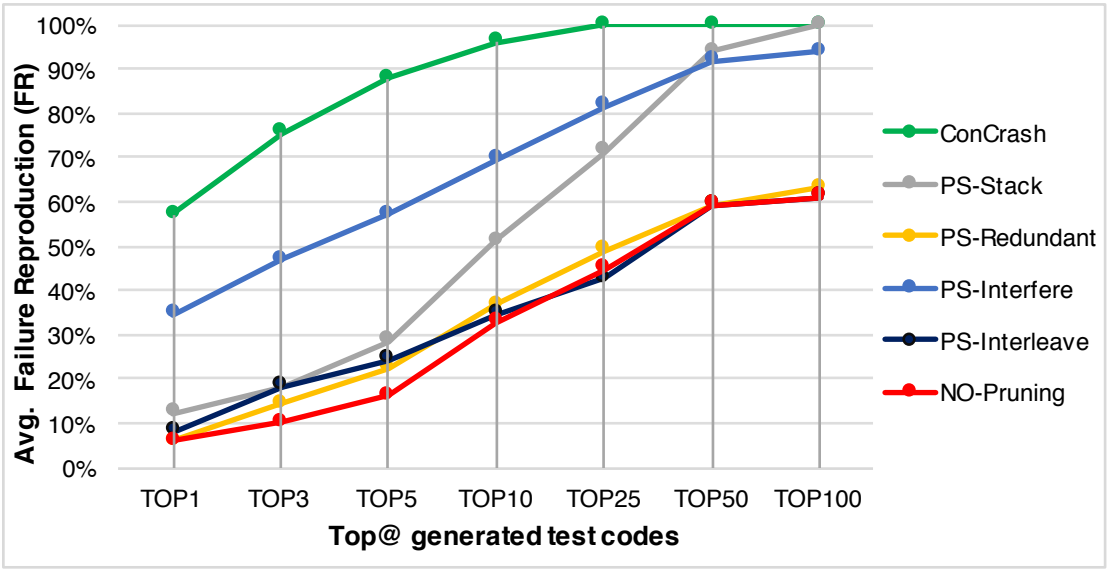
`PS-Interleave` is particularly effective when a CUT largely adopts synchronization mechanisms to access its object fields. This is the case, for instance, of a Java class that declares most of its methods as synchronized. In such case, `PS-Interleave` prunes many irrelevant test cases. For instance, `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 achive the highest speedup for every subjects. For instance, `PS-Stack` and `PS-Interleave` are more effective than `PS-Interfere` for two subjects (ID 4 and ID 8, respectively). This result suggests that 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.



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



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

Figure 6.1. Aggregate comparison of the ConCrash pruning strategies for the ten subjects

Table 6.5. Comparison with test case generators

ID	ConTeGe [104]				AutoConTest [132]			
	FR %	FRT avg	FTID avg	FTS avg	FR %	FRT avg	FTID avg	FTS avg
1	0%	18,000	14,177	N/A	0%	18,000	N/A	N/A
2	0%	18,000	7,736	N/A	0%	18,000	N/A	N/A
3	0%	18,000	25,712	N/A	100%	23	1	56
4	80%	4,487	7,465	12	0%	18,000	6	N/A
5	0%	18,000	1,491	N/A	0%	18,000	6	N/A
6	20%	14,510	5,796	10	-	-	-	-
7	0%	18,000	34,960	N/A	100%	93	1	124
8	40%	12,387	25,215	10	0%	18,000	N/A	N/A
9	40%	14,410	41,461	15	-	-	-	-
10	0%	18,000	39,912	N/A	-	-	-	-
AVG	**18%**	**15,379**	**20,392**	**12**	**28%**	**12,874**	**4**	**90**

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 AUTOCONTEST is comparable with CONCRASH in the cases it succeeds, but AUTOCONTEST generates much larger test cases than CONCRASH.

Our results suggest that testing techniques that are designed as failure-oblivious 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 AUTOCONTEST 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 pseudo-deterministic, 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.

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 evaluation appeared in the proceedings of the Joint Meeting on Foundations of Software Engineering (FSE) in 2017 [11].

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 proposed 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 CONCRASH, the first automated technique that reproduces concurrency failures from the limited information available in crash stack traces. CONCRASH differs from the approaches proposed in the literature in that it requires commonly 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 pruning 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

[1] Agarwal, R., Sasturkar, A., Wang, L. and Stoller, S. D. [2005]. Optimized run-time race detection and atomicity checking using partial discovered types, *Proceedings of the International Conference on Automated Software Engineering*, ASE '05, ACM, pp. 233–242.

[2] Altekar, G. and Stoica, I. [2009]. Odr: output-deterministic replay for multicore debugging, *Proceedings of the Symposium on Operating Systems Principles*, SOSP '09, ACM, pp. 193–206.

[3] Andrews, G. R. [1991]. *Concurrent programming: principles and practice*, Benjamin/Cummings Publishing Company.

[4] Andrews, G. R. and Schneider, F. B. [1983]. Concepts and notations for concurrent programming, *ACM Computing Surveys* **15**(1): 3–43.

[5] Armstrong, J. [2013]. *Programming Erlang*, Pragmatic Bookshelf.

[6] Artho, C., Havelund, K. and Biere, A. [2003]. High-level data races, *Software Testing, Verification and Reliability* **13**(4): 207–227.

[7] Artzi, S., Ernst, M. D., Kieżun, A., Pacheco, C. and Perkins, J. H. [2006]. Finding the needles in the haystack: Generating legal test inputs for object-oriented programs, *Workshop on Model-Based Testing and Object-Oriented Systems*, M-TOOS '06.

[8] Artzi, S., Kim, S. and Ernst, M. D. [2008]. Recrash: Making software failures reproducible by preserving object states, *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '08, Springer, pp. 542–565.

[9] Batty, M., Owens, S., Sarkar, S., Sewell, P. and Weber, T. [2011]. Mathematizing c++ concurrency, *Proceedings of the Symposium on Principles of Programming Languages*, POPL '11, ACM, pp. 55–66.

[10] Bianchi, F. A., Margara, A. and Pezzè, M. [2018]. A survey of recent trends in testing concurrent software systems, *IEEE Transactions on Software Engineering* **44**(8): 747–783.

[11] Bianchi, F. A., Pezzè, M. and Terragni, V. [2017]. Reproducing concurrency failures from crash stacks, *Proceedings of the Joint Meeting on Foundations of Software Engineering*, ESEC/FSE '17, ACM, pp. 705–716.

[12] Billes, M., Møller, A. and Pradel, M. [2017]. Systematic black-box analysis of collaborative web applications, *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI '17, ACM, pp. 171–184.

[13] Billes, M., Møller, A. and Pradel, M. [2017]. Systematic black-box analysis of collaborative web applications, *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI '17, ACM, pp. 171–184.

[14] Biswas, S., Huang, J., Sengupta, A. and Bond, M. D. [2014]. Doublechecker: efficient sound and precise atomicity checking, *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI '14, ACM, pp. 28–39.

[15] Boehm, H.-J. and Adve, S. V. [2008]. Foundations of the c++ concurrency memory model, *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI '08, ACM, pp. 68–78.

[16] Bond, M. D., Coons, K. E. and McKinley, K. S. [2010]. Pacer: Proportional detection of data races, *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI '10, ACM, pp. 255–268.

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

[18] Burckhardt, S., Kothari, P., Musuvathi, M. and Nagarakatte, S. [2010]. A randomized scheduler with probabilistic guarantees of finding bugs, *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '10, ACM, pp. 167–178.

[19] Burnim, J., Sen, K. and Stergiou, C. [2011]. Testing concurrent programs on relaxed memory models, *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '11, ACM, pp. 122–132.

[20] Cai, Y. and Cao, L. [2015]. Effective and precise dynamic detection of hidden races for java programs, *Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE '15, ACM, pp. 450–461.

[21] Cai, Y. and Chan, W. K. [2012]. Magicfuzzer: Scalable deadlock detection for large-scale applications, *Proceedings of the International Conference on Software Engineering*, ICSE '12, IEEE Computer Society, pp. 606–616.

[22] Cai, Y., Wu, S. and Chan, W. K. [2014]. Conlock: A constraint-based approach to dynamic checking on deadlocks in multithreaded programs, *Proceedings of the International Conference on Software Engineering*, ICSE '14, ACM, pp. 491–502.

[23] Chen, F., Serbanuta, T. F. and Rosu, G. [2008]. jpredictor: A predictive runtime analysis tool for java, *Proceedings of the International Conference on Software Engineering*, ICSE '08, ACM, pp. 221–230.

[24] Chen, J., Hierons, R. and Ural, H. [2004]. Conditions for resolving observability problems in distributed testing, *Proceedings of the IFIP International Conference on Formal Techniques for Networked and Distributed Systems*, Springer, pp. 229–242.

[25] Chen, J. and MacDonald, S. [2007]. Testing concurrent programs using value schedules, *Proceedings of the International Conference on Automated Software Engineering*, ASE '07, ACM, pp. 313–322.

[26] Chen, N. and Kim, S. [2015]. Star: stack trace based automatic crash reproduction via symbolic execution, *IEEE Transactions on Software Engineering* **41**(2): 198–220.

[27] Chen, Q., Wang, L., Yang, Z. and Stoller, S. D. [2009]. Have: Detecting atomicity violations via integrated dynamic and static analysis, *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*, FASE '09, Springer, pp. 425–439.

[28] Choi, J.-D., Lee, K., Loginov, A., O'Callahan, R., Sarkar, V. and Sridharan, M. [2002]. Efficient and precise datarace detection for multithreaded object-oriented programs, *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI '02, ACM, pp. 258–269.

[29] Choudhary, A., Lu, S. and Pradel, M. [2017]. Efficient detection of thread safety violations via coverage-guided generation of concurrent tests, *Proceedings of the International Conference on Software Engineering*, ICSE '17, IEEE Computer Society, pp. 266–277.

[30] Cogumbreiro, T., Hu, R., Martins, F. and Yoshida, N. [2015]. Dynamic deadlock verification for general barrier synchronisation, *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, PPoPP '15, ACM, pp. 150–160.

[31] Computerworld [2012]. Nasdaq's facebook glitch came from race conditions, `https://www.computerworld.com/article/2504676/financial-it/nasdaq-s-facebook-glitch-came-from--race-conditions-.html`. Last access: december 2017.

[32] Coons, K. E., Burckhardt, S. and Musuvathi, M. [2010]. Gambit: Effective unit testing for concurrency libraries, *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, ACM, pp. 15–24.

[33] De Moura, L. and Bjørner, N. [2008]. Z3: An efficient SMT solver, *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS/ETAPS '08, Springer, pp. 337–340.

[34] Dimitrov, D., Raychev, V., Vechev, M. and Koskinen, E. [2014]. Commutativity race detection, *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI '14, ACM, pp. 305–315.

[35] Dingel, J. and Liang, H. [2004]. Automating comprehensive safety analysis of concurrent programs using verisoft and txl, *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '12, ACM, pp. 13–22.

[36] Dinning, A. and Schonberg, E. [1990]. An empirical comparison of monitoring algorithms for access anomaly detection, *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, PPOPP '90, ACM, pp. 1–10.

[37] Elmas, T., Qadeer, S. and Tasiran, S. [2007]. Goldilocks: A race and transaction-aware java runtime, *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI '07, ACM, pp. 245–255.

[38] Eslamimehr, M. and Palsberg, J. [2014a]. Race directed scheduling of concurrent programs, *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, ACM, pp. 301–314.

[39] Eslamimehr, M. and Palsberg, J. [2014b]. Sherlock: scalable deadlock detection for concurrent programs, *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '14, ACM, pp. 353–365.

[40] Farzan, A., Madhusudan, P., Razavi, N. and Sorrentino, F. [2012]. Predicting null-pointer dereferences in concurrent programs, *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '12, ACM, pp. 1–11.

[41] Flanagan, C. and Freund, S. N. [2004]. Atomizer: A dynamic atomicity checker for multithreaded programs, *Proceedings of the Symposium on Principles of Programming Languages*, POPL '04, ACM, pp. 256–267.

[42] Flanagan, C. and Freund, S. N. [2009]. Fasttrack: Efficient and precise dynamic race detection, *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI '09, ACM, pp. 121–133.

[43] Flanagan, C., Freund, S. N. and Yi, J. [2008]. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs, *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI '08, ACM, pp. 293–303.

[44] Fonseca, P., Li, C. and Rodrigues, R. [2011]. Finding complex concurrency bugs in large multi-threaded applications, *Proceedings of the ACM SIGOPS EuroSys European Conference on Computer Systems*, EuroSys '11, ACM, pp. 215–228.

[45] Francesco Bianchi [2017]. ConCrash, `http://star.inf.usi.ch/star/software/concrash/`. Last access: 5 April 2018.

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

[47] Ganai, M. K. [2011]. Scalable and precise symbolic analysis for atomicity violations, *Proceedings of the International Conference on Automated Software Engineering*, ASE '11, IEEE Computer Society, pp. 123–132.

[48] Gao, Q., Zhang, W., Chen, Z., Zheng, M. and Qin, F. [2011]. 2ndstrike: Toward manifesting hidden concurrency typestate bugs, *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '11, ACM, pp. 239–250.

[49] Godefroid, P. [1996]. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*, Springer.

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

[51] Hammer, C., Dolby, J., Vaziri, M. and Tip, F. [2008]. Dynamic detection of atomic-set-serializability violations, *Proceedings of the International Conference on Software Engineering*, ICSE '08, IEEE Computer Society, pp. 231–240.

[52] Havelund, K. [2000]. Using runtime analysis to guide model checking of java programs, *Proceedings of the International SPIN Workshop on SPIN Model Checking and Software Verification*, SPIN '00, Springer, pp. 245–264.

[53] Herlihy, M. P. and Wing, J. M. [1990]. Linearizability: A correctness condition for concurrent objects, *ACM Transactions on Programming Languages and Systems* **12**(3): 463–492.

[54] Hong, S., Ahn, J., Park, S., Kim, M. and Harrold, M. J. [2012]. Testing concurrent programs to achieve high synchronization coverage, *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '12, ACM, pp. 210–220.

[55] Hovemeyer, D., Pugh, W. and Spacco, J. [2002]. Atomic instructions in java, *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '02, Springer, pp. 133–154.

[56] Hsiao, C.-H., Yu, J., Narayanasamy, S., Kong, Z., Pereira, C. L., Pokam, G. A., Chen, P. M. and Flinn, J. [2014]. Race detection for event-driven mobile applications, *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI '14, ACM, pp. 326–336.

[57] Huang, J., Liu, P. and Zhang, C. [2010]. Leap: Lightweight deterministic multi-processor replay of concurrent java programs, *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, ACM, pp. 207–216.

[58] Huang, J., Luo, Q. and Rosu, G. [2015]. Gpredict: Generic predictive concurrency analysis, *Proceedings of the International Conference on Software Engineering*, ICSE '15, ACM.

[59] Huang, J., Meredith, P. O. and Rosu, G. [2014]. Maximal sound predictive race detection with control flow abstraction, *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI '14, ACM, pp. 337–348.

[60] Huang, J. and Zhang, C. [2011]. Persuasive prediction of concurrency access anomalies, *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '11, ACM, pp. 144–154.

[61] Huang, J., Zhang, C. and Dolby, J. [2013]. Clap: Recording local executions to reproduce concurrency failures, *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI '13, ACM, pp. 141–152.

[62] Jin, W. and Orso, A. [2012]. Bugredux: Reproducing field failures for inhouse debugging, *Proceedings of the International Conference on Software Engineering*, ICSE '12, IEEE Computer Society, pp. 474–484.

[63] Joshi, P., Naik, M., Sen, K. and Gay, D. [2010]. An effective dynamic analysis for detecting generalized deadlocks, *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, ACM, pp. 327–336.

[64] Joshi, P., Park, C., Sen, K. and Naik, M. [2009]. A randomized dynamic program analysis technique for detecting real deadlocks, *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI '09, ACM, pp. 110–120.

[65] Joshi, P. and Sen, K. [2008]. Predictive typestate checking of multithreaded java programs, *Proceedings of the International Conference on Automated Software Engineering*, ASE '08, IEEE Computer Society, pp. 288–296.

[66] Kahlon, V. and Wang, C. [2010]. Universal causality graphs: A precise happens-before model for detecting bugs in concurrent programs, *Proceedings of the International Conference on Computer Aided Verification*, CAV '10, Springer, pp. 434–449.

[67] Karmani, R. K., Madhusudan, P. and Moore, B. M. [2011]. Thread contracts for safe parallelism, *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, ACM, pp. 125–134.

[68] Kasikci, B., Zamfir, C. and Candea, G. [2012]. Data races vs. data race bugs: Telling the difference with portend, *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, ACM, pp. 185–198.

[69] Kasikci, B., Zamfir, C. and Candea, G. [2013]. Racemob: Crowdsourced data race detection, *Proceedings of the Symposium on Operating Systems Principles*, SOSP '13, ACM, pp. 406–422.

[70] Kim, K., Yavuz-Kahveci, T. and Sanders, B. A. [2009]. Precise data race detection in a relaxed memory model using heuristic-based model checking, *Proceedings of the International Conference on Automated Software Engineering*, ASE '09, IEEE Computer Society, pp. 495–499.

[71] Lai, Z., Cheung, S. C. and Chan, W. K. [2010]. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing, *Proceedings of the International Conference on Software Engineering*, ICSE '10, ACM, pp. 235–244.

[72] Lamport, L. [1978]. Time, clocks, and the ordering of events in a distributed system, *Communications of the ACM* **21**(7): 558–565.

[73] Lamport, L. [1979]. How to make a multiprocessor computer that correctly executes multiprocess programs, *IEEE Transactions on Computers* **C-28**(9): 690–691.

[74] Lauterburg, S., Dotta, M., Marinov, D. and Agha, G. A. [2009]. A framework for state-space exploration of java-based actor programs, *Proceedings of the International Conference on Automated Software Engineering*, ASE '09, IEEE Computer Society, pp. 468–479.

[75] Lee, D., Chen, P. M., Flinn, J. and Narayanasamy, S. [2012]. Chimera: Hybrid program analysis for determinism, *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI '12, ACM, pp. 463–474.

[76] Leveson, N. G. and Turner, C. S. [1993]. An investigation of the Therac-25 accidents, *Computer* **26**(7): 18–41.

[77] Li, D., Srisa-an, W. and Dwyer, M. B. [2011]. Sos: Saving time in dynamic race detection with stationary analysis, *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, ACM, pp. 35–50.

[78] Lipton, R. J. [1975]. Reduction: A method of proving properties of parallel programs, *Communications of the ACM* **18**(12): 717–721.

[79] Lu, S., Jiang, W. and Zhou, Y. [2007]. A study of interleaving coverage criteria, *Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC-FSE companion '07, ACM, pp. 533–536.

[80] Lu, S., Park, S., Hu, C., Ma, X., Jiang, W., Li, Z., Popa, R. A. and Zhou, Y. [2007]. Muvi: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs, *Proceedings of the Symposium on Operating Systems Principles*, SOSP '07, ACM, pp. 103–116.

[81] Lu, S., Park, S., Seo, E. and Zhou, Y. [2008]. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics, *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '08, ACM, pp. 329–339.

[82] Lu, S., Tucek, J., Qin, F. and Zhou, Y. [2006]. AVIO: Detecting atomicity violations via access interleaving invariants, *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '06, ACM, pp. 37–48.

[83] Machado, N., Lucia, B. and Rodrigues, L. [2015]. Concurrency debugging with differential schedule projections, *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI '15, ACM, pp. 586–595.

[84] Machado, N., Lucia, B. and Rodrigues, L. [2016]. Production-guided concurrency debugging, *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, ACM, pp. 29:1–29:12.

[85] Maiya, P., Kanade, A. and Majumdar, R. [2014]. Race detection for Android applications, *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI '14, ACM, pp. 316–325.

[86] Manson, J., Pugh, W. and Adve, S. V. [2005]. The java memory model, *Proceedings of the Symposium on Principles of Programming Languages*, POPL '05, ACM, pp. 378–391.

[87] Marino, D., Musuvathi, M. and Narayanasamy, S. [2009]. Literace: Effective sampling for lightweight data-race detection, *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI '09, ACM, pp. 134–143.

[88] Musuvathi, M. and Qadeer, S. [2007]. Iterative context bounding for systematic testing of multithreaded programs, *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI '07, ACM, pp. 446–455.

[89] Mutlu, E., Tasiran, S. and Livshits, B. [2015]. Detecting javascript races that matter, *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '15, ACM.

[90] Narayanasamy, S., Wang, Z., Tigani, J., Edwards, A. and Calder, B. [2007]. Automatically classifying benign and harmful data races using replay analysis, *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI '07, ACM, pp. 22–31.

[91] Netzer, R. H. B. [1991]. *Race Condition Detection for Debugging Shared-memory Parallel Programs*, PhD thesis.

[92] Nistor, A., Luo, Q., Pradel, M., Gross, T. R. and Marinov, D. [2012]. Ballerina: Automatic generation and clustering of efficient random unit tests for multithreaded code, *Proceedings of the International Conference on Software Engineering*, ICSE '12, IEEE Computer Society, pp. 727–737.

[93] Norris, B. and Demsky, B. [2013]. Cdschecker: Checking concurrent data structures written with c/c++ atomics, *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, ACM, pp. 131–150.

[94] O'Callahan, R. and Choi, J.-D. [2003]. Hybrid dynamic data race detection, *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, PPOPP '03, ACM, pp. 167–178.

[95] Pacheco, C., Lahiri, S. K., Ernst, M. D. and Ball, T. [2007]. Feedback-directed random test generation, *Proceedings of the International Conference on Software Engineering*, ICSE '07, ACM, pp. 75–84.

[96] Papadimitriou, C. H. [1979]. The serializability of concurrent database updates, *Journal of the ACM* **26**(4): 631–653.

[97] Park, C.-S. and Sen, K. [2008]. Randomized active atomicity violation detection in concurrent programs, *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '08, ACM, pp. 135–145.

[98] Park, S., Lu, S. and Zhou, Y. [2009]. Ctrigger: Exposing atomicity violation bugs from their hiding places, *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '09, ACM, pp. 25–36.

[99] Park, S., Vuduc, R. W. and Harrold, M. J. [2010]. Falcon: Fault localization in concurrent programs, *Proceedings of the International Conference on Software Engineering*, ICSE '10, ACM, pp. 245–254.

[100] Park, S., Zhou, Y., Xiong, W., Yin, Z., Kaushik, R., Lee, K. H. and Lu, S. [2009]. Pres: probabilistic replay with execution sketching on multiprocessors, *Proceedings of the Symposium on Operating Systems Principles*, ACM, pp. 177–192.

[101] Perkovic, D. and Keleher, P. J. [1996]. Online data-race detection via coherency guarantees, *Proceedings of the Symposium on Operating Systems Design and Implementation*, OSDI '96, ACM, pp. 47–57.

[102] Petrov, B., Vechev, M., Sridharan, M. and Dolby, J. [2012]. Race detection for web applications, *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI '12, ACM, pp. 251–262.

[103] Pozniansky, E. and Schuster, A. [2003]. Efficient on-the-fly data race detection in multithreaded c++ programs, *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, ACM, pp. 179–190.

[104] Pradel, M. and Gross, T. R. [2012]. Fully automatic and precise detection of thread safety violations, *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI '12, ACM, pp. 521–530.

[105] Pradel, M. and Gross, T. R. [2013]. Automatic testing of sequential and concurrent substitutability, *Proceedings of the International Conference on Software Engineering*, ICSE '13, IEEE Computer Society, pp. 282–291.

[106] Pradel, M., Huggler, M. and Gross, T. R. [2014]. Performance regression testing of concurrent classes, *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA 2014, ACM, pp. 13–25.

[107] Prvulovic, M. and Torrellas, J. [2003]. Reenact: Using thread-level speculation mechanisms to debug data races in multithreaded codes, ISCA '03, ACM, pp. 110–121.

[108] Pugh, W. [1999]. Fixing the java memory model, *Proceedings of the ACM Conference on Java Grande*, JAVA '99, ACM, pp. 89–98.

[109] Rajagopalan, A. K. and Huang, J. [2015]. Rdit: Race detection from incomplete traces, *Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE '15, ACM.

[110] Raychev, V., Vechev, M. and Sridharan, M. [2013]. Effective race detection for event-driven programs, *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, ACM, pp. 151–166.

[111] Samak, M. and Ramanathan, M. K. [2014a]. Multithreaded test synthesis for deadlock detection, *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '14, ACM, pp. 473–489.

[112] Samak, M. and Ramanathan, M. K. [2014b]. Trace driven dynamic deadlock detection and reproduction, *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, ACM, pp. 29–42.

[113] Samak, M. and Ramanathan, M. K. [2015]. Synthesizing tests for detecting atomicity violations, *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '15, ACM.

[114] Samak, M., Ramanathan, M. K. and Jagannathan, S. [2015]. Synthesizing racy tests, *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI '15, ACM, pp. 175–185.

[115] Samak, M., Tripp, O. and Ramanathan, M. K. [2016]. Directed synthesis of failing concurrent executions, *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '16, ACM, pp. 430–446.

[116] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P. and Anderson, T. E. [1997]. Eraser: A dynamic data race detector for multithreaded programs, *ACM Transactions on Computer Systems* **15**(4): 391–411.

[117] Security Focus [2004]. Software Bug Contributed to Blackout, `http://www.securityfocus.com/news/8016`. Last access: november 2017.

[118] Sen, K. [2007]. Effective random testing of concurrent programs, *Proceedings of the International Conference on Automated Software Engineering*, ASE '07, ACM, pp. 323–332.

[119] Sen, K. [2008]. Race directed random testing of concurrent programs, *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI '08, ACM, pp. 11–21.

[120] Sen, K. and Agha, G. [2006]. Automated systematic testing of open distributed programs, *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*, FASE '06, Springer, pp. 339–356.

[121] Shacham, O., Bronson, N., Aiken, A., Sagiv, M., Vechev, M. and Yahav, E. [2011]. Testing atomicity of composed concurrent operations, *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, ACM, pp. 51–64.

[122] Shacham, O., Sagiv, M. and Schuster, A. [2005]. Scaling model checking of dataraces using dynamic information, *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, ACM, pp. 107–118.

[123] Sheng, T., Vachharajani, N., Eranian, S., Hundt, R., Chen, W. and Zheng, W. [2011]. Racez: A lightweight and non-invasive race detection tool for production applications, *Proceedings of the International Conference on Software Engineering*, ICSE '11, ACM, pp. 401–410.

[124] Shi, Y., Park, S., Yin, Z., Lu, S., Zhou, Y., Chen, W. and Zheng, W. [2010]. Do i use the wrong definition?: Defuse: Definition-use invariants for detecting concurrency and sequential bugs, *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, ACM, pp. 160–174.

[125] Shirako, J., Peixotto, D. M., Sarkar, V. and Scherer, W. N. [2008]. Phasers: A unified deadlock-free construct for collective and point-to-point synchronization, *Proceedings of the Annual International Conference on Supercomputing*, ICS '08, ACM, pp. 277–288.

[126] Singhal, M. [1989]. Deadlock detection in distributed systems, *IEEE Computer* **22**(11): 37–48.

[127] Sinha, N. and Wang, C. [2010]. Staged concurrent program analysis, *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, ACM, pp. 47–56.

[128] Smaragdakis, Y., Evans, J., Sadowski, C., Yi, J. and Flanagan, C. [2012]. Sound predictive race detection in polynomial time, *Proceedings of the Symposium on Principles of Programming Languages*, POPL '12, ACM, pp. 387–400.

[129] Sorrentino, F., Farzan, A. and Madhusudan, P. [2010]. Penelope: Weaving threads to expose atomicity violations, *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, ACM, pp. 37–46.

[130] Steenbuck, S. and Fraser, G. [2013]. Generating unit tests for concurrent classes, *Proceedings of the International Conference on Software Testing, Verification and Validation*, ICST '13, IEEE Computer Society, pp. 144–153.

[131] Tasharofi, S., Pradel, M., Lin, Y. and Johnson, R. E. [2013]. Bita: Coverage-guided, automatic testing of actor programs, *Proceedings of the International Conference on Automated Software Engineering*, ASE '13, IEEE Computer Society, pp. 114–124.

[132] Terragni, V. and Cheung, S.-C. [2016]. Coverage-driven test code generation for concurrent classes, *Proceedings of the International Conference on Software Engineering*, ICSE '16, ACM, pp. 1121–1132.

[133] Terragni, V., Cheung, S.-C. and Zhang, C. [2015]. Recontest: Effective regression testing of concurrent programs, *Proceedings of the International Conference on Software Engineering*, ICSE '15, IEEE Computer Society, pp. 246–256.

[134] Tian, C., Nagarajan, V., Gupta, R. and Tallam, S. [2008]. Dynamic recognition of synchronization operations for improved data race detection, *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '08, ACM, pp. 143–154.

[135] Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P. and Sundaresan, V. [2010]. Soot: A java bytecode optimization framework, *CASCON First Decade High Impact Papers*, IBM Corp., pp. 214–224.

[136] Vaziri, M., Tip, F. and Dolby, J. [2006]. Associating synchronization constraints with data in an object-oriented language, *Proceedings of the Symposium on Principles of Programming Languages*, POPL '06, ACM, pp. 334–345.

[137] Veeraraghavan, K., Chen, P. M., Flinn, J. and Narayanasamy, S. [2011]. Detecting and surviving data races using complementary schedules, *Proceedings of the Symposium on Operating Systems Principles*, SOSP '11, ACM, pp. 369–384.

[138] Visser, W., Păsăreanu, C. S. and Khurshid, S. [2004]. Test input generation with java pathfinder, *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '04, ACM, pp. 97–107.

[139] von Praun, C. and Gross, T. R. [2001]. Object race detection, *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '01, ACM, pp. 70–82.

[140] Wang, C., Said, M. and Gupta, A. [2011]. Coverage guided systematic concurrency testing, *Proceedings of the International Conference on Software Engineering*, ICSE '11, ACM, pp. 221–230.

[141] Wang, L. and Stoller, S. D. [2006]. Runtime analysis of atomicity for multithreaded programs, *IEEE Transactions on Software Engineering* **32**(2): 93–110.

[142] Weeratunge, D., Zhang, X. and Jagannathan, S. [2010]. Analyzing multicore dumps to facilitate concurrency bug reproduction, *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '10, ACM, pp. 155–166.

[143] Wester, B., Devecsery, D., Chen, P. M., Flinn, J. and Narayanasamy, S. [2013]. Parallelizing data race detection, *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, ACM, pp. 27–38.

[144] Wu, R., Xiao, X., Cheung, S.-C., Zhang, H. and Zhang, C. [2016]. Casper: An efficient approach to call trace collection, *Proceedings of the Symposium on Principles of Programming Languages*, POPL '16, ACM, pp. 678–690.

[145] Wu, R., Zhang, H., Cheung, S.-C. and Kim, S. [2014]. Crashlocator: Locating crashing faults based on crash stacks, *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '14, ACM, pp. 204–214.

[146] Wyatt, D. [2013]. *Akka Concurrency*, Artima Incorporation.

[147] Xu, M., Bodík, R. and Hill, M. D. [2005]. A serializability violation detector for shared-memory server programs, *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI '05, ACM, pp. 1–14.

[148] Yu, J., Narayanasamy, S., Pereira, C. and Pokam, G. [2012]. Maple: a coverage-driven testing tool for multithreaded programs, *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, ACM, pp. 485–502.

[149] Yu, T., Srisa-an, W. and Rothermel, G. [2013]. Simracer: An automated framework to support testing for process-level races, *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '13, ACM, pp. 167–177.

[150] Yu, T., Srisa-an, W. and Rothermel, G. [2014]. Simrt: An automated framework to support regression testing for data races, *Proceedings of the International Conference on Software Engineering*, ICSE 2014, ACM, pp. 48–59.

[151] Yu, T., Zaman, T. S. and Wang, C. [2017]. Descry: Reproducing system-level concurrency failures, *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '17, ACM, pp. 694–704.

[152] Yu, Y., Rodeheffer, T. and Chen, W. [2005]. Racetrack: Efficient detection of data race conditions via adaptive tracking, *Proceedings of the Symposium on Operating Systems Principles*, SOSP '05, ACM, pp. 221–234.

[153] Yuan, X., Wu, C., Wang, Z., Li, J., Yew, P.-C., Huang, J., Feng, X., Lan, Y., Chen, Y. and Guan, Y. [2015]. ReCBuLC: reproducing concurrency bugs using local clocks, *Proceedings of the International Conference on Software Engineering*, IEEE Computer Society, pp. 824–834.

[154] Zamfir, C. and Candea, G. [2010]. Execution synthesis: A technique for automated software debugging, *Proceedings of the ACM SIGOPS EuroSys European Conference on Computer Systems*, EuroSys '10, ACM, pp. 321–334.

[155] Zhai, K., Xu, B., Chan, W. K. and Tse, T. H. [2012]. Carisma: A context-sensitive approach to race-condition sample-instance selection for multi-threaded applications, *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '12, ACM, pp. 221–231.

[156] Zhang, W., Lim, J., Olichandran, R., Scherpelz, J., Jin, G., Lu, S. and Reps, T. [2011]. Conseq: Detecting concurrency bugs through sequential errors, *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, ACM, pp. 251–264.

[157] Zhang, W., Sun, C. and Lu, S. [2010]. Conmem: Detecting severe concurrency bugs through an effect-oriented approach, *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, ACM, pp. 179–192.

[158] Zhou, J., Xiao, X. and Zhang, C. [2012]. Stride: Search-based deterministic replay in polynomial time via bounded linkage, *Proceedings of the International Conference on Software Engineering*, IEEE Computer Society, pp. 892–902.