A Uniform Classification of Common Concurrency Errors

FIT BUT Technical Report Series

Jan Fiedor, Bohuslav Křena, Zdeněk Letko, and Tomáš Vojnar

Technical Report No. FIT-TR-2010-03
Faculty of Information Technology, Brno University of Technology

Last modified: December 11, 2010

A Uniform Classification of Common Concurrency Errors

Jan Fiedor, Bohuslav Křena, Zdeněk Letko, and Tomáš Vojnar

Brno University of Technology, Božetěchova 2,
612 66, Brno, Czech Republic
{ifiedor, krena, iletko, vojnar}@fit.vutbr.cz
http://www.fit.vutbr.cz

Abstract. Nowadays, multi-threaded programs are quite common and so are concurrency errors. Many works devoted to detection of concurrency errors have been published in recent years and many of them presented definitions of concurrency errors that the proposed algorithms are able to handle. These definitions are usually expressed in different terms suitable for a description of the particular considered algorithms and they surprisingly often differ from each other in the meaning they assign to particular errors. To help understanding the errors and developing techniques for detecting them, this report strives to provide a uniform taxonomy of concurrency errors common in current programs, with a stress on those written in Java, together with a brief overview of techniques so far proposed for detecting such errors.

1 Introduction

The arrival of multi-core processors into regular computers accelerated development of software products that use multi-threaded design to utilise the available hardware resources. Modern object-oriented programming languages allow programmers to create multi-threaded programs, which, however, significantly increases chances of errors appearing in the code. Indeed, errors in concurrency are not only easy to cause, but also very difficult to discover and localise due to the non-deterministic nature of multi-threaded computation.

Due to the above, a lot of research effort is currently devoted to all sorts of methods of analysis and verification targeted at errors in concurrency. Plenty of papers describing new tools and techniques for detection of concurrency errors are presented each year. However, different authors describe the same concurrency errors in different terms and, surprisingly often, they even give the same concurrency error a different meaning. Therefore, this report strives to provide a uniform taxonomy of concurrency errors common in current programs, with a stress on those written in object-oriented programming languages, that could help a better understanding of these errors and their possible treatment.

The inconsistencies in definitions of concurrency errors are often related to the fact that authors of various analyses adjust the definitions according to the method they propose. Sometimes the definitions differ fundamentally (e.g.,

one can find works claiming that an execution leads to a deadlock if all threads terminate or end up waiting on a blocking instruction, without requiring any circular relationship between such threads required in the most common definition of deadlocks). However, often, the definitions have some shared basic *skeleton* which is *parameterised* by different underlying notions (such as the notion of behavioural equivalence of threads). In our description, we try to systematically identify the generic skeletons of the various notions of concurrency errors as well as the underlying notions parameterising them.

For the considered errors, we try to also provide a brief overview of the different existing techniques for detecting them. In these overviews, we (mostly) do not mention the approach of model checking which can, of course, be used for detecting all the different errors, but its use in practice is often limited by the state explosion problem (or, even worse, by a need to handle infinite state spaces) as well as the need to model the environment of a program being verified. That is why the use of model checking is usually limited in practice to relatively small, especially critical programs or components (e.g., drivers). For similar reasons, we do not discuss the use of theorem proving in the rest of the report either.

Related work. Of course, there have been several attempts to provide a taxonomy of concurrency errors in the past decades, c.f., e.g., [44, 46, 12]. In [12], authors focus on concrete bug patterns bound to concrete synchronisation constructs in Java like, e.g., the `sleep()` command. In [46], a kind of taxonomy of bug patterns can also be found. The authors report results of analysis of concurrency errors in several real-life programs. A detailed description of all possible concurrency errors that can occur when the `synchronised` construct is used in Java is provided in [44] where a Petri net model of the synchronisation construct is analysed. In comparison to these works, our aim is to provide uniform definitions of common concurrency errors that are not based on some specific set of programs or some specific synchronisation means, and we always stress the generic skeleton of the definitions and the notions parameterising it. We do not rely on concrete bug patterns because they are always incomplete, characterising only some specific ways how a certain type of error can arise.

Plan of the report. Our discussion of common concurrency errors is divided into two main sections. (1) Section 2 covers various errors in *safety*, i.e., errors that cause something bad to happen. In particular, data races, atomicity violation, order violation, deadlocks, and missed signals are discussed. (2) Section 3 covers various errors in *liveness*, i.e., errors that prevent something good from happening, as well as errors mixing liveness and safety. Concretely, starvation, livelocks, non-progress behaviour, and blocked threads are discussed. In all cases, the particular error is first defined, trying to stress the main concept of the error and the notions by which it is parameterised, followed by a brief discussion of various known techniques for detecting the particular error. Finally, in Section 4, a short conclusion is given.

2 Safety Errors

Safety errors violate safety properties of a program, i.e., cause something bad to happen. They always have a finite witness leading to an error state.

2.1 Data Races

Data races are one of the most common (mostly) undesirable phenomena in concurrent programs. To be able to identify an occurrence of a data race in an execution of a concurrent program, one needs to be able to say (1) which variables are shared by any two given threads and (2) whether any given two accesses to a given shared variable are synchronised in some way. A data race can then be defined as follows.

Definition 1 *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.*

Note, however, that not all data races are harmful—data races that are not errors are often referred to as *benign races*.

2.1.1 Detection of Data Races

Data races are a well studied concurrency problem and therefore there exist many different techniques for their detection. Dynamic techniques which 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 concrete code patterns that are likely to cause a data race or they compute locksets and/or happens-before relations over all executions considered feasible by the static analyser. There also exist static detection techniques that use type systems to detect data races. We discuss the basic principles of some of these techniques in the rest of this subsection.

The techniques based on *locksets* [61] build on the idea that all accesses to a shared variable should be guarded by a lock. The lockset is defined as a set of locks that guard all accesses to a given variable. Detectors then use an observation that if the lockset associated with a certain shared variable is non-empty, then there is at least one lock such that every access to the shared variable from any thread is protected by this lock, and hence there is no possibility of simultaneous accesses, and so a data race is not possible.

The happens-before-based techniques exploit the so-called *happens-before relation* [41] (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 $x \rightarrow y$. Also, when 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, locking and unlocking 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 the happens-before relation between them.

Type systems provide a syntactic method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute [58]. A formal type system provides a powerful and efficient checker of correctness of the code mainly if the programming language is strongly and statically typed (i.e., each variable has a deterministic type in each point of computation, and the types can be inferred without executing the code). Detection of concurrency bugs is usually done by extending the initial type system with a number of additional types that handle concurrency. These additional types are usually expressed by code annotations. A type system then searches for violations of rules defined over the newly defined types. Sometimes, the notion of *typestates*, introduced in [64], is used. Typestates extend the ordinary types that do not change through the lifetime of an object by allowing them to change during the course of the computation. A typestate property can be captured by a finite state machine where the nodes represent states of the type and the arcs correspond to operations that lead to state transitions.

Lockset-based algorithms. The first algorithm which used the idea of locksets was Eraser [61]. 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. Eraser updates $C(v)$ by intersecting $C(v)$ and the set $L(t)$ of locks held by the current thread whenever v is accessed. Eraser warns about a 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 $s(v)$ for each shared variable v used to identify whether v is used exclusively by one thread, v is read by multiple threads, or multiple threads change the value of v. The lockset $C(v)$ is then modified only when the variable is shared and a data race is reported only if $C(v)$ becomes empty and v is in the state denoting the situation when multiple threads access v for writing.

The original Eraser algorithm designed for C programs was then modified for programs written in object-oriented languages, c.f., e.g., [69, 13, 11, 77]. The main modification (usually called as the *ownership model*) is inspired by the common idiom used in object-oriented programs where a 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 by the Java virtual machine. This idea is reflected by inserting a new internal state of the shared variables. The modification introduces a small possibility of having false negatives [40, 69] but greatly reduces the number of false alarms caused by this object-oriented programming idiom.

A static data race detector that approximates locksets using an interprocedural data-flow analysis has been presented in [19]. The detection has three phases: (1) The control flow of each procedure is obtained and a call graph of the whole system is constructed. (2) A top-down data-flow analysis based on locksets is performed over the constructed graphs. A data race is suspected if an access to a shared variable is not guarded by a lock that is present in the lockset constructed for the variable along a path being analysed. (3) A final inspection algorithm then ranks each detected possible bug and reports only those that are real with a higher probability. However, the approach still produces many false alarms and it also has high memory requirements.

Better results were obtained, e.g., in [52, 38] where two more static analyses were incorporated into the machinery. An *alias analysis* [55] identifies a set of variables that refer to the same memory location, and an *escape analysis* [55] identifies a set of objects that are accessible in more than one thread. The first analysis enables the method to join locksets of different variables referencing the same data, and the escape analysis allows the method to limit the computation of locksets for variables that could be shared only.

A computation of locksets has also been implemented in the Java PathFinder (JPF) model checker [68]. JPF performs explicit state model checking, and therefore the *RaceDetector* checker computes a lockset for each shared variable and each state of the state space.

A problem of techniques based on locksets is that they do not support other synchronisation than locks and therefore produce too many false alarms when applied to common concurrent software.

Happens-before-based algorithms. Most happens-before-based algorithms use the so-called *vector clocks* introduced in [50]. The idea of vector clocks for a message passing system is as follows. Each thread has a vector of clocks T_{vc} indexed by thread identifiers. One position in T_{vc} represents the own 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 $T_{vc} = T_{vc} \sqcup T'_{vc}$.

Recently, a new variation of the vector-clock algorithm has been published [3]. The modification allows a distributed computation of vector clocks. Each thread maintains not a vector but a tree structure holding values of vector clocks. However, according to the best of our knowledge, there is no detector that uses this new algorithm yet.

Algorithms [59, 60] 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 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 of variables from above were mostly replaced by the so-called *epochs* associated with each variable v that are represented as tuples (t, c) 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.

The happens-before relation can also be approximated statically. A very expensive and quite precise static analysis has been introduced in [49]. The algorithm proposed in this paper aims at computing the so-called *may-happen-in-parallel* relation (MHP) which is the complement of the happens-before relation. MHP is approximated in two steps. Initially, any pair of instructions is assumed to be able to happen in parallel (the MHP is total). Then, the initial set of pairs is pruned such that only those which cannot be seen to happen in succession remain in the set. A data race is reported if two accesses to a shared variable are in the MHP relation.

The original algorithm for computing MHP relations is very inefficient and is able to handle very small programs only. Therefore, several modifications have been proposed. The approach presented in [54] tries to compute MHP relations using a data-flow framework. The data-flow analysis is performed over the so-called *parallel execution graph*. This graph combines control-flow graphs of all threads that could be started during the execution with special edges induced by synchronisation actions in the code. The size of the graph increases exponentially with the size of the program and with the number of threads. Further, the data-flow computation of MHP relations has been slightly improved in [8] by the so-called *thread creation trees* (TCT) that help to compute a rough over-approximation of MHP relations before the data-flow evaluation is used. However, the obtained static analysis is still quite expensive.

A somewhat similar approach has also been used in the JPF model checker [68]. The *PreciseRaceDetector* detects data races by checking whether two accesses to the same variable may happen in parallel. In every state that JPF visits, the algorithm checks all actions that can be performed next. If this collection of actions contains at least two accesses to the same variable from different threads, then a data race is reported. Compared to the static analyses presented above, the detection in JPF is completely precise: It announces a data race only if it can really happen.

A bit different detection approach has been introduced in TRaDe [14] where a *topological race detection* [30] 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 a considerable overhead.

An advantage of the algorithms mentioned above is their precision. However, the big cost of these algorithms inspired many researches to come up with some combination of happens-before-based and lockset-based algorithms. These combinations are often called *hybrid algorithms*.

Happens-before-based algorithms. Hybrid algorithms such as [57, 18, 77, 24] combine the two approaches described above.

In RaceTrack [77], a notion of a *threadset* was introduced. The threadset is maintained for each shared variable and contains information concerning 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 consisting 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 labels that correspond to accesses that are ordered before the current access. Hence the threadset contains solely 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 currently accesses it.

One of the most advanced lockset-based algorithms that also uses the happens-before relation is Goldilocks presented in [18]. The main idea 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. An appearance of a thread t in a lockset of a shared variable 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 the transitive closure of the happens-before relation via the locksets. An advantage of Goldilocks is that it allows locksets to grow during a 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 cheaper checks that are sufficient for race freedom between the two last accesses on a variable are used) and using *a lazy computation of the locksets* (the locksets are computed only if the previous optimisation is not able to detect that some events are in the happens-before re-

lation). The optimised algorithm has a considerably lower overhead approaching in some cases pure lockset-based algorithms.

A similar approach to Goldilocks but for the JPF model checker has been presented in [39]. This algorithm does not map variables to locksets containing threads and synchronisation elements (such as locks) but 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. The modification can be done because model checking allows the method to modify structures associated with different threads at once. In a dynamic analysis, this cannot be done and locksets must be maintained in a distributed manner.

In [70], an abstract interpretation over abstract heaps is performed. The algorithm maintains the so-called *object use graphs* (OUG) capturing accesses from different threads to particular objects. Nodes of an OUG represent events performed with the object for which the OUG is built, and edges denote approximated happens-before relations between the events. The data race detection algorithm performed after the OUGs are built then does not need to analyse the entire program but only relatively small OUGs. A data race is detected if there exist two events such that: (1) There is no ordering between the events, (2) the events originate from different threads, (3) at least one event is a write action, and (4) the events are not done under a common lock protection.

Type-based algorithms. In [23, 25], a clone of classic Java called *ConcurrentJava* has been proposed. The papers presented several annotations of using synchronisation primitives and of accessing shared variables allowing race-freeness be checked by an extended typing system of ConcurrentJava. For instance, each definition of a shared variable can be annotated with an annotation `guarded-by` l which requires each access to the particular variable to be guarded by a lock l.

A combination of type-based and data flow analysis has been presented in [75]. The proposed algorithm uses typestates to handle locking states for each variable and does not need any annotation provided by the user. Typestates are computed using an intraprocedural data-flow analysis and symbolic path simulation. The technique is able to handle large programs but still produces false alarms mainly due to unsupported synchronisation mechanisms (only locks are supported).

2.2 Atomicity Violation

Atomicity is a non-inference property. The notion of atomicity is rather generic. It is parametrised by (1) a specification of when two program executions may be considered equivalent from the point of view of their overall impact and (2) a specification of which code blocks are assumed to be atomic. Then an atomicity violation can be defined as follows.

Definition 2 *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.*

An execution that violates atomicity of some code blocks is often denoted as an *unserialisable* execution. The precise meaning of unserialisability of course depends on the employed notion of equivalence of program executions.

2.2.1 Detection of Atomicity Violation

Taking into account the generic notion of atomicity, methods of detecting atomicity violation can be classified according to: (1) The way they obtain information about which code blocks should, in fact, be expected to execute atomically. (2) The notion of equivalence of executions used (we will get to several commonly used equivalences in the following). (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 [28]. Some other works come with predefined patterns of code which should typically execute atomically [47, 67, 32]. Still other authors try to infer blocks to be assumed to execute atomically, e.g., by analysing data and control dependencies between program statements [74], where dependent program statements form a block which should be executed atomically, or by finding out access correlations between shared variables [45], where a set of accesses to correlated shared variables should be executed atomically (together with all statements between them).

Below, we first discuss approaches of 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 try to detect a situation where two accesses to a shared variable should be executed atomically, but are interleaved by an access from another thread.

In [74], 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 tries to partition this dependency graph to obtain a set of distinct subgraphs which are 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 [20] 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 variables). 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 a program trace is first recorded and then analysed using the off-line version of the algorithm.

A much simpler approach of discovering atomicity violations was presented in [47]. 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 an 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 slow down of the monitored application.

A more complicated approach was introduced in [24, 72], where atomicity violations are sought using the Lipton's reduction theorem [43]. The approach is in particular 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 left or right in the execution). Both [24] and [72] 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) *Right-mover* instructions R which can be swapped with immediately following instructions. (2) *Left-mover* instructions L which can be swapped with immediately preceding instructions. (3) *Both-mover* instructions B which can be swapped with preceding or following instructions. (4) *Non-mover* instructions N which 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 or no non-mover instruction and both-mover instructions B can be taken as right-mover instructions R or left-mover instructions L. Algorithms in both [24] and [72] use dynamic analysis to detect atomicity violation using the reduction algorithm described above.

Other approaches using the Lipton's reduction theorem [43] can be found in [27, 71] where type systems based on this theorem are used to deal with atomicity violations.

In [27], atomicity is analysed in programs written in a language called AtomicJava, a subset of Java with a type system for atomicity. The type system works with atomicity-related types denoting an expression as compound, atomic, mover (in the sense of Lipton), etc. These types may, moreover, be conditional upon the locks held. The user may annotate methods by the atomicity types and he/she can also annotate variables by locks the variables are supposed to be guarded with. The type inference rules proposed in the paper then automatically derive type constraints whose solution (if any) provides atomicity types for particular methods. If the methods were annotated by the user, conformance of the automatically derived and manually provided atomicity types is checked. The use of the conditional atomicity types makes the analysis more precise than the previous approaches.

In [71], a more simple type system for programs with non-blocking synchronisation is used which operates with five atomicity-related types: atomic, non-atomic, right-, left-, and both-mover expressions are distinguished. Again, procedures are considered as the main unit of atomicity. In this case, no annotations are provided, the technique just informs on which methods it considers atomic and which not, which is a bit restricting. Methods which are not executed atomically may, but need not violate atomicity assumptions of the programmer.

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 three different variables representing a point in a three-dimensional space. Even if we ensure that every consecutive read and write accesses to each of these variables are executed atomically, the program can still have an unserialisable execution. This is because the three atomic blocks guarding each pair of accesses to each of these variables can be interleaved with other atomic blocks operating with these variables. Some of these variables can then end up modified by a different thread than the others which cannot happen in a serial execution. Nevertheless, the above discussed detectors would not detect any atomicity violation here.

In [6], 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 `synchronised` 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 the thread. An execution is then serialisable if each thread is only using views which 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 a program trace. Since the algorithm has to operate with a big number of sets (each view is a set), it suffers from a big overhead.

A different approach is associated with the Velodrome detector [28]. 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 if it contains a cycle. If yes, the program contains an atomicity violation. Since creating the graph for an entire execution is inconvenient, nodes that cannot be involved in a cycle are garbage collected or not created at all. Like the previous algorithm, Velodrom too may suffer from a considerable overhead in some cases.

The simple idea of *AI invariants* described in [47] has been generalised for checking atomicity over pairs of variables in [67, 32], where 11 or 14, respectively, 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 [67] there is proposed an algorithm which infers which procedure bodies should be the so-called *units of work* w.r.t. the given atomic sets. This is done statically using a 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], 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 detection to be able to detect atomicity violations over multiple variables. One method can be found in [45] 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 [2] 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 [2]. 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 for every pair of accesses to a shared variable that 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 variables, but also with accesses to the correlated variables. The detection can be done statically or dynamically, depends on the data race detector which is used.

2.3 Order Violations

Order violations form a much less studied class of concurrency errors than data races and atomicity violations, which is, however, starting to gain more attention lately. An order violation is a problem of a missing enforcement of some higher-level ordering requirements. For detecting order violations, one needs to be able to decide for a given execution whether the instructions executed in it have been executed in the right order. An order violation can be defined as follows.

Definition 3 *A program execution exhibits an order violation if some instructions executed in it are not executed in an expected order.*

2.3.1 Detection of Order Violations

Like in the case of atomicity violations, a prerequisite for detecting order violations is to know which order restrictions are assumed. These can be specified manually, generic order requirements may be used (e.g., an object must be first initialised and only then used), or some restrictions may be automatically inferred. The order restrictions considered in current approaches are often quite simple, frequently considering only pairs of instructions. The actual discovery of order violations can in theory be done dynamically as well as statically. Currently, however, there are not many works dealing with order violation detection as has been pinpointed in [46].

In [76], authors introduce, for each memory operation o, a set of memory operations $PSet(o)$ which o depends upon and which can safely occur before o. These sets are extracted from a set of correct executions of the analysed program. Then, order violations are sought in further runs by looking for a memory operation o such that the previous memory operation dependent upon o is not in $PSet(o)$.

The ConMem tool [78] detects several behavioural patterns corresponding to order violations that can lead to a program crash. For each test input, ConMem monitors one execution of the given program. It uses a dynamic analysis to first identify parts of executions (denoted as ingredients) that may lead to

a crash if ordered differently than in the given execution (e.g., assignments of `null` to a shared pointer and dereferences of this shared pointer from different treads). Then, ConMem analyses synchronisation around these potentially problematic constructions to see whether fatal interleavings exist to trigger an error (e.g., an interleaving where a thread t_1 assigns `null` to a shared variable v, and subsequently, a thread t_2 dereferences v). The paper describes four problematic patterns consisting of ingredients and timing conditions that lead to an error and that ConMem is able to detect. One example is the *Con-NULL* pattern with ingredients *rp*—a thread t_1 reads a pointer *ptr*, *wp*—a thread t_2 writes `null` to *ptr*, and timing conditions requiring *wp* to execute before *rp* with no write operation on *ptr* happening in between of *wp* and *rp*. Another example is the *Con-UnInit* pattern with ingredients *r*—a thread t_1 reads a variable v without previously writing to v, *w*—a thread t_2 initialises v, and the timing condition requiring *r* to execute before *w*.

2.4 Deadlocks

Deadlocks are a class of safety errors which is quite often studied in the literature. However, despite that, the understanding of deadlocks still varies in different works. We stick here to the meaning common, e.g., 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 4 *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 from S.*

Most works consider a special case of deadlocks, namely, the so-called *Coffman deadlock* [15]. 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 until the resources are used to completion (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).

2.4.1 Detection of Deadlocks

Detection of deadlocks usually involves various graph algorithms as it is, for instance, in the case of the algorithm introduced in [56] where a *thread-wait-for*

graph is dynamically constructed and analysed 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 could 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., [9, 1] improve the algorithm to detect deadlocks among multiple threads. In [1], a support for semaphores and wait-notify synchronisation was added. The recent work [37] modified the original algorithm so that runtime lock trees are not constructed. Instead, the algorithm uses a stack to handle the so-called *lock dependency relation*. 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.

A purely data-flow-based interprocedural static detector of deadlocks called RacerX has been presented in [19]. The detection it implements has two phases: (1) The control flow graph of each procedure is obtained and the complete control flow graph of the whole system is constructed. (2) A data-flow analysis is performed over the constructed graph and the order in which locks are nested is analysed. A deadlock is reported when locks are not obtained every time in the same order. In [73], a bottom-up data-flow static analysis is used to detect deadlocks. The algorithm traverses the call graph bottom-up and builds a lock-order graph per method. Each node of the lock-order graph represents a set of objects that may be aliased and an edge in the graph indicates nested locking of objects along some code path. If the obtained graph contains cycles, a possible deadlock is reported. Both algorithms produce many false alarms due to the approximations they use.

The algorithm presented in [53] reduces the number of false alarms obtained by a data-flow interprocedural analysis described in the previous paragraph using six conditions. The first four represent results of reachability, alias, escape, and approximated may-happen-in-parallel analyses. The next two conditions handle

special cases of using reentrant locks and a guarding lock. The algorithm filters all potential deadlocks through these conditions and reports only those which fulfil all the conditions.

A combination of symbolic execution, static analysis, and SMT solving is used in [17] to automatically derive the so-called *method contracts* guaranteeing deadlock free executions. The algorithm does not focus on detection of deadlock if the whole program is available. Instead, the algorithm analyse pieces of code (usually libraries) and automatically infers conditions that must be fulfilled when these libraries are used in order to avoid deadlocks.

2.5 Missed Signals

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 *A program execution contains a missed signal iff there is sent a signal that is not delivered to the thread or threads to which it is intended to be delivered.*

Since signals are often used to unblock waiting threads, a missed signal error typically leads to a thread or threads being blocked forever.

2.5.1 Detection of Missed Signals

There are not many works focusing specially on missed signals. Usually, the problem is studied as a part of detecting other concurrency problems. In [1], a lost notification error is reported if there is a notify event e in a trace tr and there exists a trace that is a *feasible permutation* of tr in which e wakes up fewer threads than it does in tr. Such a situation is possible when the wait event of one of the threads woken in tr is not constrained to happen before the event e.

In [36], several code patterns that might lead to a lost notification are listed. For instance, each call of `wait()` must be enclosed by a loop checking some external condition. A pattern-based static analysis is then used to detect such bug patterns.

3 Liveness and Mixed Errors

Liveness errors are errors which violate liveness properties of a program, i.e., prevent something good from happening. They have infinite (or finite, but complete) witnesses. Dealing with liveness errors is much harder than with safety errors because algorithms dealing with them have to find out that there is no way something could (or could not) happen in the future, which often boils down to a necessity of detecting loops. Mixed errors are then errors that that have both finite witnesses as well as infinite ones, whose any finite prefix does not suffice as a witness.

Before we start discussing more concrete notions of liveness and mixed errors, let us first introduce the very general notion of *starvation* [66].

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

Starvation can be seen to cover as special cases various safety as well as liveness (mixed) errors such as deadlocks, missed signals, and the below discussed livelocks or blocked threads. In these situations, an event for which a thread is waiting cannot happen, and the situations are clearly to be avoided. On the other hand, there are cases where the event for which a thread is waiting can always eventfully happen despite there is a possibility that it never happens. Such situations are not welcome since they may cause performance degradation, but they are sometimes tolerated (one expects that if an event can always eventually happen, it will eventually happen in practice).

3.1 Livelocks and Non-progress Behaviour

There are again 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 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. However, by analogy with deadlocks, we feel it more appropriate to restrict the notion of livelocks to the case when threads are looping in a useless way while trying to synchronise (which is a notion common, e.g., in various works on operating systems). That is why, we first define a general notion of non-progress behaviour and then we specialise it to livelocks.

Definition 7 *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.*

A non-progress behaviour is a special case of starvation within an infinite behaviour. On the other hand, starvation may exhibit even in finite behaviours and also in infinite progress behaviours in which a thread is for a while waiting for an event that is not guaranteed to happen. As we have said already above, livelocks may be seen as a special case of non-progress behaviour [66].

Definition 8 *Within an infinite execution, a set S of threads is in a livelock iff each of the threads in S keeps running forever in some loop in which it is not intended to run forever, but which it could leave only if some thread from S could leave the loop it is running in.*

As was mentioned before, there are many, often inconsistent, definitions of a livelock. Moreover, many works do not distinguish between livelocks and a non-progress behaviour [10, 63, 42, 65, 34]. Other papers [51, 48] take a livelock to be

a situation where a task has such a low priority that it does not run (it is not allowed to make any progress) because there are many other, higher priority, tasks which run instead. We do not consider such a situation a livelock and not even a non-progress behaviour but a form of starvation. There are even works [4] for which a thread is in a livelock whenever it is executing an infinite loop, regardless of what the program does within the loop. However, there are many reactive programs which run intentionally in an infinite loop, e.g., controllers, operating systems and their components, etc., and it is not appropriate to considered them to be in a livelock.

3.1.1 Detection of Livelocks and Non-progress Behaviour

We are not aware of any works specialising in detection of livelocks in the sense we defined them above, which requires not only detection of a looping behaviour but also of the fact that this behaviour could be escaped only if some of the livelocked threads could escape it. There are, however, works considering detection of non-progress behaviour (sometimes under the name of livelock detection, but we stick here to speaking about detection of non-progress behaviour).

The first issue the non-progress detection methods have to deal with is getting to know what is to be considered a non-progress behaviour. In the literature there are used various different notions of progress. Some works, e.g., [63], define progress by looking at the *communication* among two or more cooperating threads. If a thread communicates, it is progressing. In [29], progress is associated with operations on the so-called *communication objects* (such as shared variables, semaphores, FIFO buffers, etc.). In [42, 65], progress is defined by reaching a so-called *progress action* or *progress statement*, respectively (e.g., delivering output, responding to the environment, etc.). In [34], progress is expressed by a so-called *liveness signature*, a set of state predicates and temporal rules, specifying which application states determine whether a program is making a progress when they are repeatedly reached.

To enable a non-progress detection, the general notions of progress from above have to be concretised for a particular program. This is mostly expected to be done by the user, e.g., by specifying progress actions [42], labelling statements as progress statements [35], annotating the code [34], calling actions of the so-called observer [10], etc. However, some works do not require any user input and use a fixed notion of progress [29].

One of the most common approaches for detecting non-progress behaviour in finite-state programs is to use model checking [35, 29] and search for non-progress cycles [21]. In [29], no non-progress cycles are being sought, instead bounded model checking is used to find a path where no progress is being made for a user-specified period of time. Such an approach can lead to false alarms, but can be used for large and infinite-state programs. Recently, there has also appeared various (infinite-state) program analyses, based, e.g., on transition predicate abstraction and ranking function synthesis, for proving (non-)termination. These can also be used for verifying liveness properties [16], but these approaches, albeit

useful, for instance, for verifying drivers, are still not applicable to large software systems.

Another often used approach is to use a dynamic analysis in the form of *dynamic monitoring* [34, 5]. In [34], a liveness signature, a set of state predicates and temporal rules, is used to determine whether a program is making progress. In other words, the liveness signature determines which program states are significant in determining whether a program is making a progress. If a program does not reach any of these program states for some time, the program is at a so-called *standstill*, i.e., is not making progress. In [5], the so-called *Q-Learning* is used to navigate a program execution to follow paths which most likely lead to an execution trace in which the program is not making progress. In case of finite-state programs, the tool searches for an execution trace containing a non-progress loop, in case of infinite-state programs, it tries to find an execution trace of a user-specified length. A problem with dynamic monitoring techniques is that they cannot distinguish between a non-progress behaviour and starvation because they are checking bounded liveness properties [62], which are in fact safety properties, not liveness properties.

In [10], a static analysis is used to detect a non-progress behaviour in programs written in Ada. Progress is defined here as a *communication with an external observer*. Each program is represented by a control flow graph. To detect a non-progress behaviour, the method searches the control flow graph for an infinite loop in which no communication with the external observer is performed.

There are also works which attempt to use approaches often used for detection of other concurrency errors. For example, in [31], the authors define *antipatterns* which may lead to a non-progress behaviour and use static and dynamic analyses to locate these antipatterns in programs.

In [42], the authors define progress as an execution of a progress action and a progress cycle as a cycle which contains at least one progress action. Then they translate a necessary condition of an existence of a non-progress behaviour, i.e., that there exists an infinite run in which progress cycles are repeated only a finite number of times, into a homogeneous integer programming problem. Subsequently, they try to find a solution to this problem. If the problem has no solution, then the program surely does not contain a non-progress behaviour. A downside of this method (apart from its cost) is that it is incomplete, so if the problem has a solution there may or may not be a non-progress behaviour.

3.2 Blocked Threads

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 9 *A program execution contains a blocked thread iff there is a thread which is waiting for some event to continue and this event never occurs in the execution.*

An absence of some unblocking event which leaves some thread blocked may have various reasons. A common reason is that a thread, which should have unblocked some other thread, ended unexpectedly, leaving the other thread in a blocked state. In such a case, one often speaks about the so-called *orphaned threads* [22]. Another reason may be that a thread is waiting for a livelocked or deadlocked thread.

3.2.1 Detection of Blocked Threads

We are not aware of any works specialising in this kind of errors. Of course, the most simple solution to deal with this error is to check that a thread is waiting for more than some time to be unblocked. This is, however, a very crude approach. In fact, a similar approach is used in MySQL to detect deadlocks and it was shown [46] that it is quite inaccurate detection method which often leads to false alarms and, in case of MySQL, to unnecessary restarts.

Like with all previous errors, another possibility is to use model checking [7], limited by its high price and problems with some program operations like input and output. In theory, one could also use, e.g., static analysis for detection of some undesirable code patterns that could cause permanent blocking, similar to deadlock antipatterns [31] in case of a deadlock. However, detection of this kind of errors remains mostly an open issue.

4 Conclusions

We have provided a uniform classification of common concurrency errors, mostly focusing on shared memory systems. In the definitions, we have have tried to stress the basic skeleton of the considered phenomena together with the various notions that parameterise these phenomena and that must be fixed before one can speak about concrete appearances of the given errors. These parameters are often used implicitly, but we feel appropriate to stress their existence so that one realizes that they have to be fixed and also that various specialised notions of errors are in fact instances of the same general principle. We decided to define all the considered errors in an informal way in order to achieve a high level of generality. For concrete and formal definitions of these errors, one has to define the memory model used and the exact semantics of all operations that may (directly or indirectly) influence synchronisation of the application, which typically leads to a significant restriction of the considered notions.

We have also mentioned various detection techniques for each studied concurrency error and briefly described the main ideas they are based on. It is evident that some concurrency errors are quite often studied (e.g., data races), and some have a shortage of algorithms for their detection. Despite some of the latter problems may appear less often than the former ones and they are also typically more difficult to detect, detection of such problems is an interesting subject for future research.

Acknowledgement

This work was supported by the Czech Science Foundation (within projects P103/10/0306 and 102/09/H042), the Czech Ministry of Education (projects COST OC10009 and MSM 0021630528), and the internal BUT project FIT-10-1.

References

1. R. Agarwal and S. D. Stoller. Run-time Detection of Potential Deadlocks for Programs with Locks, Semaphores, and Condition Variables. In *Proc. of PADTAD'06*. ACM Press, 2006.
2. R. Agrawal, T. Imieliński, and A. Swami. Mining Association Rules Between Sets of Items in Large Databases. In *Proc. of SIGMOD'93*. ACM Press, 1993.
3. P. S. Almeida, C. Baquero, and V. Fonte. Interval Tree Clocks. In *Proc. of OPODIS'08*. Springer-Verlag, 2008.
4. G. R. Andrews. *Concurrent Programming: Principles and Practice*. Benjamin-Cummings Publishing, 1991.
5. T. Araragi and S. M. Cho. Checking Liveness Properties of Concurrent Systems by Reinforcement Learning. In *Model Checking and Artificial Intelligence*, LNCS 4428. Springer-Verlag, 2007.
6. C. Artho, K. Havelund, and A. Biere. High-level Data Races. In *Proc. of VVEIS'03*. Angers, 2003.
7. C. Baier and J.-P. Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.
8. R. Barik. Efficient Computation of May-happen-in-parallel Information for Concurrent Java Programs. In *Languages and Compilers for Parallel Computing*, LNCS 4339. Springer-Verlag, 2006.
9. S. Bensalem and K. Havelund. Dynamic Deadlock Analysis of Multi-threaded Programs. In *Proc. of PADTAD'05*. Springer-Verlag, 2005.
10. J. Blieberger, B. Burgstaller, and R. Mittermayr. Static Detection of Livelocks in Ada Multitasking Programs. In *Proc. of Ada-Europe'07*. Springer-Verlag, 2007.
11. E. Bodden and K. Havelund. Racer: Effective Race Detection Using AspectJ. In *Proc. of ISSTA'08*. ACM Press, 2008.
12. J. S. Bradbury and K. Jalbert. Defining a Catalog of Programming Anti-patterns for Concurrent Java. In *Proc. of SPAQu'09*. ACM Press, 2009.
13. 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. of PLDI'02*. ACM Press, 2002.
14. M. Christiaens and K. D. Bosschere. Trade: Data Race Detection for Java. In *Proc. of ICCS'01*. Springer-Verlag, 2001.
15. E. G. Coffman, M. Elphick, and A. Shoshani. System Deadlocks. *ACM Comput. Surv.*. ACM Press, 1971.
16. B. Cook, A. Gotsman, A. Podelski, A. Rybalchenko, and M. Y. Vardi. Proving That Programs Eventually Do Something Good. In *Proc. of POPL'07*. ACM Press, 2007.
17. J. Deshmukh, E. A. Emerson, and S. Sankaranarayanan. Symbolic Deadlock Analysis in Concurrent Libraries and Their Clients. In *Proc. of ASE'09*. IEEE, 2009.
18. T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A Race and Transaction-aware Java Runtime. In *Proc. of PLDI'07*. ACM Press, 2007.

19. D. Engler and K. Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proc. of SIGOPS'03*. ACM Press, 2003.
20. K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM*. ACM Press, 1976.
21. D. Faragó and P. H. Schmitt. Improving Non-progress Cycle Checks. In *Proc. of SPIN'09*. Springer-Verlag, 2009.
22. E. Farchi, Y. Nir, and S. Ur. Concurrent Bug Patterns and How to Test Them. In *Proc. of IPDPS'03*. IEEE, 2003.
23. C. Flanagan and S. N. Freund. Type-based Race Detection for Java. In *Proc. of PLDI'00*. ACM Press, 2000.
24. C. Flanagan and S. N. Freund. Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs. In *Proc. of SIGPLAN'04*. ACM Press, 2004.
25. C. Flanagan and S. N. Freund. Type Inference Against Races. *Sci. Comput. Program.*, 64(1):140–165, 2007.
26. C. Flanagan and S. N. Freund. Fasttrack: Efficient and Precise Dynamic Race Detection. In *Proc. of SIGPLAN'09*. ACM Press, 2009.
27. C. Flanagan, S. N. Freund, M. Lifshin, and S. Qadeer. Types for Atomicity: Static Checking and Inference for Java. *ACM Trans. Program. Lang. Syst.*, 30(4):1–53, 2008.
28. C. Flanagan, S. N. Freund, and J. Yi. Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. *SIGPLAN Not.*, 43(6):293–303, 2008.
29. P. Godefroid. Software Model Checking: The Verisoft Approach. *Form. Methods Syst. Des.*, 26(2):77–101, 2005.
30. E. Goubault. Geometry and Concurrency: A User's Guide. *Mathematical. Structures in Comp. Sci.*, 10(4):411–425, 2000.
31. H. H. Hallal, E. Alikacem, W. P. Tunney, S. Boroday, and A. Petrenko. Antipattern-based Detection of Deficiencies in Java Multithreaded Software. In *Proc. of QSIC'04*. IEEE, 2004.
32. C. Hammer, J. Dolby, M. Vaziri, and F. Tip. Dynamic Detection of Atomic-set-serializability Violations. In *Proc. of ICSE'08*. ACM Press, 2008.
33. K. Havelund. Using Runtime Analysis to Guide Model Checking of Java Programs. In *Proc. SPIN'00*. Springer-Verlag, 2000.
34. A. Ho, S. Smith, and S. Hand. On Deadlock, Livelock, and Forward Progress. Technical report, University of Cambridge, 2005.
35. G. Holzmann. *Spin Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.
36. D. Hovemeyer and W. Pugh. Finding Concurrency Bugs in Java. In *Proc. of PODC'04*. ACM Press, 2004.
37. P. Joshi, C.-S. Park, K. Sen, and M. Naik. A Randomized Dynamic Program Analysis Technique for Detecting Real Deadlocks. In *SIGPLAN'09*. ACM Press, 2009.
38. V. Kahlon, Y. Yang, S. Sankaranarayanan, and A. Gupta. Fast and Accurate Static Data-race Detection for Concurrent Programs. In *Proc. of CAV'07*. ACM Press, 2007.
39. 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. of ASE'09*. IEEE, 2009.
40. B. Křena, Z. Letko, R. Tzoref, S. Ur, and T. Vojnar. Healing Data Races On-the-fly. In *Proc. of PADTAD'07*. ACM Press, 2007.

41. L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, 1978.
42. S. Leue, A. Ştefănescu, and W. Wei. A Livelock Freedom Analysis for Infinite State Asynchronous Reactive Systems. In *Proc. of CONCUR'06*, LNCS 4137. Springer-Verlag, 2006.
43. R. J. Lipton. Reduction: A Method of Proving Properties of Parallel Programs. *Commun. ACM*, 18(12):717–721, 1975.
44. B. Long and P. Strooper. A Classification of Concurrency Failures in Java Components. In *Proc. of IPDPS'03*. IEEE, 2003.
45. S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. Muvi: Automatically Inferring Multi-variable Access Correlations and Detecting Related Semantic and Concurrency Bugs. *SIGOPS Oper. Syst. Rev.*, 41(6):103–116, 2007.
46. S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proc. of ASPLOS'08*. ACM Press, 2008.
47. S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: Detecting Atomicity Violations via Access Interleaving Invariants. In *Proc. of ASPLOS'06*. ACM Press, 2006.
48. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.
49. S. P. Masticola and B. G. Ryder. Non-concurrency Analysis. In *Proc. of PPOPP'93*. ACM Press, 1993.
50. F. Mattern. Virtual Time and Global States of Distributed Systems. In *Proc. of PDA'88*. Elsevier Science Publishers, 1988.
51. J. C. Mogul and K. K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-driven Kernel. *ACM Trans. Comput. Syst.*, 15(3):217–252, 1997.
52. M. Naik, A. Aiken, and J. Whaley. Effective Static Race Detection for Java. *SIGPLAN Not.*, 41(6):308–319, 2006.
53. M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective Static Deadlock Detection. In *Proc. of ICSE'09*. IEEE, 2009.
54. G. Naumovich and G. S. Avrunin. A Conservative Data Flow Algorithm for Detecting All Pairs of Statements That May Happen in Parallel. In *Proc. of SIGSOFT'98*. ACM Press, 1998.
55. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.
56. Y. Nonaka, K. Ushijima, H. Serizawa, S. Murata, and J. Cheng. A Run-time Deadlock Detector for Concurrent Java Programs. In *Proc. of APSEC'01*. IEEE, 2001.
57. R. O'Callahan and J.-D. Choi. Hybrid Dynamic Data Race Detection. In *Proc. of PPoPP'03*. ACM Press, 2003.
58. B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
59. E. Pozniansky and A. Schuster. Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs. In *Proc. of PPoPP'03*. ACM Press, 2003.
60. E. Pozniansky and A. Schuster. Multirace: Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs. *Concurr. Comput. : Pract. Exper.*, 19(3):327–340, 2007.
61. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multi-threaded Programs. In *Proc. of SOSP'97*. ACM Press, 1997.
62. V. Schuppan. *Liveness Checking as Safety Checking to Find Shortest Counterexamples to Linear Time Properties*. PhD thesis, ETH Zrich, 2006.

63. W. Stallings. *Operating Systems: Internals and Design Principles (6th edition)*. Prentice Hall, 2008.
64. R. E. Strom and S. Yemini. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986.
65. K.-C. Tai. Definitions and Detection of Deadlock, Livelock, and Starvation in Concurrent Programs. In *Proc. of ICPP'94*. IEEE, 1994.
66. A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, 2007.
67. M. Vaziri, F. Tip, and J. Dolby. Associating Synchronization Constraints with Data in an Object-oriented Language. In *Proc. of POPL'06*. ACM Press, 2006.
68. W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In *Proc. of ASE'00*. IEEE, 2000.
69. C. von Praun and T. R. Gross. Object Race Detection. In *Proc. of OOPSLA'01*. ACM Press, 2001.
70. C. von Praun and T. R. Gross. Static Conflict Analysis for Multi-threaded Object-oriented Programs. In *Proc. of SIGPLAN'03*. ACM Press, 2003.
71. L. Wang and S. D. Stoller. Static Analysis of Atomicity for Programs with Non-blocking Synchronization. In *Proc. of SIGPLAN'05*. ACM Press, 2005.
72. L. Wang and S. D. Stoller. Runtime Analysis of Atomicity for Multithreaded Programs. *IEEE Trans. Softw. Eng.*, 32(2):93–110, 2006.
73. A. Williams, W. Thies, and M. D. Ernst. Static Deadlock Detection for Java Libraries. In *Proc. of ECOOP'05*. ACM Press, 2005.
74. M. Xu, R. Bodík, and M. D. Hill. A Serializability Violation Detector for Shared-memory Server Programs. *SIGPLAN Not.*, 40(6):1–14, 2005.
75. Y. Yang, A. Gringauze, D. Wu, and H. Rohde. Detecting Data Race and Atomicity Violation via Typestate-guided Static Analysis. Technical Report MSR-TR-2008-108, Microsoft Research, 2008.
76. J. Yu and S. Narayanasamy. A Case for an Interleaving Constrained Shared-memory Multi-processor. *SIGARCH Comput. Archit. News*, 37(3):325–336, 2009.
77. Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. *SIGOPS Oper. Syst. Rev.*, 39(5):221–234, 2005.
78. W. Zhang, C. Sun, and S. Lu. Conmem: Detecting Severe Concurrency Bugs Through an Effect-oriented Approach. In *Proc. of ASPLOS'10*. ACM Press, 2010.