# A Survey of Recent Trends in Testing Concurrent Software Systems

Francesco Adalberto Bianchi, Alessandro Margara, and Mauro Pezzè

**Abstract**—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*. Concurrent systems are difficult to test, since the faults that derive from their concurrent nature depend on the interleavings of the actions performed by the individual execution flows. Testing techniques that target these faults must take into account the concurrency aspects of the systems. The increasingly rapid spread of parallel and distributed architectures led to a deluge of concurrent software systems, and the explosion of testing techniques for such systems in the last decade. The current lack of a comprehensive classification, analysis and comparison of the many testing techniques for concurrent systems limits the understanding of the strengths and weaknesses of each approach and hampers the future advancements in the field. This survey provides a framework to capture the key features of the available techniques to test concurrent software systems, identifies a set of classification criteria to review and compare the available techniques, and discusses in details their strengths and weaknesses, leading to a thorough assessment of the field and paving the road for future progresses.

**Index Terms**—Survey, classification, testing, concurrent systems, parallel systems, distributed systems

## 1 Introduction

Concurrent systems are common in several application domains: many interactive applications exploit the multi-threaded paradigm to decouple the input-output processing from the back-end computation; applications developed for single nodes often exploit multiple threads to enable parallel computations on multi-core architectures; Web applications implement the client-server communication paradigm with client-side and server-side computations; mobile applications often access data and interact with remote servers; peer-to-peer services coordinate a multitude of computing nodes.

Concurrent software systems are composed of multiple execution flows that execute simultaneously, and the need to synchronize the execution flows leads to new problems and introduces new design and verification challenges. The behavior of concurrent systems depends not only on the sequence of actions executed within each individual flow, but also on the interleavings of the actions in the different execution flows. Wrong interleavings may lead to concurrency faults regardless of the correctness of the computation of each execution flow. The problem of developing reliable concurrent systems has attracted a lot of interest in the software engineering community, and has led to several solutions for designing [1], implementing [2], refactoring [3], modeling, verifying and validating concurrent software systems [4].

Concurrency faults are intrinsically non-deterministic, since they occur only in the presence of specific interleavings, and the interleavings depend on execution conditions that are not under the direct control of the program. The testing techniques that address the problem of efficiently exploring the space of the interleavings consider one or more of the following activities: (i) generating test cases, which are sequences of operations that stimulate the system; (ii) selecting a subset of interleavings for the execution flows; (iii) executing the system with the selected test cases and interleavings and validating the results.

Although the problem of testing concurrent systems has attracted the attention of the research community since the late seventies, and has grown considerably in the last decade, to the best of our knowledge a precise survey and classification of the progresses and the results in the field is still missing.

In this paper, we provide a comprehensive survey of the state-of-the-art in testing concurrent software systems. We studied the recent literature by systematically browsing the main publishers and scientific search engines, and we traced 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 testing concurrent software systems 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 testing concurrent software systems.

• *The authors are with the Faculty of Informatics, Univeristà della Svizzera italiana, USI Lugano, Lugano 6900, Switzerland. E-mail: {francesco.bianchi, mauro.pezze}@usi.ch, alessandro.margara@polimi.it.*

*Manuscript received 14 June 2016; revised 5 May 2017; accepted 17 May 2017. Date of publication 22 May 2017; date of current version 21 Aug. 2018. (Corresponding author: Alessandro Margara.)*
*Recommended for acceptance by C. Zhang.*
*For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.*
*Digital Object Identifier no. 10.1109/TSE.2017.2707089*

The remainder of the paper is organized as follows. Section 2 introduces background information on concurrent systems, defines the terminology used in the paper and presents a general framework that captures the key aspects of the testing techniques for concurrent systems. Section 3 presents the historical trends in the research on testing concurrent systems and describes the methodology we adopt to select the relevant work. Section 4 introduces a set of criteria to classify testing techniques for concurrent systems. Section 5 surveys and classifies the state-of-the-art techniques. Section 6 discusses the key observations that emerge from the analysis of the literature. Section 7 briefly overviews some important areas that lie on the borders of the scope of this survey, and Section 8 summarizes the contribution of the paper.

## 2 Concurrent Software Systems

In this section, we define the scope of our analysis and introduce the terminology that we adopt in this paper. To do so, we define a conceptual framework that captures the main elements of the different approaches to test concurrent software systems. In the remainder of the paper, we use the framework to structure our survey.

### 2.1 Concurrent Systems

In this survey, we follow the definition of concurrent system proposed by Andrews and Schneider in their popular survey and book [5], [6] 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, like concurrent programs executed on single-core architectures. Depending on the specific architecture and programming paradigm, execution flows can be concretely implemented as processes on different physical machines, processes within the same machine or threads within the same process, 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 only possible when the execution flows are 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 survey.

### 2.2 Interleaving of Execution Flows

The behavior of a concurrent system depends not only on the input parameters and the sequences of instructions of the individual flows, but also on the interleaving of instructions from the different execution flows that comprise the system.

Following the vast majority of approaches that we discuss in Section 5, we introduce the main concepts of concurrency under the assumption of a *sequentially consistent* model [7]. 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, and in the survey we consider approaches regardless of this assumption.

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.

1. Although in the paper we adopt the terminology of Andrews and Schneider, we prefer the term *execution flow* over *process* to avoid biases towards a specific technology.

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 1 exemplifies the impact of the interleaving of instructions from multiple execution flows on the result of an execution. In Listing 1 both execution flows $f_1$ and $f_2$ access a common 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 $x == 0$ 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]

**Listing 1.** A non-deterministic concurrent system

```
begin f_1                begin f_2
if (x==0) {               x=1
  print OK               end f_2
}
end f_1
```

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. This is allowed in the memory models of both Java [8], [9] and C++ [10].

The histories of relaxed systems may violate the properties that characterize the history of sequential consistent systems, leading to new types of possible concurrency faults that cannot occur under the sequential consistency hypothesis. In this survey we review both the many approaches that work under the sequential consistency hypothesis and the few approaches that extend to relaxed models.

### 2.3 Synchronization Mechanisms

Concurrent programming languages offer various *synchronization mechanisms* to constrain the order of instructions and thus prevent erroneous behaviors. The synchronization mechanisms depend on the concurrency paradigm, the granularity of the synchronization structures and the constraints imposed on the system architecture. For instance, Java offers *synchronized* blocks and *atomic* instructions to assure that program blocks are executed without the interleaving of instructions of other execution flows [11]. 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 [12]. 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 [13]. Synchronous message passing ensures that the sender of a message can progress only after the message has been successfully delivered to the recipient [14].

### 2.4 Testing Concurrent Systems

In this paper we focus on software *testing* techniques that target *concurrency faults*, which are faults caused by unexpected interleavings of instructions of otherwise correct execution flows. Concurrency faults can be extremely hard to reveal and reproduce, since they manifest only in the presence of specific interleavings that may be rarely executed. To expose concurrency faults, testing techniques for concurrent systems need 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 comprise the flows.

The many approaches for testing concurrent systems 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 testing concurrent systems. Fig. 1 presents the conceptual framework that we use to provide a comprehensive view of the problem and to organize this survey.

*Approaches* for *testing* concurrent systems deal with specific types of *target system*s and address one or more of the three main aspects of the problem visualized with rectangles in Fig. 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.

Fig. 1 presents a conceptual framework for the testing techniques, but does not prescribe 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.

The 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 both.

The approaches for *selecting interleavings* identify a subset of relevant interleavings to be executed, and target either

2. In this example, we assume that read and write of $x$ are atomic. Relaxing the atomicity assumption would produce even more interleavings.

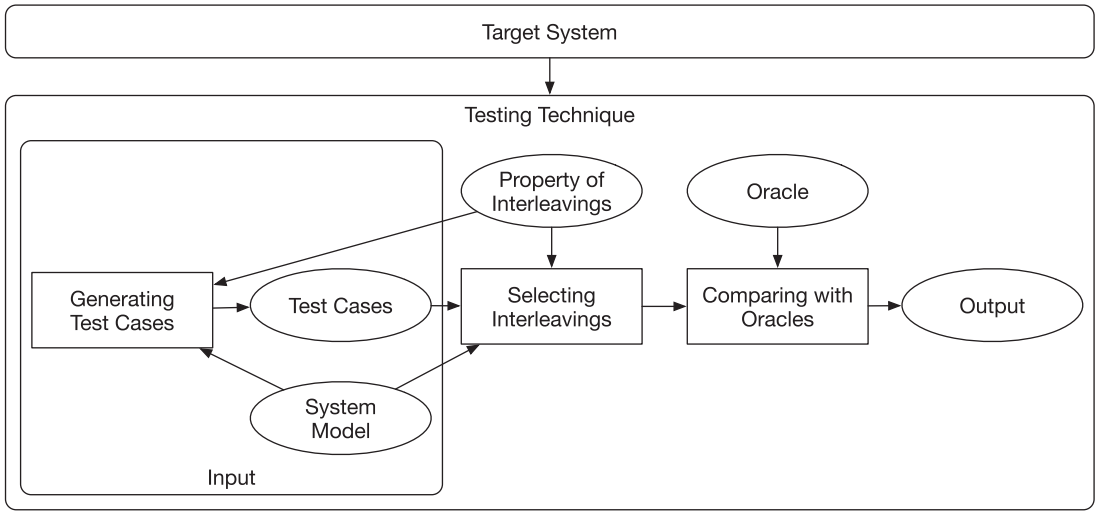


Fig. 1. A general framework for testing concurrent software systems.

the interleaving space as a whole or some specific properties of interleaving. The 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 4, properties of interleaving typically identify patterns of interactions across execution flows that are likely to expose concurrency faults. 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.

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 Trends in Research on Testing Concurrent Systems

In this section we present an analysis of the research on testing concurrent systems conducted in the last fifteen years referring to the seminal work of the last forty years.

Concurrency has been investigated since the early sixties with pioneer work on models for concurrent systems, like the research of Karl Adam Petri [15] and the inspiring work on process algebras of Tony Hoare [16] and Robin Milner [17].

In the seventies, with the emergence of distributed architectures, the focus of the research extended towards the analysis and verification of distributed systems with the Lipton's influential work on the theory of reduction [18] and Lamport's seminal work on distributed systems [7].

The nineties have seen the introduction of the term *testing concurrent systems* with continuity in the literature [19], [20], [21], and the appearing of analysis techniques that are at the core of many popular approaches for testing concurrent systems [22], [23], [24], [25], [26].

The research on testing concurrent systems 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. Our survey indicates that most of the concurrent software testing techniques developed in the last fifteen years target shared memory systems, and only few cope with (distributed) message passing systems, which are addressed mainly by runtime monitoring and model based verification approaches.

To provide a comprehensive survey of the emerging trends in testing concurrent software systems, we systematically review the literature from 2000 to 2015: (i) we 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", (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, as defined

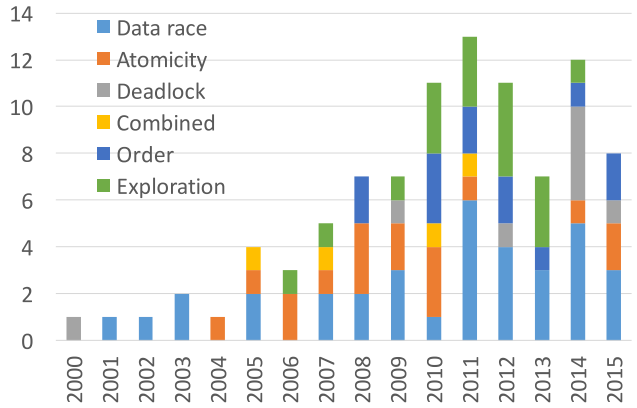


Fig. 2. Number of publications from 2000 to 2015 that witness novel research contributions and address different concurrency properties.

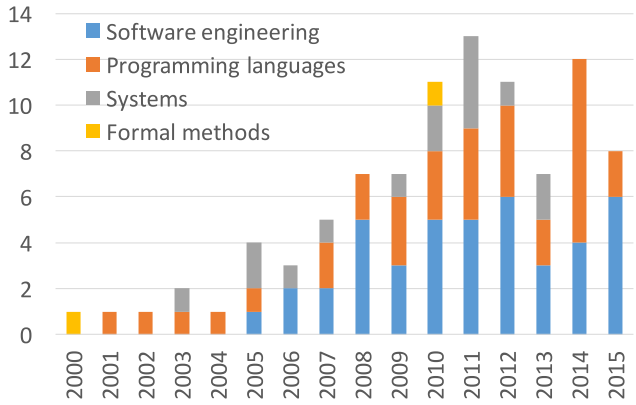


Fig. 3. Number of publications from 2000 to 2015 that witness novel research contributions in different research communities.

in Section 2, for example papers on hardware testing and papers on theoretical aspects or purely monitoring approaches with no direct application to software testing (see Section 7 for an overview of such papers), and (v) filtered out workshop papers and preliminary work that has been later subsumed by conference or journal publications.

We selected 94 papers that were published in the last 15 years and that we identified as unique representatives of clusters of related publications. The papers focus on the main concurrency properties, and reflect the interest of different communities in the area of testing concurrent systems. We discuss the possible biases of our choices in Section 6.8.

Figs. 2 and 3 show the distribution of the papers over the years according to the concurrency properties they deal with and the research communities they refer to,[3] respectively. The figures indicate a relevant increase of interest and results: more than 80 percent of the papers have been published in the second half of the considered period (2008-2015), and more than almost 65 percent have appeared since 2010. Fig. 2 indicates that the recent research has focused on specific interleaving properties, which we use in Section 4 as the main classification criterion. Fig. 3 suggests a growing interest in the software engineering and programming languages communities, and a stable interest in the systems and formal methods communities, whose main focus has been on techniques for monitoring concurrent systems and theoretical investigations, respectively. We briefly discuss both topics in the related work Section 7 since they are relevant but not central to the topic of this paper.

## 4 Towards a Classification Schema

Our analysis of the literature led to the definition of the framework of Fig. 1 in Section 2, which inspired a classification schema presented in this section and used in Section 5 to organize our survey. Fig. 4 overviews our classification schema, which includes seven distinct classification criteria that distinguish techniques based on:

*Input.* Most approaches assume the availability of a set of input test cases, few approaches work on some models of the system under test, yet others require both test cases and models.

3. We identify the communities through the publication venues.

*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*). Property based *detection* techniques simply check for the presence of patterns of interest in the execution traces, while *prediction* techniques also select new interleavings that potentially expose the property of interest.

*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*).

*Testing Technique.* The techniques differ in terms of the type of *analysis*, which can be *dynamic* or *hybrid*, the *type of testing*, which can focus on either *functional* or *non-functional* properties, the *granularity of testing* (*unit*, *integration* or *system* testing), the *scope of testing*, and the *testing architecture* used to implement the technique, which can be either *centralized* or *distributed*.

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

### 4.1 Input

All the 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 Fig. 1.

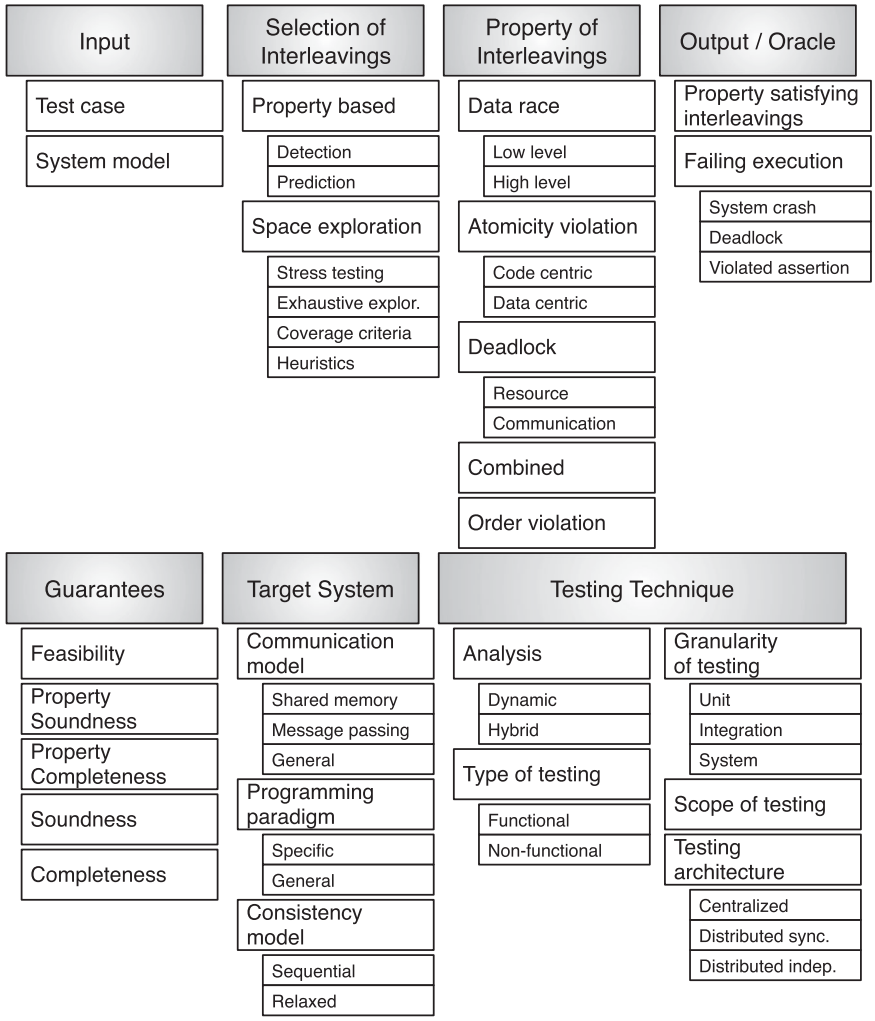


Fig. 4. A classification schema for the approaches to test concurrent systems.

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 [27]. 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.

## 4.2 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.

### 4.2.1 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 [23]. Happens-before analysis extends this approach by capturing general order constraints. Different types of happens-before analysis apply to different synchronization mechanisms [28], and present various cost-accuracy trade offs [29], [30].

Property based techniques exploit the order information identified with the analysis to either simply 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).

### 4.2.2 Space Exploration Techniques

Space exploration techniques explore the space of interleavings without referring to a specific property, and include:

*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 [31]. They also often adopt reduction techniques such as dynamic partial order reduction [32] 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.

### 4.3 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* and *deadlocks*. Other techniques *combine* multiple classical properties. Yet other techniques focus on program-specific or domain-specific *order violations*.

#### 4.3.1 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 shows an example of data race. Both execution flows `f_1` and `f_2` write on the data item `x` of type string. Let us assume that the underlying programming language can write atomically 32 bits (4 characters). If `f_1` and `f_2` start writing on `x` concurrently, the resulting value of `x` will be any combination of `'AAAA'` and `'BBBB'`, for example `x='AAAABBBB'`, where the first 4 characters come from the write instruction in `f_1` and the remaining ones from the write instruction in `f_2`.

**Listing 2.** An example of data race

```
begin f_1              begin f_2
x='AAAAAAAA'           x='BBBBBBBB'
end f_1                end f_2
```

Data races might represent 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 [33]. 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.[4] A data race implies an uncontrolled access to a data item, which may or may not be an error. For instance, in the example of Listing 2 the value `x='AAAABBBB'` may be valid or not depending on the application logic.

Data race techniques target either *low level* or *high level* data races, and propose different kinds of analysis and algorithms. Techniques for detecting low level data races work at a fine granularity level, and typically consider accesses to individual memory locations. Techniques for detecting high level data races check misbehaviors due to different execution flows invoking 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.

#### 4.3.2 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. Many atomicity violation techniques use serializability to validate the correctness of interleavings. However, checking for the serializability of an interleaving can be very expensive, since it requires comparing the interleaving with all the possible serial histories. Thus, many techniques often check for specific patterns of interleavings that are known to be non serializable.

Listing 3 shows 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 conflicting operation `x = 0` in `f_2` can occur between the two operations of `f_1`, causing the value of `x` to become negative ($-1$).

**Listing 3.** An example of atomicity violation

```
begin f_1              begin f_2
if (x>0)               x = 0
 x = x-1               end f_2
end f_1
```

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.

We distinguish two main classes of approaches to detect atomicity violations, namely *code centric* and *data centric* approaches, based on their definition of atomic blocks.

4. In literature such property is referred to as *view serializability* opposite to *conflict serializability* [34].

*Code Centric.* Code centric techniques work at a coarse granularity level. They target code blocks that should be executed atomically according to some specification of the system, and verify if the system implementation guarantees the atomicity requirements. Atomic code blocks can be either explicitly specified [27], [35] or implicitly assumed [36], [37], based on some heuristics or some hypotheses of the programming paradigm.

*Data Centric.* 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 [38]. 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.

Some testing techniques use patterns that violate atomic-set serializability to select dangerous interleavings. Since atomic-set serializability approaches require the definition of atomic sets, these techniques either rely on some code annotations or infer atomic sets using either heuristics or assumptions about the programming paradigm in use.

#### 4.3.3 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 4 shows 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.

**Listing 4.** An example of deadlock

```
begin f_1            begin f_2
acquire(L1)          acquire(L2)
  acquire(L2)          acquire(L1)
  ...                  ...
  release(L2)          release(L1)
release(L1)          release(L2)
end f_1              end f_2
```

In the example of Listing 4, 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 may depend either on an incorrect order of accesses to shared resources (*resource* deadlocks) or on an incorrect communication protocol between execution flows (*communication* deadlocks) [39].

*Resource Deadlocks.* 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 4 is an example of resource deadlock.

*Communication Deadlocks.* 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.

#### 4.3.4 Combined

Some testing techniques address *combined* properties of interleavings, that is more than one of the properties defined above.

#### 4.3.5 Order Violation

Data races, atomicity violations and deadlocks are classic and well studied properties that in general result in specific violations of the order of interleavings. Some techniques target program- or domain-specific violations of the order of interleavings that cannot be traced back to classic interleaving properties. We refer to these techniques as *order violation* approaches. For instance, some techniques target the domain of concurrent object oriented programs, and focus on interleavings that lead to null pointer dereferencing that occur when an execution flow erroneously tries to dereference an object reference *after* it has been set to null by another execution flow [40], even if suitable protected with locks that avoid data races.

### 4.4 Output and Oracle

Testing techniques produce two types of outputs, some simply produce *property satisfying interleavings* that are 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 Fig. 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.

### 4.5 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 that 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 [28] or causally-precedes relations [29], can introduce approximations that hamper both soundness and completeness [30].

Soundness and completeness describe the accuracy of a technique in detecting concurrency faults. Property soundness and property completeness describe the accuracy of a technique in detecting interleavings that expose a given property.

### 4.6 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 testing techniques assume a *sequential consistency model*, while other techniques can be applied to *relaxed consistency models*.

### 4.7 Testing Technique

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

#### 4.7.1 Analysis

Testing techniques 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. Strictly speaking, techniques that rely on static information only are not testing techniques, and thus fall outside the scope of this survey. We discuss the most relevant static analysis techniques in the related work section (Section 7).

#### 4.7.2 Type of Testing

Testing techniques may target either *functional* or *non-functional* properties. Functional testing techniques check the correctness of the program results with respect to a given oracle, while non-functional testing techniques verify non-functional properties such as response time and scalability.

#### 4.7.3 Granularity of Testing

Different testing techniques work at various *granularity* levels that span from *unit* to *integration* and *system*. Unit testing techniques target individual units in isolation. For instance, in object oriented programs, unit testing considers classes in isolation, without taking into account their interactions with other components of the system. Integration testing techniques focus on the interactions between units, and check that the communication interface between the units under test works as expected. System testing techniques target the system as a whole, and verify that it meets its requirements. In Section 5 we indicate both the granularity level and the type of system components considered by the testing technique.

#### 4.7.4 Scope of Testing

Some testing techniques have a limited *scope* in the development process. For example, many techniques are used for system validation, while other techniques target regression testing.

#### 4.7.5 Testing Architecture

Testing techniques refer to different *testing architectures* that characterize the concrete infrastructure used to exercise the system under test. Such infrastructures are composed of one or more *test 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 test 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.

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

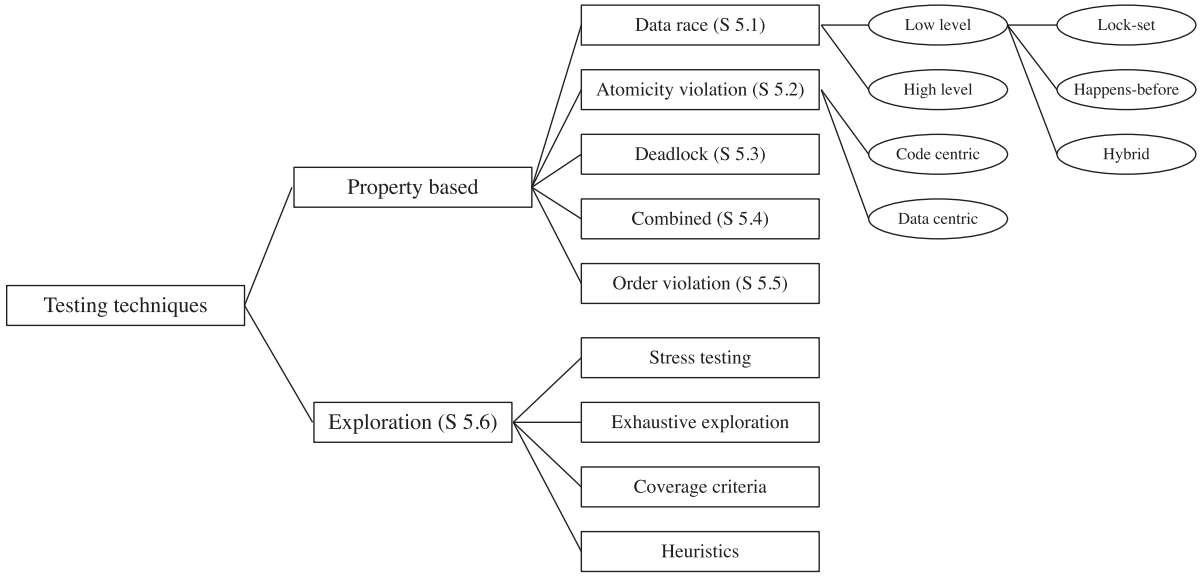


Fig. 5. Organization of the detailed survey.

We distinguish between *synchronized* and *independent* distributed testing architecture. In *synchronized* testing architectures, the drivers coordinate each other by exchanging messages, while in *independent* testing architectures the drivers execute independently and do not exchange messages to coordinate their actions. In distributed testing 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 [41].

## 5 Detailed Survey

We classify the main approaches for testing concurrent systems according to the criteria discussed in Section 4. Fig. 5 shows the organization of this section. We use *selection of interleavings* as the main classification criterion, and distinguish between *property based* and *space exploration* techniques. We classify property based approaches according to the target *property of interleaving* as *data race* (Section 5.1), *atomicity violation* (Section 5.2), *deadlock* (Section 5.3), *combined* (Section 5.4) and *order violation* (Section 5.5), and discuss space exploration techniques in Section 5.6.

This organization groups together approaches that implement related classes of methodologies and algorithms to detect faults. 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 approaches, which typically target generic types of faults.

We summarize the classification of the techniques in six tables according to the criteria. Tables 1, 2, 3, 4, and 5 overview property based techniques. Table 6 overviews space exploration techniques. The six 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 Fig. 4.

In the six tables we do not report explicitly *type of testing*, *scope of testing* and *testing architecture*, since all the techniques we analyze (i) implement functional testing, with the only exception of SpeedGun, which focuses on performance, (ii) are in the scope of validation testing, with the exception of ReConTest, SimRT and SpeedGun, which are explicitly designed for regression testing, and (iii) implement a centralized architecture. In Table 6, we also omit columns *prediction*, *property completeness* and *property soundness*, since they apply only to property based techniques.

### 5.1 Property Based: 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 interprets the two operations as

TABLE 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. | ✓ | | | ✓ | ✓ | ✓ | ✓ | SM | General | Seq | S | D | ✓ | | ✓ | ✓ | ✓ |
| Racez | ✓ | | ✓ | | | | | SM | General | Seq | S | D | | | - | - | ✓ |
| von Praun and Gross | ✓ | | ✓ | | | | | SM | OO | Seq | S | H | | | - | - | ✓ |
| ACCORD | ✓ | ✓ | ✓ | | | | ✓ | SM | Fork-join | Seq | S | D | | | - | - | |
| **Low level — Happens-before** | | | | | | | | | | | | | | | | | |
| FastTrack | ✓ | | ✓ | | | | | SM | General | Seq | S | D | | | - | - | ✓ |
| LiteRace | ✓ | | ✓ | | | | | SM | General | Seq | U | D | | | - | - | ✓ |
| Pacer | ✓ | | ✓ | | | | | SM | General | Seq | S | D | | | - | - | ✓ |
| SOS | ✓ | | ✓ | | | | | SM | General | Seq | S | D | | | - | - | ✓ |
| Carisma | ✓ | | ✓ | | | | | SM | General | Seq | S | D | | | - | - | ✓ |
| ReEnact | ✓ | ✓ | ✓ | | | | | SM | General | Seq | S | D | | | - | - | ✓ |
| Narayanasamy et al. | ✓ | | ✓ | | | | ✓ | SM | General | Seq | S | D | | ✓ | - | - | ✓ |
| Frost | ✓ | | | ✓ | ✓ | ✓ | ✓ | SM | General | Seq | S | D | | | | | ✓ |
| Portend | ✓ | | ✓ | | | | ✓ | SM | General | Seq | S | D | | ✓ | - | - | ✓ |
| Tian et al. | ✓ | | ✓ | | | | | SM | General | Seq | S | D | | | - | - | ✓ |
| RDIT | ✓ | | ✓ | | | | ✓ | SM | General | Seq | S | D | | | - | - | |
| Smaragdakis et al. | ✓ | | ✓ | | | | ✓ | SM | General | Seq | S | D | | | - | - | |
| DrFinder | ✓ | | ✓ | | | | ✓ | SM | General | Seq | S | D | | | - | - | ✓ |
| RVPredict | ✓ | | ✓ | | | | ✓ | SM | General | Seq | S | D | | | - | - | |
| WebRacer | ✓ | | ✓ | | | | | SM | Web platforms | Seq | S | D | | | - | - | ✓ |
| EventRacer | ✓ | | ✓ | | | | | SM | Event-based | Seq | S | D | | | - | - | ✓ |
| DroidRacer | ✓ | | ✓ | | | | | SM | Android apps | Seq | S | D | | | - | - | ✓ |
| Java RaceFinder | ✓ | | ✓ | | | | ✓ | SM | General | Rel | S | D | ✓ | | - | - | ✓ |
| Relaxer | ✓ | | ✓ | | | | ✓ | SM | General | Rel | S | D | | | - | - | ✓ |
| **Low level — Hybrid** | | | | | | | | | | | | | | | | | |
| Choi et al. | ✓ | | ✓ | | | | | SM | General | Seq | S | H | | | - | - | ✓ |
| Wester et al. | ✓ | | ✓ | | | | | SM | General | Seq | S | D | | | - | - | ✓ |
| RaceMob | ✓ | | ✓ | | | | | SM | General | Seq | S | H | | ✓ | - | - | ✓ |
| RaceFuzzer | ✓ | | | ✓ | | | ✓ | SM | General | Seq | S | D | | | | ✓ | ✓ |
| RaceTrack | ✓ | | ✓ | | | | | SM | General | Seq | S | D | | | - | - | ✓ |
| Goldilocks | ✓ | | ✓ | | | | | SM | General | Seq | S | H | | | - | - | ✓ |
| MultiRace | ✓ | | ✓ | | | | | SM | General | Rel | S | D | | | - | - | ✓ |
| SimRT | ✓ | | | ✓ | | | ✓ | SM | General | Seq | S | D | | | | ✓ | ✓ |
| Racageddon | | | ✓ | | | | ✓ | SM | General | Seq | S | D | | | - | - | ✓ |
| Narada | ✓ | | ✓ | | | | ✓ | SM | General | Seq | S | D | | | | ✓ | ✓ |
| **High level** | | | | | | | | | | | | | | | | | |
| Colt | ✓ | ✓ | ✓ | | | | ✓ | SM | OO | Seq | U | D | | | - | - | ✓ |
| Dimitrov et al. | ✓ | ✓ | ✓ | | | | | SM | OO | Seq | U | D | | | - | - | ✓ |

*In this and all the tables in the paper, the techniques are sorted according to their publication date within each subcategory, and are named as indicated by the proposers, or by the authors of the seminal paper when a name is not available.*
*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* |

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 [29].

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 [29], [30], 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 [42].

*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

TABLE 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 | ✓ | ✓ | ✓ | | | | | SM | General | Seq | S | D | | | - | - | ✓ |
| SVD | ✓ | | ✓ | | | | | SM | General | Seq | S | D | | | - | - | ✓ |
| AVIO | ✓ | | ✓ | | | | | SM | General | Seq | S | D | | | - | - | ✓ |
| Velodrome | ✓ | | ✓ | | | | | SM | General | Seq | S | D | | | - | - | ✓ |
| AtomFuzzer | ✓ | | | ✓ | ✓ | | ✓ | SM | General | Seq | S | D | | | | ✓ | ✓ |
| HAVE | ✓ | | ✓ | | | | ✓ | SM | General | Seq | S | H | | | - | - | |
| Penelope | ✓ | | | ✓ | | ✓ | ✓ | SM | General | Seq | U | D | | | | ✓ | ✓ |
| Wang and Stoller | ✓ | | ✓ | | | | ✓ | SM | OO | Seq | S | D | ✓ | | - | - | |
| CTrigger | ✓ | | | ✓ | ✓ | ✓ | ✓ | SM | General | Seq | S | D | | | | ✓ | ✓ |
| Falcon | ✓ | | ✓ | | | | | SM | General | Seq | U | D | | | - | - | ✓ |
| Best | ✓ | | ✓ | | | | ✓ | SM | General | Seq | S | H | | | - | - | ✓ |
| DoubleChecker | ✓ | ✓ | ✓ | | | | | SM | General | Seq | U | D | | | - | - | ✓ |
| Intruder | ✓ | | n.a. | n.a. | n.a. | n.a. | | SM | General | Seq | U | D | n.a. | n.a. | n.a. | n.a. | n.a. |
| **Data centric** | | | | | | | | | | | | | | | | | |
| Muvi | ✓ | | 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. | ✓ | | ✓ | | | | ✓ | SM | General | Seq | S | H | | | - | - | ✓ |
| AssetFuzzer | ✓ | ✓ | ✓ | | | | ✓ | SM | General | Seq | S | D | | | - | - | ✓ |
| ReConTest | ✓ | | | ✓ | ✓ | ✓ | ✓ | SM | General | Seq | S | D | ✓ | | ✓ | ✓ | ✓ |

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.

#### 5.1.1 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 [18]. Savage et al.'s Eraser approach [23] 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 both 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

TABLE 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 | ✓ | | ✓ | | | | ✓ | SM | Lock sync | Seq | S | D | | | - | - | |
| DeadlockFuzzer | ✓ | | | | ✓ | | ✓ | SM | Lock sync | Seq | S | D | | ✓ | | ✓ | ✓ |
| MagicFuzzer | ✓ | | | | ✓ | | ✓ | SM | Lock sync | Seq | S | D | | ✓ | | ✓ | ✓ |
| Wolf | ✓ | | | | ✓ | | ✓ | SM | Lock sync | Seq | S | D | | ✓ | | ✓ | ✓ |
| ConLock | ✓ | | | | ✓ | | ✓ | SM | Lock sync | Seq | S | D | | ✓ | | ✓ | ✓ |
| Sherlock | ✓ | | | | ✓ | | ✓ | SM | Lock sync | Seq | S | D | | ✓ | | ✓ | ✓ |
| OMEN | ✓ | | | | ✓ | | ✓ | SM | Lock sync | Seq | S | D | | ✓ | | ✓ | ✓ |
| Armus | ✓ | | | | ✓ | | | SM | Barrier sync | Seq | S | D | | ✓ | | ✓ | ✓ |

TABLE 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. | ✓ | | ✓ | | | | | SM | General | Seq | S | H | | | - | - | ✓ |
| Chen and MacDonald | ✓ | | | ✓ | ✓ | ✓ | ✓ | SM | General | Seq | S | H | | ✓ | | ✓ | ✓ |
| Kahlon and Wang | ✓ | | ✓ | | | | ✓ | SM | General | Seq | S | D | | | - | - | |
| PECAN | ✓ | ✓ | ✓ | | | | ✓ | SM | General | Seq | S | D | | | - | - | ✓ |

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. [43] and Racez [44] improve the precision and accuracy, respectively, while Praun and Gross [45] and ACCORD [46] extend lockset analysis to object oriented and array based concurrent programs, respectively.

*Improving Precision. Shacham et al., PPoPP 2005 [43].* Shacham et al. combine dynamic lockset analysis with model checking to detect data races in general shared memory programs. They capture locking constraints with dynamic lockset analysis, generate alternative interleavings for the executed trace with model checking, and execute the interleavings identified by the model checker to reveal faults. Lockset analysis ensures property completeness but not property soundness, while the final execution phase guarantees both soundness and completeness with respect to the oracle.

*Improving Efficiency. Racez, ICSE 2011 [44].* Racez reduces the overhead of lockset analysis with a sampling approach that captures only a subset of the synchronization operations and the memory accesses performed by the target system, thus trading accuracy for complexity. Racez inherits the idea of sampling from previous work such as LiteRace and Pacer, that we discuss in the remainder of this section. Racez further reduces the overhead by both only instrumenting synchronization operations and delegating the detection of memory operations to the hardware performance monitoring unit (PMU). The PMU stores hardware information in a buffer that is accessed asynchronously with system-level calls, and is usually adopted for performance monitoring. This combination of user-level instrumentation to monitor synchronization operations and hardware monitoring of memory accesses leads to a very low overhead—Sheng et al. report an overhead as low as 2.8 percent in some experiments performed on real systems—. The approach is neither property complete, due to the approximations introduced by the sampling, nor property sound, since it relies on the imprecise lockset analysis.

TABLE 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 | ✓ | | ✓ | | | | ✓ | SM | OO | Seq | U | H | | | - | - | |
| 2ndStrike | ✓ | ✓ | | ✓ | | | ✓ | SM | OO | Seq | U | D | | ✓ | | ✓ | ✓ |
| ConMem | ✓ | | | ✓ | | | ✓ | SM | General | Seq | S | D | | ✓ | | ✓ | ✓ |
| ConSeq | ✓ | | | ✓ | | ✓ | ✓ | SM | General | Seq | S | H | | ✓ | | ✓ | ✓ |
| ExceptioNULL | ✓ | | | ✓ | | | ✓ | SM | General | Seq | S | D | | ✓ | | ✓ | ✓ |
| Maple | ✓ | | | ✓ | ✓ | ✓ | ✓ | SM | General | Seq | S | D | | ✓ | | ✓ | ✓ |
| jPredictor | ✓ | ✓ | ✓ | | | | ✓ | SM | General | Seq | S | H | | ✓ | | ✓ | ✓ |
| Sinha and Wang | ✓ | | ✓ | | | | ✓ | SM | General | Seq | S | D | | | - | - | |
| DefUse | ✓ | | ✓ | | | | | SM | General | Seq | S | D | | | - | - | ✓ |
| GPredict | ✓ | ✓ | ✓ | | | | ✓ | SM | General | Seq | S | D | | | - | - | |
| SimRacer | ✓ | | | ✓ | | ✓ | ✓ | SM | Process level | Seq | S | D | | ✓ | | ✓ | ✓ |
| Cafa | ✓ | | ✓ | | | | ✓ | MP | Event driven | Seq | S | D | | | - | - | |
| Mutlu et al. | ✓ | | ✓ | | | | ✓ | MP | Javascript app | Seq | S | D | | | - | - | |

TABLE 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 | ✓ | ✓ | ✓ | | | | SM | General | Seq | S | D | | ✓ | ✓ |
| SpeedGun | | | | | | ✓ | SM | General | Seq | n.a. | n.a. | ✓ | | |
| **Exhaustive exploration** | | | | | | | | | | | | | | |
| Ballerina | | | ✓ | | | | SM | OO | Seq | U | D | | ✓ | ✓ |
| Sen and Agha | | | | ✓ | ✓ | | MP | General | Seq | S | D | | ✓ | ✓ |
| Basset | ✓ | | | ✓ | ✓ | ✓ | MP | Actors | Seq | S | D | ✓ | ✓ | ✓ |
| CheckMate | ✓ | | ✓ | | | | SM | General | Seq | S | D | | | |
| CPPMem | ✓ | | ✓ | | | | SM | General | Rel | S | D | | | |
| ViP | ✓ | | ✓ | | | | SM | General | Seq | S | D | ✓ | | |
| CDSChecker | ✓ | | ✓ | | | | SM | General | Rel | S | D | | | |
| **Coverage criteria** | | | | | | | | | | | | | | |
| Wang et al. | ✓ | | | ✓ | | | SM | General | Seq | U | D | | ✓ | ✓ |
| Hong et al. | ✓ | | | ✓ | | ✓ | SM | General | Seq | S | D | | ✓ | ✓ |
| Bita | ✓ | | | ✓ | | ✓ | MP | Actors | Seq | S | D | | ✓ | ✓ |
| **Heuristics** | | | | | | | | | | | | | | |
| Rapos | ✓ | | | ✓ | ✓ | ✓ | SM | General | Seq | S | D | | ✓ | ✓ |
| PCT | ✓ | | | ✓ | | ✓ | SM | General | Seq | S | D | | ✓ | ✓ |
| Gambit | ✓ | | | ✓ | ✓ | ✓ | SM | General | Seq | S | D | | ✓ | ✓ |

*Extending to New Paradigms von Praun and Gross, OOPSLA 2001 [45].* von Praun and Gross exploit the encapsulation features of object oriented programs to reduce the time and space overhead of the instrumentation to optimize lockset analysis. The approach takes advantage of the *confinement* property that characterizes objects and object fields that can be accessed only by one execution flow, and excludes them during the instrumentation. The technique pairs static analysis to detect encapsulation properties and dynamic lockset analysis to detect memory accesses. It searches for data races within the executed interleavings, without predicting faults, thus, it is not property complete.

*ACCORD, PPoPP 2011 [46].* ACCORD targets data races in array-based concurrent programs characterized by fork-join parallelism, like programs written in OpenMP, CILK and TBB. ACCORD strengthens lockset analysis by relying on an input model expressed as source code annotations that specify the intended concurrency coordination strategy. The model specifies the set of memory locations that each execution flow reads and writes, distinguishing between accesses that should only occur when a lock is acquired and accesses that might occur without mutual exclusion. ACCORD is one of the few approaches that exploit annotations from the developers to improve the accuracy of the results. ACCORD automatically verifies that the concurrency coordination strategy is data race free using a constraint solver. It also automatically generates assertions to check that the implementation conforms to the specified concurrency coordination strategy during testing. ACCORD is neither property complete, since the constraint solving might not be able to generate solutions in all cases, nor property sound, because the specification only targets lock based synchronization mechanisms.

#### 5.1.2 Low Level—Happens-Before

Happens-before analysis was introduced by Leslie Lamport in the late seventies [47] and has been adopted in several early techniques to detect data races in concurrent programs [48], [49].

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 testing 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 by maintaining additional information at runtime (FastTrack [50]), reducing the expensive monitoring activities to a subset of samples (LiteRace [51], Pacer [52], SOS [53] and Carisma [54]), exploiting the parallelism of multi-core systems (Wester [55]), and implementing happens-before analysis in hardware (ReEnact [56]).

Some approaches reduce the amount of false positives by (i) classifying data races in harmful and benign (Narayanasamy et al. [57], Frost [58] and Portend [59]), (ii) exploiting program specific synchronization mechanisms (Tian et al. [60])

or (iii) considering synchronization events generated from external libraries (RDIT [61]).

Smaragdakis et al. and DrFinder improve the completeness of happens-before analysis by taking into account alternative interleavings of locked code regions [29], [62], while RVPredict improves completeness by integrating happens-before analysis with control flow information [30].

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 [63] and EventRacer [64] adapt the analysis to Web and event-based applications, respectively; DroidRacer [65] applies the analysis to Android applications.

RaceFinder [66] and Relaxer [67] augment the analysis with model checking to capture order relations in the relaxed Java memory model.

*Improving Performance FastTrack, PLDI 2009 [50].* FastTrack introduces a runtime happens-before analysis with a constant time and space complexity, thus improving over the linear complexity of traditional vector clocks approaches. FastTrack builds upon the observation that in the context of data race detection the full generality of vector clocks is not needed to characterize a large fraction of read and write operations. Thus, it proposes a lightweight representation of the happens-before information that records only the information about the last write operation on each data item. In this way, it reduces the cost of vector clock comparison up to an order of magnitude compared to the original happens-before analysis. FastTrack is neither property complete nor property sound.

*LiteRace, PLDI 2009 [51].* LiteRace is the first technique that introduces sampling to reduce the analysis overhead. LiteRace instruments only *cold regions* that are defined as the less frequently accessed code elements, based on the assumption that frequently accessed code elements (*hot regions*) have fewer probabilities to be involved in data races. LiteRace trades property completeness (number of detected data races) for efficiency. The technique is not property sound, since the happens-before analysis might be unaware of program-specific synchronization mechanisms.

*Pacer, PLDI 2010 [52].* Similar to LiteRace, Pacer targets the efficiency of happens-before analysis through sampling. Differently from LiteRace that relies on a heuristic approach, Pacer estimates the probability of finding data races within code regions based on the sampling rate, thus trading precision for costs. Pacer extends FastTrack by alternating a sampling period that implements the classic FastTrack algorithm, and a non-sampling period that reduces the amount of recorded information and simplifies the management of vector clocks. The non-sampling period reduces the overhead of precise happens-before analysis. Pacer is not property sound since the happens-before analysis does not capture program-specific synchronization mechanisms. It is not property complete due to the sampling approach that does not analyze the entire execution.

*SOS, OOPLA 2011 [53].* SOS improves FastTrack and Pacer by reducing the overhead of dynamic happens-before analysis, while maintaining the precision of the original approach. SOS introduces the concept of *stationary objects*, which are objects that are only read after being written during the initialization period, and optimizes happens-before analysis by excluding stationary objects from the analysis, since they cannot lead to data races. SOS reduces the average overhead of FastTrack by 45 percent, while increasing the amount of detected races, and reveals over five times more races than Pacer when considering 50 percent runtime overhead.

*Carisma, ISSTA 2012 [54].* Carisma improves both the performance and the accuracy of LiteRace and Pacer by exploiting the *similarity* between multiple accesses to the same data structures, such as arrays or lists. When processing many accesses to the same data structure, techniques such as LiteRace and Pacer waste time with redundant memory access sampling. Carisma dynamically infers the application *contexts* that specify mappings between memory locations and high level data structures, and uses the contexts to compute the distribution of memory locations across data structure to better balance the sampling budget. Being based on FastTrack, Carisma is neither property sound nor property complete.

*ReEnact, ISCA 2003 [56].* ReEnact improves the performance of happens-before analysis by proposing an original hardware implementation. It segments a program execution into epochs, saves the state of the program in cache prior to each epoch, and persistently saves the state of the epoch only at the end of its execution, if they do not suffer from data races. Otherwise ReEnact rolls back, restores the previous state, and re-executes the epoch under the same interleaving with additional instrumentation to collect useful information for the developers. ReEnact implements happens-before analysis based on vector clocks, and reduces the analysis overhead to a bare 5.8 percent of the program execution time in average. ReEnact automatically repairs data race problems that match a set of predefined bug patterns, such as a missing lock-unlock synchronization, and relies on program annotations to distinguish benign data races. Hence, it is neither property sound nor property complete since it is not predictive.

*Reducing False Positives. Narayanasamy et al., PLDI 2007 [57].* Narayanasamy et al.'s technique reduces the false positive rate by automatically classifying the detected races as either benign or harmful. For a given data race, the approach replays the execution for the different orders among the memory operations involved in the data race, and classifies the race as harmful only if the executions result in different program states. The approach can still lead to misclassifications, since it would erroneously classify as benign a data race between two interleavings that lead to indistinguishable state and results only incidentally [59]. The approach is not property complete, due to imprecisions in the dynamic analysis. It is property sound, since it checks that the two different orders between the memory operations involved in the data race are feasible.

*Frost, SOSP 2011 [58].* Frost detects non-benign data races by comparing the results and program state obtained by executing multiple replicas of the same program with different interleavings. Frost segments an execution into epochs, and runs each epoch on three replicas. It executes a replica instrumented with dynamic happens-before analysis to detect synchronization points in the program, and the other two replicas with a non-preemptive controlled scheduler on a single thread. Frost schedules these two replicas as *complementary* as possible, meaning that it schedules two

statements in reversed order in the two replicas whenever the synchronization mechanisms allow so. Frost both infers the presence of harmful data races and identifies the most likely faulty replica by comparing the output of the three replicas. When used at runtime, Frost can also recover from a faulty execution by rolling back and re-executing the faulty epoch. Frost incurs a utilization cost of 3 $\times$, since it executes three replicas of each epoch. Simultaneously executions replicas on spare cores can reduce the increased utilization to a bare 3 to 12 percent overhead. Frost is neither property sound nor property complete because of the imprecision introduced by the happens-before analysis and by the epoch splitting mechanism.

*Portend, ASPLOS 2012 [59].* Portend proposes a precise classification of data races, which includes different types of harmful data races based on the effects that they have on the system under test. It addresses the limitations of data race classification approaches that rely on the identity of both outputs and state of the pair of interleavings induced by a data race: accidental identity and differences that do not depend on data races. Portend considers data races as benign only if they produce both the same results and state under all possible test inputs, and checks this property with symbolic execution. Portend is not property complete, due to imprecisions of dynamic analysis. It is property sound, since it checks if the two different orders between the memory operations involved in the data race are both feasible.

*Tian et al., ISSTA 2008 [60].* Tian et al.'s dynamic technique proposes an automated mechanism to infer program-specific synchronization mechanisms to improve the accuracy of happens-before analysis and thus reduce the rate of false positives. Specifically, Tian et al. target synchronizations based on flags, test-and-set locks and barriers. They infer common patterns that indicate the presence of such synchronization mechanisms, and dynamically detect these patterns. The approach improves the precision of data race detection, but does not detect all possible synchronization constraints, and thus cannot guarantee either property soundness or property completeness.

*RDIT, FSE 2015 [61].* RDIT reduces the amount of false positives of happens-before analysis by analyzing *missing events*, which are events not captured by happens-before analysis because they belong to portions of the code that cannot be instrumented, such as external libraries. Missing events are responsible for many false positives. RDIT tackles this problem by extending happens-before relation to take into account also the invocations of external functions even if their implementation cannot be instrumented. RDIT assumes that two external functions that operate on some shared memory locations can possibly use such locations for synchronization purposes, and adds a happens-before relation between the invocation of the two functions, thus reducing the number of false positives that could be generated if this relation is ignored. RDIT relies on RVPredict, discussed in the remainder of this section, for the data race detection based on the computed happens-before relations. Thus, it does not ensure either property completeness or property soundness.

*Improving Completeness. Smaragdakis et al., POPL 2012 [29].* Smaragdakis et al. observe that, by analysing single execution traces, happens-before analysis may infer incorrect order relations and thus miss some data races. Indeed the order relation that the happens-before analysis infers between the release and a subsequent acquisition of the same lock may be violated in other execution traces. Smaragdakis et al. mitigate this problem, by proposing a new *causally-precedes* (CP) relation that captures the order of execution of statements. The CP relation 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. Smaragdakis et al. detect *CP-races* that occur when two conflicting memory accesses are not CP related, and demonstrate that a CP-race always corresponds to a feasible interleaving with a data race. The technique is neither property complete nor property sound, since it can miss relations due to program-specific synchronization mechanisms.

*DrFinder, FSE 2015 [62].* DrFinder targets the problem of detecting *hidden data races*, which are data races that previous the happens-before analysis–as well as the causally-precedes analysis and the precise analysis implemented in RVPredict–may miss due to the over-constraining nature of the analysis. A hidden data race consists of a pair of accesses to the same shared memory location that are in a happens-before relation only for a subset of all possible interleavings. For instance, hidden data races can occur when a happens-before relation depends on the order of acquisition of a lock, which can change from execution to execution. The key intuition of DrFinder is that many hidden races can be detected by reversing the order of execution of one or more operations in a happens-before relation. For instance, reversing the execution order of two locking operations could remove a happens-before relation and expose a hidden data race. DrFinder introduces a new *may-trigger relation* that specifies if a function may trigger a lock acquisition either directly or indirectly, through a chain of function invocations. DrFinder is a predictive technique. It computes the may-trigger relation on an execution trace, looks for alternative interleavings that might expose data races, and executes the selected interleavings to check their feasibility. DrFinder is neither property complete nor property sound.

*RVPredict, PLDI 2014 [30].* RVPredict defines an order relation to detect data races that improves the accuracy of the classic happens-before analysis and the causally-precedes analysis of Smaragdakis et al. The key insight of the proposed relation consists in taking into account control flow information. Previous dynamic data race prediction approaches permute an execution trace to identify possible feasible traces that may suffer from a data race by relying on a conservative definition of feasibility that may miss some valid interleavings. In particular, they assume *read-write* consistency, meaning that every read in a valid permutation returns the same value as in the original trace. RVPredict exploits a more precise definition of feasible permutations: it encodes the order relation as a set of constraints, and invokes a constraint solver to detect races. Thanks to the additional information, RVPredict can detect up to two order of magnitude more concurrency faults than approaches based on happens-before or causally-precedes analysis. RVPredict is not property complete, since it relies on constraint solving, and is not property sound since it can miss program-specific synchronization mechanisms like all the techniques based on classic happens-before analysis.

*Tailoring to Specific Paradigms. WebRacer, PLDI 2012 [63].* WebRacer enhances happens-before analysis by taking advantage of the semantics of Web platforms and in particular information about the specification and implementation of the different browsers, like the order constraints between loading, parsing and executing. WebRacer focuses on (i) variable races, which represent data races caused by concurrent accesses to shared memory locations, such as JavaScript variables, (ii) HTML races, which occur when accesses of DOM nodes that represent HTML elements may occur both before and after their creations, (iii) function races, which occur when function invocations may occur both before and after the parsing of the functions, (iv) event dispatch races, which occur when events may fire both before and after adding the corresponding event handlers. WebRacer is neither property sound, since happens-before analysis can miss program-specific synchronization mechanisms, nor property complete, since it does not check for alternative interleavings of the analyzed execution trace. WebRacer automatically generates sets of events to interact with Web sites, thus producing concrete test case.

*EventRacer, OOPSLA 2013 [64].* EventRacer extends WebRacer by formulating the happens-before relation for event-based programs, and improves the classification of data races to reduce false positives. EventRacer extends the analysis to ad-hoc synchronization mechanisms that developers often use to extend the few synchronization primitives available in Web platforms, thus eliminating many benign data races that data race detectors such as WebRacer report. EventRacer prunes benign races by introducing the notion of race coverage: a race $a$ covers a race $b$ if the race on $a$ is used as a synchronization element to eliminate the race on $b$. An inexpensive inspection of uncovered races can quickly identify races on synchronization variables, and eliminate the false positives covered by those races. The set of data items with uncovered races is 14 times smaller on average than the set of all data items with races. EventRacer implements an efficient vector clock algorithm to perform happens-before analysis based on chain decomposition [68].

*DroidRacer, PLDI 2014 [65].* DroidRacer dynamically exploits the concurrency semantics of the Android programming model to derive a precise happens-before relation that reduces the amount of false positives for Android applications. DroidRacer takes into account the non determinism that derives from both multi-threaded executions and asynchronous tasks, a concurrency mechanism implemented in Android to prevent complex computations from running on the user interface thread. Although DroidRacer is the only technique specifically conceived for Android applications so far, the concurrency model of Android that is based on asynchronous callbacks is similar to the model of Web platform that is targeted by WebRacer and EventRacer. Compared with EventRacer, DroidRacer does not consider ad-hoc synchronization mechanisms, and thus it is not property sound. It is not property complete either, due to the limitations in the happens-before analysis it relies on.

*Extending to Relaxed Memory Models. Java RaceFinder, ASE 2009 [66].* Differently from most data race detection techniques, Java RaceFinder (JRF) does not assume a sequentially consistent memory model and introduces a new happens-before analysis to capture ordering relations in the relaxed Java memory model. JRF relies on the Java PathFinder model checker to generate interleavings that may result in data races, and explores the interleaving space driven by patterns that increase the probability to identify a data race. JRF is property complete, since it relies on model checking and exploits a precise variant of happens-before analysis that does not produce false order constraints. It is not property sound, since it can miss program-specific synchronization mechanisms.

*Relaxer, ISSTA 2011 [67].* In line with Java RaceFinder, Relaxer defines a dynamic analysis approach to detect data races in a relaxed memory model. The analysis detects potential data races in a sequentially consistent execution trace by computing the set of potential happens-before cycles, which represent possible violations of sequential consistency, uses the detected races to predict alternative interleavings on a relaxed memory model, and exploits a biased-random scheduler to force the occurrence of such interleavings. Relaxer is not property complete, since it uses a random scheduler that could miss some interleavings that suffer from a data race, and is not property sound, since it may miss program-specific synchronization mechanisms.

### 5.1.3 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. [69], 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.

In line with the seminal work of Choi et al., Wester et al. and RaceMob improve performance by pipelining the analysis on multicore hardware [55] and crowdsourcing distribute data race detection across several executions [70], respectively. RaceFuzzer [71] reduces the amount of false positives by randomly generating executions that expose data races and observing their effects. RaceTrack [72] targets both lock-based and fork-join synchronization primitives, while Goldilocks [73] targets the synchronization mechanisms of software transactions. MultiRace [74] covers relaxed memory models. SimRT [75] complements interleavings with test case priorities, while Racageddon [76] and Narada [77] complement interleavings with test cases.

*Seminal Work. Choi et al., PLDI 2002 [69].* Choi et al.'s reduce the overhead of data-race detection by combining static and dynamic optimization techniques. They determine a set of statements that can cause data races with static lockset analysis, and verify if the canditate statements lead to a data race by tracing the runtime memory accesses of the selected statements only. They introduce a *weaker-than* dependence relation among statements that identifies statements whose instrumentation would be redundant, They cache the memory accesses to further optimize the analysis. The combination of these optimizations reduces the runtime overhead down to 13

to 42 percent, by trading performance for completeness. The weaker-than relation guarantees to report *at least one* (not *every*) memory access operations involved in a data race on a memory location. As a consequence, the technique is neither property complete nor complete. It is not sound, since it could miss ad-hoc synchronization mechanisms. O'Callahan and Choi later extend this approach by relying on dynamic lockset analysis [42].

*Improving Performance. Wester et al., ASPLOS 2013 [55].* Wester et al. propose a parallel infrastructure that exploits multiple cores to speed up lockset and happens-before analyses. The technique splits an execution trace into multiple epochs, which are executed on different cores in a pipeline. A preliminary cheap analysis identifies epochs whose initial conditions depend on the results of other epochs to determine the minimum amount of information needed to initialize all epochs. The technique takes advantage of multiple cores to execute different epochs in parallel, and guarantees that an epoch runs entirely on a single core thus significantly optimizing the analysis due to the limited synchronization on data structures that store information on individual epochs. The technique inherits the limitations of the tools it uses for data race detection, and thus it is neither property complete nor property sound.

*RaceMob, SOSP 2013 [70].* RaceMob optimizes dynamic analysis with crowdsourcing. RaceMob statically identifies candidate data races with lockset analysis, and distributes the dynamic validation of these candidate data races to many people. Each individual monitors only the memory accesses related a subset of candidate races, thus requiring to instrument only a small subset of memory accesses for each individual, with a conceivable reduction of the overhead of the single executions. RaceMob is property sound. It is not property complete, since it is not predictive.

*Reducing False Positives. RaceFuzzer, PLDI 2008 [71].* RaceFuzzer extends the hybrid lockset and happens-before analyses proposed by Choi et al. to predict data races in alternative executions [42]. RaceFuzzer dynamically computes order information using an imprecise but efficient combination of lockset and happens-before analyses to reduce the computational cost. It then computes alternative interleavings of the execution trace using a probabilistic algorithm that pauses an execution flow with random sleeps when trying to perform unprotected accesses to shared data items. RaceFuzzer executes the different interleavings determined by a data race and uses either program crashes or assertions to discriminate between faulty and benign races. RaceFuzzer is not property complete since it selects interleavings with a probabilistic mechanism, and is not property sound either since the hybrid analysis might miss some synchronization constraints. Nevertheless, the final execution phase guarantees the soundness with respect to the oracle.

*Targeting Specific Synchronization Mechanisms. RaceTrack, SOSP 2005 [72].* RaceTrack detects data races in shared memory programs that are executed on the virtual machine of the Microsoft Common Language Runtime. RaceTrack improves the accuracy of the analysis by means of a hybrid algorithm that supports both lock-based and fork-join synchronization, and that monitors memory accesses up to the granularity of individual object fields and array elements. RaceTrack reduces the runtime overhead by implementing an adaptive approach that dynamically adjusts both the granularity of the detection and the amount of history of memory accesses maintained for each unit. RaceTrack analyzes a system execution to detect possible data races, without predicting new ones, and is neither property complete nor property sound. Indeed, RaceTrack might miss some data races as a consequence of the adaptive mechanism it adopts to adjust the granularity of the analysis, and might generate false warnings, due to user defined locking mechanisms not captured by the happens-before relation.

*Goldilocks, PLDI 2007 [73].* Goldilocks is the first data race detection technique that considers software transactions in addition to other synchronization mechanisms. Similarly to Choi et al's approach approach, Goldilocks exploits a sound lockset static analysis to avoid instrumenting memory accesses that are guaranteed to be race free. Goldilocks dynamically detects data races using a happens-before analysis that takes into account the semantic of memory transactions. In this way, Goldilocks combines lockset and happens-before analyses to improve precision and efficiency. The technique does not predict alternative interleaving, thus it is not complete. It is also not sound, since it can miss ad-hoc synchronization mechanisms.

*Covering Relaxed Memory Models. MultiRace, PPoPP 2003 [74].* MultiRace detects data races both at variable and object granularity levels by considering the relaxed C++ memory model. MultiRace combines lockset and happens-before analyses and takes into account both lock-based and barrier synchronization mechanisms. MultiRace aims to detect data races in production mode and introduces analysis optimizations that reduce the number of checks to memory accesses thus reducing the overhead. MultiRace is neither property complete nor complete since it is not predictive. It is not property sound, since it could miss program-specific synchronization mechanisms.

*Completing Interleavings with Test Cases. SimRT, ICSE 2014 [75].* SimRT targets regression testing. It 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 both impacted by a program change and accessed by multiple execution flows in the modified program. It implements a classic greedy test case prioritization to early detect possible data races, thus reducing the analysis time with a saving of up to 95 percent of the overall testing time. SimRT builds on top of RaceFuzzer to detect data races, and presents the same guarantees.

*Racageddon, PPoPP 2014 [76].* Racageddon introduces *race directed scheduling*, a technique to generate a test input together with an interleaving that lead to executing a target data race. Racageddon combines the hybrid data race detection technique introduced by O'Callahan and Choi [42] to identify a set of candidate data races, with dynamic symbolic execution to find an input and an interleaving that expose the candidate data race. Racageddon alternates dynamic symbolic execution with an *improve* function that permutes some instructions in the schedule to increase the probability of observing the candidate data race. Racageddon is neither property sound nor property complete, because it inherits the limitations of the underlying analysis and exploration approaches.

*Narada, PLDI 2015 [77].* Narada automates the task of generating a test suite to detect data races. It monitors the execution of a sequential test suite with lockset analysis to identify unprotected accesses to shared data elements, and to infer the state of the system and the sequences of function invocations that can trigger a data race. It uses this information to synthesize concurrent test cases that can expose the data race. Narada relies on RaceFuzzer [71] to execute and analyze the generated test cases.

#### 5.1.4 High Level

Few approaches move from the analysis of direct shared memory accesses to the analysis of complex data structures. Both Shacham et al. with Colt, [78] and Dimitrov et al. [79] rely on system specifications to detect high level data races, that is, data races that involve complex data structures.

*Colt, OOPSLA 2011 [78].* Colt 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. It prunes the space of interleavings by exploiting both the commutativity properties of operations on data structures and a set of specified linearization points, which are program locations in which an operation is assumed to take place [80]. Colt is not property complete because it relies on a random definition of the adversary execution flow and executes only a subset of all possible interleavings. It is not property sound because the randomly generated sequences of operations may be infeasible due to semantics of the program.

*Dimitrov et al., PLDI 2014 [79].* Dimitrov et al. target commutative data races, which are pairs of operations on the same object that are not ordered according to the happens-before relation, and do not commute according to commutativity specifications given in expressive ECL logic formulas, which predicate on the arguments and return values of each method. The technique converts the commutativity specifications into intermediate executable representations that enrich the happens-before relation and reduce the amount of interleavings to consider. The approach is not property sound because happens-before analysis might miss some synchronization constraint. It is not property complete since it only checks for data races in a given trace and does not consider alternative interleavings.

### 5.2 Property Based: Atomicity Violation

We classify atomicity violation techniques based on the considered atomicity constraints as *code centric*, which target code regions, and *data centric*, which target data items. Similarly to the data race detection techniques, atomicity violation techniques 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.

#### 5.2.1 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 correspond to check 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 [27]. 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 [81] and AVIO [36] infer atomic code regions by relying on information of both the structure and the dynamic behavior of the system under test. Velodrome, AtomFuzzer, HAVE and Penelope reduce the amount of false positives by either taking into account static information (HAVE [82]) or analyzing the results of interleavings that lead to atomicity violations (Velodrome [35], AtomFuzzer [83] and Penelope [37]). The approach by Wang and Stoller [84], CTrigger [85], Falcon [86], Best [87] and DoubleChecker [88] 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 [89] completes the interleaving by generating test cases that expose atomicity violations.

*Seminal Work. Atomizer, POPL 2004 [27].* Atomizer is the first work to target atomicity violations. Atomizer exploits lockset analysis—and in particular a reduction algorithm—to identify non serializable interleavings, relying on code annotations that specify atomic code blocks. The technique is not predictive, since it only analyzes the executed interleaving.

*Inferring Atomic Regions. SVD, PLDI 2005 [81].* SVD automatically infers approximations of atomic regions by exploiting data and control dependencies. As SVD atomic region is the largest group of mutually related and control dependent statements that do not read again any variable written in the region itself. The heuristic inference of atomic regions does not rely on any synchronization construct, hence it identifies atomic regions even when locks are mistakenly omitted. In the case of atomicity violation, SVD reports a detailed log of the conflicts on data items to

simplify software debugging and repairing. SVD is neither property complete nor property sound, because of the heuristic approach used to infer atomic regions.

*AVIO, ASPLOS 2006 [36].* Similar to SVD, AVIO automatically infers atomic code blocks, but referring to a relaxed definition of atomicity, to detect also atomicity violations that involve write-write conflicts, which are not considered by the SVD heuristic. AVIO models atomicity in terms of *access interleaving invariants* that it iteratively infers from system executions. AVIO starts from an initial set of candidate invariants—all pairs of accesses in the program—and repeatedly executes the system to remove unserializable interleavings that do not lead to system misbehaviors. AVIO monitors the execution of the system to detect four different access patterns that violate the access interleaving invariants and thus are not serializable. The authors present both a software and a hardware implementation of AVIO. The hardware implementation takes advantage of hardware cache coherence protocols and virtually eliminates the analysis overhead, similarly to the ReEnact data race detector.

*Reducing False Positives. Velodrome, PLDI 2008 [35].* Velodrome reduces the number of false positives with respect to previous approaches such as Atomizer, by relying on a specification of atomic code blocks to detect (and not predict) atomicity violations. Velodrome looks for cyclic patterns in the happens-before graph, which represent sufficient and necessary conditions for atomicity violations [90]. Velodrome locates the operations that most likely lead to the atomicity violations by pruning unrelated operations.

*AtomFuzzer, FSE 2008 [83].* AtomFuzzer exploits annotations that specify which code blocks are intended to be atomic, and limits the analysis to pairs of execution flows that use a single lock to ensure the atomicity of a code region. It randomly generates interleavings for a test case by exploiting happens-before analysis to capture order relations among execution flows. It executes the test case with random pauses in correspondence of accesses to critical memory regions, to maximize the probability of observing an atomicity violation. AtomFuzzer is neither property complete nor complete, since its prediction relies on a probabilistic algorithm. It is sound, since its final output consists only of executions that lead to oracle violations.

*HAVE, FASE 2009 [82].* HAVE extends the predictive techniques by taking into account the control flow: It implements a hybrid static and dynamic analysis that speculatively approximates the results that could have occurred in branches that have not been observed in the executed traces yet. HAVE statically infers a *static summary tree* for each method in the system to model the control flow, the memory accesses and the synchronization primitives of the method. HAVE combines the static summary trees with dynamic happens-before analysis to produce a model that captures both the events that happen during the execution and the events that would have happened in different unexplored branches. As in other techniques such as Velodrome, HAVE looks for non serializable cyclic patterns in the model. HAVE does not guarantee feasibility and thus is not property sound. It is not property complete either since it does not consider all the alternative interleavings of the analyzed execution trace.

*Penelope, FSE 2010 [37].s* Penelope extends the happens-before analysis to take into account alternative orders of lock acquisitions and releases that violate a set of predefined atomicity violation patterns, and re-executes the target program under the predicted atomicity violating schedules to prune false positives with test oracles. Penelope is not property complete, since it detects only a subset of all the alternative interleavings of the observed execution trace. It is not property sound since the dynamic analysis it implements can miss some ordering constraints.

*Improving Performance. Wang and Stoller, TSE 2006 [84].* Wang and Stoller propose the first technique to *predict* rather than detect atomicity violations. They propose two approaches for optimizing the detection of atomicity violation in object oriented programs. The first approach extends Atomizer based on Lipton's reduction [18], and introduces several optimizations, for instance identify read-only and thread-local variables to detect atomicity violations online. The second approach reasons on the commutativity of code blocks. It is more precise but also more expensive, and thus works offline. The proposed approaches are property complete but not property sound, since they do not consider program-specific synchronization mechanisms.

*CTrigger, ASPLOS 2009 [85].* Like Velodrome, CTrigger is a predictive technique. It targets C programs and refers to specific access patterns that involve pairs of execution flows. CTrigger executes a test case and relies on happens-before analysis to predict interleavings that expose the patterns of interest. CTrigger builds on the observation that, when multiple accesses to a shared variable in an execution flows are close to each other, the probability of an interleaved access from a different execution flow is low. Thus, CTrigger ranks the interleavings based on the distance of the accesses to the shared variables, aiming to prioritize interleavings that have a lower probability of occurrence. CTrigger is not property sound, due the limitations of happens-before analysis. It is not property complete either, since it detects all possible interleavings that expose the considered patterns, but can still miss some faulty interleavings.

*Falcon, ICSE 2010 [86].* Falcon builds on the concept of *fixed-sized sliding window* to detect suspicious patterns that lead to unserializable memory accesses. It maintains access information for each shared data item in a fixed-size window, and uses the information stored in such a window to detect suspicious memory access patterns. The sliding window keeps focus on the closely related accesses, by substituting the oldest accesses in the window with new ones. The size of the window is parametric and determines the trade off between the performance and the accuracy of the analysis. Because of this trade off, Falcon does not guarantee property soundness and completeness. Falcon ranks access patterns according to the probability of leading to concurrency faults. It executes the target system multiple times, registering the set of memory accesses occurring in failing and non-failing executions, and ranking memory access patterns according to their frequency.

*Best, ASE 2011 [87].* Best improves the accuracy and the performance of prediction techniques, like HAVE and Penelope, by defining and targeting new non-serializable patterns. Best monitors the system execution, identifies synchronization constraints, and exploits an SMT solver to

derive alternative interleavings that might lead to atomicity violations. It improves scalability by applying simplifications that aggregate interleavings into equivalence classes, and executing only one interleaving per class. A distinguishing characteristic of Best is the analysis of non-instrumented binary code: Best does not need the source code or any modified version of the target program, but only debug information, such as the mapping of instructions to source code lines. Indeed, Best infers the atomicity of code blocks by relying on the structure of the code, including the use of non-synchronization variables and the maximum distance, in terms of lines of code, between two uses of such variables. Due to imprecisions in the analysis and in the inference of atomicity assumptions, Best does not guarantee property soundness and property completeness.

*DoubleChecker, PLDI 2014 [88].* DoubleChecker reduces the cost of detecting atomicity violations by combining precise and imprecise analyses. The imprecise analysis tracks all the possible data dependencies between execution flows, and identifies a set of possible atomicity violations that contains many false positives, due to approximations in the analysis. Similar to the Velodrome and HAVE, DoubleChecker precise analysis looks for cyclic patterns in the happens-before graph and tracks precisely the data dependencies by considering the execution flows involved in the atomicity violations identified with the imprecise analysis. In this way DoubleChecker prunes many of the false positives generated in the first step, with a small overhead due to the limited scope of the analysis. DoubleChecker is not property complete, since it detects but does predict atomicity violations, and is not property sound, since happens-before analysis might miss some synchronization constraints.

*Completing Interleavings with Test Cases. Intruder, FSE 2015 [89].* Intruder automatically generates test cases to detect atomicity violations in multi-threaded Java libraries. Similarly to Narada, Intruder executes a sequential test suite and instruments the target system to profile the lock acquisitions and releases, and the field accesses. It then performs a lock-based analysis to infer possible atomicity violations based on four memory access patterns that are known to be non-serializable. Based on the results of this analysis, Intruder combines sequential test cases available in the test suite to generate concurrent test cases that have a high probability to expose atomicity violations. Intruder focuses on test case generation, and relies on other techniques to detect atomicity violations. For this reason, in Table 2 we do not specify any aspect that depends on the adopted detection technique (*n.a.* in the table).

### 5.2.2 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 [38]. The recent data centric approaches focus on (i) analyzing the correlation between data items (Muvi [91]), (ii) revealing data access patterns that characterize some form of atomicity violation (Hammer et al. and AssetFuzzer [34], [92]) or (iii) producing regression tests that take advantage of the differences among system versions (ReConTest [93]).

*Analyzing Data Correlations Muvi, SOSP 2007 [91].* Muvi considers atomicity violations that involve multiple semantically correlated variables. Muvi builds on the observation that shared variables that are semantically correlated should be accessed and modified in a consistent way. Muvi checks that correlated variables are accessed within the same atomic region, and signals an atomicity violation in the presence of accesses not consistently protected within atomic regions. Muvi identifies variable correlations by means of a static analysis approach that relies on the observation that (i) correlated variables are usually accessed together, and thus their accesses are commonly mutually close in the source code, and (ii) correlated variables are accessed within the same function. Muvi considers two variables $x$ and $y$ to be correlated if they are accessed together in at least a minimum number of functions within a maximum distance of lines of code. Muvi relies on existing approaches to detect and predict atomicity violations that involve correlated variables. The features that depend on the specific approach adopted are left unspecified in Table 2.

*Revealing Access Patterns. Hammer et al., ICSE 2008 [34].* Hammer et al. propose an automata-based algorithm to detect violations of atomic-set serializability constraints. They identify eleven problematic data access patterns that characterize non atomic-set serializable executions and that apply to general shared memory programs. The technique is predictive. It statically identifies the object fields that may be accessed by multiple threads, instruments the identified objects to dynamically generate an automaton for each field, and relies on the automaton to detect access patterns that violate atomic-set serializability. The technique generates new interleavings relying on pseudo-random noise injection. All the generated interleavings are feasible, since they are observed from concrete executions. The technique is not property complete, since it can miss some interleavings that expose the patterns of interest due to the probabilistic nature of the interleavings selection process. It is not property sound either, because of possible imprecisions in the analysis to detect order relations between instructions in different execution flows.

*AssetFuzzer, ICSE 2010 [92].* AssetFuzzer uses a relaxed happens-before analysis to detect Hammer et al.'s patterns, and a probabilistic process to steer the execution towards the relevant interleavings identified with happens-before analysis. AssetFuzzer extends RaceFuzzer and AtomFuzzer approaches that target data races and code centric atomicity violations, respectively, to data centric violations. AssetFuzzer is not property complete, since it relies on a probabilistic algorithm to select interleavings, and not property sound either since a relaxed happens-before analysis might miss some synchronization constraints.

*Regression Testing. ReConTest, ICSE 2015 [93].* ReConTest selects a subset of interleavings from a test suite for optimizing regression test suites of concurrent programs. ReConTest reduces the large amount of interleavings to be re-executed in the context of a regression testing of concurrent systems by identifying the interleavings that might be affected by the program changes, and by selecting the interleavings that may both occur only in the new version and lead to atomic-set serializability violations. ReConTest executes both the old and the new versions of the program to collect information about memory accesses, and uses this

information to detect memory accesses potentially affected by the change with change impact analysis. It executes the identified interleavings focusing both on memory accesses that can occur in the new but not in the old version of the system and memory accesses that can be executed with different concurrency contexts (sets of retained locks) in the old and new versions. The identified interleavings are feasible and violate atomic-set serializability, thus the technique is sound.

### 5.3 Property Based: Deadlock

Most testing approaches for detecting deadlocks build on the seminal work of Goodlock [94] and DeadlockFuzzer [95]. 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 case generation, (iii) consider synchronization mechanisms other than locks.

MagicFuzzer [96] and Wolf [97] improve performance by pruning the lock trees, while ConLock [98] improves the randomized scheduler of DeadlockFuzzer to avoid artificially-generated deadlocks. Sherlock [99] symbolically executes the program to identify relevant test inputs. OMEN [100] extends DeadlockFuzzer to generate concurrent test cases starting from sequential ones. Armus [101] does not consider lock synchronization but targets deadlocks caused by barrier synchronization primitives.

*Seminal Work. Goodlock, SPIN 2000 [94].* Goodlock targets lock based synchronization between pairs of execution flows. It dynamically instruments the program under test to build a lock tree, which captures the locking pattern of an execution flow, and compares pairs of lock trees to detect circular dependencies that can lead to possible *resource* deadlocks. Goodlock does not execute the identified potential deadlocks. Thus, it does not guarantee the feasibility of the identified deadlocks and is neither property sound nor sound.

*DeadlockFuzzer, PLDI 2009 [95].* DeadlockFuzzer extends Goodlock to execute the predicted deadlocks, thus confirming their actual fasibility. DeadlockFuzzer relies on the Goodlock analysis and thus considers only lock-based synchronization faults. DeadlockFuzzer uses the collected information to guide a randomized scheduler, thus improving the probability of selecting interleavings that expose deadlocks.

*Improving Performance. MagicFuzzer, ICSE 2012 [96].* MagicFuzzer improves the scalability of the Goodlock analysis by introducing a technique to prune the lock graph. MagicFuzzer observes that a deadlock that corresponds to a cycle in the lock graph contains only nodes that have both incoming and outgoing edges. Thus, it iteratively removes all the nodes that do not satisfy this property, as well as their connected edges. MagicFuzzer introduces a novel algorithm to analyze the pruned graph, relying on the observation that an execution flow can participate only in a single node of a deadlock cycle. MagicFuzzer partitions the nodes based on the execution flow they belong to, and implements a search strategy that does not explore redundant paths.

*Wolf, PPoPP 2014 [97].* Wolf improves the accuracy of the monitoring and analysis features of Goodlock by adopting happens-before analysis to prune candidate deadlocks that are infeasible due to order relations among events. In this way, Wolf reduces the number of candidate deadlocks to be confirmed by the randomized scheduler.

*ConLock, ICSE 2014 [98].* ConLock addresses the *thrashing* problem of randomized scheduling algorithms that occurs when the randomized scheduler generates an artificial deadlock. In this case the execution flows are suspended by the scheduler and cannot progress, but a deadlock cannot be confirmed. This problem reduces the confirmation capabilities of the randomized scheduler and thus impacts on the effectiveness of the approach. ConLock addresses the thrashing problem by introducing a should-happen-before order relation that is captured during the dynamic analysis. The relation is used by the randomized scheduler to increase the probability to reach and thus confirm a deadlock.

*Complementing Interleavings with Test Cases. Sherlock, FSE 2014 [99].* Sherlock extends Goodlock with dynamic symbolic execution to select new inputs and interleavings. The goal of this improved analysis is to detect deadlocks that occur after long computations, even millions of lines of code. Sherlock iteratively invokes two *execute* and *permute* functions. The former computes the path condition of the current execution by means of dynamic symbolic execution. The latter uses the path condition to generate a new set of program inputs that execute a new schedule. The two functions are executed iteratively until reaching either a schedule that leads to the deadlock candidate or an infeasible schedule.

*OMEN, OOPSLA 2014 [100].* OMEN automatically generates concurrent test cases to reveal deadlocks by exploiting properties of sequential executions. OMEN adopts an approach similar to Intruder to generate concurrent test cases: it executes a sequential test suite, builds a lock dependency relation that captures the lock acquisitions of the executed methods, and generates concurrent test cases that include the methods and parameters of the sequential test cases, but are invoked in different execution flows. For instance, a sequential execution can identify a set of method calls that acquire nested locks, and can results in a deadlock if executed from multiple execution flows.

*Extending to Other Synchronization Mechanisms. Armus, PPoPP 2015 [101].* Armus targets deadlocks caused by barrier synchronization primitives. It introduces a graph analysis technique to detect deadlocks that is both property sound and property complete with respect to programs that only adopt barrier synchronization primitives, but not property complete in general, since deadlocks can derive from other synchronization mechanisms. The implementation of Armus for X10 and Java is characterized by a low overhead, which improves the scalability to industrial size programs.

### 5.4 Property Based: Combined

Few techniques target both atomicity violations and data races, and rely on happens-before analysis. Agarwal et al. [102] focus on performance by complementing happens-before analysis with static type checking. Chen and MacDonald [103] focus on accuracy, by considering control flow information. Kahlon and Wang [104] and PECAN [105]

focus on generality, by providing developers with languages to express undesired patterns of memory accesses.

*Improving Performance. Agarwal et al., ASE 2005 [102].* Agarwal et al. propose a low overhead testing technique for detecting both data races and atomicity violations. They reduce the testing overhead by combining static type checking with a modified lockset analysis. Agarwal et al.'s static type checking is a linear time, albeit incomplete, hybrid *type discovery* mechanism that identifies types for a large portion of the system under test, in line with hybrid data race detection techniques that use lockset analysis to isolate unprotected memory accesses that they monitor with a precise happens-before analysis. Agarwal et al.'s modified lockset analysis exploits the type information inferred though type discovery to efficiently detect both data races and atomicity violations in the target system. The technique is neither property sound, nor property complete, because of the approximations in the type discovery and dynamic analysis phases.

*Improving Accuracy. Chen and MacDonald, ASE 2007 [103]* Chen and MacDonald's hybrid analysis identifies statements that may happen in parallel, and generates interleavings that exercise a selected set of concurrent access pairs. Chen and MacDonald's approach selects the relevant interleavings to be exercised according to three types of concurrent access pairs, read/write shared data items, concurrent acquisition of the same lock, concurrent wait/notify on the same monitor. Differently from Agrawal et al., and similar to HAVE, Chen and MacDonald's static approach takes into account both control flow and order information, and analyzes all the portions of the code that are reachable through a set of test cases. The technique is property sound but not property complete, since the JPF model checker does not guarantee to explore all possible systems states.

*Providing Languages for Pattern Specification. Kahlon and Wang, CAV 2010 [104].* Kahlon and Wang's approach traces the causality relation in concurrent programs with *Universal Causality Graphs* (UCGs) that extend the notion of happens-before by taking into account the constraints introduced by a property to be verified. For instance, in the case of a data race, the property to be verified is the presence of multiple concurrent accesses to a shared data item without synchronization. The approach prunes a large number of interleavings that are not relevant for the target property by combining the constraints that derive from the program synchronization structure with the constraints that derive from the target property. Kahlon and Wang discuss the approach referring to data races and atomicity violations, but UCGs can be extended to capture other custom properties. The approach is not property sound, since happens-before analysis might miss some synchronization constraints, and is not property complete either, since the prediction phase relies on an SMT solver that can be unable to solve some constraints.

*PECAN, ISSTA 2011 [105].* The key novelty introduced by PECAN is a simple language to define memory access anomalies as patterns of interaction of different execution flows with shared data items. The patterns take into account both the type of the executed memory operation (read or write) and the atomic region the memory operation belongs to, if any. The pattern language captures well known memory access anomalies, such as data races and atomicity violations. PECAN implements happens-before analysis to infer the order relations between instructions, creates a schedule for each of the identified interleaving and checks its feasibility. Pecan does not guarantee property soundness since happens-before analysis can miss some synchronization constraints, and the prediction algorithm guarantees property completeness only under specific synchronization patterns (nested locks).

### 5.5 Property Based: 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 [106] and 2ndStrike [107] address typestate faults, that is, faults that involve violations of the high level semantics of the data structures. ConMem [108], ConSeq [109] and ExceptionNULL [40] target violations of faults that impact on the program behavior, for instance interleaving that lead to null pointer dereferences, dangling pointers and infinite cycles. Maple [110] 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 [111], Sinha and Wang [112] and GPredict [113] look for concurrent behaviors that violate program specifications, while DefUse [114] targets both sequential and concurrent faults that violate automatically generated data-flow invariants.

SimRacer [115] targets process level violations, Cafa targets Android programs [116], Mutlu et al. target JavaScript programs [117].

*Violations of Specific Types of Faults. PRETEX, ASE 2008 [106].* PRETEX predicts *concurrency typestate faults*. A typestate is the state associated with an object of a given type, and defines a set of operations that can be applied to the object in that state [118]. A typestate fault occurs when invoking an operation *op* on an object *obj* in a typestate that does not support *op*. Reading a file after the file has been closed is a simple example of a typestate fault. Differently from many commonly studied concurrency faults such as data races and atomicity violations, which are related to low level memory accesses, concurrency typestate faults are related to the high level semantics of the specific target system. PRETEX instruments the target Java programs to trace events such as method calls and thread creations. When executing a test case, PRETEX (i) computes the happens-before relation among events, (ii) determines which objects are shared, and (iii) infers typestate properties of each shared object relying on mining techniques. PRETEX generates a finite state machine model of the concurrent execution, and checks the generated model for typestate property violations. PRETEX is not property complete, since automatically inferring typestate specifications may miss some specifications, and is not property sound either, since it uses an approximation of the happens-before relation that can identify infeasible violations.

*2ndStrike, ASPLOS 2011 [107].* In line with PRETEX, 2ndStrike detects *concurrency typestate faults* that involve files, pointers and locks. The technique dynamically analyzes a test case execution to generate a set of candidate faults, expressed as pairs of operations in different threads, and that may lead to a typestate fault if executed in a swapped order. The technique relies on the happens-before relation to identify operations that cannot be reordered. Differently from PRETEX, 2ndStrike uses a deterministic scheduler to force the execution of the candidate faults computed during the analysis. 2ndStrike can be extended to work with other objects. It only requires (i) an input finite state machine that captures the typestates of the target object and the operations that can be applied in each state, and (ii) an instrumentation of the program to profile the relevant object methods. The technique is both sound and property sound, but not property complete since the analysis does not take into account the control flow of the program and can thus miss some faults.

*ConMem, ASPLOS 2010 [108].* ConMem builds on the observation that testing techniques based on traditional properties of interleavings, such as data races and atomicity violations, suffer both from false positives and from the impossibility to differentiate faults based on the severity of their effects only. ConMem overcomes these limitations focusing on the *effects* of concurrency faults rather than on the memory access patterns that may *cause* such effects. ConMem selects interleavings that can lead to null pointer dereference, read of an uninitialized data item and access to invalid memory locations, as in buffer overflows. ConMem analyzes execution traces offline to identify alternative interleavings that can potentially lead to concurrency problems of the considered categories. ConMem generates concrete executions for each of the selected interleavings to prune infeasible ones and false positives. The overall approach is neither sound nor complete due to the initial dynamic analysis.

*ConSeq, ASPLOS 2011 [109].* ConSeq extends ConMem by considering a larger set of fault types, including assertion violations, infinite loops, calls to error message procedures and invalid memory accesses, and by including indirect faults, triggered through a chain of cause-effect relationships. ConSeq improves the ConMem approach with slicing and dynamic analysis. It slices the program statically with respect to the data items involved in the fault to identify accesses to shared data items that can influence the read value through a chain of control and data dependencies. It dynamically analyzes correct program executions to infer alternative interleavings that could change the dynamic control and data dependencies and cause the reading of an incorrect value. ConSeq generates a deterministic schedule for each selected interleaving, and executes the generated schedules to prune infeasible interleavings and false positives. ConSeq is not property complete due to approximations in the static analysis.

*ExceptioNULL, FSE 2012 [40].* ExceptioNULL detects interleavings that can lead to null pointer dereferences of shared data items in Java programs. ExceptioNULL implements a hybrid lockset and happens-before approach. It analyzes execution traces to capture memory accesses, order relations between operations, lock acquisition and releases. It identifies pairs of events $e_1$ and $e_2$ of two different execution flows, such that $e_1$ writes a $null$ value to a shared data item $x$ and $e_2$ reads $x$. It uses an SMT solver to detect interleavings that lead to $null$ reads, and executes the identified interleaving in a controlled environment to check for its feasibility. ExceptioNULL is both property sound and sound, but not property complete, since the SMT solver may fail to provide a solution for some input constraints.

*Maple, OOPSLA 2012 [110].* Maple extends techniques to detect and predict data race and atomicity violation with six *interleavings idioms* that represent suspicious memory access patterns that encode general violations of order constraints. Maple analyzes pairs of execution flows, under the hypothesis that they can reveal most concurrency faults. It captures order relations with happens-before analysis, computes alternative interleavings that expose some of the identified idioms, and executes the selected interleavings using a deterministic scheduler looking for faults. Maple records the executed interleavings but not the specific values observed during the execution, and can miss some faults that depend on the values adopted in the execution. Thus Maple is not property complete.

*Violations of Program Specifications. jPredictor, ICSE 2008 [111].* jPredictor idenfies violations of program specification with a two-step analysis inspired from hybrid techniques for detecting data races and atomicity violations. It shrinks an execution trace to only events relevant for the property to be checked with static analysis, builds a causality graph involving the selected events based on the notion of sliced causality—a variant of the happens-before relation—and predicts and executes alternative interleavings that might lead to property violations. Differently from the techniques to detect data races and atomicity violations discussed in the previous sections, jPredictor detects more general violations of user-defined properties. jPredictor is both sound and property sound since it executes and checks the selected interleavings. It is not property complete due to imprecisions in the dynamic analysis.

*Sinha and Wang, FSE 2010 [112].* Sinha and Wang predict interleavings that can lead to assertion violations or data races in C programs. Similar to HAVE, Sinha and Wang take into account control flow relations and build a concurrent control flow graph (CCFG) that corresponds to the execution of a test case, captures the memory accesses and the synchronization primitives in the execution flow, and includes an error node for each violated assertion in the execution. They inspect each error node to compute a path condition that is composed of a set of constraints on the program inputs, and that indicates the order of statements that reach error nodes. They then solve the path condition with a constraint solver to identify interleavings that bring the system to the error node. The technique is not property sound, due to imprecision of the analysis, and is not property complete either since the constraint solver might fail to find some solutions that lead the system to an error state.

*DefUse, OOPSLA 2010 [114].* The key contribution of DefUse is the automated generation of test oracles from data flow invariants to detect both sequential and concurrency faults. DefUse considers (i) local/remote invariants, which specify that a read operation on a data item uses definitions either only from the local or from a remote execution flow, but not from both; local/remote invariants capture order violations as well as read-write atomicity violations in concurrent programs; (ii) follower invariants, which specify

that two consecutive read accesses to the same data item from the same execution flow use always the same definition; follower invariants capture read-read atomicity violations in concurrent programs, and (iii) definition set invariants, which specify the set of write operations that a given read access is allowed to see. DefUse trains the analysis by inferring invariants from a large set of correct runs, and checks for violations against the extracted invariants in specific interleaving. DefUse is neither property complete nor property sound, since the learning phase can both miss invariants and generate false positives.

*GPredict, ICSE 2015 [113].* Similar to jPredictor, GPredict verifies high level properties expressed as regular expressions on the order of statements. GPredict infers the order relations between events that are dynamically identified on execution traces relying solely on thread-local traces, and ignoring global synchronizations, thus reducing the overhead of the dynamic analysis. GPredict checks for the feasibility of interleavings that violate the concurrency properties by means of a constraint solver to predict possible concurrency faults. GPredict is not property sound, since the analysis can miss some order relations, and is not property complete since the constraint solver might fail to generate results for some constraints.

*Violations of Programming Model Semantics. SimRacer, ISSTA 2013 [115].* SimRacer focuses on process level order violations that involve resources shared among processes—such as files and devices—and detects concurrency faults that involve user processes, software signals, software handlers and kernel processes. The technique is predictive: it dynamically collects information on synchronization operations and accesses to shared resources that occur during a test case execution, analyzes the collected information with happens-before analysis to identify potential order violations, and executes the program in a controlled environment to force such order violations to occur. The technique relies on test oracles to distinguish between benign and faulty order violations. The technique is not property complete, since it does not consider all the potential shared resources that can be accessed by different processes. It is sound, since it executes the selected interleavings and compares them with the test oracles provided by the developers.

*Cafa, PLDI 2014 [116].* Cafa analyzes event-driven (Android) programs. In event-driven programs, each execution flow owns an event queue, and processes events pushed in such queue in a non deterministic order. In line with ExceptionNULL, Cafa detects a specific form of order violation that occurs in event driven programs and that leads to a *use-after-free* violation when a pointer is dereferenced after being released. Cafa implements happens-before analysis of execution traces to identify the order relations between events and dereferencing instructions. It heuristically prunes false positives by ignoring accesses that are protected by branching conditions that check their safety, not considering as dangerous the dereferencing operations that cannot propagate outside a program unit. Cafa selects alternative interleavings that could lead to a use-after-free violation, and reports them with an offline analysis that does not ensure the feasibility of the output interleavings and thus is not property sound. Cafa is not property complete either because of the approximations in the analysis.

*Mutlu et al., FSE 2015 [117].* Mutlu et al. detect specific order violations in client-side JavaScript programs that perform multiple asynchronous requests whose callbacks can race each other, due to possible network delays. Mutlu et al. focus on the `XmlHttpRequest`, an asynchronous mechanism used to request data from a server. When the answer of a request becomes available, JavaScript invokes a callback function. In the presence of multiple requests, the invocation of the corresponding callback functions can occur in different orders, thus potentially leading to order violations. The technique distinguishes benign from harmful order violations by focusing on dangerous races between callback functions that write on the persistent state of the system, for instance on client-local cookies. In these cases, the value of the persistent state at the end of the execution strictly depends on the executed interleaving. The technique is predictive, since it monitors an execution trace of the target system to identify interleavings that potentially lead to harmful races, according to the definition above. The technique introduces a specific happens-before analysis for JavaScript programs to detect events that cannot be reordered. The technique does not guarantee the feasibility of the selected interleavings, and is neither property sound nor property complete.

## 5.6 Space Exploration

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.

### 5.6.1 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. The few recent stress testing techniques for concurrent systems check either the linearizability of the executions [119] or the impact of changes on performance [120].

*Checking Linearizability. Pike, EuroSys 2011 [119].* Pike detects *semantic* and *latent* concurrency faults. Semantic faults violate the application semantics, for example by returning an unexpected value, and are hard to find since they do not usually lead to system failures. Latent faults corrupt the internal state of the system, and may manifest much later than when triggered. Pike 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. While comparing the output of two executions is straightforward, comparing the the internal state is complicated by the presence of nested data items. Pike compares internal states on the basis of a specification expressed as a state summary function that captures the semantics of the internal state of the system and allows for an efficient logical comparison.

*Checking the Impact of Changes on Performance. SpeedGun, ISSTA 2014 [120].* SpeedGun is the only approach in our survey that targets performance—and in particular performance regression testing—. SpeedGun targets Java programs and automatically generates a set of test cases that

consist of a sequential prefix and $n$ suffixes executed concurrently on $n$ different executions flows. SpeedGun executes the test cases several times both on the old and on the new version of the class under test, and reports relevant performance differences between the two versions. SpeedGun executes each test several times to observe various interleavings, and to alleviate the impact of external factors, such as garbage collection and just-in-time compilation. It does not control the interleavings to avoid impacts on performance. SpeedGun optimizes the size of the test cases to trade off precision and execution time.

#### 5.6.2 Exhaustive Exploration

Exhaustive exploration approaches analyze *all* possible interleavings for a given test input on either a space reduced to a manageable size with a suitable test suite [121], [122], [123] or on the complete space by exploiting either symbolic execution [124] or model checking [125], [126], [127], [128], [129]. Different approaches studied exhaustive exploration in the context of message passing MPI programs [124], actor based programs [125], the C and C++ relaxed memory model [127], [129], invariant violations [128], and deadlocks [126].

*Exploring a Reduced Space of Interleavings. Ballerina, ICSE 2012 [121].* Ballerina generates and executes test cases for concurrent object oriented programs. As most approaches, Ballerina builds on the assumption that most faults can be revealed using two concurrent execution flows [130]. The Ballerina test cases are composed of two execution flows that pair a randomly generated sequential prefix with a concurrent suffix. Ballerina exploits different strategies for executing thread interleavings, including exhaustive stateful search, preemption bounding, stateless search and parallelized test execution, and introduces a technique to cluster linearizability violations based on their characteristics to simplify the classification of faults and the detection of false positive results. Pradel et al. [122] extend Ballerina with oracles that consider non linearizable execution as faults if and only if they lead to a system crash or a deadlock. In this way, they identify only non linearizable behaviors that lead to a visible fault. Pradel and Gross [123] further extend Ballerina with a technique to predict substitutability violations in concurrent object-oriented programs. Substitutability is a desired property in a properly designed object oriented program. It ensures that an instance of a subclass can always substitute an instance of a superclass without causing behavioral differences for the clients of the superclass. Pradel and Gross' approach executes automatically generated test cases on both a superclass and a corresponding subclass, and use the behavior of the superclass as an oracle for the subclass. Pradel and Gross' oracles are behavioral differences, exceptions and crashes. Ballerina identifies non linearizable behaviors and guarantees property soundness by executing each selected interleaving, but not completeness due to the probabilistic nature of the generation phase and the limitation to two execution flows.

*Exploring the Space of Interleavings with Symbolic Execution. Sen and Agha, FASE 2006 [124].* Sen and Agha's approach automatically tests distributed systems that rely on the message passing paradigm, and in particular MPI programs. The technique generates a set of test cases that execute all the reachable statements of the program, and exercises all the feasible interleavings of the generated test cases. The technique dynamically executes an initial test case to generate the path conditions for all the executed branching points. It explores different interleavings of the execution that are identified with a happens-before analysis that captures only feasible interleavings. It generates new test cases that reach not-yet-executed statements by solving constraints that violate the path condition of a branch. The technique uses system crashes and deadlocks as oracles, and returns test cases and interleavings that expose such problems.

*Exploring the Space of Interleavings with Model Checking. Basset, ASE 2009 [125].* Basset generates efficient test cases for actor based programs by extending the Java PathFinder model checker [131] to support message scheduling. Basset prunes the exploration state space by exploiting dynamic partial order reduction and happens-before analysis. Dynamic partial order reduction prevents the execution of equivalent interleavings by stopping the exploration of states that are equivalent to already explored ones, while happens-before analysis avoids producing infeasible interleavings. Basset also introduces an optimization that substitutes external libraries with stubs to speed up the space exploration, relying on the assumption that the exchange of messages between actors is only marginally affected by the behavior of external libraries.

*CheckMate, FSE 2010 [126].* CheckMate detects deadlocks, without relying on any specific synchronization paradigm, and thus differs from classic deadlock detection techniques that target deadlocks deriving from lock-based synchronization. CheckMate monitors an execution of the target system and builds a trace program, which is a simplified model of the execution that includes relevant synchronization aspects. Then, it generates all possible interleavings from the executed program trace. Similar to Basset, CheckMate relies on model checking to identity the interleavings that can lead to deadlocks. CheckMate limits the analysis cost by building a simplified model, which eliminates computations that are not relevant for detecting deadlocks. The approximations introduced in the model make the approach unsound and incomplete.

*CPPMem, POPL 2011 [127].* CPPMem exhaustively explores all possible interleavings of a test case to identify data races and other patterns of interleavings that might lead to faults in the presence of a relaxed memory model. CPPMem exploits a mathematically rigorous semantics for the C++0x concurrency model, which takes into account both operations that guarantee sequential consistency and low level instructions that accept relaxed consistency guarantees to enable for a higher level of code optimization and system performance. The technique is designed to analyze small code fragments that might exhibit non sequential behavior. CPPMem precisely defines order relations but ignores the contributions of control and data flow constraints, and thus can identify infeasible interleavings. CPMem does not guarantee either soundness or completeness. An alternative implementation of CPPMem that relies on the Nipkin model checker provides better performance while retaining the same target and guarantees [132].

*ViP, FSE 2012 [128].* ViP explores the space of interleavings of C and C++ programs with the VeriSoft model checker [25] to identify violations of user-defined properties

expressed in past-time Linear Temporal Logic (pLTL). ViP does not guarantee the feasibility of the selected interleavings, and thus is not sound.

*CDSChecker, OOPSLA 2013 [129].* CDSChecker targets the relaxed memory model implemented in C++ low level instructions, and extends CPPMem largely improving its performance. CDSChecker encodes the memory coherence guarantees offered by the C++ semantics as a set of constraints, and combines these constraints with happens-before constraints defined for sequentially consistent executions to prune redundant and infeasible executions. Similar to CPPMem, CDSChecker precisely defines order relations but ignores contributions of control and data flow constraints, thus it can identify infeasible interleavings and does not guarantee either soundness or completeness.

#### 5.6.3 Coverage Criteria

Coverage criteria identify subsets of the program space to be explored by exploiting data-flow relations in shared memory locations (Wang et al. [133]), the order of synchronized code blocks (Hong et al. [134]) or the message ordering in actor programs (Bita [135]).

*Wang et al., ICSE 2011 [133].* Wang et al. introduce the concept of *HaPSet* that characterizes the set of relevant interleavings of a test case. They define the *HaPSet* of an operation $op$ as the set of operations $op'$ that can access the same memory location as $op$ in at least one interleaving without any intermediate operation between $op'$ and $op$. The technique iteratively computes the HaPSets of a set of test cases, and explores different interleavings that exercise not-yet-executed HaPSets. At each iteration, the technique augments the initial set of HaPSets with all those HaPSets that have been explored at runtime and did not produce any system crash, until no more HaPSets can be generated. HaPSets capture well common concurrency faults such as data races and atomicity violations, thus by covering the HaPSet, the technique can find many concurrency faults. Since many interleavings share the same HaPSets, focusing on them largely reduces the amount of interleaving to be exercised.

*Hong et al., ISSTA 2012 [134].* Hong et al.'s coverage criterion focuses on synchronized blocks, and requires to execute all synchronization pairs in different orders. They extend previous work on synchronization pairs coverage [136] by also considering code blocks that belong to the same thread. The approach dynamically builds a *thread model* that captures memory accesses and synchronization operations among threads, It iteratively identifies the not-yet-covered synchronization pairs by analyzing the model, and executes not-yet-executed synchronization pairs in a controlled environment. The approach ensures both feasibility and soundness, but not completeness.

*Bita, ASE 2013 [135].* Bita targets actor based programs and introduces three coverage criteria over pairs of events received by an actor: the *'pair-of-consecutive-receives*, *pair-of-receives* and *pair-of-behavior-change-and-receive* criteria, which requie covering different sequences of receive events. Bita incrementally builds a test suite by selecting interleavings that improve the coverage according to a selected criterion, and executes them in a controlled environment, while checking for system crashes, thus guaranteeing feasibility and soundness, but not completeness.

#### 5.6.4 Heuristics

Heuristic approaches bound the space of interleavings to be explored, and prioritize their execution: Rapos [137] exploits partial-order reduction to identify equivalent interleavings, PCT [138] bounds the number of scheduling constraints, Gambit [139] prioritizes interleavings by their diversity with respect to the executed ones.

*Rapos, ASE 2007 [137].* Rapos reduces the exploration space by exploiting partial-order reduction, an optimization technique that model checkers exploit to identify equivalent states. Rapos targets Java programs and is implemented on top of CalFuzzer.

*PCT, ASPLOS 2010 [138].* Probabilistic Concurrency Testing (PCT) detects general concurrency faults in C and C++ programs with disciplined random testing. PCT guarantees to find faults of depth $d$ with a probability of at least $1/nk^{d-1}$ in programs that spawn at most $n$ threads and execute at most $k$ instructions, where the depth $d$ is defined as the minimum number of scheduling constraints that need to be enforced to reveal the fault.

*Gambit, PPoPP 2010 [139].* Gambit dynamically records the differences between the executed interleavings in a compressed representation of the interleaving space, and uses the model to prioritize the interleaving exploration according to two priority functions: a randomized search with progress guarantees, and a function that favors interleavings that expose new happens-before relations. Gambit controls the execution using the CHESS model checker, and uses assertion violations and deadlocks as oracles. The execution of the selected interleavings guarantees both feasibility and soundness.

## 6 Discussion

The detailed analysis of the state of the research presented in Section 5 highlights some relevant trends, which we summarize in this section referring to the classification criteria introduced in Section 4 and illustrated in Fig. 4.

### 6.1 Input

More than 95 percent of the analyzed approaches focus on the problem of selecting execution interleavings for a given test suite. The problem of generating test cases for concurrent software systems, although important, has not been widely explored yet. The few techniques that target test case generation either (i) build concurrent test cases from a set of sequential test cases that identify the operations to be executed, thus reducing the problem of generating concurrent test cases to the problem of distributing operations over multiple execution flows (*parallelization* approaches), or (ii) generate test cases randomly, and rely on the assumption that most concurrency faults can be triggered with a small number of execution flows and operations, thus usually generating small test cases, and simplifying the problem of selecting interleavings (*random* approaches). In addition, the interaction between generating test cases and selecting interleavings has been only marginally investigated so far.

About 10 percent of the analyzed approaches rely on system models to obtain semantic information about the behavior of the system. The models encode desired properties of

the system and are used to guide the techniques towards interleavings that violate such properties.

### 6.2 Selection of Interleavings

More than 80 percent of the analyzed approaches are property based, and about 60 percent of them are predictive, that is, they analyze an execution trace to identify alternative interleavings that can expose violations of the property of interest. Predictive techniques have become increasingly popular over time, starting from the seminal paper on Predictive Trace Analysis by Sen et al. [140]. While some techniques to detect data races and atomicity violations are not predictive, almost all the techniques that target deadlocks or order violations are predictive.

There exist several mature monitoring approaches that detect and avoid deadlocks at runtime,[5] thus testing techniques are mainly used to predict deadlocks in alternative interleavings before their occurrence in the field, as shown in Table 3. As indicated in Table 5, order violation techniques mainly target expected program invariants, and predict alternative interleavings that violate such invariants. As detailed in Table 6, exploration techniques are losing popularity in the research context, and are dominated by exhaustive exploration techniques, sometimes paired with automated generation of test cases to limit the number of execution flows and the amount of concurrent operations to be explored for the sake of feasibility. Stress testing techniques have attracted little attention, due to their inability to effectively explore the space of interleavings. Finally, only a few approaches based on heuristic or coverage criteria have been proposed so far.

### 6.3 Property of Interleavings

Most property based techniques deal with data races and atomicity violations, with emphasis on low level data races (Tables 1, 2, and 4). Patterns that involve high level data structures and programming abstractions are gaining increasing popularity. This is the case of atomic set serializability, introduced in the seminal paper by Vaziri et al. [38] and addressed in Colt of Shacham et al. and by Dimitrov et al. (rows *high level* in Table 1).

Deadlock detection has received less attention in testing, due to the presence of effective and efficient techniques to avoid, detect and recover deadlocks at runtime.

As shown in Table 5, testing techniques to detect order violations are becoming increasingly popular, likely because they target properties that better encode the semantics of the program. Furthermore, recent studies showed that order violations represent about one third of all concurrency faults [130]. Techniques in this area target either general properties such as null pointer dereferences or domain-specific properties.

### 6.4 Output and Oracle

More than 65 percent of the analyzed approaches that select interleavings do not consider the problems of executing the identified sequences and of comparing the results with oracles. Limiting the effort to the identification of suspicious interleavings without executing them reduces the costs, but results in many false positives.

As indicated in Table 3, most approaches to detect deadlocks execute the selected interleavings to prove that the predicted deadlocks are indeed feasible. Similarly, many recent techniques to detect order violations exploit explicit or implicit oracles for discriminating failing executions from false positives. In general, we observe an increasing interest in reducing the false positive rate that represents a main limitation in many approaches.

### 6.5 Guarantees

More than 80 percent of the analyzed approaches guarantee the feasibility of the identified interleavings. All property based detection approaches are feasible since they focus on a single execution trace. Property based prediction techniques guarantee feasibility by executing the selected interleavings; this is the case of almost all deadlock detection techniques, as indicated in Table 3. As summarized in Table 6, the vast majority of exploration approaches guarantee feasibility by executing the interleavings observed during exploration.

Property soundness and property completeness only apply to property based techniques. Almost no technique is property complete, due to one or more of the following considerations: (i) some techniques only perform detection (and not prediction), and thus can miss relevant properties in alternative interleavings, (ii) some techniques rely on an over constraining analysis, such as some forms of happens-before analysis, (iii) some techniques rely on bounded model checking or constraint solving to generate alternative interleavings, which can fail to produce some valid interleavings, (iv) some techniques exploit probabilistic approaches to generate or execute alternative interleavings, (v) some techniques are limited in scope, for example some techniques focus on data races that involve at most two execution flows.

Techniques that do not execute the selected interleavings are not property sound. Indeed, the selected interleavings might be infeasible due to the limited precision of the adopted dynamic analysis, which may miss some order constraints defined through program-specific synchronization mechanisms.

Soundness and completeness apply to the techniques that compare the results of executing test cases with oracles. Since most techniques are not property complete, they are also not complete. Almost all the techniques that compare the results of executing the test cases with oracles are sound. Only the techniques that implement oracles as model checking over an abstract system representation might not guarantee soundness, due to approximations in the model.

In general, we observe an increasing interest in improving the fault prediction rate and reducing the false positive rate pursued by (i) improving the precision of dynamic analysis techniques, and (ii) targeting specific patterns of faults or programming paradigms.

### 6.6 Target System

Almost all state-of-the-art techniques for testing concurrent systems address shared memory systems. Only few

5. These monitoring approaches are outside the boundaries of our survey, which focuses on testing techniques. We discuss them in some details in Section 7.

techniques (we discuss five of them in this survey) deal with message passing systems, which are a main focus of the research on runtime monitoring and model based verification. Monitoring techniques observe the behavior of the target system at runtime to either analyze the root causes of observed faults or predict future faults to prevent their occurrence. Model based techniques verify the correctness of message passing systems on abstract models. We discuss both classes of approaches in Section 7.

Most state-of-the-art techniques target general concurrent systems. Few exceptions focus on specific programming paradigms: (i) data race detection techniques that rely on lockset analysis target systems that adopt lock-based synchronization mechanisms (rows *Low level-Lockset* in Table 1); (ii) most deadlock detection techniques target deadlocks that derive from lock-based synchronization faults (Table 3); (iii) techniques for high level data races work at the level of abstract objects and in particular object-oriented systems (rows *High level* in Table 1); (iv) some techniques improve effectiveness by relying on the semantics of operations defined in specific programming paradigms. For instance, some techniques focus on event-based systems, such as Web platforms, the Android platform or Javascript, where the framework enforces some order relations among operations.

Almost all recent approaches assume a sequential consistency model; only few approaches (we discuss five of them in this survey) consider the memory models implemented in specific programming languages, such as Java and C++. Relaxing the assumption of a sequential consistency model remains an open area for future investigations.

### 6.7 Testing Technique

Most techniques for testing concurrent systems rely on some sort of dynamic analysis of the system to capture some information about order relations between operations in multiple execution flows. Most techniques adopt happens-before analysis. Only few approaches (we discuss nine of them in this survey) propose some form of hybrid static and dynamic analysis to improve the accuracy of the produced results. For instance, some techniques use static analysis to infer relevant properties of the system, such as atomic regions or correlated variables in the case of atomic set serializability.

Most techniques focus on the impact of concurrency faults on the system functionality, leaving largely open the problem of violations of non functional properties, such as temporal or security properties. Most techniques apply at the system granularity level. Only a few techniques focus on the unit granularity level, typically objects in object oriented systems. Most techniques focus on validation testing. Only few techniques (ReConTest [93], SimRT [75] and SpeedGun [55] address regression testing.

All techniques refer to a centralized testing architecture, which reflects the limited attention to the message passing paradigm, mostly used to develop distributed applications. Distributed message passing systems possibly deployed in large scale settings remain a largely open research area that involves the design of efficient and potentially distributed testing architectures.

### 6.8 Threats to Validity

The main threats to the validity of our survey derive from the strategy we chose to review the literature, as discuss in Section 3. The threats include the validity of the criteria we used to select the papers, the likelihood of missing some relevant work, and the coherence of the filters we used to bound the scope of the survey. In this section, we acknowledge the threats that may limit the validity of our conclusions, and briefly discuss the countermeasures that we adopted to mitigate the related risks.

*Selection Criteria.* We grounded our selection on a search of relevant repositories by keywords. Testing techniques might be referred to with different terms in different communities and may have been published in different venues. The choice of keywords and repositories for the initial search is crucial for the success of our survey. We complemented a search through the most popular repositories (IEEE Explore, ACM Digital Library, Springer Online Library and Elsevier Online Library) with general search through the Web, and we adopted a multivariate set of keywords that include "testing + concurrent", "testing + multithread", "testing + parallel", "testing + distributed". To mitigate the intrinsic limitations of any choice of terms and repositories, we also included in our search the publications that are cited or cite the papers that we selected. As shown in Fig. 3, our selection criteria let us identify many results presented in different research communities.

*Missed Work.* Our literature search based on repositories, keywords and mutual relations among papers may have missed some papers not cited in and not citing any of the selected papers. We mitigated this risk by analyzing both conference proceedings and journal issues that have been published in the last fifteen years.

*Boundary Filters.* Bounding the scope of a survey always carries the risk of arbitrarily excluding some relevant work. We mitigated this risk by precisely defining the boundaries of our survey. We list closely related albeit out of bound work in Section 7, where we present the main topics that are excluded from this survey, namely empirical study of concurrency faults, interleaving exploration strategies, coverage criteria for concurrent systems, mutation analysis, monitoring and debugging approaches, static analysis, model checking, and programming paradigms. We are aware that these choices may bias the defined framework and may privilege the work of some communities over others. By explicitly listing the work excluded from our survey, we fairly indicate the boundaries and the limits of our work.

The large number of publications included in our survey—over 90 publications in 15 years—and the heterogeneity of the communities and venues of these publications give us a good confidence on the validity of the analysis.

## 7 Related Work

We frame the scope of this survey to software *testing* techniques that target *concurrency faults*, and intentionally leave out other aspects of concurrent systems. In this section, we briefly overview some important areas that lie on the borders of the scope of this survey: in Section 7.1, we overview studies and techniques that support approaches for testing concurrent software systems and that are particularly

relevant in the framework of this survey; in Section 7.2, we outline the main approaches for verifying concurrent systems that are alternative to the testing techniques surveyed in this paper; in Section 7.3, we present programming paradigms to simplify the development of concurrent systems.

## 7.1 Related Studies

The software testing techniques that we survey in this paper are strongly influenced by the empirical studies of concurrency faults, take advantage of different strategies for exploring the interleaving state, and complement coverage criteria defined for concurrent software systems.

### 7.1.1 Empirical Studies of Concurrency Faults

There exist only few papers that report the results of large empirical studies of in-development and in-field concurrency faults, but they strongly influenced some of the techniques presented in this survey. The most direct liaison between empirical studies and testing techniques is the observation that a large amount of concurrency faults manifest in the presence of only two execution flows, which has strongly influenced many techniques for generating test cases for concurrent systems.

The first comprehensive and widely cited study of concurrency faults is due to Lu et al. [130], who consider 105 concurrency faults randomly selected from four open source applications. Previous studies, such as the work of Chandra and Chen [141], were performed on a small number of concurrency faults, which limits the generalizability of the results. Lu et al.'s experimental results lead to some interesting considerations: (i) about one third of non-deadlock concurrency faults are violations of order assumptions; (ii) about one third of non-deadlock concurrency faults involve multiple variables; (iii) about 92 percent of concurrency faults can be reliably triggered by enforcing certain order among at most four memory accesses; (iv) about 73 percent of non-deadlock concurrency faults are not fixed by simply adding or changing locks, and many times the first fix is not correct, indicating the difficulty of reasoning about concurrent systems; (v) concurrency faults that can cause program crash constitute approximately 50 percent of non-deadlock faults; (vi) many well known properties of interleavings, such as data races and atomicity violations, have little correlation with fault severity; (vii) about 85 percent of the crashes due to concurrency faults involve wrong memory accesses to shared memory objects; (viii) concurrency-memory faults can be further classified into null shared pointer dereference, shared buffer overflow, uninitialized read to shared data items and dangling pointers to shared memory.

Fonseca et al. [142] study concurrency faults in the context of MySQL applications and observe that 15 percent of concurrency faults do not lead to a deadlock or a system fault, but rather silently corrupt some data structure, and violate the intended semantics of the application. Lin and Dig [143] empirically study misuses of concurrent collections in Java, focusing specifically on *check-then-act* errors that occur when the code is developed under the erroneous assumption of the atomic execution of a condition evaluation and a subsequent operation that depends on the outcome of such evaluation. The authors consider various patterns of use of different types of collections, and observe that some patterns have a high probability to expose concurrency faults, indicating that they represent a good target for testing.

### 7.1.2 Benchmarks

Only few studies focus on benchmarks to evaluate the approaches for testing concurrent software systems. Lin et al. [144] review the status of benchmarks in the field, and observe that (i) researchers often follow ad-hoc processes to choose case studies for evaluation, and (ii) existing benchmarks are not explicitly designed for concurrency fault detection, are typically outdated, and consist of a selection of real-world case studies that might introduce a bias when evaluating the effectiveness of testing techniques. In their survey, Lin et al. introduce *JaConTeBe*, a publicly available benchmark with 47 faults in 8 software projects, which might constitute a reference benchmark for research work.

### 7.1.3 Interleavings Exploration Strategies

The space of interleavings grows exponentially both with the number of execution flows and with the number of operations within each execution flow. The problem of exploring the space is addressed with algorithms that either *reduce* or *bound* the space to be explored.

Approaches to *reduce* the interleaving space prune the search space by exploiting known properties of the target system. Java Path Finder [131], [145] introduces partial order reduction, which exploits the commutativity of independent operations to avoid the repeated analysis of interleavings that lead to the same results. Partial order reduction [32], [146] removes equivalent interleavings during exploration, thus limiting the memory consumption during the analysis. The level of reduction can be exponential in the size of the search space in the best case. More aggressive approaches further limit the execution of some specific interleavings: for instance, Kähkönen et al.'s approach [147] trades completeness for performance by combining dynamic symbolic execution with the unfolding of a Petri Net model of the target system.

While approaches to reduce the interleaving space aims to completely cover the search space, approaches that *bind* the interleaving space ensure the completeness of exploration within some given bounds. *Depth bounding* is commonly adopted in sequential program analysis to limit the exploration of the execution space to a bounded number of steps [148]. In the domain of concurrent programs, the CHESS model checker introduces *context bounding* that limits the maximum number of context switches to be considered during the analysis [31], [149], [150]. *Delay bounding* [151] bounds the number of times a schedule can deviate from a predefined deterministic scheduler. Thomson et al.'s empirical comparison of delay and context bounding reveals that context bounding does not introduce visible improvements over a naïve random exploration, while delay bounding is more effective in finding faults [152]. Thomson et al. observe that many faults can be found within limited bounds; however, most concurrency faults do not result in immediate system crashes, but in more subtle errors, such as data corruption.

Bindal et al. [153] propose *variable and thread* bounding to rank and explore the space of interleavings. They move

from the observation that real-world concurrency faults usually involve a low number of shared variables and execution flows and use these numbers as bounds. ReEx [154] focuses on regression testing and prioritizes interleavings that can be affected by a change in the system under test.

#### 7.1.4 Coverage Criteria for Concurrent Systems

Some space explorations techniques presented in Section 5.6 (rows *Coverage criteria* in Table 6) identify subsets of the program space to be explored referring to coverage criteria for concurrent systems, which are studied also independently from testing approaches.

Bron et al. [155] propose synchronization coverage that refers to the synchronization mechanisms. The criterion builds on a model of the available concurrency mechanisms and on knowledge of common fault patterns. For instance, the authors discuss the design of synchronization coverage for the concurrency abstractions provided by Java. Lu et al. [156] propose a set of coverage criteria, discuss their cost, which ranges from exponential to linear with respect to the number of accesses to shared data items, organize the criteria hierarchically according to the subsume relation, and discuss the types of faults addressed by each criterion. Křena et al. [157] propose a general approach to build coverage criteria starting from target error patterns, focusing on criteria that are suitable to guide search-based techniques, such as genetic algorithms. Hong et al. [158] study the impact of some coverage criteria on the effectiveness of testing, observing that the criteria are moderate to good predictors of concurrent testing effectiveness, and are generally reasonable targets for test case generation.

### 7.2 V&V of Concurrent Software Systems

The testing techniques discussed in this survey are one of the many V&V approaches for concurrent software systems that also include mutation analysis, runtime monitoring and debugging, static analysis, model checking and model conformance verification.

#### 7.2.1 Mutation Analysis for Concurrent Systems

Mutation analysis estimates the effectiveness of a test suite for a target program by executing the suite on a set of mutated programs obtained by systematically seeding program changes, and statistically analyzing the results. Mutation analysis applies a set of mutation operators to the target program to generate modified versions of the system, called *mutants*, executes a test suite on each mutant, and checks if the mutants can be *killed*, that is, distinguished from the original program. The ratio of killed mutants with respect to the overall number of (non-equivalent) mutants is the *mutation score* that estimates the quality of the test suite. The interested readers can refer to the recent excellent survey on mutation testing from Jia and Harman [159]. Mutation analysis has been extended to concurrent systems by (i) defining mutation operators for concurrent systems, and (ii) introducing techniques and tools to apply mutation analysis to concurrent systems.

Bradbury et al. [160] define a set of mutation operators for the concurrency and synchronization mechanisms introduced in Java since version 5. Bernhard and Delgado [161] propose to generate mutants for concurrent systems by applying mutation operators at the specification level. Sen and Abadir [162] provide mutation operators for concurrent real-time systems.

A relevant problem that emerges when applying mutation analysis to concurrent systems consists in comparing the results produced by executing the original program with the results produced by executing the mutants. In principle, all possible interleavings of operations should be evaluated to identify differences in the results. The seminal work of Carver [163] proposes a heuristic approach that considers only a subset of all possible interleavings. The MutMut framework [164] exploits information collected during the execution of the original code under test to speed up the exploration and execution of the same test in the mutant code. In particular, MutMut prunes interleavings that cannot be affected by the considered mutation.

Gligoric et al. [165] tackle the problem of the large number of mutants by proposing a selective mutation technique that identifies a subset of operators that have high effectiveness and provide high savings. Selective mutation generates fewer mutants that approximate the mutation score.

Aichernig et al. [161] mutate the specifications and generate test cases that check whether the system conforms to the faulty specification. Some of Aichernig et al.'s mutation operators are strictly correlated to concurrency, such as swapping the order of two events, or replacing a message with another message.

#### 7.2.2 Monitoring and Debugging Concurrent Systems

Monitoring techniques collect information on a running system to understand whether the execution violates some desired properties and to provide information for replicating a failing execution. The approaches to monitor concurrent systems, and in particular complex distributed systems, work either *online* or *offline*. Online monitoring approaches analyze the target system during the execution, while offline approaches analyze logging and checkpointing information offline.

Magpie [166] collects performance traces at runtime and uses them to build a performance model that can be analyzed offline. Magpie is based on a black-box instrumentation that requires no source code modification to the measured system. X-Trace [167] identifies offline correlations between tasks at different layers of the protocol stack. It augments tasks with metadata, and forces the propagation of such metadata across layers to provide a unified view of the behavior of a distributed application. Pip [168] proposes a language to specify desired properties for a distributed system, and checks that these properties are satisfied at runtime. Friday [169] and WiDS-checker [170] replay (failing) executions of distributed systems to support debugging. Dystalizer [171] focuses on performance monitoring and uses machine learning techniques to detect the set of events that are most likely responsible for performance issues. Singh et al. [172] monitor distributed systems online through queries to the state of the system. $D^3S$ [173] and MaceODB [174] check online global concurrency properties. CrystallBall [175], [176] goes one step further by both predicting potential failures in a deployed system and trying to prevent them, steering the execution away from states that violate safety properties.

#### 7.2.3 Static Analysis of Concurrent Systems

Static analysis approaches aim to verify some desired properties of systems by relying on statically available information, that is the code or some form of specifications, without requiring any execution of the systems. Static analysis techniques aim at a different completeness-precision tradeoff than the testing approaches surveyed in this paper. Static analysis improves completeness by addressing the whole program space, but may suffer from lack of precision due to false positive warnings, which derive from conservative approximations that can lead to infeasible interleavings.

RacerX [177], the seminal Dawson's work on static analysis for detecting both data races and deadlocks, relies on a static lockset analysis of the control flow graph of the target system, and limits the problem of false positives with a mechanism that estimates the severity of the detected faults. von Praun and Gross [178] exploit static analysis to detect atomicity violations. They build on the notion of *method consistency*, which checks that the accesses in the scope of a method are not interleaved with conflicting accesses of other methods. von Praun and Gross statically analyze the program to infer information about the locking discipline and the memory access patterns of each method, and report an atomicity violation if the method consistency is violated. Williams et al. [179] statically analyze the code to build the lock-order graph of a library and detect sequences of calls to the public methods of the library that lead to a deadlock when executed concurrently. Jade [180] statically computes the lock acquisition graph and reduces the number of false positives by checking six properties that represent necessary conditions for deadlock. Marino et al. [181] target programs that use synchronization primitives for atomic set serializability. They statically build a partial order of lock acquisitions on atomic sets and generate an alert, representing a potential deadlock, whenever a partial order cannot be found.

#### 7.2.4 Model Checking

Model checking is a method to formally verify finite-state concurrent systems [182]. Model checking was introduced in the early eighties by Emerson and Clarke [183] and Queille and Sifakis [184], and consists of exhaustively traversing an abstract model of the system and checking if some properties of interest, expressed in some logical formalism, hold.

In this survey, we review several approaches that rely on model checking for generating test cases and interleavings for concurrent systems by combining model checking with lockset analysis [29], [30], [43], happens-before analysis [66], [67] hybrid analysis [103], exhaustive exploration [125], [126], [127], [128], [129] and heuristics [137].

Model checking is also widely exploited in the context of the analysis of concurrent software systems, and shares the advantages and the limitations of other static analysis techniques. A complete survey of the many applications of model checking is out of the scope of this survey. The interested reader can refer to Clarke's book [182] and the proceedings of the many symposia on model checking.

#### 7.2.5 Model Conformance

Model conformance approaches generate test cases from some abstract models of the system under test to verify that the system conforms to the model.

In this survey we review the model conformance approaches that produce concrete test cases, and in particular techniques for atomicity violations that take in input a specification of atomic code blocks and check if the implementation ensures the atomicity of such blocks [27], [35], [36]. Here we briefly describe the model conformance approaches that are not implemented in concrete testing techniques, and thus are out of the scope of this survey. Model conformance dates back to the research on protocol conformance verification [185], and has been extended to many formalisms and notations: multi-port Finite State Machines [186], Input/Output Transition Systems (IOTS) [187], [188], Partial Order Automata [189], [190], and Petri Nets [191]. Even if Partial Order Automata and Petri Nets embed concurrency notations that simplify the specification of concurrent systems, most work in the area has focused on multi-port FSMs, in which the components of the distributed systems are modeled as ports, and an input to one port can trigger a transition that produces outputs at multiple ports.

### 7.3 Programming Paradigms for Concurrency

Several programming paradigms have been proposed to develop concurrent software systems and to express concurrency patterns. Good examples of successful programming paradigms are the primitives for data parallelism and task parallelism in OpenMP [192], Apple's Grand Central Dispatch and operation queues [193], programmatic event loops [194], Microsoft's Task Parallel Library in .NET [195], primitives for asynchronous programming such as asynch-await, futures and promises [196], and the Java concurrency utilities introduced since Java 5. Here, we briefly summarize how different programming paradigms either prevent the occurrence of some classes of concurrency faults or enable efficient static detection at compile time.

Functional programming languages such as Haskell [197] do not allow mutable state and thus guarantee race freedom by design. For this reason, most modern frameworks for big data parallel and distributed processing such as MapReduce [198], Apache Spark [199] and Apache Flink [200] provide functional interfaces. Similarly, several programming languages and libraries provide functional abstractions to manipulate collections. This is the case of the ReactiveX framework [201] and of the *stream* operators introduced in Java 8. Actor-based languages such as Erlang [13], [202] and Scala/Akka [14] implement the communication between execution flows using messages, and ensure that messages are processed atomically. This guarantees data race freedom by design, since two execution flows cannot access a shared resource concurrently.

Some type systems constrain the concurrency patterns to ensure the validity of some given properties by design: some avoid deadlocks [203], [204], others data races [205], [206], [207] yet others atomicity violations [208]. The object oriented programming framework SCOOP offers high level primitives for concurrent programming that ensure data race freedom, exclude some forms of deadlocks, and enable pre and postcondition reasoning guarantees between execution flows [209]. The TWEJava [210] model for task parallelism allows the declaration of the effects of each task on the share state. The runtime ensures that two tasks are executed concurrently only if their effects do not conflict. P# [211]

extends C# for event-based programming, and forces the declarative specification of concurrency with state machines, thus enabling sound static data race analysis.

Other programming paradigms and type systems ensure deterministic execution order. Deterministic Parallel Java (DPJ) [212] introduces a type and effect system that guarantees deterministic semantics with compile time checking in Java. Grace [213] is a runtime system for C/C++ programs that masks the concurrent execution of code designed as sequential on multiple cores. CoreDet [214] and DThreads [215] replace the pthread library for C/C++ programs, providing deterministic execution with limited overhead. Determinator [216] provides API for low overhead deterministic concurrent execution on top of a microkernel.

## 8 Conclusion

This paper provides a comprehensive survey of the recent results in testing concurrent systems. The research in this area has focused mainly on the problem of selecting interleavings to be tested and has explored two classes of approaches: property based approaches, which target patterns of interleavings that are more likely to lead to faults, and exploration approaches, which explore the space of interleavings exhaustively, based on coverage criteria or heuristically.

The current research trends are towards *predictive* property based techniques and violations of expected order invariants rather then low level memory access conflicts such as data races.

The research of the last decade has produced several efficient and effective testing techniques for concurrent systems that open promising directions for future investigations:

(i) Most testing techniques for concurrent systems target the selection of relevant interleavings, and few techniques focus on test case generation. Exploiting the synergy between these two aspects remains an open research topic.

(ii) The vast majority of testing approaches target shared memory systems. Validation and verification of distributed message passing systems has exploited mostly static analysis and model checking approaches, leaving the important area of testing message passing systems open for future research.

(iii) The last decade has seen a bloom of new programming paradigms for concurrent software systems, which enforce patterns of interactions among execution flows that prevent the occurrence of some kinds of concurrency faults such as data races and deadlocks. The new programming paradigms shift the testing problem from low level memory access conflicts to high level order violations, and open the opportunity of devising new testing approaches that exploit the semantics of modern programming paradigms.

## References


[1] T. Mattson, B. Sanders, and B. Massingill, *Patterns for Parallel Programming*. Reading, MA, USA: Addison-Wesley, 2004.

[2] J. Diaz, C. Munoz-Caro, and A. Nino, "A survey of parallel programming models and tools in the multi and many-core era," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 8, pp. 1369–1386, Aug. 2012.

[3] D. Dig, "A refactoring approach to parallelism," *IEEE Softw.*, vol. 28, no. 1, pp. 17–22, Jan./Feb. 2011.

[4] K. Jensen and L. M. Kristensen, *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Berlin, Germany: Springer, 2009.

[5] G. R. Andrews and F. B. Schneider, "Concepts and notations for concurrent programming," *ACM Comput. Surveys*, vol. 15, no. 1, pp. 3–43, 1983.

[6] G. R. Andrews, *Concurrent Programming: Principles and Practice*. San Francisco, CA, USA: Benjamin/Cummings Publishing Company, 1991.

[7] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput.*, vol. C-28, no. 9, pp. 690–691, Sep. 1979.

[8] W. Pugh, "Fixing the Java memory model," in *Proc. ACM Conf. Java Grande*, 1999, pp. 89–98.

[9] J. Manson, W. Pugh, and S. V. Adve, "The Java memory model," in *Proc. Symp. Principles Program. Languages*, 2005, pp. 378–391.

[10] H.-J. Boehm and S. V. Adve, "Foundations of the C++ concurrency memory model," in *Proc. Conf. Program. Language Des. Implementation*, 2008, pp. 68–78.

[11] D. Hovemeyer, W. Pugh, and J. Spacco, "Atomic instructions in Java," in *Proc. Eur. Conf. Object-Oriented Program.*, 2002, pp. 133–154.

[12] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer, "Phasers: A unified deadlock-free construct for collective and point-to-point synchronization," in *Proc. Annu. Int. Conf. Supercomputing*, 2008, pp. 277–288.

[13] J. Armstrong, *Programming Erlang*. Raleigh, NC, USA: Pragmatic Bookshelf, 2013.

[14] D. Wyatt, *Akka Concurrency*. Walnut Creek, CA, USA: Artima Incorporation, 2013.

[15] C. A. Petri, "Communication with automata," Tech. Rep. RADC-TR-65-377, Vol. I, Suppl. I of the PhD Thesis, Technische Universität Darmstadt, 1966.

[16] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[17] R. Milner, *A Calculus of Communicating Systems*. Berlin, Germany: Springer, 1982.

[18] R. J. Lipton, "Reduction: A method of proving properties of parallel programs," *Commun. ACM*, vol. 18, no. 12, pp. 717–721, 1975.

[19] S. Morasca and M. Pezzè, "Using high-level petri nets for testing concurrent and real-time systems," *Real-Time Syst.: Theory Appl.*, vol. 132, pp. 119–131, 1990.

[20] R. N. Taylor, D. L. Levine, and C. D. Kelly, "Structural testing of concurrent programs," *IEEE Trans. Softw. Eng.*, vol. 18, no. 3, pp. 206–215, Mar. 1992.

[21] R. H. Carver and K.-C. Tai, "Use of sequencing constraints for specification-based testing of concurrent programs," *IEEE Trans. Softw. Eng.*, vol. 24, no. 6, pp. 471–490, Jun. 1998.

[22] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Languages Syst.*, vol. 12, no. 3, pp. 463–492, 1990.

[23] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, 1997.

[24] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Trans. Program. Languages Syst.*, vol. 8, no. 2, pp. 244–263, 1986.

[25] P. Godefroid, "Model checking for programming languages using VeriSoft," in *Proc. Symp. Principles Program. Languages*, 1997, pp. 174–186.

[26] G. J. Holzmann, "The model checker SPIN," *IEEE Trans. Softw. Eng.*, vol. 23, no. 5, pp. 279–295, May 1997.

[27] C. Flanagan and S. N. Freund, "Atomizer: A dynamic atomicity checker for multithreaded programs," in *Proc. Symp. Principles Program. Languages*, 2004, pp. 256–267.

[28] D. Perkovic and P. J. Keleher, "Online data-race detection via coherency guarantees," in *Proc. Symp. Operating Syst. Des. Implementation*, 1996, pp. 47–57.

[29] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan, "Sound predictive race detection in polynomial time," in *Proc. Symp. Principles Program. Languages*, 2012, pp. 387–400.

[30] J. Huang, P. O. Meredith, and G. Rosu, "Maximal sound predictive race detection with control flow abstraction," in *Proc. Conf. Program. Language Des. Implementation*, 2014, pp. 337–348.

[31] M. Musuvathi and S. Qadeer, "Iterative context bounding for systematic testing of multithreaded programs," in *Proc. Conf. Program. Language Des. Implementation*, 2007, pp. 446–455.

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

[33] C. H. Papadimitriou, "The serializability of concurrent database updates," *J. ACM*, vol. 26, no. 4, pp. 631–653, 1979.

[34] C. Hammer, J. Dolby, M. Vaziri, and F. Tip, "Dynamic detection of atomic-set-serializability violations," in *Proc. Int. Conf. Softw. Eng.*, 2008, pp. 231–240.

[35] C. Flanagan, S. N. Freund, and J. Yi, "Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs," in *Proc. Conf. Program. Language Des. Implementation*, 2008, pp. 293–303.

[36] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "AVIO: Detecting atomicity violations via access interleaving invariants," in *Proc. Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2006, pp. 37–48.

[37] F. Sorrentino, A. Farzan, and P. Madhusudan, "PENELOPE: Weaving threads to expose atomicity violations," in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2010, pp. 37–46.

[38] M. Vaziri, F. Tip, and J. Dolby, "Associating synchronization constraints with data in an object-oriented language," in *Proc. Symp. Principles Program. Languages*, 2006, pp. 334–345.

[39] M. Singhal, "Deadlock detection in distributed systems," *IEEE Comput.*, vol. 22, no. 11, pp. 37–48, Nov. 1989.

[40] A. Farzan, P. Madhusudan, N. Razavi, and F. Sorrentino, "Predicting null-pointer dereferences in concurrent programs," in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2012, pp. 1–11.

[41] J. Chen, R. Hierons, and H. Ural, "Conditions for resolving observability problems in distributed testing," in *Proc. IFIP Int. Conf. Formal Techn. Netw. Distrib. Syst.*, 2004, pp. 229–242.

[42] R. O'Callahan and J.-D. Choi, "Hybrid dynamic data race detection," in *Proc. Symp. Principles Practice Parallel Program.*, 2003, pp. 167–178.

[43] O. Shacham, M. Sagiv, and A. Schuster, "Scaling model checking of dataraces using dynamic information," in *Proc. Symp. Principles Practice Parallel Program.*, 2005, pp. 107–118.

[44] T. Sheng, N. Vachharajani, S. Eranian, R. Hundt, W. Chen, and W. Zheng, "RACEZ: A lightweight and non-invasive race detection tool for production applications," in *Proc. Int. Conf. Softw. Eng.*, 2011, pp. 401–410.

[45] C. von Praun and T. R. Gross, "Object race detection," in *Proc. Conf. Object-Oriented Program. Syst. Appl.*, 2001, pp. 70–82.

[46] R. K. Karmani, P. Madhusudan, and B. M. Moore, "Thread contracts for safe parallelism," in *Proc. Symp. Principles Practice Parallel Program.*, 2011, pp. 125–134.

[47] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[48] A. Dinning and E. Schonberg, "An empirical comparison of monitoring algorithms for access anomaly detection," in *Proc. Symp. Principles Practice Parallel Program.*, 1990, pp. 1–10.

[49] R. H. B. Netzer, "Race condition detection for debugging shared-memory parallel programs," Ph.D. dissertation, Dept. Comput. Sci., Univ. Wisconsin-Madison Madison, WI, USA, 1991.

[50] C. Flanagan and S. N. Freund, "FastTrack: Efficient and precise dynamic race detection," in *Proc. Conf. Program. Language Des. Implementation*, 2009, pp. 121–133.

[51] D. Marino, M. Musuvathi, and S. Narayanasamy, "LiteRace: Effective sampling for lightweight data-race detection," in *Proc. Conf. Program. Language Des. Implementation*, 2009, pp. 134–143.

[52] M. D. Bond, K. E. Coons, and K. S. McKinley, "PACER: Proportional detection of data races," in *Proc. Conf. Program. Language Des. Implementation*, 2010, pp. 255–268.

[53] D. Li, W. Srisa-an, and M. B. Dwyer, "SOS: Saving time in dynamic race detection with stationary analysis," in *Proc. Conf. Object-Oriented Program. Syst. Appl.*, 2011, pp. 35–50.

[54] K. Zhai, B. Xu, W. K. Chan, and T. H. Tse, "CARISMA: A context-sensitive approach to race-condition sample-instance selection for multithreaded applications," in *Proc. Int. Symp. Softw. Testing Anal.*, 2012, pp. 221–231.

[55] B. Wester, D. Devecsery, P. M. Chen, J. Flinn, and S. Narayanasamy, "Parallelizing data race detection," in *Proc. Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2013, pp. 27–38.

[56] M. Prvulovic and J. Torrellas, "ReEnact: Using thread-level speculation mechanisms to debug data races in multithreaded codes," in *Proc. Annu. Int. Symp. Comput. Archit.*, 2003, pp. 110–121.

[57] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder, "Automatically classifying benign and harmful data races using replay analysis," in *Proc. Conf. Program. Language Des. Implementation*, 2007, pp. 22–31.

[58] K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy, "Detecting and surviving data races using complementary schedules," in *Proc. Symp. Operating Syst. Principles*, 2011, pp. 369–384.

[59] B. Kasikci, C. Zamfir, and G. Candea, "Data races versus data race bugs: Telling the difference with portend," in *Proc. Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2012, pp. 185–198.

[60] C. Tian, V. Nagarajan, R. Gupta, and S. Tallam, "Dynamic recognition of synchronization operations for improved data race detection," in *Proc. Int. Symp. Softw. Testing Anal.*, 2008, pp. 143–154.

[61] A. K. Rajagopalan and J. Huang, "RDIT: Race detection from incomplete traces," in *Proc. Eur. Softw. Eng. Conf. Held Jointly ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2015, pp. 914–917.

[62] Y. Cai and L. Cao, "Effective and precise dynamic detection of hidden races for Java programs," in *Proc. Eur. Softw. Eng. Conf. Held Jointly ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2015, pp. 450–461.

[63] B. Petrov, M. Vechev, M. Sridharan, and J. Dolby, "Race detection for Web applications," in *Proc. Conf. Program. Language Des. Implementation*, 2012, pp. 251–262.

[64] V. Raychev, M. Vechev, and M. Sridharan, "Effective race detection for event-driven programs," in *Proc. Conf. Object-Oriented Program. Syst. Appl.*, 2013, pp. 151–166.

[65] P. Maiya, A. Kanade, and R. Majumdar, "Race detection for android applications," in *Proc. Conf. Program. Language Des. Implementation*, 2014, pp. 316–325.

[66] K. Kim, T. Yavuz-Kahveci, and B. A. Sanders, "Precise data race detection in a relaxed memory model using heuristic-based model checking," in *Proc. Int. Conf. Automated Softw. Eng.*, 2009, pp. 495–499.

[67] J. Burnim, K. Sen, and C. Stergiou, "Testing concurrent programs on relaxed memory models," in *Proc. Int. Symp. Softw. Testing Anal.*, 2011, pp. 122–132.

[68] H. V. Jagadish, "A compression technique to materialize transitive closure," *ACM Trans. Database Syst.*, vol. 15, no. 4, pp. 558–598, 1990.

[69] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan, "Efficient and precise datarace detection for multithreaded object-oriented programs," in *Proc. Conf. Program. Language Des. Implementation*, 2002, pp. 258–269.

[70] B. Kasikci, C. Zamfir, and G. Candea, "RaceMob: Crowdsourced data race detection," in *Proc. Symp. Operating Syst. Principles*, 2013, pp. 406–422.

[71] K. Sen, "Race directed random testing of concurrent programs," in *Proc. Conf. Program. Language Des. Implementation*, 2008, pp. 11–21.

[72] Y. Yu, T. Rodeheffer, and W. Chen, "RaceTrack: Efficient detection of data race conditions via adaptive tracking," in *Proc. Symp. Operating Syst. Principles*, 2005, pp. 221–234.

[73] T. Elmas, S. Qadeer, and S. Tasiran, "Goldilocks: A race and transaction-aware Java runtime," in *Proc. Conf. Program. Language Des. Implementation*, 2007, pp. 245–255.

[74] E. Pozniansky and A. Schuster, "Efficient on-the-fly data race detection in multithreaded C++ programs," in *Proc. Symp. Principles Practice Parallel Program.*, 2003, pp. 179–190.

[75] T. Yu, W. Srisa-an, and G. Rothermel, "SimRT: An automated framework to support regression testing for data races," in *Proc. Int. Conf. Softw. Eng.*, 2014, pp. 48–59.

[76] M. Eslamimehr and J. Palsberg, "Race directed scheduling of concurrent programs," in *Proc. Symp. Principles Practice Parallel Program.*, 2014, pp. 301–314.

[77] M. Samak, M. K. Ramanathan, and S. Jagannathan, "Synthesizing racy tests," in *Proc. Conf. Program. Language Des. Implementation*, 2015, pp. 175–185.

[78] O. Shacham, N. Bronson, A. Aiken, M. Sagiv, M. Vechev, and E. Yahav, "Testing atomicity of composed concurrent operations," in *Proc. Conf. Object-Oriented Program. Syst. Appl.*, 2011, pp. 51–64.

[79] D. Dimitrov, V. Raychev, M. Vechev, and E. Koskinen, "Commutativity race detection," in *Proc. Conf. Program. Language Des. Implementation*, 2014, pp. 305–315.

[80] M. Vechev, E. Yahav, and G. Yorsh, "Experience with model checking linearizability," in *Model Checking Software*. Berlin, Germany: Springer, 2009, pp. 261–278.

[81] M. Xu, R. Bodík, and M. D. Hill, "A serializability violation detector for shared-memory server programs," in *Proc. Conf. Program. Language Des. Implementation*, 2005, pp. 1–14.
[82] Q. Chen, L. Wang, Z. Yang, and S. D. Stoller, "HAVE: Detecting atomicity violations via integrated dynamic and static analysis," in *Proc. Int. Conf. Fundam. Approaches Softw. Eng.*, 2009, pp. 425–439.
[83] C.-S. Park and K. Sen, "Randomized active atomicity violation detection in concurrent programs," in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2008, pp. 135–145.
[84] L. Wang and S. D. Stoller, "Runtime analysis of atomicity for multithreaded programs," *IEEE Trans. Softw. Eng.*, vol. 32, no. 2, pp. 93–110, Feb. 2006.
[85] S. Park, S. Lu, and Y. Zhou, "CTrigger: Exposing atomicity violation bugs from their hiding places," in *Proc. Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2009, pp. 25–36.
[86] S. Park, R. W. Vuduc, and M. J. Harrold, "Falcon: Fault localization in concurrent programs," in *Proc. Int. Conf. Softw. Eng.*, 2010, pp. 245–254.
[87] M. K. Ganai, "Scalable and precise symbolic analysis for atomicity violations," in *Proc. Int. Conf. Automated Softw. Eng.*, 2011, pp. 123–132.
[88] S. Biswas, J. Huang, A. Sengupta, and M. D. Bond, "DoubleChecker: Efficient sound and precise atomicity checking," in *Proc. Conf. Program. Language Des. Implementation*, 2014, pp. 28–39.
[89] M. Samak and M. K. Ramanathan, "Synthesizing tests for detecting atomicity violations," in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2015, pp. 131–142.
[90] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Reading, MA, USA: Addison-Wesley, 1987.
[91] S. Lu, et al., "MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs," in *Proc. Symp. Operating Syst. Principles*, 2007, pp. 103–116.
[92] Z. Lai, S. C. Cheung, and W. K. Chan, "Detecting atomic-set serializability violations in multithreaded programs through active randomized testing," in *Proc. Int. Conf. Softw. Eng.*, 2010, pp. 235–244.
[93] V. Terragni, S.-C. Cheung, and C. Zhang, "RECONTEST: Effective regression testing of concurrent programs," in *Proc. Int. Conf. Softw. Eng.*, 2015, pp. 246–256.
[94] K. Havelund, "Using runtime analysis to guide model checking of Java programs," in *Proc. Int. SPIN Workshop SPIN Model Checking Softw. Verification*, 2000, pp. 245–264.
[95] P. Joshi, C. Park, K. Sen, and M. Naik, "A randomized dynamic program analysis technique for detecting real deadlocks," in *Proc. Conf. Program. Language Des. Implementation*, 2009, pp. 110–120.
[96] Y. Cai and W. K. Chan, "MagicFuzzer: Scalable deadlock detection for large-scale applications," in *Proc. Int. Conf. Softw. Eng.*, 2012, pp. 606–616.
[97] M. Samak and M. K. Ramanathan, "Trace driven dynamic deadlock detection and reproduction," in *Proc. Symp. Principles Practice Parallel Program.*, 2014, pp. 29–42.
[98] Y. Cai, S. Wu, and W. K. Chan, "ConLock: A constraint-based approach to dynamic checking on deadlocks in multithreaded programs," in *Proc. Int. Conf. Softw. Eng.*, 2014, pp. 491–502.
[99] M. Eslamimehr and J. Palsberg, "Sherlock: Scalable deadlock detection for concurrent programs," in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 353–365.
[100] M. Samak and M. K. Ramanathan, "Multithreaded test synthesis for deadlock detection," in *Proc. Conf. Object-Oriented Program. Syst. Appl.*, 2014, pp. 473–489.
[101] T. Cogumbreiro, R. Hu, F. Martins, and N. Yoshida, "Dynamic deadlock verification for general barrier synchronisation," in *Proc. Symp. Principles Practice Parallel Program.*, 2015, pp. 150–160.
[102] R. Agarwal, A. Sasturkar, L. Wang, and S. D. Stoller, "Optimized run-time race detection and atomicity checking using partial discovered types," in *Proc. Int. Conf. Automated Softw. Eng.*, 2005, pp. 233–242.
[103] J. Chen and S. MacDonald, "Testing concurrent programs using value schedules," in *Proc. Int. Conf. Automated Softw. Eng.*, 2007, pp. 313–322.
[104] V. Kahlon and C. Wang, "Universal causality graphs: A precise happens-before model for detecting bugs in concurrent programs," in *Proc. Int. Conf. Comput. Aided Verification*, 2010, pp. 434–449.
[105] J. Huang and C. Zhang, "Persuasive prediction of concurrency access anomalies," in *Proc. Int. Symp. Softw. Testing Anal.*, 2011, pp. 144–154.
[106] P. Joshi and K. Sen, "Predictive typestate checking of multithreaded Java programs," in *Proc. Int. Conf. Automated Softw. Eng.*, 2008, pp. 288–296.
[107] Q. Gao, W. Zhang, Z. Chen, M. Zheng, and F. Qin, "2ndStrike: Toward manifesting hidden concurrency typestate bugs," in *Proc. Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2011, pp. 239–250.
[108] W. Zhang, C. Sun, and S. Lu, "ConMem: Detecting severe concurrency bugs through an effect-oriented approach," in *Proc. Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2010, pp. 179–192.
[109] W. Zhang, et al., "ConSeq: Detecting concurrency bugs through sequential errors," in *Proc. Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2011, pp. 251–264.
[110] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam, "Maple: A coverage-driven testing tool for multithreaded programs," in *Proc. Conf. Object-Oriented Program. Syst. Appl.*, 2012, pp. 485–502.
[111] F. Chen, T. F. Serbanuta, and G. Rosu, "jPredictor: A predictive runtime analysis tool for Java," in *Proc. Int. Conf. Softw. Eng.*, 2008, pp. 221–230.
[112] N. Sinha and C. Wang, "Staged concurrent program analysis," in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2010, pp. 47–56.
[113] J. Huang, Q. Luo, and G. Rosu, "GPredict: Generic predictive concurrency analysis," in *Proc. Int. Conf. Softw. Eng.*, 2015, pp. 847–857.
[114] Y. Shi, et al., "Do I use the wrong definition?: Defuse: Definition-use invariants for detecting concurrency and sequential bugs," in *Proc. Conf. Object-Oriented Program. Syst. Appl.*, 2010, pp. 160–174.
[115] T. Yu, W. Srisa-an, and G. Rothermel, "SimRacer: An automated framework to support testing for process-level races," in *Proc. Int. Symp. Softw. Testing Anal.*, 2013, pp. 167–177.
[116] C.-H. Hsiao, et al., "Race detection for event-driven mobile applications," in *Proc. Conf. Program. Language Des. Implementation*, 2014, pp. 326–336.
[117] E. Mutlu, S. Tasiran, and B. Livshits, "Detecting JavaScript races that matter," in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2015, pp. 381–392.
[118] R. E. Strom and S. Yemini, "Typestate: A programming language concept for enhancing software reliability," *IEEE Trans. Softw. Eng.*, vol. SE-12, no. 1, pp. 157–171, Jan. 1986.
[119] P. Fonseca, C. Li, and R. Rodrigues, "Finding complex concurrency bugs in large multi-threaded applications," in *Proc. ACM SIGOPS EuroSys Eur. Conf. Comput. Syst.*, 2011, pp. 215–228.
[120] M. Pradel, M. Huggler, and T. R. Gross, "Performance regression testing of concurrent classes," in *Proc. Int. Symp. Softw. Testing Anal.*, 2014, pp. 13–25.
[121] A. Nistor, Q. Luo, M. Pradel, T. R. Gross, and D. Marinov, "BALLERINA: Automatic generation and clustering of efficient random unit tests for multithreaded code," in *Proc. Int. Conf. Softw. Eng.*, 2012, pp. 727–737.
[122] M. Pradel and T. R. Gross, "Fully automatic and precise detection of thread safety violations," in *Proc. Conf. Program. Language Des. Implementation*, 2012, pp. 521–530.
[123] M. Pradel and T. R. Gross, "Automatic testing of sequential and concurrent substitutability," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 282–291.
[124] K. Sen and G. Agha, "Automated systematic testing of open distributed programs," in *Proc. Int. Conf. Fundam. Approaches Softw. Eng.*, 2006, pp. 339–356.
[125] S. Lauterburg, M. Dotta, D. Marinov, and G. A. Agha, "A framework for state-space exploration of Java-based actor programs," in *Proc. Int. Conf. Automated Softw. Eng.*, 2009, pp. 468–479.
[126] P. Joshi, M. Naik, K. Sen, and D. Gay, "An effective dynamic analysis for detecting generalized deadlocks," in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2010, pp. 327–336.
[127] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber, "Mathematizing C++ concurrency," in *Proc. Symp. Principles Program. Languages*, 2011, pp. 55–66.
[128] J. Dingel and H. Liang, "Automating comprehensive safety analysis of concurrent programs using verisoft and TXL," in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2004, pp. 13–22.
[129] B. Norris and B. Demsky, "CDSchecker: Checking concurrent data structures written with C/C++ atomics," in *Proc. Conf. Object-Oriented Program. Syst. Appl.*, 2013, pp. 131–150.

[130] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," in *Proc. Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2008, pp. 329–339.

[131] W. Visser, C. S. Păsăreanu, and S. Khurshid, "Test input generation with Java pathfinder," in *Proc. Int. Symp. Softw. Testing Anal.*, 2004, pp. 97–107.

[132] J. C. Blanchette, T. Weber, M. Batty, S. Owens, and S. Sarkar, "Nitpicking C++ concurrency," in *Proc. Int. Symp. Principles Practices Declarative Program.*, 2011, pp. 113–124.

[133] C. Wang, M. Said, and A. Gupta, "Coverage guided systematic concurrency testing," in *Proc. Int. Conf. Softw. Eng.*, 2011, pp. 221–230.

[134] S. Hong, J. Ahn, S. Park, M. Kim, and M. J. Harrold, "Testing concurrent programs to achieve high synchronization coverage," in *Proc. Int. Symp. Softw. Testing Anal.*, 2012, pp. 210–220.

[135] S. Tasharofi, M. Pradel, Y. Lin, and R. E. Johnson, "Bita: Coverage-guided, automatic testing of actor programs," in *Proc. Int. Conf. Automated Softw. Eng.*, 2013, pp. 114–124.

[136] E. Trainin, Y. Nir-Buchbinder, R. Tzoref-Brill, A. Zlotnick, S. Ur, and E. Farchi, "Forcing small models of conditions on program interleaving for detection of concurrent bugs," in *Proc. Workshop Parallel Distrib. Syst.: Testing Anal. Debugging*, 2009, pp. 1–6.

[137] K. Sen, "Effective random testing of concurrent programs," in *Proc. Int. Conf. Automated Softw. Eng.*, 2007, pp. 323–332.

[138] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte, "A randomized scheduler with probabilistic guarantees of finding bugs," in *Proc. Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2010, pp. 167–178.

[139] K. E. Coons, S. Burckhardt, and M. Musuvathi, "GAMBIT: Effective unit testing for concurrency libraries," in *Proc. Symp. Principles Practice Parallel Program.*, 2010, pp. 15–24.

[140] K. Sen, G. Rosu, and G. Agha, "Runtime safety analysis of multithreaded programs," in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2003, pp. 337–346.

[141] S. Chandra and P. M. Chen, "Whither generic recovery from application faults? A fault study using open-source software," in *Proc. Int. Conf. Depend. Syst. Netw.*, 2000, pp. 97–106.

[142] P. Fonseca, C. Li, V. Singhal, and R. Rodrigues, "A study of the internal and external effects of concurrency bugs," in *Proc. Int. Conf. Depend. Syst. Netw.*, 2010, pp. 221–230.

[143] Y. Lin and D. Dig, "Check-then-act misuse of Java concurrent collections," in *Proc. Int. Conf. Softw. Testing Verification Validation*, 2013, pp. 164–173.

[144] Z. Lin, D. Marinov, H. Zhong, Y. Chen, and J. Zhao, "JaConTeBe: A benchmark suite of real-world Java concurrency bugs (T)," in *Proc. Int. Conf. Automated Softw. Eng.*, 2015, pp. 178–189.

[145] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *Automated Softw. Eng.*, vol. 10, no. 2, pp. 203–232, 2003.

[146] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," in *Proc. Symp. Principles Program. Languages*, 2005, pp. 110–121.

[147] K. Kähkönen, O. Saarikivi, and K. Heljanko, "Using unfoldings in automated testing of multithreaded programs," in *Proc. Int. Conf. Automated Softw. Eng.*, 2012, pp. 150–159.

[148] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. Upper Saddle River, NJ, USA: Pearson Education, 2003.

[149] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing Heisenbugs in concurrent programs," in *Proc. Symp. Operating Syst. Des. Implementation*, 2008, pp. 267–280.

[150] M. Musuvathi and S. Qadeer, "Fair stateless model checking," in *Proc. Conf. Program. Language Des. and Implementation*, 2008, pp. 362–371.

[151] M. Emmi, S. Qadeer, and Z. Rakamarić, "Delay-bounded scheduling," in *Proc. Symp. Principles Program. Languages*, 2011, pp. 411–422.

[152] P. Thomson, A. F. Donaldson, and A. Betts, "Concurrency testing using schedule bounding: An empirical study," in *Proc. Symp. Principles Practice Parallel Program.*, 2014, pp. 15–28.

[153] S. Bindal, S. Bansal, and A. Lal, "Variable and thread bounding for systematic testing of multithreaded programs," in *Proc. Int. Symp. Softw. Testing Anal.*, 2013, pp. 145–155.

[154] V. Jagannath, Q. Luo, and D. Marinov, "Change-aware preemption prioritization," in *Proc. Int. Symp. Softw. Testing Anal.*, 2011, pp. 133–143.

[155] A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur, "Applications of synchronization coverage," in *Proc. Symp. Principles Practice Parallel Program.*, 2005, pp. 206–212.

[156] S. Lu, W. Jiang, and Y. Zhou, "A study of interleaving coverage criteria," in *Proc. Eur. Softw. Eng. Conf. Held Jointly ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2007, pp. 533–536.

[157] B. Křena, Z. Letko, and T. Vojnar, "Coverage metrics for saturation-based and search-based testing of concurrent software," in *Proc. Int. Conf. Runtime Verification*, 2012, pp. 177–192.

[158] S. Hong, M. Staats, J. Ahn, M. Kim, and G. Rothermel, "The impact of concurrent coverage metrics on testing effectiveness," in *Proc. Int. Conf. Softw. Testing Verification Validation*, 2013, pp. 232–241.

[159] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649–678, Sep./Oct. 2011.

[160] J. S. Bradbury, J. R. Cordy, and J. Dingel, "Mutation operators for concurrent Java (J2SE 5.0)," in *Proc. Workshop Mutation Anal.*, 2006, pp. 11–11.

[161] B. K. Aichernig and C. C. Delgado, "From faults via test purposes to test cases: On the fault-based testing of concurrent systems," in *Proc. Int. Conf. Fundam. Approaches Softw. Eng.*, 2006, pp. 324–338.

[162] A. Sen and M. S. Abadir, "Coverage metrics for verification of concurrent SystemC designs using mutation testing," in *Proc. Int. High Level Des. Validation Test Workshop*, 2010, pp. 75–81.

[163] R. H. Carver, "Mutation-based testing of concurrent programs," in *Proc. Int. Test Conf.*, 1993, pp. 845–853.

[164] M. Gligoric, V. Jagannath, and D. Marinov, "MuTMuT: Efficient exploration for mutation testing of multithreaded code," in *Proc. Int. Conf. Softw. Testing Verification Validation*, 2010, pp. 55–64.

[165] M. Gligoric, L. Zhang, C. Pereira, and G. Pokam, "Selective mutation testing for concurrent code," in *Proc. Int. Symp. Softw. Testing Anal.*, 2013, pp. 224–234.

[166] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using magpie for request extraction and workload modelling," in *Proc. Symp. Operating Syst. Des. Implementation*, 2004, pp. 18–18.

[167] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-trace: A pervasive network tracing framework," in *Proc. Conf. Netw. Syst. Des. Implementation*, 2007, pp. 20–20.

[168] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat, "Pip: Detecting the unexpected in distributed systems," in *Proc. Conf. Netw. Syst. Des. Implementation*, 2006, pp. 9–9.

[169] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica, "Friday: Global comprehension for distributed replay," in *Proc. Conf. Netw. Syst. Des. Implementation*, 2007, pp. 21–21.

[170] X. Liu, W. Lin, A. Pan, and Z. Zhang, "WiDS checker: Combating bugs in distributed systems," in *Proc. Conf. Netw. Syst. Des. Implementation*, 2007, pp. 257–270.

[171] K. Nagaraj, C. Killian, and J. Neville, "Structured comparative analysis of systems logs to diagnose performance problems," in *Proc. Conf. Netw. Syst. Des. Implementation*, 2012, pp. 26–26.

[172] A. Singh, P. Maniatis, T. Roscoe, and P. Druschel, "Using queries for distributed monitoring and forensics," in *Proc. ACM SIGOPS EuroSys Eur. Conf. Comput. Syst.*, 2006, pp. 389–402.

[173] X. Liu, et al., "D3S: Debugging deployed distributed systems," in *Proc. Conf. Netw. Syst. Des. Implementation*, 2008, pp. 423–437.

[174] D. Dao, J. Albrecht, C. Killian, and A. Vahdat, "Live debugging of distributed systems," in *Proc. Int. Conf. Compiler Construction*, 2009, pp. 94–108.

[175] M. Yabandeh, N. Knezevic, D. Kostic, and V. Kuncak, "CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems," in *Proc. Conf. Netw. Syst. Des. Implementation*, 2009, pp. 229–244.

[176] M. Yabandeh, N. Knežević, D. Kostić, and V. Kuncak, "Predicting and preventing inconsistencies in deployed distributed systems," *ACM Trans. Comput. Syst.*, vol. 28, no. 1, pp. 1–49, 2010.

[177] D. R. Engler and K. Ashcraft, "RacerX: Effective, static detection of race conditions and deadlocks," in *Proc. Symp. Operating Syst. Principles*, 2003, pp. 237–252.

[178] C. von Praun and T. R. Gross, "Static detection of atomicity violations in object-oriented programs," *J. Object Technol.*, vol. 3, no. 6, pp. 103–122, 2004.

[179] A. Williams, W. Thies, and M. D. Ernst, "Static deadlock detection for Java libraries," in *Proc. Eur. Conf. Object-Oriented Program.*, 2005, pp. 602–629.

[180] M. Naik, C. Park, K. Sen, and D. Gay, "Effective static deadlock detection," in *Proc. Int. Conf. Softw. Eng.*, 2009, pp. 386–396.

[181] D. Marino, C. Hammer, J. Dolby, M. Vaziri, F. Tip, and J. Vitek, "Detecting deadlock in programs with data-centric synchronization," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 322–331.
[182] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, MA USA: MIT Press, 1999.
[183] E. A. Emerson and E. M. Clarke, "Characterizing correctness properties of parallel programs using fixpoints," in *Proc. Colloq. Automata Languages Program.*, 1980, pp. 169–181.
[184] J. P. Queille and J. Sifakis, *Specification and Verification of Concurrent Systems in CESAR*. Berlin, Germany: Springer, 1982, pp. 337–351.
[185] B. Sarikaya and G. V. Bochmann, "Synchronization and specification issues in protocol testing," *IEEE Trans. Commun.*, vol. C-32, no. 4, pp. 389–395, Apr. 1984.
[186] R. M. Hierons and H. Ural, "The effect of the distributed test architecture on the power of testing," *Comput. J.*, vol. 51, no. 4, pp. 497–510, 2008.
[187] E. Brinksma, L. Heerink, and J. Tretmans, "Factorized test generation for multi-input/output transition systems," in *Testing of Communicating Systems*. Berlin, Germany: Springer, 1998, pp. 67–82.
[188] R. Hierons, M. Merayo, and M. Nez, "Implementation relations and test generation for systems with distributed interfaces," *Distrib. Comput.*, vol. 25, no. 1, pp. 35–62, 2012.
[189] G. Bochmann, S. Haar, C. Jard, and G.-V. Jourdan, "Testing systems specified as partial order input/output automata," in *Testing of Software and Communicating Systems*. Berlin, Germany: Springer, 2008, pp. 169–183.
[190] S. Haar, C. Jard, and G.-V. Jourdan, "Testing input/output partial order automata," in *Testing of Software and Communicating Systems*. Berlin, Germany: Springer, 2007, pp. 171–185.
[191] H. Ponce de Leòn, S. Haar, and D. Longuet, "Unfolding-based test selection for concurrent conformance," in *Testing Software and Systems*. Berlin, Germany: Springer, 2013, pp. 98–113.
[192] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel Programming in OpenMP*. San Mateo, CA, USA: Morgan Kaufmann Publishers, 2001.
[193] K. Sakamoto and T. Furumoto, "Grand central dispatch" in *Pro Multithreading and Memory Management for iOS and OS X*. New York, NY, USA: Apress, 2012, pp. 139–145.
[194] A. Santhiar, S. Kaleeswaran, and A. Kanade, "Efficient race detection in the presence of programmatic event loops," in *Proc. Int. Symp. Softw. Testing Anal.*, 2016, pp. 366–376.
[195] C. Campbell, R. Johnson, A. Miller, and S. Toub, *Parallel Programming with Microsoft .NET: Design Patterns for Decomposition and Coordination on Multicore Architectures*. Redmond, WA, USA: Microsoft Press, 2010.
[196] S. Okur, D. L. Hartveld, D. Dig, and A. V. Deursen, "A study and toolkit for asynchronous programming in C#," in *Proc. Int. Conf. Softw. Eng.*, 2014, pp. 1117–1127.
[197] S. Peyton Jones, A. Gordon, and S. Finne, "Concurrent haskell," in *Proc. Symp. Principles Program. Languages*, 1996, pp. 295–308.
[198] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[199] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. USENIX Conf. Hot Topics Cloud Comput.*, 2010, pp. 10–10.
[200] A. Alexandrov, et al., "The stratosphere platform for big data analytics," *VLDB J.*, vol. 23, no. 6, pp. 939–964, 2014.
[201] E. Meijer, "Your mouse is a database," *ACM Queue*, vol. 10, no. 3, pp. 20–33, 2012.
[202] R. Virding, C. Wikström, and M. Williams, *Concurrent Programming in ERLANG*, J. Armstrong, Ed., 2nd ed. London, U.K.: Prentice Hall, 1996.
[203] C. Flanagan and M. Abadi, "Types for safe locking," in *Proc. Eur. Symp. Program. Languages Syst.*, 1999, pp. 91–108.
[204] N. Kobayashi, "A new type system for deadlock-free processes," in *Proc. Int. Conf. Concurrency Theory*, 2006, pp. 233–247.
[205] C. Flanagan and S. N. Freund, "Type-based race detection for java," in *Proc. Conf. Program. Language Des. Implementation*, 2000, pp. 219–232.
[206] C. Boyapati and M. Rinard, "A parameterized type system for race-free Java programs," in *Proc. Conf. Object-Oriented Program. Syst. Appl.*, 2001, pp. 56–69.
[207] Z. Anderson, D. Gay, R. Ennals, and E. Brewer, "SharC: Checking data sharing strategies for multithreaded C," in *Proc. Conf. Program. Language Des. Implementation*, 2008, pp. 149–158.
[208] C. Flanagan and S. Qadeer, "A type and effect system for atomicity," in *Proc. Conf. Program. Language Des. Implementation*, 2003, pp. 338–349.
[209] S. West, S. Nanz, and B. Meyer, "Efficient and reasonable object-oriented concurrency," in *Proc. Symp. Principles Practice Parallel Program.*, 2015, pp. 273–274.
[210] S. T. Heumann, V. S. Adve, and S. Wang, "The tasks with effects model for safe concurrency," in *Proc. Symp. Principles Practice Parallel Program.*, 2013, pp. 239–250.
[211] P. Deligiannis, A. F. Donaldson, J. Ketema, A. Lal, and P. Thomson, "Asynchronous programming, analysis and testing with state machines," in *Proc. Conf. Program. Language Des. and Implementation, ser. PLDI* 2015. ACM, 2015, pp. 154–164.
[212] R. L. Bocchino, Jr., et al., "A type and effect system for deterministic parallel Java," in *Proc. Conf. Object-Oriented Program. Syst. Appl.*, 2009, pp. 97–116.
[213] E. D. Berger, T. Yang, T. Liu, and G. Novark, "Grace: Safe multithreaded programming for C/C++," in *Proc. Conf. Object-Oriented Program. Syst. Appl.*, 2009, pp. 81–96.
[214] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman, "CoreDet: A compiler and runtime system for deterministic multithreaded execution," in *Proc. Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2010, pp. 53–64.
[215] T. Liu, C. Curtsinger, and E. D. Berger, "DTHREADS: Efficient deterministic multithreading," in *Proc. Symp. Operating Syst. Principles*, 2011, pp. 327–336.
[216] A. Aviram, S.-C. Weng, S. Hu, and B. Ford, "Efficient system-enforced deterministic parallelism," *Commun. ACM*, vol. 55, no. 5, pp. 111–119, 2012.

**Francesco Adalberto Bianchi** received the bachelor's and master's degree in computer science from the Università degli Studi Milano Bicocca, Milano. He is working toward the PhD degree in the Faculty of Informatics, Università della Svizzera italiana, Lugano. He currently works under the supervision of Prof. Mauro Pezzè. His research interests include program analysis and testing concurrent and distributed systems.

**Alessandro Margara** received the PhD degree from the Dipartimento di Elettronica e Informazione, Politecnico di Milano, under the supervision of Prof. Gianpaolo Cugola. He is an assistant professor with Politecnico di Milano. His research interests focus on middleware solutions to ease the design and development of complex distributed systems, with a special emphasis on event stream processing and reactive systems. His work includes the definition of languages for event and stream processing and the implementation of parallel and distributed algorithms to efficiently support such languages. He has been a postdoctoral researcher with Vrije Universiteit Amsterdam and at Università della Svizzera italiana, Lugano. Some of his recent publications appear in the *ACM Computing Surveys*, the *IEEE Transactions on Parallel and Distributed Systems*, DEBS'14, ICDCS'14, DEBS'15, and ICSE'15.

**Mauro Pezzè** is a professor of software engineering with the University of Milano-Bicocca and the Università della Svizzera italiana. He is associate editor of the *IEEE Transactions on Software Engineering*, and has served as associate editor of the *ACM Transactions on Software Engineering and Methodology*, as general chair or the ACM International Symposium on Software Testing and Analysis in 2013, program chair of the International Conference on Software Engineering in 2012 and of the ACM International Symposium on Software Testing and Analysis in 2006. He is known for his work on software testing, program analysis, self-healing, and self-adaptive software systems.