Discovering Concurrency Errors

João M. Lourenço[1], Jan Fiedor[2], Bohuslav Křena[2], and Tomáš Vojnar[2]

[1] NOVA LINCS and NOVA University Lisbon, FCT, DI, Portugal
[2] Brno University of Technology, Faculty of Information Technology, IT4Innovations Centre of Excellence, Czech Republic

Abstract. Lots of concurrent software is being developed for the now ubiquitous multicore processors. And concurrent programming is difficult because it is quite easy to introduce errors that are really hard to diagnose and fix. One of the main obstacles to concurrent programming is that threads are scheduled nondeterministically and their interactions may become hard to predict and to devise. This chapter addresses the nature of concurrent programming and some classes of concurrency errors. It discusses the application of dynamic program analysis techniques to detect, locate and diagnose some common concurrency errors like data races, atomicity violations and deadlocks. This chapter also mentions some techniques that can help with quality assurance of concurrent programs, regardless of any particular class of concurrency errors, like noise injection and systematic testing, and it is closed by some prospects of concurrent software development.

Keywords: Software Correctness, Quality Assurance, Nondeterminism, Concurrency Errors, Atomicity Violations, Data Races, Deadlocks, Dynamic Analysis, Noise Injection.

1 Introduction

The arrival of multi-core processors into computers, laptops, tablets, phones, and other devices demands the development of software products that make use of multi-threaded design to better use the available hardware resources. Modern programming languages allow programmers to easily create multi-threaded programs, at the expense of a significant increase in the number and variety of errors appearing in the code. The basic difficulty is introduced by the conflict between safety and efficiency. It is not easy to set up the appropriate synchronisation among threads ensuring safety and low overhead simultaneously. If the synchronisation is too strict, the performance of the application degrades as the computation power brought by the presence of several computational cores becomes underused. On the other hand, if the concurrent program is under-synchronised, some failures may occur, like wrong results and application crashes. As an example of what can be the impact of a concurrency error, we refer to the Northeastern U.S. blackout in August 2003, where a race condition error was identified as one of the causes [59, 60].

Creating concurrent programs is more demanding on programmers, since people usually think in a linear way. It is not easy to imagine the parallel execution and the possible interactions of dozens of threads, even if the programmer concentrates on it; on the contrary, programmers think most of the time about snippets of sequential code despite they may be executed in parallel. Moreover, errors in concurrency are not only easy to create, but also very difficult to detect and diagnose due to the nondeterministic nature of multi-threaded computation. For instance, the error from Northeastern blackout has been unmasked about eight weeks after the blackout [60]. Some concurrent errors may manifest rarely or under special circumstances, making it difficult to discover them by testing as well as to reproduce them while debugging.

This chapter is organised as follows. It starts with a discussion on the nature of concurrent computations and on the errors that can arise from the execution of concurrent programs, including a motivating example, in Section 2. Different concurrency errors are then presented and classified in Section 3. Section 4 describes various approaches for monitoring the execution of concurrent programs as monitoring allows one to obtain all the necessary information to perform an analysis and detect errors. Follows a discussion on the detection of common classes of concurrency errors, with Section 5 addressing errors related to atomicity violations, and Section 6 addressing deadlocks. The detection of less common concurrency errors is discussed in Section 7. Techniques that can help with quality assurance of concurrent programs, regardless of any particular class of concurrency errors, like noise injection and systematic testing, are discussed in Section 8. Section 9 sums up the chapter and provides some prospects for concurrent software development.

2 Errors in Concurrency

To understand errors in concurrency, one first needs to understand the nature of concurrent execution. The execution of a concurrent program is performed simultaneously by several processes (they can be called threads or nodes as well). All the processes have access to a shared memory that serves as a communication mean among them.[1] Additionally, each process has its own local memory that can be typically accessed much faster than the shared memory. Although memory in computers is usually organised in a hierarchy with several levels, each with different size, speed, and price per bit, simple differentiation between shared and local memory of processes is enough to show the basis of concurrent errors.

As the shared memory is usually much slower than the local memory, a typical scenario of the execution performed by a process is copying data from the shared memory to its local memory, performing the given computation using the local memory, and storing the result back to the shared memory. At the first sight, there is no problem with this operation model, however, this is true only if a single

[1] In this Chapter, we concentrate on the shared memory paradigm, leaving behind the distributed memory and message passing paradigms, which are covered elsewhere in this book.

process is working with that particular data in the shared memory. If two or more processes are operating concurrently over the same data in the shared memory (perceived by a programmer, for instance, as a shared variable), some problems may arise. We illustrate this with the help of a very simple concurrent system.

Let us have a shared variable `x`, initialised to zero, and two threads operating concurrently on `x`, one thread adding one to `x` and the other adding two to `x`.

`x=0 . ( x++ || x+=2 )`

What will be the final value of the variable `x` after this computation ends? The most obvious answer would be 3. However, it is not necessarily the case if we take concurrency into account. First, it may happen that incrementing and adding are not implemented using single instructions. For instance, they can be implemented by a three steps procedure: 1) loading the value of a shared variable `x` from the shared memory to the local memory (e.g., to a processor register); 2) adding the intended value to the local copy of `x`; and 3) storing the new value of `x` back into the shared memory.

Here is an example in Java bytecode (no knowledge of Java neither of Java bytecode is required to understand the example):

Thread 1:	Thread 2:
`load x`	`load x`
`inc`	`add 2`
`store x`	`store x`

Nothing bad happens if the threads do not interfere with each other while executing. In the following, Thread 2 starts its work only after Thread 1 completes its whole execution. On the right-hand side we can see the evolution of the values of the shared variable `x` and of its local copies $\mathtt{x}_1$ for Thread 1 and $\mathtt{x}_2$ for Thread 2. The outcome of the execution is highlighted by a frame.

Thread 1:	Thread 2:	`x`	$\mathtt{x}_1$	$\mathtt{x}_2$
`load x`		0	0	
`inc`		0	1	
`store x`		1	1	
	`load x`	1		1
	`add 2`	1		3
	`store x`	$\boxed{3}$		3

The same outcome is achieved if Thread 1 starts its work after Thread 2 completes its executions. The problem occurs when the executions of these two threads are interleaved. For instance, the first thread starts executing as in the previous example, however, the second thread starts and loads the value of the shared variable `x` before the first thread stores its result back. Then, the result of Thread 1 is lost because it is overwritten by Thread 2 when it stores its own result back to `x`, as illustrated by the example below.

Thread 1:	Thread 2:	x	x_1	x_2
`load x`		0	0	
`inc`		0	1	
	`load x`	0	1	0
`store x`		1	1	0
	`add 2`	1		2
	`store x`	$\boxed{2}$		2

Yet another outcome can be seen if the result produced by Thread 2 is overwritten by Thread 1.

Thread 1:	Thread 2:	x	x_1	x_2
`load x`		0	0	
`inc`		0	1	
	`load x`	0	1	0
	`add 2`	1	1	2
	`store x`	2	1	2
`store x`		$\boxed{1}$	1	

In this example, we can highlight two issues related to concurrency. First, programmers usually write code in a more abstract level than what it is actually executed. When writing code, the programmer does not need to care (and sometimes does not even know) about the underlying implementation. Rather, it is quite natural to assume that adding a value to a variable is an atomic operation. Thus, the non-atomicity of some operations is actually hidden from the programmer and the concurrency errors are not easily visible in the source code.

It is commonly accepted that errors that are not realised by the programmer, because they are hidden in less frequent operations or branches of the source code, can frequently be uncovered by a proper testing activity. And this raises the second issue. When a code block contains a concurrency related error, frequently there is a huge number of different ways this erroneous code block can interact with the other code blocks of the program, and all but a few of them will trigger the error. Thus, even when involving the erroneous code block, the testing procedures most probably will not uncover the error, which will stay hidden. The error will be uncovered only if one of the low-probable erroneous interactions is exercised by the testing procedures. And this may happen very seldom or, in some cases, even never!

The execution order of particular instructions by some given threads is called *thread interleaving*. The set of all possible thread interleavings defines the semantics of a concurrent program. In the previous example, there are 20 possible thread interleavings from which most probably only two are common: one where thread 1 executes before thread 2 and another where thread 2 executes before thread 1. These two common interleavings have, moreover, special importance because they can also be achieved by a sequential execution. Interleavings which

are equivalent to some sequential execution are called *serialisable*. If all the possible thread interleavings are serialisable, the multi-threaded program is correct (of course, provided that the sequential program is correct). This notion of correctness of multiprocess programs called *sequential consistency* has been introduced by Lamport in [42]. Obviously, our example is not sequentially consistent because it can produce results that cannot be obtained by a sequential execution of the given threads.

3 Classification of Concurrency Errors

In general, we can think of a concurrency error as a behaviour that does not respect the sequential consistency model, which, in a nutshell, means the behaviour/result could not be obtained by a sequential execution (i.e., it is *unserialisable*). For efficient handling of concurrency errors, however, one needs to use the divide-and-conquer strategy and concentrate and deal with only some particular kinds of such errors at a time.

To classify concurrency errors, we can adopt classification of general programming errors and distinguish between safety errors and liveness errors like we have done in [19]. Safety errors violate safety properties of a program, i.e., they cause something bad to happen. They always have a finite witness leading to an error state, so, they may be seen as easier to detect. Liveness errors are errors which violate liveness properties of a program, i.e., prevent something good from happening. To detect them one needs to find an infinite path showing that the intended behaviour cannot be achieved, and thus, it may be more complicated to detect liveness errors.

In the following, however, we present the classes of concurrency errors rather with respect to the underlying mechanism that leads to a failure. It allows us later to address the practical aspects of error detection, focusing on some particular errors that violate program safety, in the same way.

Atomicity Violation. The first group of concurrency errors we address in this book chapter is related to wrong atomicity, i.e., some operations unintentionally interfere with the execution of some other operations. We can define it more formally as follows.

Definition 1 Atomicity Violation — *A program execution violates atomicity iff it is not equivalent to any other execution in which all code blocks which are assumed to be atomic are executed serially.*

We have already seen an atomicity violation in our example above where the operations of incrementing and adding a value to a shared variable were not executed atomically. This kind of concurrency errors is, however, usually treated as a special subclass of atomicity violation called data race or race condition.

Definition 2 Data Race — *A program execution contains a data race iff it contains two unsynchronised accesses to a shared variable and at least one of them is a write access.*

Data races are one of the most common and most undesirable phenomena in concurrent programs. However, one should note that not all data races are harmful. Data races that cannot cause application failures are often referred to as *benign data races* and are sometimes intentionally left in concurrent programs. As an example of a benign data race consider two unsynchronised threads, one thread updating a shared integer variable with the percentage of the completed work, and another thread reading this shared variable and drawing a progress bar. Even in the absence of synchronisation between the threads, this program will behave as expected and, thus, we are facing a benign data race.

Similar to data races with respect to the behaviour scheme, but rather different regarding their origins, are the so-called *high-level data races* [4]. For instance, consider a complex number whose real and imaginary values are protected by two separate locks. Updating such complex number can never cause a data race as presented in Definition 2, because the accesses to both parts of the complex number are always protected by the corresponding lock. Nevertheless, when the complex number is updated concurrently by two threads, the complex number may, after both the updates, contain the real part from one of the updates and the imaginary part from the other. This inconsistency did not occur directly on the shared variables (the complex number real and imaginary parts) but on the complex number itself as a whole, which is at the higher level of abstraction and, therefore, it is called a high-level data race.

Deadlock. Deadlocks are another kind of safety concurrency errors. Actually, one may see them as a consequence of tackling atomicity violations—to avoid, for instance, data races, one should use locks to guard accesses to shared resources, e.g., shared variables. Using locks in a wrong way may, however, cause a deadlock, which is definitely undesirable because the application stops working.

Despite deadlocks being quite often studied in the literature, the understanding of deadlocks still varies, depending on the specific setting being considered. Here we stick to the meaning common in the classical literature on operating systems. To define deadlocks in a general way, we assume that given any state of a program: (1) one can identify threads that are blocked and waiting for some event to happen; and (2) for any waiting thread t, one can identify threads that could generate an event that would unblock t.

Definition 3 Deadlock — *A program state contains a set S of deadlocked threads iff each thread in S is blocked and waiting for some event that could unblock it, but such an event could only be generated by a thread also in S.*

Most works consider a special case of deadlocks, namely, the so-called *Coffman deadlock* [10]. A Coffman deadlock happens in a state in which four conditions are met: (1) Processes have an exclusive access to the resources granted to them, (2) processes hold some resources and are waiting for additional resources, (3) resources cannot be forcibly removed from the tasks holding them (no preemption on the resources), and (4) a circular chain of tasks exists in which each task holds one or more resources that are being requested by the next task in the

chain. Such a definition perfectly fits deadlocks caused by blocking lock operations but does not cover deadlocks caused by message passing (e.g., a thread t_1 can wait for a message that could only be sent by a thread t_2, but t_2 is waiting for a message that could only be sent by t_1).

Order Violation. Order violations form a much less studied class of concurrency errors than atomicity violations and deadlocks. An order violation is a problem of a missing enforcement of some higher-level ordering requirements.[2] An order violation can be defined as follows.

Definition 4 Order Violation — *A program execution exhibits an order violation if some of its instructions are not executed in the expected order.*

Missed Signal. Missed signals are another less studied class of concurrency errors. The notion of missed signals assumes that it is known which signal is *intended* to be delivered to which thread or threads. A missed signal error can be defined as follows.

Definition 5 Missed Signal — *A program execution contains a missed signal iff there is a signal sent that is not delivered to the thread or threads for which it was intended.*

Since signals are often used to unblock waiting threads, a missed signal error typically leads to a thread or threads being blocked forever and can lead to a deadlock as well.

Starvation. Starvation is a behaviour that can cover several safety as well as liveness (or mixed[3]) errors, such as the already discussed deadlocks and missed signals, and the to be discussed livelocks and blocked threads. Starvation occurs when a thread is waiting for an event that never happens. If the probability of the event is very low but will eventually happen, the thread is not exhibiting a starvation, but in these cases the performance degradation imposed by the waiting for the event may become unacceptable and render the solution invalid as would a starvation situation.

Definition 6 Starvation — *A program execution exhibits starvation iff there exists a thread which waits (blocked or continually performing some computation) for an event that needs not to occur.*

[2] Some atomicity violations can be, actually, seen as a low-level violations of ordering expectations and deadlocks, in addition, are often caused by a wrong order of locking operations. Here, we do not consider atomicity violations and deadlocks as order violations.

[3] Mixed errors are errors that have both finite witnesses as well as infinite ones whose any finite prefix does not suffice as a witness.

Livelock and Non-Progress Behaviour. There are various different definitions of a livelock in the literature. Often, the works consider some kind of a *progress* notion for expressing that a thread is making some useful work, i.e., doing something of what the programmer intended to be done. Then they see a livelock as a problem when a thread is not blocked but is not making any progress as well.

Definition 7 Livelock and Non-Progress Behaviour — *An infinite program execution exhibits a non-progress behaviour iff there is a thread which is continually performing some computation, i.e., it is not blocked, but it is not making any progress either.*

Blocked Thread. We speak about a *blocked thread* appearing within some execution when a thread is blocked and waiting forever for some event which can unblock it. Like for a deadlock, one must be able to say what the blocking and unblocking operations are. The problem can then be defined as follows.

Definition 8 Blocked Thread — *A program execution contains a blocked thread iff the thread is waiting for some event to continue, and this event never occurs in the execution.*

The absence of some unblocking event may leave a thread blocked forever. There may be many reasons for leaving a thread blocked. A common reason is that a thread that was holding a lock ends unexpectedly, leaving another thread (or threads) waiting forever for that lock to be released. Another common reason are missed signals. Blocked threads may also be called *orphaned threads* [18].

4 Monitoring of Concurrent Program Execution

One of the strategies to find errors in concurrent programs makes use of dynamic program analysis that, in turn, requires to observe and monitor the execution of the program under analysis. To monitor the program execution, one needs to inject additional code into selected locations of the original program, which when executed will generate observation points for program analysis. There are several levels at which such additional code can be inserted, including the source code level, the level of the intermediate code, and the binary level.

From the three approaches above, inserting the monitoring code at the binary level has a big advantage of not requiring the source code of the program under analysis. This is particularly important when dealing with proprietary or legacy libraries whose source files are not available even for the developers of the program under analysis. Another advantage might be that this kind of instrumentation is more precise in that the monitoring code can be inserted exactly where necessary, and the placement is not affected by any optimisations possibly made by the compiler. Yet another advantage is getting access to some low-level information, such as register allocation, which might be important for some

analyses. All these advantages come at the expense of sometimes loosing access to various pieces of high-level information about the program (organisation of complex data objects, names of variables, etc.).

There exist several frameworks for binary instrumentation, which can be used to insert the execution-monitoring code into a program. They might be divided into two groups: static and dynamic binary instrumentation frameworks. *Static binary instrumentation frameworks*, e.g., PEBIL [44], insert monitoring code to a program by rewriting the object or executable code of the program before it is executed, thus generating a new modified version of the original program's binary file, which will be executed afterwards. *Dynamic binary instrumentation frameworks*, e.g., PIN [49] and Valgrind [55], insert execution-monitoring code to the program image in memory at runtime, leaving the program's binary file untouched.

An advantage of static binary instrumentation is that it does not suffer from the overhead of instrumenting the code of a program every time it is executed. On the other hand, it cannot handle constructions such as self-modifying and self-generating code, which is not known before the program actually executes. On the contrary, dynamic binary instrumentation is slower, but it can cover all the code that is executed by a program. Furthermore, since the binary file of the program is not modified in any way, the instrumentation is more transparent to the user who can run some (possibly lengthy) analysis on the program and, at the same time, use the program as usual. As the dynamic binary instrumentation changes the in-memory image of the program, it also allows to instrument and monitor shared libraries without requiring to generate and maintain two versions of the library, one normal and the other instrumented.

Regardless of which type of the instrumentation approaches is used, there are some issues that need to be dealt with when analysing multi-threaded programs at the binary level [21]. One of these problems is the monitoring of function calls/execution. This is because the monitoring code has to handle cases where the control is passed among several functions by jumps, and the return is from a different function than the one that was called. Another problem is that the monitoring code must properly trigger notifications for various special types of instructions such as atomic instructions, which access several memory locations at once but in an atomic way, and conditional and repeatable instructions, which might be executed more than once or not at all. Further, some pieces of information about the execution of instructions and functions (such as the memory locations accessed by them), which are crucial for various analyses, may be lost once the instruction or function finishes its execution, and it is necessary to explicitly preserve this information for later use. Finally, in order to support various multithreading libraries, the analysers must be abstracted from the particular library used.

Inserting additional code needed for monitoring at the intermediate code level is suitable for programming languages that use intermediate code like Java. It does not require the source code of the application while it stays at a level of abstraction that retains more information about the original program, making it

easier to explain the errors found than if the monitoring code is inserted at the binary level. One may find useful tools, like RoadRunner [27], which provide an instrumentation facility and allows one to fully concentrate on the development of the analyser.

When the source code of the application is available, one may insert monitoring code directly into the application source code. Preparation of each such application for analysis, however, requires some manual work even when aspect-oriented programming is employed.

5 Detection of Atomicity Errors

In this section, we describe possible ways for detecting atomicity violations. We start with the detection of data races as a special and very important case of atomicity violations, and then we follow with the detection of general atomicity violations, detecting first single and then multiple variable atomicity violations.

5.1 Detection of Data Races

To recall Definition 2, a data race occurs if there are two unsynchronised accesses to a shared variable within the execution of a concurrent program and at least one of them is a write access. To be able to identify an occurrence of a data race, one thus needs to detect (1) which variables are shared by any two given threads, and (2) whether all pairs of accesses to a given shared variable are synchronised.

As data races are a well-studied concurrency problem, many different techniques have been proposed to tackle their detection. Dynamic techniques that analyse one particular execution of a program are usually based on computing the so-called locksets and/or happens-before relations along the witnessed execution. Static techniques often either look for particular code patterns that are likely to cause a data race, or compute locksets and/or happens-before relations over all executions considered feasible by a static analyser [16, 39, 50, 54]. It is also possible to use type systems to detect and/or avoid data races by design [23, 25, 73]. One may also consider model checking approach [67]. However, we discuss dynamic techniques and their principles in the remainder of this section.

Lockset-based algorithms. The techniques based on *locksets* build on the idea that all accesses to a shared variable should be guarded by a lock. A lockset is defined as a set of locks that guard all accesses to a given variable. Detectors then use the assumption that, if the lockset associated with a certain shared variable is non-empty, i.e., all accesses to that variable are protected by at least one lock, then no data race is possible.

The first algorithm which used the idea of locksets was Eraser [64]. The algorithm maintains for each shared variable v a set $C(v)$ of candidate locks for v. When a new variable is initialised, its candidate set $C(v)$ contains all possible locks. Whenever a variable v is accessed, Eraser updates $C(v)$ by intersecting

$C(v)$ and the set $L(t)$ of locks held by the current thread at the moment. Eraser warns about a possible data race if $C(v)$ becomes empty for some shared variable v along the execution being analysed. In order to reduce the number of false alarms, Eraser introduces an internal state $s(v)$ used to identify the access pattern for each shared variable v: if the variable is used exclusively by one thread, if it is written by a single thread and read by multiple threads, or if it can be changed by multiple threads. The lockset $C(v)$ is then modified only if the variable is shared, and a data race is reported only if $C(v)$ becomes empty and $s(v)$ is in the state denoting the case where multiple threads can access v for writing.

The original Eraser algorithm designed for C programs has been modified for programs written in object-oriented languages, e.g., [7, 8, 63, 76]. The main modification (usually called as the *ownership model*) is inspired by the common idiom used in object-oriented programs where the creator of an object is actually not the owner of the object. Then, one should take into account that the creator always accesses the object first and no explicit synchronisation with the owner is needed, because the synchronisation is implicitly taken care of by the Java virtual machine. This idea is reflected by adding a new internal state for the shared variables. The modification introduces a small possibility of having false negatives [41, 63], but greatly reduces the number of false alarms caused by the object-oriented programming idiom.

Locksets-based techniques do not support other synchronisation mechanisms than locks and thus, if other mechanisms are also used, these techniques may produce too many false alarms.

Happens-before-based algorithms. The happens-before-based techniques exploit the so-called *happens-before relation* [43] (denoted $\rightarrow$), which is defined as the least strict partial order that includes every pair of causally ordered events. For instance, if an event x occurs before an event y in the same thread, then it is denoted as $x \rightarrow y$. Also, if x is an event creating some thread and y is an event in that thread, then $x \rightarrow y$. Similarly, if some synchronisation or communication means is used that requires an event x to precede an event y, then $x \rightarrow y$. All notions of synchronisation and communication, such as sending and receiving a message, unlocking and locking a lock, sending and receiving a notification, etc., are to be considered. Detectors build (or approximate) the happens-before relation among accesses to shared variables and check that no two accesses (out of which at least one is for writing) can happen simultaneously, i.e., without a happens-before relation between them.

Most happens-before-based algorithms use the so-called *vector clocks* introduced in [51]. The idea of vector clocks for a message passing system is as follows. Each thread t has a vector of clocks T_{vc} indexed by thread identifiers. One position in T_{vc} holds the value of the clock of t. The other entries in T_{vc} hold logical timestamps indicating the last event in a remote thread that is known to be in the happens-before relation with the current operation of t. Vector clocks are partially-ordered in a point-wise manner ($\sqsubseteq$) with an associated join operation

($\sqcup$) and the minimal element (0). The vector clocks of threads are managed as follows: (1) initially, all clocks are set to 0; (2) each time a thread t sends a message, it sends also its T_{vc} and then t increments its own logical clock in its T_{vc} by one; (3) each time a thread receives a message, it increments its own logical clock by one and further updates its T_{vc} according to the received vector T'_{vc} to $T_{vc} = T_{vc} \sqcup T'_{vc}$.

Algorithms [61, 62] detect data races in systems with locks via maintaining a vector clock C_t for each thread t (corresponding to T_{vc} in the original terminology above), a vector clock L_m for each lock m, and two vector clocks for write and read operations for each shared variable x (denoted W_x and R_x, respectively). W_x and R_x simply maintain a copy of the C_t of the last thread that accessed x for writing or reading, respectively. A read from x by a thread is race-free if $W_x \sqsubseteq C_t$ (it happens after the last write of each thread). A write to x by a thread is race-free if $W_x \sqsubseteq C_t$ and $R_x \sqsubseteq C_t$ (it happens after all accesses to the variable).

Maintaining such a big number of vector clocks as above generates a considerable overhead. Therefore, in [26], the vector clocks from above that were associated with variables were mostly replaced by the so-called *epochs*. The epoch of a variable v is represented as a tuple $(t, c)_v$, where t identifies the thread that last accessed v and c represents the value of its clock. The idea behind this optimisation is that, in most cases, a data race occurs between two subsequent accesses to a variable. In such cases, epochs are sufficient to detect unsynchronised accesses. However, in cases where a write operation needs to be synchronised with multiple preceding read operations, epochs are not sufficient and the algorithm has to build an analogy of vector clocks for sequences of read operations.

A bit different detection approach has been introduced in TRaDe [9] where a *topological race detection* [31] is used. This technique is based on an exact identification of objects which are reachable from a thread. This is accomplished by observing manipulations with references which alter the interconnection graph of the objects used in a program—hence the name topological. Then, vector clocks are used to identify possibly concurrently executed segments of code, called *parallel segments*. If an object is reachable from two parallel segments, a race has been detected. A disadvantage of this solution is its considerable overhead.

Although the algorithms mentioned above exhibit good precision, their computational demands are sometimes prohibitive, which inspired researchers to come up with some combinations of happens-before-based and lockset-based algorithms. These combinations are often called *hybrid algorithms*.

Hybrid algorithms. Hybrid algorithms such as [15, 24, 58, 76] combine the two approaches described above.

In RaceTrack [76], the notion of a *threadset* was introduced. The threadset is maintained for each shared variable and contains information concerning the threads currently working with the variable. The method works as follows. Each time a thread performs a memory access on a variable, it forms a label consist-

ing of the thread identifier and its current private clock value. The label is then added to the threadset of the variable. The thread also uses its vector clock to identify and remove from the threadset the labels that correspond to accesses that are ordered before the current access. Hence, the threadset contains solely the labels for accesses that are concurrent. At the same time, locksets are used to track locking of variables, which is not tracked by the used approximation of the happens-before relation. Intersections on locksets are applied if the approximated happens-before relation is not able to assure an ordered access to shared variables. If an ordered access to a shared variable is assured by the approximated happens-before relation, the lockset of the variable is reset to the lockset of the thread that is currently accessing it.

One of the most advanced lockset-based algorithms that also uses the happens-before relation is Goldilocks [15]. The main insight of this algorithm is that locksets can contain not only locks but also volatile variables (i.e., variables with atomic access that may also be used for synchronisation) and, most importantly, also threads. The appearance of a thread t in the lockset of a shared variable v means that t is properly synchronised for using the given variable. The information about threads synchronised for using certain variables is then used to maintain a transitive closure of the happens-before relation via the locksets. The advantage of Goldilocks is that it allows locksets to grow during the computation when the happens-before relation is established between operations over v. The basic Goldilocks algorithm is relatively expensive but can be optimised by *short circuiting the lockset computation* (three cheap checks are sufficient for ensuring race freedom between the last two accesses on a variable) and using *a lazy computation of the locksets* (locksets are computed only if the previous optimisation is not able to detect that some events are in the happens-before relation). The optimised algorithm has a considerably lower overhead, in some cases approaching the pure lockset-based algorithms.

A similar approach to Goldilocks but for the Java Path Finder model checker has been presented in [40]. This algorithm does not map variables to locksets containing threads and synchronisation elements (such as locks), but rather threads and synchronisation elements to sets of variables. This modification is motivated by the fact that the number of threads and locks is usually much lower than the number of shared variables. Such a modification is feasible because model checking allows the method to modify structures associated with different threads at once. Methods based on dynamic analysis cannot use this modification and locksets must be kept using the original relation.

5.2 Detection of Atomicity Violations

Taking into account the generic notion of atomicity, methods for detecting atomicity violations can be classified according to: (1) the way they obtain information about which code blocks should be expected to execute atomically; (2) the notion of equivalence of executions used (we will get to several commonly used equivalences in the following); and (3) the actual way in which an atomicity violation is detected (i.e., using static analysis, dynamic analysis, etc.).

As for the blocks to be assumed to execute atomically, some authors expect the programmers to annotate their code to delimit such code blocks [29]. Some other works come with predefined patterns of code which should typically execute atomically [32, 48, 66]. Yet other authors try to infer blocks to be executed atomically, e.g., by analysing data and control dependencies between program statements [72], where dependent program statements form a block which should be executed atomically, or by finding access correlations between shared variables [47], where a set of accesses to correlated shared variables should be executed atomically (together with all statements between them).

Below, we first discuss approaches for detecting atomicity violations when considering accesses to a single shared variable only and then those which consider accesses to several shared variables.

Atomicity over one variable. Most of the existing algorithms for detecting atomicity violations are only able to detect atomicity violations within accesses to a single shared variable. They mostly attempt to detect situations where two accesses to a shared variable should be executed atomically, but are interleaved by an access from another thread.

In [72], blocks of instructions which are assumed to execute atomically are approximated by the so-called *computational units* (CUs). CUs are inferred automatically from a single program trace by analysing data and control dependencies between instructions. First, a dependency graph is created which contains control and read-after-write dependencies between all instructions. Then, the algorithm partitions this dependency graph to obtain a set of distinct subgraphs which form the CUs. The partitioning works in such a way that each CU is the largest group of instructions where all instructions are control or read-after-write dependent, but no instructions which access shared variables are read-after-write dependent, i.e., no read-after-write dependencies are allowed between shared variables in the same computational unit. Since these conditions are not sufficient to partition the dependency graph to distinct subgraphs, additional heuristics are used. Atomicity violations are then detected by checking if the strict 2-phase locking (2PL) discipline [17] is violated in a program trace. Violating the strict 2PL discipline means that some CU has written or accessed a shared variable which another CU is currently reading from or writing to, respectively (i.e., some CU accessed a shared variable and before its execution is finished, another CU accesses this shared variable). If the strict 2PL discipline is violated, the program trace is not identical to any serial execution, and so seen as violating atomicity. Checking if the strict 2PL discipline is violated is done dynamically during a program execution in case of the online version of the algorithm, or on a previously recorded execution trace using the off-line version of the algorithm.

A much simpler approach of discovering atomicity violations was presented in [48]. Here, any two consecutive accesses from one thread to the same shared variable are considered an atomic section, i.e., a block which should be executed atomically. Such blocks can be categorised into four classes according to the types of the two accesses (read or write) to the shared variable. Serialisability is

then defined based on analysis of what can happen when a block b of each of the possible classes is interleaved with some read or write access from another thread to the same shared variable which is accessed in b. Out of the eight total cases arising in this way, four (namely, r-w-r, w-w-r, w-r-w, r-w-w) are considered to lead to an unserialisable execution. However, the detection algorithm does not consider all the unserialisable executions as errors. Detection of atomicity violations is done dynamically in two steps. First, the algorithm analyses a set of correct (training) runs in which it tries to detect atomic sections which are never unserialisably interleaved. These atomic sections are called *access interleaving invariants* (AI invariants). Then, the algorithm checks if any of the obtained AI invariants is violated in a monitored run, i.e., if there is an AI invariant which is unserialisably interleaved by an access from another thread to a shared variable which the AI invariant (atomic section) accesses. While the second step of checking AI invariants violation is really simple and can be done in a quite efficient way, the training step to get the AI invariants can lead to a considerable slowdown of the monitored application and has to be repeated if the code base of the application changes (e.g., for a new version of the application).

A more complicated approach was introduced in [24, 69], where atomicity violations are sought using the Lipton's reduction theorem [46]. The approach is based on checking whether a given run can be transformed (reduced) to a serial one using commutativity of certain instructions (or, in other words, by moving certain instructions back or forward in the execution timeline). Both [24] and [69] use procedures as atomic blocks by default, but users can annotate blocks of code which they assume to execute atomically to provide a more precise specification of atomic sections for the algorithm. For the reduction used to detect atomicity violations, all instructions are classified, according to their commutativity properties, into 4 groups: (1) *Left-mover* instructions L that may be swapped with the immediately preceding instruction; (2) *Right-mover* instructions R that may be swapped with the immediately succeeding instruction; (3) *Both-mover* instructions B that are simultaneously left and right mover, i.e., they may be swapped with both the immediately preceding and succeeding instructions; and (4) *Non-mover* instructions N that are not known to be left or right mover instructions.

Classification of instructions to these classes is based on their relation to synchronisation operations, e.g., lock acquire instructions are right-movers, lock release instructions are left-movers, and race free accesses to variables are both-movers (a lockset-based dynamic detection algorithm is used for checking race freeness). An execution is then serialisable if it is deadlock-free and each atomic section in this execution can be reduced to a form $R^*N^?L^*$ by moving the instructions in the execution in the allowed directions. Here, $N^?$ represents a single non-mover instruction and both-mover instructions B can be taken as either right-mover instructions R or left-mover instructions L. Algorithms in both [24] and [69] use dynamic analysis to detect atomicity violations using the reduction algorithm described above.

Other approaches using the Lipton's reduction theorem [46] can be found in [28, 68] where type systems based on this theorem are used to deal with atomicity violations.

Atomicity over multiple variables. The above mentioned algorithms consider atomicity of multiple accesses to the same variable only. However, there are situations where we need to check atomicity over multiple variables, e.g., when a program modifies two or more interrelated variables in several atomic blocks (such variables can represent, for instance, a point in a three-dimensional space or the real and imaginary parts of a complex number). Even if we ensure that all the read and write accesses to these variables are executed atomically, the program can still have an unserializable execution. This happens when the boundaries of the atomic block guarding the access to these variables are misdefined, and what should be a single atomic block was split into two or more smaller atomic blocks. The interleaving of these smaller atomic blocks with other atomic blocks may violate the integrity of the data and expose states that would never be observed in a sequential execution. Nevertheless, the algorithms and the detectors discussed above cannot address these multiple-variable atomicity errors.

In [4], the problem of violation of atomicity of operations over multiple variables is referred to as a *high-level data race*. In the work, all synchronised blocks (i.e., blocks of code guarded by the `synchronized` statement) are considered to form atomic sections. The proposed detection of atomicity violations is based on checking the so-called *view consistency*. For each thread, a set of views is generated. A view is a set of fields (variables) which are accessed by a thread within a single synchronised block. From this set of views, a set of maximal views (maximal according to set inclusion) is computed for each thread. An execution is then serialisable if each thread only uses views that are compatible, i.e., form a chain according to set inclusion, with all maximal views of other threads. Hence, the detection algorithm uses a dynamic analysis to check whether all views are compatible within a given program trace. Since the algorithm has to operate over a high number of sets (each view is a set), it suffers from a big overhead. Dias et al in [13] adapted this approach to apply static analysis techniques and extended it to reduce the number of false warnings.

A different approach is associated with the Velodrome detector [29]. Here, atomic sections (called transactions) are given as methods annotated by the user. Detection of atomicity violations is based on constructing a graph of the *transactional happens-before relation* (the happens-before relation among transactions). An execution is serialisable if the graph does not contain a cycle. The detection algorithm uses a dynamic analysis to create the graph from a program trace and then checks it for a cycle. If a cycle is found, the program contains an atomicity violation. Since creating the graph for the entire execution is inconvenient, nodes that cannot be involved in a cycle are garbage-collected or not created at all. Like the previous algorithm, Velodrome may suffer from a considerable overhead in some cases, too.

The simple idea of *AI invariants* described in [48] has been generalised for checking atomicity over pairs of variables in [32, 66], where a number of problematic interleaving scenarios were identified. The user is assumed to provide the so-called *atomic sets* that are sets of variables which should be operated atomically. In [66] an algorithm to infer which procedure bodies should be the so-called *units of work* w.r.t. the given atomic sets is proposed. This is done statically using dataflow analysis. An execution is then considered serialisable if it does not correspond to any of the problematic interleavings of the detected units of work. An algorithm capable of checking unserialisability of execution of units of work (called atomic-set-serialisability violations) is described in [32]. It is based on a dynamic analysis of program traces. The algorithm introduces the so-called *race automata*, which are simple finite state automata used to detect the problematic interleaving scenarios.

There are also attempts to enhance well-known approaches for data race analysis to detect atomicity violations over multiple variables. One method can be found in [47], where data mining techniques are used to determine *access correlations* among an arbitrary number of variables. This information is then used in modified lockset-based and happens-before-based detectors. Since data race detectors do not directly work with the notion of atomicity, blocks of code accessing correlated variables are used to play the role of atomic sections. Access correlations are inferred statically using a correlation analysis. The correlation analysis is based on mining association rules [3] from frequent itemsets, where items in these sets are accesses to variables. The obtained association rules are then pruned to allow only the rules satisfying the minimal support and minimal confidence constraints. The resulting rules determine access correlations between various variables. Using this information, the two mentioned data race detector types can then be modified to detect atomicity violations over multiple variables as follows. Lockset-based algorithms must check if, for every pair of accesses to a shared variable, the shared variable and all variables correlated with this variable are protected by at least one common lock. Happens-before-based algorithms must compare the logical timestamps not only with accesses to the same variable, but also with accesses to the correlated variables. The detection can be done statically or dynamically, depending on the data race detector used.

6 Detection of Deadlocks

As deadlock is connected with circular dependency among threads and shared resources, the detection of deadlocks usually involves various graph algorithms. For instance, the algorithm introduced in [57] constructs a *thread-wait-for graph* dynamically and analyses it for a presence of cycles. Here, a thread-wait-for graph is an arc-labelled digraph $G = (V, E)$ where vertices V are threads and locks, and edges E represent waiting arcs, which are classified (labelled) according to the synchronisation mechanism used (join synchronisation, notification, finalisation, and waiting on a monitor). A cycle in this graph involving at least two threads represents a deadlock. A disadvantage of this algorithm is that it is able to

detect only deadlocks that actually happen. The following works can detect also potential deadlocks that can happen but did not actually happen during the witnessed execution.

In [33], a different algorithm called GoodLock for detecting deadlocks was presented. The algorithm constructs the so-called *runtime lock trees* and uses a depth-first search to detect cycles in it. Here, a runtime lock tree $T_t = (V, E)$ for a thread t is a tree where vertices V are locks acquired by t and there is an edge from $v_1 \in V$ to $v_2 \in V$ when v_1 represents the most recently acquired lock that t holds when acquiring v_2. A path in such a tree represents a nested use of locks. When a program terminates, the algorithm analyses lock trees for each pair of threads. The algorithm issues a warning about a possible deadlock if the order of obtaining the same locks (i.e., their nesting) in two analysed trees differs and no "gate" lock guarding this inconsistency has been detected.

The original GoodLock algorithm is able to detect deadlocks between two threads only. Later works, e.g., [2,6] improve the algorithm to detect deadlocks among multiple threads. In [2], a support for semaphores and wait-notify synchronisation was added. A stack to handle the so-called *lock dependency relation* is used in [38] instead of lock trees. The algorithm computes the transitive closure of the lock dependency relation instead of performing a depth first search in a graph. The modified algorithm uses more memory but the computation is much faster.

Static approaches can be employed also for deadlock detection. A purely data-flow-based interprocedural static detector of deadlocks called RacerX has been presented in [16] while a bottom-up data-flow static analysis is used to detect deadlocks in [70]. Both algorithms produce many false alarms due to the approximations they use. A combination of symbolic execution, static analysis, and SMT solving is used in [11] to automatically derive the so-called *method contracts* guaranteeing deadlock-free executions.

7 Detection of Other Errors in Concurrency

So far, we have covered detection of data races, atomicity violations, and deadlocks that are the most common concurrency errors in practice. In this section, we briefly touch detection of other concurrency errors, such as order violations, missed signals, and non-progress behaviour.

For detecting order violations one needs to be able to decide if, for a given execution, the instructions were or were not executed in the right order. There are only a few detection techniques which are able to detect order violations. These techniques try to detect that some instructions are executed in a wrong order by searching for specific behavioural patterns [77] or by comparing the order of instructions in a testing run with the order witnessed in a series of preceding, correct program runs [74].

Similarly to order violations, there are just a few methods for detecting missed signals. Usually, the problem is studied as part of detecting other concurrency

problems, e.g., deadlocks. There is also an approach that uses pattern-based static analysis to search for code patterns that may lead to missed signals [37].

It is also possible to use *contracts for concurrency* [65] to detect some of the errors mentioned above. A contract for concurrency allows one to enumerate sequences of public methods of a module that are required to be executed atomically. Even though such contracts were designed to primarily capture atomicity violations, they are capable of capturing order violations and missed signals as well. Contracts may be written by a developer or inferred automatically from the program (based on its typical usage patterns) [65].

There are two methods [12, 20] for dynamically verifying that such contracts are respected at program runtime. In particular, the first method [20] belongs among the so-called *lockset-based* dynamic analyses, whose classic example is the Eraser algorithm for data race detection [64]. Their common feature is that they track sets of locks that are held by various threads and used for various synchronisation purposes. The tracked lock sets are used to extrapolate the synchronisation behaviour seen in the witnessed test runs, allowing one to warn about possible errors even when they do not directly appear in the witnessed test runs.

While the lockset-based method works well in many cases, it may produce both false positives and negatives. Some of these problems are caused by the method itself as lockset-based methods are imprecise in general. However, many of the problems are caused by the limitations of the (basic) contracts which do not allow one to precisely describe which situations are errors and which are not. To address this problem, the notion of contracts for concurrency was extended in [12] to allow them to reflect both the *data flow* between the methods (in that a sequence of method calls only needs to be atomic if they manipulate the same data) and the *contextual information* (in that a sequence of method calls needs not be atomic wrt all other sequences of methods but only some of them). The paper then proposes a method for dynamic validation of contracts based on the *happens-before relation* which utilises *vector clocks* in a way optimised for contract validation. This method does not suffer from false alarms and supports the extended contracts.

One of the most common approaches for detecting non-progress behaviour in finite-state programs is to use model checking and search for non-progress cycles [35]. In case of infinite-state programs, a long enough path of non-progress actions in the state space is often sufficient for proving a non-progress behaviour [30]. A similar approach is also used in dynamic techniques where *dynamic monitoring* [34] of an execution is performed in order to find an execution where no progress action is reached for a long period of time.

8 Boosting Detection of Errors in Concurrency

In this section, we discuss techniques that can help with quality assurance of concurrent programs regardless of particular class of concurrency errors, namely, noise injection and systematic testing.

Noise injection. Noise injection inserts delays into the execution of selected threads aiming at promoting the occurrence of low-probable interleavings that otherwise would happen very seldom or even never. This approach allows to test time-sensitive synchronisation interleavings that could hide latent errors. Noise injection is also able to test legal interleavings of actions which are far away from each other in terms of execution time and in terms of the number of concurrency-relevant events [14] between those actions during average executions provided that the appropriate noise is injected into some of the threads. In a sense, the approach is similar to running the program inside a model checker such as JPF [67] with a random exploration algorithm enabled. However, making purely random scheduling decisions may be less efficient than using some of the noise heuristics which influence the scheduling at some carefully selected places important from the point of view of synchronisation only. The approach of noise injection is mature enough to be used for testing of real-life software, and it is supported by industrial-strength tools, such as IBM Java Concurrency Testing Tool (ConTest) [14] and the Microsoft Driver Verifier, where the technique is called delay fuzzing [1]. A recent tool supporting noise-based testing of concurrent C/C++ code on the binary level is ANaConDA [12, 22].

Systematic testing. Systematic testing of concurrent programs [36, 52, 53, 71] has become popular recently. The technique uses a deterministic control over the scheduling of threads. A deterministic scheduler is sometimes implemented using intense noise injection keeping all threads blocked except the one chosen for making a progress. Often, other threads which do not execute synchronisation-relevant instructions or which do not access shared memory are also allowed to make progress concurrently.

The systematic testing approach can be seen as execution-based model checking which systematically tests as many thread interleaving scenarios as possible. Before execution of each instruction which is considered as relevant from the point of view of detecting concurrency-related errors, the technique computes all possible scheduler decisions. The concrete set of instructions considered as concurrency-relevant depends on the particular implementation of the technique (often, shared memory accesses and synchronisation relevant instructions are considered as concurrency-relevant). Each such a decision point is considered a state in the state space of the system under test, and each possible decision is considered an enabled transition at that state. The decisions that are explored from each state are recorded in the form of a partially ordered happens-before graph [52], totally ordered list of synchronisation events [71], or simply in the form of a set of explored decisions [36]. During the next execution of the program, the recorded scheduling decisions can be enforced again when doing a replay or changed when testing with the aim of enforcing a new interleaving scenario.

As the number of possible scheduling decisions is high for complex programs, various optimisations and heuristics reducing the number of decisions to explore have been proposed. For example, the *locality hypothesis* [52] says that most concurrency-related errors can be exposed using a small number of preemptions. This hypothesis is exploited in the CHESS tool [52], which limits the

number of context switches taking place in the execution (iteratively increasing the bound on the allowed number of context switches). Moreover, the tool also utilises a partial-order reduction algorithm blocking exploration of states equal to the already explored states (based on an equivalence defined on happens-before graphs). Some further heuristics are then mentioned below when discussing the related approach of coverage-driven testing.

However, despite a great impact of the various reductions, the number of thread interleavings to be explored remains huge for real-life programs, and therefore the approach provides great benefit mainly in the area of unit testing [36, 52]. The systematic testing approach is not as expensive as full model checking, but it is still quite costly because one needs to track which scheduling scenarios of possibly very long runs have been witnessed and systematically force new ones. The approach makes it easy to replay an execution where an error was detected, but it has problems with handling various external sources of nondeterminism (e.g., input events).

Systematic testing offers several important benefits over noise injection. Its full control over the scheduler allows systematic testing to precisely navigate the execution of the program under test, to explore different interleavings in each run, and to also replay interesting runs (if other sources of nondeterminism, such as input values, are handled). It allows the user to get information about what fraction of (discovered) scheduling decisions has already been covered by the testing process. On the other hand, the approach suffers from various problems, such as handling external sources of nondeterminism (user actions in GUI, client requests) as well as with continuously running programs where its ability to reuse already collected information is limited. In all those problematic cases, noise injection can be successfully used. Moreover, the performance degradation introduced by noise injection is significantly lower.

Coverage-driven testing. An approach related to systematic testing is the approach of *coverage-driven testing* implemented in the Maple tool [75]. Maple attempts to influence the scheduling such that the obtained coverage of several important synchronisation idioms (called iRoots) is maximised. These idioms capture several important memory access patterns that are shown to be often related with error occurrences. Maple uses several heuristics to likely increase the coverage of iRoots. The technique provides lower guarantees of finding an error than systematic testing, but it is more scalable. The Maple tool [75] limits the number of context switches to two and additionally gets use of the *value-independence hypothesis* which states that exposing a concurrency error does not depend on data values. Moreover, the Maple tool does not consider interleavings where two related actions executed in different threads are too far away from each other. The distance of such actions is computed by counting actions in one of the threads, and the threshold is referred to as a *vulnerability window* [75]. The approach of Maple does not support some kinds of errors (e.g., value-dependent errors or some forms of deadlocks). Multiple of the heuristics that Maple uses are based on randomisation. Maple can thus be viewed as being in between of systematic testing and noise-based testing.

9 Conclusions

In this chapter, we have explained problems connected with concurrent execution of programs on modern multi-core devices, listed common errors that can arise from concurrency and described possibilities how particular kinds of concurrency errors can be detected. We have also mentioned noise injection and systematic testing that can support the discovery of concurrency errors.

Understanding the problems connected with concurrency is the first and inevitable step to produce high quality concurrent software. We, however, believe that detection of concurrency errors followed by their manual elimination by a programmer is not the only way to handle failures caused by concurrency. In contrast to many other kinds of software defects, concurrency errors have one admirable feature emerging from usually huge space of possible thread interleavings which provides us with redundancy that can be employed for automatic correction of the error. If we are able to automatically detect and localise the concurrency defect, we can propose a fix to a programmer or heal it fully automatically at runtime because the set of interleavings contains, besides interleavings leading to a failure, also interleavings that lead to a correct outcome. This is not possible for most of other programmers mistakes because the intended behaviour of the program cannot be inferred automatically.

For instance, if a data race over a shared variable is detected, it can be healed by introducing a new lock for guarding this shared variable. That means that a code acquiring the new lock is inserted before each access to the variable, while a code releasing the lock is inserted after the access. As a reader can guess, adding a lock without knowledge of other locking operations can introduce a deadlock which may be even worse problem than the original data race. Another approach of healing data races we have studied [41] exploits noise injection technique. As noise injection can be used for increasing probability of spotting a concurrency errors by changing probabilities of particular interleavings it can also be used in an opposite manner to significantly reduce probability of interleavings that lead to a fault. This approach cannot introduce a deadlock, however, it does not guarantee that the error is fully covered. Other types of concurrency problems such as deadlocks [56] and atomicity violations [45] can be covered automatically as well.

One may also expect that there will be available programming paradigms like transactional memory [5] in the future that reduce or eliminate chances of creating concurrency errors.

Acknowledgment. This work was partially supported by the ARVI EU COST ACTION IC1402. Further, the Czech authors were supported by the EU ECSEL project Aquas, the internal BUT FIT project FIT-S-17-4014, and the IT4IXS project: IT4Innovations Excellence in Science (LQ1602). The Portuguese author was also supported by the Portuguese Science Foundation and NOVA LINCS (ref. UID/CEC/04516/2013).

References

1. Power Framework Delay Fuzzing. Online at: http://msdn.microsoft.com/en-us/library/hh454184(v=vs.85).aspx (April 2013)
2. Agarwal, R., Stoller, S.D.: Run-time Detection of Potential Deadlocks for Programs with Locks, Semaphores, and Condition Variables. In: Proc. of PADTAD'06. pp. 51–60. ACM, New York, NY, USA (2006)
3. Agrawal, R., Imieliński, T., Swami, A.: Mining Association Rules Between Sets of Items in Large Databases. In: SIGMOD'93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data. pp. 207–216. ACM, New York, NY, USA (1993)
4. Artho, C., Havelund, K., Biere, A.: High-Level Data Races. In: VVEIS'03: The First International Workshop on Verification and Validation of Enterprise Information Systems. Angers, France (2003)
5. Ayguade, E., Cristal, A., Unsal, O.S., Gagliardi, F., Smith, B., Valero, M., Harris, T.: Transactional memory: An overview. IEEE Micro 27, 8–29 (2007)
6. Bensalem, S., Havelund, K.: Dynamic Deadlock Analysis of Multi-threaded Programs. In: Proc. of HVC'05. pp. 208–223. volume 3875 of LNCS, Springer-Verlag, Berlin, Heidelberg (2006)
7. Bodden, E., Havelund, K.: Racer: Effective Race Detection Using Aspectj. In: ISSTA'08: Proceedings of the 2008 international symposium on Software testing and analysis. pp. 155–166. ACM, New York, NY, USA (2008)
8. Choi, J.D., Lee, K., Loginov, A., O'Callahan, R., Sarkar, V., Sridharan, M.: Efficient and Precise Datarace Detection for Multithreaded Object-oriented Programs. In: PLDI'02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation. pp. 258–269. ACM, New York, NY, USA (2002)
9. Christiaens, M., Bosschere, K.D.: TRaDe: Data Race Detection for Java. In: ICCS'01: Proceedings of the International Conference on Computational Science-Part II. pp. 761–770. Springer-Verlag, London, UK (2001)
10. Coffman, E.G., Elphick, M., Shoshani, A.: System Deadlocks. ACM Comput. Surv. 3, 67–78 (June 1971)
11. Deshmukh, J., Emerson, E.A., Sankaranarayanan, S.: Symbolic Deadlock Analysis in Concurrent Libraries and Their Clients. In: ASE'09: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering. pp. 480–491. IEEE, Washington, DC, USA (2009)
12. Dias, R.F., Ferreira, C., Fiedor, J., Lourenço, J.M., Smrčka, A., Sousa, D.G., Vojnar, T.: Verifying concurrent programs using contracts. In: Proc. of ICST'17. IEEE Computer Society, Washington, DC, USA (2017)
13. Dias, R.J., Pessanha, V., Lourenço, J.M.: Precise detection of atomicity violations. In: Biere, A., Nahir, A., Vos, T. (eds.) Hardware and Software: Verification and Testing, Lecture Notes in Computer Science, vol. 7857, pp. 8–23. Springer Berlin / Heidelberg (Nov 2013)
14. Edelstein, O., Farchi, E., Goldin, E., Nir, Y., Ratsaby, G., Ur, S.: Framework for Testing Multi-threaded Java Programs. Concurrency and Computation: Practice and Experience 15(3-5), 485–499 (2003)
15. Elmas, T., Qadeer, S., Tasiran, S.: Goldilocks: A Race and Transaction-aware Java Runtime. In: Proc. of PLDI'07. pp. 245–255. ACM, New York, NY, USA (2007)
16. Engler, D., Ashcraft, K.: RacerX: Effective, Static Detection of Race Conditions and Deadlocks. SIGOPS Oper. Syst. Rev. 37(5), 237–252 (2003)

17. Eswaran, K.P., Gray, J.N., Lorie, R.A., Traiger, I.L.: The Notions of Consistency and Predicate Locks in a Database System. Commun. ACM 19, 624–633 (November 1976), http://doi.acm.org/10.1145/360363.360369
18. Farchi, E., Nir, Y., Ur, S.: Concurrent bug patterns and how to test them. In: IPDPS'03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing. p. 286.2. IEEE Computer Society, Washington, DC, USA (2003)
19. Fiedor, J., Křena, B., Letko, Z., Vojnar, T.: A uniform classification of common concurrency errors. In: Proceedings of the 13th International Conference on Computer Aided Systems Theory - Volume Part I. pp. 519–526. EUROCAST'11, Springer-Verlag, Berlin, Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-27549-4_67
20. Fiedor, J., Letko, Z., Lourenço, J., Vojnar, T.: Dynamic validation of contracts in concurrent code. In: Computer Aided Systems Theory–EUROCAST 2015. pp. 555–564. No. 9520, Springer-Verlag (2015)
21. Fiedor, J., Vojnar, T.: Noise-based testing and analysis of multi-threaded C/C++ programs on the binary level. In: PADTAD'12. pp. 36–46. ACM (2012)
22. Fiedor, J., Vojnar, T.: ANaConDA: A Framework for Analysing Multi-threaded C/C++ Programs on the Binary Level. In: Proc. of RV'13. pp. 35–41. volume 7687 of LNCS, Springer-Verlag (2013)
23. Flanagan, C., Freund, S.N.: Type-based Race Detection for Java. In: PLDI'00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation. pp. 219–232. ACM, New York, NY, USA (2000)
24. Flanagan, C., Freund, S.N.: Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs. SIGPLAN Not. 39(1), 256–267 (2004)
25. Flanagan, C., Freund, S.N.: Type Inference Against Races. Sci. Comput. Program. 64(1), 140–165 (2007)
26. Flanagan, C., Freund, S.N.: FastTrack: efficient and precise dynamic race detection. In: PLDI'09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation. pp. 121–133. ACM, New York, NY, USA (2009)
27. Flanagan, C., Freund, S.N.: The roadrunner dynamic analysis framework for concurrent programs. In: Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. pp. 1–8. PASTE '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1806672.1806674
28. Flanagan, C., Freund, S.N., Lifshin, M., Qadeer, S.: Types for Atomicity: Static Checking and Inference for Java. ACM Trans. Program. Lang. Syst. 30(4), 1–53 (2008)
29. Flanagan, C., Freund, S.N., Yi, J.: Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. SIGPLAN Not. 43(6), 293–303 (2008)
30. Godefroid, P.: Software model checking: The verisoft approach. Form. Methods Syst. Des. 26(2), 77–101 (2005)
31. Goubault, E.: Geometry and Concurrency: A User's Guide. Mathematical. Structures in Comp. Sci. 10(4), 411–425 (2000)
32. Hammer, C., Dolby, J., Vaziri, M., Tip, F.: Dynamic Detection of Atomic-set-serializability Violations. In: ICSE'08: Proceedings of the 30th international conference on Software engineering. pp. 231–240. ACM, New York, NY, USA (2008)

33. Havelund, K.: Using Runtime Analysis to Guide Model Checking of Java Programs. In: Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification. pp. 245–264. Springer-Verlag, London, UK (2000)
34. Ho, A., Smith, S., Hand, S.: On deadlock, livelock, and forward progress. Tech. rep., University of Cambridge (2005)
35. Holzmann, G.: Spin model checker, the: primer and reference manual. Addison-Wesley Professional (2003)
36. Hong, S., Ahn, J., Park, S., Kim, M., Harrold, M.J.: Testing Concurrent Programs to Achieve High Synchronization Coverage. In: Proc. of ISSTA'12. pp. 210–220. ACM, New York, NY, USA (2012)
37. Hovemeyer, D., Pugh, W.: Finding concurrency bugs in java. In: 23rd Annual ACM SIGACTSIGOPS Symposium on Principles of Distributed Computing (PODC 2004) Workshop on Concurrency and Programs (July 2004)
38. Joshi, P., Park, C.S., Sen, K., Naik, M.: A Randomized Dynamic Program Analysis Technique for Detecting Real Deadlocks. In: PLDI'09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation. pp. 110–120. ACM, New York, NY, USA (2009)
39. Kahlon, V., Yang, Y., Sankaranarayanan, S., Gupta, A.: Fast and Accurate Static Data-race Detection for Concurrent Programs. In: CAV'07. pp. 226–239 (2007)
40. Kim, K., Yavuz-Kahveci, T., Sanders, B.A.: Precise Data Race Detection in a Relaxed Memory Model Using Heuristic-Based Model Checking. In: ASE. pp. 495–499. IEEE (2009)
41. Křena, B., Letko, Z., Tzoref, R., Ur, S., Vojnar, T.: Healing Data Races On-the-fly. In: Proc. of PADTAD'07. pp. 54–64. ACM, New York, NY, USA (2007)
42. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput. 28(9), 690–691 (Sep 1979), http://dx.doi.org/10.1109/TC.1979.1675439
43. Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. Commun. ACM 21(7), 558–565 (1978)
44. Laurenzano, M., Tikir, M., Carrington, L., Snavely, A.: Pebil: Efficient static binary instrumentation for linux. In: ISPASS'10. pp. 175–183 (2010)
45. Letko, Z., Vojnar, T., Křena, B.: Atomrace: Data race and atomicity violation detector and healer. In: Proceedings of the 6th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging. pp. 7:1–7:10. PADTAD '08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1390841.1390848
46. Lipton, R.J.: Reduction: A Method of Proving Properties of Parallel Programs. Commun. ACM 18(12), 717–721 (1975)
47. Lu, S., Park, S., Hu, C., Ma, X., Jiang, W., Li, Z., Popa, R.A., Zhou, Y.: MUVI: Automatically Inferring Multi-variable Access Correlations and Detecting Related Semantic and Concurrency Bugs. SIGOPS Oper. Syst. Rev. 41(6), 103–116 (2007)
48. Lu, S., Tucek, J., Qin, F., Zhou, Y.: AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In: Proc. of ASPLOS'06. pp. 37–48. ACM, New York, NY, USA (2006)
49. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In: Proc. of PLDI'05. ACM (2005)
50. Masticola, S.P., Ryder, B.G.: Non-concurrency Analysis. In: PPOPP'93: Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming. pp. 129–138. ACM, New York, NY, USA (1993)

51. Mattern, F.: Virtual time and global states of distributed systems. In: Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel and Distributed Algorithms. Elsevier Science Publishers (1988), citeseer.ist.psu.edu/mattern89virtual.html
52. Musuvathi, M., Qadeer, S., Ball, T.: CHESS: A Systematic Testing Tool for Concurrent Software. Tech. Rep. MSR-TR-2007-149, Microsoft Research (2007)
53. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing heisenbugs in concurrent programs. In: OSDI'08. pp. 267–280. USENIX Association, Berkeley, CA, USA (2008), http://dl.acm.org/citation.cfm?id=1855741.1855760
54. Naik, M., Aiken, A., Whaley, J.: Effective Static Race Detection for Java. SIGPLAN Not. 41(6), 308–319 (2006)
55. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: PLDI'07. pp. 89–100. ACM, New York, NY, USA (2007), http://doi.acm.org/10.1145/1250734.1250746
56. Nir-Buchbinder, Y., Tzoref, R., Ur, S.: Deadlocks: From Exhibiting to Healing, pp. 104–118. Springer Berlin Heidelberg, Berlin, Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-89247-2_7
57. Nonaka, Y., Ushijima, K., Serizawa, H., Murata, S., Cheng, J.: A Run-Time Deadlock Detector for Concurrent Java Programs. In: APSEC'01: Proceedings of the Eighth Asia-Pacific on Software Engineering Conference. p. 45. IEEE, Washington, DC, USA (2001)
58. O'Callahan, R., Choi, J.D.: Hybrid Dynamic Data Race Detection. In: PPoPP'03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming. pp. 167–178. ACM, New York, NY, USA (2003)
59. Park, S., Vuduc, R.W., Harrold, M.J.: Falcon: Fault localization in concurrent programs. In: Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1. pp. 245–254. ICSE '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1806799.1806838
60. Poulsen, K.: Tracking the blackout bug (2004), http://www.securityfocus.com/news/8412
61. Pozniansky, E., Schuster, A.: Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs. In: Proc. of PPoPP'03. pp. 179–190. ACM, New York, NY, USA (2003)
62. Pozniansky, E., Schuster, A.: MultiRace: Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs: Research Articles. Concurr. Comput. : Pract. Exper. 19(3), 327–340 (2007)
63. von Praun, C., Gross, T.R.: Object Race Detection. In: Proc. of OOPSLA'01. pp. 70–82. ACM, New York, NY, USA (2001)
64. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: A Dynamic Data Race Detector for Multi-threaded Programs. In: Proc. of SOSP'97. pp. 27–37. ACM, New York, NY, USA (1997)
65. Sousa, D.G., Dias, R.J., Ferreira, C., Lourenço, J.M.: Preventing atomicity violations with contracts. arXiv preprint arXiv:1505.02951 (may 2015), http://arxiv.org/abs/1505.02951
66. Vaziri, M., Tip, F., Dolby, J.: Associating Synchronization Constraints with Data in an Object-oriented Language. In: POPL'06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 334–345. ACM, New York, NY, USA (2006)
67. Visser, W., Havelund, K., Brat, G., Park, S.: Model Checking Programs. In: Proc. of ASE'00. p. 3. IEEE Computer Society, Washington, DC, USA (2000)

68. Wang, L., Stoller, S.D.: Static Analysis of Atomicity for Programs with Non-blocking Synchronization. In: PPoPP'05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming. pp. 61–71. ACM, New York, NY, USA (2005)
69. Wang, L., Stoller, S.D.: Runtime Analysis of Atomicity for Multithreaded Programs. IEEE Trans. Softw. Eng. 32(2), 93–110 (2006)
70. Williams, A., Thies, W., Ernst, M.D.: Static Deadlock Detection for Java Libraries. In: ECOOP 2005—Object-Oriented Programming, 19th European Conference. pp. 602–629. Glasgow, Scotland (July 27–29, 2005)
71. Wu, J., Tang, Y., Hu, G., Cui, H., Yang, J.: Sound and Precise Analysis of Parallel Programs through Schedule Specialization. In: Proc. of PLDI'12. pp. 205–216. ACM, New York, NY, USA (2012)
72. Xu, M., Bodík, R., Hill, M.D.: A Serializability Violation Detector for Shared-memory Server Programs. SIGPLAN Not. 40(6), 1–14 (2005)
73. Yang, Y., Gringauze, A., Wu, D., Rohde, H.: Detecting Data Race and Atomicity Violation via Typestate-Guided Static Analysis. Tech. Rep. MSR-TR-2008-108, Microsoft Research (2008)
74. Yu, J., Narayanasamy, S.: A case for an interleaving constrained shared-memory multi-processor. SIGARCH Comput. Archit. News 37(3), 325–336 (2009)
75. Yu, J., Narayanasamy, S., Pereira, C., Pokam, G.: Maple: A Coverage-driven Testing Tool for Multithreaded Programs. In: Proc. of OOPSLA'12. pp. 485–502. ACM, New York, NY, USA (2012)
76. Yu, Y., Rodeheffer, T., Chen, W.: RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. SIGOPS Oper. Syst. Rev. 39(5), 221–234 (2005)
77. Zhang, W., Sun, C., Lu, S.: Conmem: detecting severe concurrency bugs through an effect-oriented approach. In: ASPLOS '10: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems. pp. 179–192. ACM, New York, NY, USA (2010)