Analysis and Sophisticated Testing of Concurrent Programs (Ph.D. thesis report)

Zdeněk Letko
iletko@fit.vutbr.cz

Faculty of Information Technology
Brno University of Technology

February 24, 2010

Abstract

The general subject of my interest is finding concurrency bugs in complex software systems written in object-oriented programming languages. Concurrent, or multi-threaded, programming has become very popular in recent years. However, as concurrent programming is far more demanding than sequential, its increased use leads to a significantly increased number of bugs that appear in commercial software due to errors in synchronization. This stimulates a more intensive research in the field of detecting of such bugs. Despite persistent effort of wide community of researchers, a satisfiable solution of this problem for common programming languages like Java does not exist. I have focused on combination of three already existing approaches: (i) dynamic analysis which is in certain cases able to precisely detect bugs along an execution path, (ii) static analysis which is able to collect various information concerning tested application, and (iii) systematic testing that helps to examine as many different execution paths as possible. Moreover, I plan to incorporate artificial intelligence algorithms into process of testing of complex concurrent software and for bugs that are hard to detect I consider development of self-healing methods that can suppress manifestation of detected bugs during execution.

Contents

Chapter 1

Introduction

Concurrent, or multi-threaded, programming has become popular. New technologies such as multi-core processors have become widely available and cheap enough to be used even in common computers. True concurrency moves to everyday life and currently popular object-oriented programming languages like Java, C++, C#, and others support concurrent programming. But, these languages put more demands on skills of programmers because it is easy to make an error when implementing concurrency. Moreover, concurrency bugs are hard to discover because of their non-deterministic nature and especially rare occurrence. Their manifestation depends on interleaving of threads.

The aim of my work is to contribute to the research in detection and healing of concurrency bugs in object-oriented programming languages. The focus is put on detection and healing of concurrency bugs in complex real-world software systems. The complexity of such products limits deep analysis of the whole system using existing techniques. Therefore, companies still exploit mainly software testing to maintain software quality. However, most of concurrency bugs are very difficult to localize just by testing. Thus, the subject of this work is to combine techniques for dynamic and static analysis of concurrent programs and systematic testing with intention to ensure certain quality of complex concurrent software with effort similar to testing. Moreover, dynamically detected bugs can be dynamically suppressed, e.g., when a bug gets into production. This leads to another goal of my work—developing of self-healing techniques for detected bugs.

In my work, I plan to focus on the Java programming language because of its clear memory model, support for various synchronization constructs, simplicity of modifications in already compiled code (bytecode), and its popularity among programmers of complex software systems. But, most of detection and healing techniques can be later modified for other languages similar to Java (C++, C#, etc.). This work extends the work I have done during my master's studies [37].

Outline. The report is organized as follows. The rest of this chapter contains introduction to concurrency bugs, an overview of techniques for their detection, and introduction to principles of self-healing. The next chapter presents the state of the art in detection and self-healing of concurrency bugs followed by a chapter presenting the goals of my research. Then, a chapter shortly presenting so-far achieved results is given and finally, general ideas for near and further future work are presented.

1.1 Concurrency Bugs

A concurrency bug can be defined as a mistake in the code that allows unwanted interleavings of threads leading to a wrong behavior of the program. This can be expressed formally as follows [18]: For a program P, let a set $I(P)$ represent all possible interleavings. Then, let $C(P) \subset I(P)$ contain all interleavings for P under which the program is *correct*. Finally, the set of all interleavings which leads to the bug can be defined as the difference of those sets $E(P) = I(P) \setminus C(P)$. The set $E(P)$ is non-empty if the program P contains one or more concurrency bugs.

There exist several taxonomies of concurrency bugs and their patterns. For instance, in [41], a Petri net model is used to describe different kinds of concurrency bugs. But the taxonomy of concurrency bugs given in [18] fits more the subject of this work, and therefore is used. Concurrency bugs (or patterns leading to concurrency bugs) are according to this taxonomy sorted into three groups: (1) code assumed to be protected, (2) interleaving assumed never to occur, and (3) blocking or dead thread. All three groups are described in more detail in the following paragraphs together with a few examples.

Code assumed to be protected. The first group of concurrency bugs is related to isolated execution of multiple instructions. A segment of code (sequence of instructions) is protected if execution of that segment cannot be disturbed by a computation done by other threads. Therefore, execution of such code segment gives the same result for any possible interleaving scenarios $I(P)$. In some sources, the code segment is then called an *atomic section* [20] and bugs in this group are called *atomicity violations*. The following bugs belong to the first class of concurrency bug patterns:

1. **Non-atomic operations assumed to be atomic**. An operation seems to be atomic (executed without interfering interleaving) but actually consists of several unprotected operations. For example, `x++;` seems to be atomic but in fact this single command consists of at least three instructions (load, increment, store) which could be interleaved.

2. **Two-Stage Access**. A sequence of operations needs to be protected but the programmer wrongly protects each operation separately. For example, consider a non trivial access to a collection in which a check whether the collection contains an item is performed and, based on the result, the operation is executed.

3. **Wrong lock or no lock**. A code segment is protected by a lock but other threads do not obtain the same lock instance when executing this segment and inference between these threads is possible. For example, a lock l is taken each time a variable v is accessed. If some thread does not acquire l before access to v, it is not synchronized with other threads.

Interleaving assumed never to occur. The second group of concurrency bugs is related to the order in which certain instructions are executed. Such bugs are also called *order-violation bugs* [42]. The cause of these bugs is that programmer wrongly assumes that certain interleavings are not possible in practice ($E(P)$ is empty) because of relative execution length of different threads, underlying hardware, or inserted instructions influencing the scheduler (e.g., `sleep()`). The following bugs belong to this class of concurrency bug patterns:

1. **Sleep does not control interleaving.** The programmer wrongly assumes that some thread t must be faster than a thread t' that contains a `sleep()` instruction. However, in some situations, the thread t' is still faster than the thread t.

2. **Losing a notification.** The bug is caused when a `notify()` statement is executed before its corresponding `wait()` statement. The notification is then missed by the receiving thread. This causes that receiving thread hangs because it waits for the missed notification. And therefore, this bug can be also listed among bugs in the third group.

Blocking or dead thread.The third group contains concurrency bugs that are related to the lifecycle of threads. In this bug category, some interleavings in $E(P)$ contains a blocking operation that blocks indefinitely (such situation is also called a *deadlock* [56]) or some interleavings in $E(P)$ contains an operation that causes that one of threads terminates unexpectedly. The following bugs belong to this class of concurrency bugs:

1. **A blocking critical section**. A thread entering the critical section is assumed to eventually exit it. For example, a thread which performs some blocking I/O operation may never exit and does not release owned lock on all paths.

2. **Orphaned thread**. When the main thread terminates abnormally, the remaining threads may continue to run. For example, a queue can be used to synchronize threads in the way that the main thread puts messages to the queue and other threads are getting the messages from the queue. If the main thread terminates abnormally without informing other threads, they can wait infinitely and block the termination of the execution.

There exist also other concurrency related problem than those mentioned above—a *data race*. The notion of data race is orthogonal to the taxonomy above. The definition of so called *(low-level) data race* [65] is as follows: A data race occurs when two concurrent threads access a shared variable and when at least one of the accesses is a write and the threads use no explicit mechanism to prevent the accesses from being simultaneous. In other words, the value of the variable over which a data race exist is not deterministic.

The reason why data races are orthogonal to the taxonomy above is that a data race is not always a bug. Moreover, many important synchronization mechanisms are based on low-level data races, e.g., a flag synchronization, barriers, or queues [59]. But, if the data race exists over a variable, a non-careful use of such variable can lead to a concurrency bug. For instance, lets look at the non-atomic operations assumed to be atomic bug. If there is a data race over a variable, there exists no explicit mechanism which prevents accesses to this variable from being simultaneous and therefore atomicity of operations can be violated. In general, it can be said that data race can cause directly or indirectly concurrency bugs from all three groups.

1.2 Bug Detection Techniques

There exist multiple approaches to detect bugs in programs including program testing, dynamic analysis, static analysis, and model checking. An ideal tech-

nique for bug detection should be *sound* and *complete*. Sound analysis does not produce so-called *false positives/false alarms* (warnings about bugs that are not real). Complete analysis does not produce so-called *false negatives* (miss to warn about a real bug). So, sound and complete analysis detects all real bugs and does not miss any bug. Let me briefly introduce all mentioned techniques.

Program testing. This is the most common way of finding bugs in programs. A programmer or tester creates a *test case* which is usually defined by inputs and corresponding outputs. If the expected outputs are not achieved or the program crashes before the output is produced, there is a bug in the program or in the test case. Program testing checks only the code along the execution path of the test case and usually does not provide information about the root cause of the bug. Program testing is accepted as unsound and incomplete technique.

Dynamic analysis. This technique also detects a bug along an execution path. But instead of checking outputs of a test, dynamic analysis automatically gathers information concerning the execution and analyzes it with an intention to discover abnormal execution conditions. Usually, an instrumentation which injects some code into original is used to gather the information. The information can be analyzed *on-the-fly*, during the execution, or *post-mortem*, after the end of the execution. Despite the analysis gathers information concerning a single execution, sometimes, if an approximation is taken into account, it can discover also bugs that are not directly on the execution path. In the best case, dynamic analysis is sound and complete with respect to an examined execution path but incomplete with respect to all possible execution paths.

Static analysis. Static analysis represents a different approach than previous two. Both techniques described above need the code to be executed and are able to observe only the code along the path of the execution. Static analysis is based on a *compile-time* analysis and it only requires code to be compilable (sometimes even this is not needed). Usually, it infers abstraction of the program behavior from the code and tries to find a bug in this abstraction. Usually, it suffers from false positives due to taken approximations. The code coverage may be total, and sometimes, the static analysis is even analyzing *dead code* which is never used along any possible execution paths (this is also source of unsoundness).

Model checking. This technique does not execute the program either, however, it requires the code to be executable. The code of the program is usually turned into a model representing the semantics of the program. Exhaustive exploring of the model state space brings the problem of state space explosion. This prevents model checking to be used on large pieces of code. Model checking provides a 100% code coverage. Model checking is sound and complete if the surrounding environment is modeled correctly.

1.3 Self-Healing

Self-healing (or simply healing) is a technique that automatically influences the program execution in such way that a detected problem does not manifest and the program executes correctly. Typical self-healing steps include:

1. **Problem detection**. Before any self-healing action can be performed, it is necessary to detect that something is wrong with the system.

2. **Problem localization**. When an incorrect behavior of the monitored system is witnessed, one has to find the root cause of the problem.

3. **Problem healing**. Applying a fix to the problem found in the localization stage.

4. **Healing assurance**. By an application of the self-healing action, the system and its behavior are modified in the hope that the problem will be resolved while no new problem will be introduced to the system. However, it is desirable to check/prove whether this goal was achieved or not.

It is important to note, that a similar approach has been around for a long time under the name of *fault tolerance*, c.f., e.g., [63]. This concept similarly as self-healing offers detection and suppressing of problems. The difference is in a way which is used to accomplish this goal. Fault tolerant techniques work with redundancy [32]. The redundancy is usually accomplished by mirroring. Detection and/or fault suppression is then implemented using comparators which compare results from duplicated parts of a system which worked on the same task.

Self-healing can be understood as a next step in fault tolerance. Self-healing capable systems perform analysis of a computational process (not only of the results) and healing is based on influencing and/or modification of this process.

Chapter 2

State of the Art

This chapter describes the state of the art in techniques for dynamic and/or static detection of concurrency bugs, self-healing of detected concurrency bugs, and an overview of testing of concurrent software.

There has been done a lot of work in the area of dynamic and/or static analysis of concurrent software in the past two decades. But as was pinpointed in a recent comprehensive study [42], most of works focus on data races, atomicity violations, or deadlocks and only little address multiple-variable bugs. Moreover, there are nearly no works focusing order-violation bugs.

Self-healing of concurrency bugs is a relatively new area of research and therefore there exist only a few works regarding the topic. In the case of systematic testing, there has been done a lot of research in the areas of systematic automatic testing but only a few works focus on applying these techniques on concurrent software.

2.1 Dynamic Analysis of Concurrent Software

Dynamic analysis is very popular for analysis of concurrent software because analysis of only one particular execution path is much easier than modeling of all possible paths and all possible interactions among threads. I divide publications mentioned in this section into three groups according to concurrency bugs they detect: (1) detection of data races, (2) detection of atomicity violations, and (3) detection of deadlocks.

2.1.1 Detection of Data Races

There are two main techniques for detection of data races. One is based on locksets and the second on happens-before relation. At first, I briefly introduce both principles and then describe tools and algorithms that use them.

The first technique is based on reasoning about so called *locksets* [65]. The lockset is defined as a set of locks that guard all accesses to a variable. Detectors then use observation that if every shared variable is protected by a lock, there is no possibility of operations on this variable being simultaneous, and therefore a race is not possible.

The happens-before based technique reasons about so called *happens-before relation* [35] (denoted $\rightarrow$) which is defined as least strict partial order on events such that: (1) If event x occur before event y in the same thread, then $x \rightarrow y$, and (2) If event x is the sender of a message and event y is the receiver of the message, then $x \rightarrow y$ after successful receive of the message. The happens-before relation is like all strict partial orders transitive, irreflexive and antisymmetric. Detectors build this relation among accesses to a variable and check that accesses cannot happen simultaneously.

Lockset-based algorithms. The first algorithm based on locksets was Eraser[65]. The algorithm maintains for each shared variable v, the set $C(v)$ of candidate locks for v. When a new variable is initialized, its candidate set $C(v)$ contains all possible locks. Eraser updates $C(v)$ by the intersection of $C(v)$ and the set of locks held by the current thread $L(t)$ when the variable is accessed. The Eraser algorithm warns about a data race if along the execution for some shared variable v the $C(v)$ becomes empty.

In order to reduce number of false alarms, Eraser introduced for each shared variable internal states *Virgin*, *Exclusive*, *Shared*, and *Shared-Modified*. When a new variable v is initialized, its state is set to *Virgin*. The variable state is changed to *Exclusive*, if v is later accessed from the thread that initialized it. If another thread access v, the state is changed to *Shared* describing that the variable v is shared among multiple threads. The state is changed to *Shared-Modified* if the variable v is shared among multiple threads and at least one thread access the variable for write. The $C(v)$ is updated only if v is in *Shared* or *Shared-Modified* states and warning concerning a data race is issued only if $C(v)$ becomes empty and the state of v is *Shared-Modified*. This modification helps to reduce false alarms but still the Eraser algorithm produces plenty of them.

The original algorithm designed for C was then modified for object-oriented languages, c.f., e.g., [73, 11, 9, 79]. The main modification (usually called *ownership model*) is inspired by the common idiom used in object oriented programs where a creator of the object is actually not the owner of the object. This idea is reflected by inserting a state *Exclusive2*. The variable is in state *Exclusive2* when it is accessed from the first thread different from thread that initialized v variable. In other words, the first owner is replaced by the thread that access variable after the first owner. This modification introduces a small possibility of having false negative [34, 73] but reduce number of false alarms caused by this object oriented attribute.

The problem of techniques based on locksets is that they do not support other synchronization than locks and therefore produce too many false alarms when applied to common concurrent software. Their advantage is in relatively low overhead.

Happens-before-based algorithms. The happens-before relation is constructed based on memory model of particular programming language. In the case of Java, memory model[39, 45, 59] induces happens-before relation in the following situations: (1) Sequence of instructions executed in the same thread induces happens-before among them according to order of their execution. (2) Releasing a lock l happens-before the following acquire of the same lock l. (3) Write to a *volatile* variable v happens-before the following read of v. (4) Instruction starting a new thread t happens-before any instruction executed in t. (5) Any instruction executed in thread t happens-before a successful *join* operation on

thread t. (6) Instruction *notify* happens-before any instruction following *wait* instruction successfully notified by the notify instruction.

Most of algorithms use so-called *vector clocks* introduced in [47] for handling happens-before relation. The idea of vector clocks is as follows. Each thread t maintains its own clock that is incremented during execution and a vector T_{vc} indexed by thread identifiers (size of the vector is equal to number of threads, one position in the vector represents thread's own clock). Each entry in T_{vc} holds a logical timestamp indicating the last event in a remote thread that could have influenced maintainer of the vector. Recently, a new variation of this algorithm was published[3]. The modification is in distributive computation of vector clocks. Each thread then maintains not a vector but a tree structure holding values of vector clocks. But, according to my knowledge there is no detector that uses this new algorithm yet.

The problem of algorithms which use vector-clocks is efficiency of handling constructed happens-before relation. This is reason why there exist only a few algorithms based purely on happens-before relation[61, 62, 22]. Algorithms[61, 62] detect data races via maintaining vector clocks for each thread C_t, each lock L_m, and two vector clocks for write W_x and read R_x operations for each shared variable x. Maintaining such a big number of vector clocks generates considerable overhead. Therefore, in [22] vector clocks for variables and locks are generated only in situations where thread vector clocks are not sufficient. This way, the overhead of vector clocks algorithm was rapidly decreased to the level of algorithms computing locksets.

The advantage of algorithms mentioned above is their preciseness in detection of data races. But big overhead generated by these algorithms forces many researches to come up with some combination of happens-before-based and lockset based algorithms. These combinations are often called *hybrid algorithms*.

Hybrid algorithms. Hybrid algorithms, c.f., e.g., [11, 57, 16, 79, 20], combine the two approaches described above. In RaceTrack [79], 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 threads current private clock value. The label is then added to the variable threadset. 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 only labels for accesses that are concurrent. Only races caused by these concurrent accesses are reported.

One of the most advanced lockset algorithms that uses notion of happens-before is Goldilocks presented in [16]. The main idea of this algorithm is that locksets can contain not only locks but also threads and volatile variables. In general, if the last action a on variable v happens-before another action b, a must be included in lockset $L(v)$ when b is going to be performed. Advantage of Goldilocks is that it allows lockset grows during computation (when happens-before relation is established on operations over v, set of locks guarding the variable is reset). The computation of basic Goldilocks algorithm is expensive but with introduction of optimizations, mainly *lazy computation* (handle thread local variables and other cases when there is no need to compute happens-before relation) and *short circuit* (a lockset is build only in the case that there is no happens-before relation between accesses in conflict and is build only addition-

ally) the algorithm has considerably lower overhead approaching in some cases lockset-based algorithms.

A quite different detection approach has been introduced in TRaDe [12] where a *topological race detection* [24] was used. This technique is based on an exact identification of objects which are reachable from the 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. The disadvantage of this solution is considerable overhead.

2.1.2 Detection of Atomicity Violations

As was explained in Section 1.1, data race is not always a bug and programs that are free of data races still can contain concurrency bugs. Therefore, researches focus on more general concurrency problem tightly coupled with data race—atomicity violations. *Atomicity* [76] is the property that every concurrent execution of a set of so-called *transactions* is equivalent to some serial execution of the same transactions. Therefore, some authors use term *serializability* [77] as synonym for atomicity despite in the area of databases and distributed computation these two terms denote distinct properties [8] and above mentioned definition of atomicity corresponds to the definition of serializability. I will continue to use the term atomicity (atomicity violation) in the following text.

The problem of atomicity is nowadays also intensively studied as a part of research devoted to *transactional memory* [36]. In systems with transactional memory, the data race problem does not exist but one has to infer set of instructions that must be executed together (in one transaction) to avoid atomicity violation. Problems related to transactional memory go behind the scope of this work and therefore I mention only algorithms related to systems without transactional memory. Algorithms described in this section can be sorted into two groups: (1) algorithms considering atomicity over one variable and (2) algorithms considering atomicity over a set of somehow connected variables.

Atomicity over one variable. Most of algorithms considering correctness of a set of accesses to a single variable are based on detecting so-called *unserialized interleavings* [77, 43, 72]. Unserialized interleavings is a set (usually very small) of accesses to a single variable from different threads that can not be transformed to some serial execution. The terminology here is not fixed yet as can be seen in the following paragraphs.

In [77], a notion of *computational units* (CU) is introduced. A CU is an approximation of a piece of code that should be executed atomically. A CU starts with the read operation on some shared variable and ends before the next read of the same variable. The end of the CU is conservative because the atomic region may have ended earlier. Interference with CUs is then checked on serializability. Checking serializability is bit complicated here and involves checking if related CUs does not overlap.

Much easier way of checking serializability was introduced in [43]. Authors introduce so-called *access interleaving invariants* (AI invariants) which reflect the idea that any of two consecutive accesses from one thread to the same shared variable should not be interleaved with an unserializable access from

another thread. Based on this observation all eight possibilities (a previous access, a current access, and an interleaving access from another thread) are discussed. Four of them are marked as unserializable. Checking is then based on simple interleaving pattern matching. AI invariants are easy to check and therefore detectors based on them are very fast. Disadvantage of this solution is that sometimes transactions span more than just two subsequent accesses to a variable.

More complicated approach has been introduced in [20, 76]. The atomicity is checked based on Lipton's reduction theorem [40] using so-called *reduction algorithm*. All instructions are classified according to their commutativity properties: *right-mover* instruction R (can be swapped with immediately following instruction), *left-mover* instruction L (can be swapped with immediately preceding instruction), *both-mover* instruction B (can be swapped with preceding or following instruction), and *non-mover* instruction N (not known to be left or right mover). Classification is based on synchronization operations, e.g., lock acquire events are right-movers, lock release events are left-movers, and race free accesses to variables (lockset-based dynamic detection algorithm is used) are both-movers. A *transaction* is considered to be equivalent to program method. A set T of transactions is atomic according to Lipton's reduction theorem if T has no potential for deadlock and each transaction in T has the form $R^*N^?L^*$. The mentioned works [20, 76] improve this principle to support reentrant locks, thread local locks, etc. Again, overhead of this approach is quite high.

Atomicity over multiple variables. Previously mentioned algorithms consider only atomicity of multiple accesses to the same variable. But, there are situations where we need to check atomicity over multiple variables. For instance, a point in a three-dimensional space is described by three coordinates x, y, z. These variables must be operated in a way that preserves *consistency* [72, 4].

In [4], the problem is referred as *high-level data race* and its detection is based on checking of so-called *view consistency*. A view is generated by a thread and consists of a set of variables that are operated together (accessed within a single method). Algorithm checks whether a set of views form a chain (can be sorted according to set inclusion) and are compatible with a current view of a thread. Algorithm has to operate with a big number of sets (each view is a set) and therefore has very big overhead.

A different approach is presented in Velodrome [23] where a graph of *transactional happens-before relation* is built (happens-before among transactions). If the graph contains a cycle, transactions involved in the cycle are unserializable and therefore atomicity violation is detected. To create a graph for the whole execution is inconvenient and therefore nodes that can not be involved in a cycle are garbage collected or even are not created. Disadvantage of Velodrome is that programmer has to annotate program to identify transactions and in some cases the overhead generated by the algorithm.

The simple idea of AI invariants described above has been enriched to support a pair of variables in [72, 26], where 11 (respectively 14) problematic interleaving scenarios was identified for a given pair of variables and detectors of these patterns were proposed.

The problem of the above described algorithms for both single or multiple variables is in inferring of set of instructions (atomic sections, transactions) that should be executed atomically. This is according to my knowledge still

considered as an open problem which is heavily studied mainly in the area of transactional memory. Some approaches tries to infer atomic sections from multiple executions of the program, i.e., [77], and others force programmers to annotate code, i.e., [23].

2.1.3 Detection of Deadlocks

Deadlocks introduced in Section 1.1 are similarly to other bugs hard to detect during testing phase but they are usually easier to analyze because threads stop their execution in an error state. Therefore, dynamic analysis focus mainly on finding *potential deadlocks* [2, 1]—deadlocks that do not occur during current execution but could occur in another execution if the scheduling changes a bit.

Detection of deadlocks usually involves graph algorithms as it is for instance in the case of algorithm introduced in [56] where a *thread-wait-for* graph (*resource allocation graph*) is dynamically constructed and analyzed for presence of cycles. Thread-wait-for graph is arc-classified digraph $G = (V, E)$ where vertexes V are threads and locks and edges E represent waiting arcs which are classified (labeled) according to synchronization mechanisms (join, notification, finalization, and monitor wait). A cycle in this graph involving at least two threads represents a deadlock.

In [27], a novel algorithm called GoodLock for detecting deadlocks was presented. The algorithm constructs *synchronization tree*-based on *runtime lock trees* and uses depth-first search to detect cycles in it. The runtime lock tree $T_t = (V, E)$ for a thread t is a tree where vertexes V are locks acquired by t and there is an edge from v_1 to v_2 when $v_1 \in V$ represents the most recently acquired lock that the thread t holds when acquiring lock $v_2 \in V$. The synchronization tree is a directed graph $G = (V, E)$ such that V contains all the nodes of all runtime lock trees, and the set E of directed edges contains: (1) Tree edges induced by runtime lock trees, and (2) so-called *inter edges*—bidirectional edges between nodes that represent the same lock and that are in different runtime lock trees. The program has potential for a deadlock if synchronization tree contains so-called *valid cycle*—a cycle that does not contain consecutive inter edges and nodes from each thread appear as at most one consecutive subsequence in the cycle.

The original GoodLock algorithm is able to detect deadlocks only between two threads. Later works [2, 7, 1] improve the algorithm to detect deadlocks among multiple threads. In [1], a support for semaphores and wait-notify constructions was added. Recent work [29] modified the original algorithm in the following three ways: (1) A lock graph is not constructed—instead the algorithm uses stack to handle so-called *lock dependency relation*. (2) The algorithm computes transitive closure of the lock dependency relation instead of performing depth first search—it uses more memory but the computation is much faster. (3) The algorithm gathers context information (thread identification and where it acquired a lock) which can be then used to identify cause of the detected deadlock.

All algorithms above can produce false alarms because they do not consider the happens-before relation among lock operations and most of them (except [1]) do not consider other synchronization primitives than locks.

2.2 Static Analysis of Concurrent Software

Static analysis is a very popular technique for detection of bugs and therefore there exist plenty of different static approaches, algorithms and techniques for analyzing programs [53]. Nondeterminism introduced by concurrency is very difficult to handle for most of existing analyzes and therefore it is subject of intensive research in few past decades. I divided static analysis techniques used for detection of concurrency bugs into the following four groups: (1) pattern-based static analyzes, (2) type system based static analyzes, (3) data-flow static analyzes, and (4) heavy-weight static analyzes. The groups are described in the following paragraphs.

Pattern-based static analysis. Searching for specific patterns in the code or in the bytecode of concurrent programs is primitive but very efficient way of finding some types of concurrency bugs. In [28], six different code patterns leading to concurrency bugs were introduced. These patterns are very simple, e.g., *wait not in loop* pattern describes situation that a `wait()` command is not enclosed by a loop checking condition. In Java 5 [39], it is possible that wait routine is interrupted unexpectedly. Checking presence of a loop around each `wait()` command is relatively simple task. It is evident, that such patterns approximate reality and can produce many false positives and negatives. For instance, described pattern omits check whether the loop really checks the condition used in the specific situation.

Type system-based static analyzes. This approach uses so-called *type systems* [53]. A type system is defined as "a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute" [60]. Formal type system provides powerful and efficient checker of correctness of the code mainly if the programming language is strongly (each variable has a deterministic type in each point of computation) and statically (types can be inferred without execution) typed. 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. Type system then checks code and search for violations of rules defined over newly defined types.

In [19, 21], a clone of classic Java called *ConcurrentJava* has been proposed. The paper presented several annotations that make ConcurrentJava race-free. For instance, each definition of a shared variable can be annotated with the annotation `guarded-by l` which express that each access to the particular variable has to be guarded by a lock l. ConcurrentJava solves problem of data races but does not address atomicity violation and deadlock problems.

Pure type analysis is powerful formal technique that is unable to fully understand threading effects of the code. The analysis also expects programmers to annotate their code what is very expensive mainly for already written code. Moreover, type analysis is usually flow insensitive (does not respect flow of control during execution) and therefore detectors based purely on this technique produce many false alarms.

Data-flow-based static analyzes. On the other hand, data-flow-based analysis respect flow of control during execution [53]. The analysis computes so-called *fact* for each statement (or basic block). A set of so-called *flow-equations* is used to pass facts along statements according to control flow of the program. Computation is usually done iteratively until a fixpoint is reached (a state where

facts do not change with further iterating). Data flow analysis can be so-called *intraprocedural* (considers control flow only within procedures) or *interprocedural* (considers control flow within procedures and among procedures).

A combination of type-based and data flow analysis has been presented in [78]. The type system presented in the paper uses so-called *typestates* introduced in [67]. Typestates extend the ordinary types defined in language. The ordinary type does not change through lifetime of the object but its typestate may be updated during the course of the computation. A typestate property can be captured by a finite state machine where the nodes represent typestates and the arcs correspond to operations that lead to state transitions. The proposed algorithm uses typestates to handle locksets and correlated variables that should be operated together (high level data race). Typestates are computed using intraprocedural data-flow analysis. The technique is able to handle relatively big programs but still produces false alarms due to unsupported synchronization mechanisms (only locks are supported) and threading effects.

Purely data-flow interprocedural static detector of data races and deadlocks called RacerX has been presented in [17]. Detection has three phases: (1) Control flow of each procedure is obtained and complement control flow graph of the whole system is constructed. (2) Data-flow analysis based on lockset algorithm is performed over the constructed graph. A datarace is reported if an access to the variable is not guarded by a lock that is mostly held during another accesses to the variable. Deadlock is reported when a locking cycle (locks are not obtained every time in the same order) is discovered. (3) Final inspection algorithm then ranks each detected bug and reported are only those that are real with a high probability. The presented approach produce many false alarms and has tremendous memory requirements because uses a lots of procedure caches gathering sets of locks.

Better results were obtained, e.g., in [51, 31] where two more static analyzes were incorporated into detecting machinery. *Alias analysis* [53] identifies a set of variables that refer to the same memory location. Detectors use so-called *may* analysis which computes over-approximation of the set of variables. The may analysis computes a set of variables that may at some point of computation refer to the same program location. Detectors can then better assume which locks are held and which objects are accessed. The second analysis that improves static detectors is called *escape analysis* [53]. The analysis identifies set of objects that are accessible in more than one thread. The set usually contains all globally accessible objects and objects that so-called *escape* thread where they were initialized. Again, usually over-approximation is used. An object escapes a thread if it is initialized in a procedure and then a pointer to the variable is returned from the procedure. Then a place of use of such object is hard to determine and therefore the object is add to the set of variables that escaped a thread.

There are plenty of works devoted to different kinds of static data race and deadlock detectors. They incorporate different alias and escape analyzes, various ways of inferring of variables that should be operated together, different kinds of ranking of detected errors and various kinds of optimizations that decrease computation and/or memory requirements of the whole machinery. But still, they can produce false alarms because of approximation they accept and because they usually rely on simple lockset-based algorithm.

Heavy-weight static analyzes. It is evident that all previously described static analyzes cannot be accurate because they do not model threads. Such analyzes are relatively cheap and can analyze hundreds of thousands of lines of code but still produce false alarms because they do not understand what is happening during multi-threaded execution. The following approaches model threads and therefore are able to reason about concurrency more accurately.

Abstract interpretation for detection of high-level data races has been presented in [75]. Abstract interpretation [53] is a static analysis technique that analyzes instructions along the control-flow and gains selected information about their semantics without performing all the calculations. The presented analysis collects information about concurrency (thread start/stop), accessed objects, and locking operations (lock/unlock). Each thread is modeled by so-called *abstract thread* identified by a place of their creation. The analysis symbolically executes the main thread first and collect threads that were created by the main thread. Then the next yet unanalyzed thread is chosen and processed. During abstract interpretation a *heap shape graph* is constructed. Heap shape graph is graph which nodes represent abstract objects (individual objects or set of aliased objects) and edges represent points-to relation induced by references. A problem is reported if inconsistency in accesses to a set of abstract objects by abstract threads is detected. The algorithm still produce many false alarms due to accepted abstractions but shows a new direction in analysis of concurrent systems.

A similar approach has been introduced in [74] where during abstract interpretation *object use graphs* are constructed. Object use graph extends heap shape graph and captures accesses from different threads to related abstract object. It can be said that object use graph approximates happens-before relation among accesses to the abstract object from different abstract threads. This solution is more precise then previously mentioned but consumes a lot of memory because to every shared object is constructed complex object use graph. Moreover, even this algorithm produces many false alarms due to approximations it accepts.

A very expensive and precise static analysis has been firstly introduced in [46]. So-called *non-concurrency* algorithm proposed in this paper constructs *may happen in parallel* (MHP) relation. The happens-before relation mentioned in previous sections can be represented as set of pairs of statements that cannot be executed in parallel. MHP is a conservative complement to this relation. In the original paper, MHP is computed in two steps. Initially, any pair of instructions can happen in parallel (MHP is total). Then, the initial set of statements are pruned using happens-before relation induced by detected happens before relation.

The original algorithm for computing MHP relation is very inefficient and therefore several modifications has been proposed in recent years. In [52], a data-flow computation of MHP has been presented. The data-flow analysis is performed over 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 synchronization actions in the code. The size of the graph increases exponentially to the size of the program and to the number of threads. Therefore, this algorithm can handle only small programs.

The data-flow computation of MHP has been then slightly improved in [6] by so-called *thread creational tree* that helps to compute rough over-approximation

of MHP relation before the data-flow evaluation is used. But still this precise analysis of concurrent programs is too expensive for common programs containing a lot of concurrency.

From the examples above, one can see that static analysis is powerful technique for checking correctness of common programs. But non-determinism introduced by multi-threaded programs present a big challenge for these techniques. Current techniques are either not precise and produce many false alarms or too expensive so they can not handle common programs. In fact, these expensive analyzes get very close to model checking.

2.3 Combinations of Static and Dynamic Analyzes

Previous sections demonstrated strong and weak sides of dynamic and static approaches. Dynamic analyzes can be very precise but results obtained by this kind of analyzes describe only particular vicinity of analyzed execution paths and overhead generated by dynamic analyzes is very important metrics for practical use. On the other hand, static analyzes are usually imprecise but are able to handle whole systems. Therefore, it is straightforward to combine these two approaches to get better analyzers. There exist plenty of works applying combination of static and dynamic analyzes (some of them were already mentioned in previous section but the combination was omitted). The goal of this section is to pinpoint basic ways used for combinations of static and dynamic analyzes. Combination scenarios are grouped according to direction of information flow.

From static analysis to dynamic analysis. Vast majority of detectors that combines static and dynamic approach use static analysis first and pass obtained results as input to dynamic analysis, c.f., e.g., [33, 2]. Usually, this combination is motivated by intention to decrease overhead generated by dynamic analysis. This is case of deadlock detector [2], where type-based static analysis is used to identify possible deadlock scenarios for a program. Based on these possible scenarios, following dynamic analysis omits to perform checks that are not necessary.

From dynamic analysis to static analysis. Less popular is other direction when dynamic analysis is used as source of information for static analyzer. For instance in recently published framework called Radar [13], a dynamic lockset-based data race detector is used to collect data concerning simultaneous accesses to a variable which are then used within classic sequential data-flow analysis. More precisely, data-flow analysis asks race detector whether there exist any concurrent write access statement to a read access statement currently being analyzed. If the answer is yes, data-flow analysis mark read statement as possibly corrupted and fact based on this statement is not spread using flow equations. This solution helped to decrease number of false alarms produced by their data-flow analysis.

Bi-directional exchange of information. There are nearly no approaches that use bi-directional flow of information between dynamic and static methods for detection of concurrency bugs. This lack was pinpointed in a recently published paper [10]. In this paper, an approach combining static and dynamic analyzes has been proposed. Static analysis uses coarse-grained analysis to

guide the dynamic analysis to concentrate on the relevant code, while dynamic analysis collects concrete runtime information during guided exploration. Proposed collaboration of static and dynamic analyzes is not only bi-directional but also iterative. The communication between static and dynamic analyzes is repeated several times during analysis. However solution described in the paper was in preliminary stage, I believe that such cooperation of static and dynamic analyzes has a big potential.

2.4 Self-Healing of Concurrent Software

Algorithms for self-healing of concurrency bugs can be grouped according to types of concurrency bugs they heal into three groups: (1) algorithms that heal data races, (2) algorithms that suppress atomicity violations, and (3) algorithms healing deadlocks. All following algorithms focus only on detection and suppressing of concurrency bugs. The algorithms do not address healing assurance which is an open problem to all of them.

Healing of data races. The idea of automatic healing of data races was probably firstly presented in our paper [34] which is described in Section 4.2. In this paper, we proposed two possibilities how to heal data races: (1) Legal influencing of the scheduler using noise injection technique, and (2) adding synchronization to force execution behaves correctly (can cause deadlock and/or considerable overhead in some cases). Problem of our solution is that detected (previously unknown) race can be successfully healed only during the next execution of the application. This is caused by the fact that we detect data race when it is already happening.

Different approach was presented in [50, 64]. The algorithm called ToleRace presented in these papers is based on ideas known from fault tolerance—specifically on mirroring. The algorithm duplicates shared data inside a critical section and so provides an illusion of atomicity when the shared data is updated. The healing is based on propagating the appropriate copy when the critical section is exited. ToleRace detects only *asymmetric races*, i.e., races caused by two threads accessing a shared variable, one that correctly acquires and releases a lock and another that does not. The lock acquiring and releasing defines the critical section.

The healing technique has to know for successful healing an atomicity of the healed application and therefore data race healers can be linked together with healers of atomicity violations described in the following paragraphs.

Healing of atomicity violations. In papers [38, 33], we presented a new algorithm called AtomRace that is able to detect and heal atomicity violations (and also data races as a special kind of atomicity violations). The algorithm uses the same healing techniques which we proposed in our previous work. Moreover, this algorithm is able to heal even atomicity violations that are about to happen. Atomicity of the application is obtained in advance using static analysis or AtomRace checks for AI invariants. The algorithm is described in more detail in Section 4.

Similar approach but focused on hardware solution was presented in [44]. The main idea presented in this paper is that a program execution is divided into so-called *chunks*. Chunks (atomic parts of execution) are chosen according to accesses to variables (based on AI invariants). Threads can be switched only

between chunks. It was shown that chunk granularity systems have a lower probability of exposing atomicity violations than instruction granularity systems. Detection algorithm then dynamically checks memory addresses which consecutive chunks access. If an address in memory is accessed by two consecutive chunks (executed in one thread), the latter one is enlarged to contain also previous access to the memory address.

The problem of healing data races and atomicity violations is that healing must preserve atomicity of the application. Inferring of correct atomicity of the application is still an open problem as was mentioned in previous sections.

Healing of deadlocks. A bit more work than in previous cases has been done on dynamic healing of deadlocks. Healing usually use analysis of simple thread-wait-for graphs. In [54], detection algorithm looks for strongly connected components (SCC) in the graph which means a potential for deadlock. These components are then during execution guarded by so-called *gate-lock*. The gate-lock for a given SCC must be acquired before any lock participating in SCC is acquired and is released after all locks participating in SCC are released. The algorithm also checks for deadlock caused by a gate-lock and when such deadlock is detected healing is canceled. Very similar idea was also presented in [81] where a term *ghost-lock* is used instead of gate-lock.

Another approach [80] tries to derive patterns from already detected deadlocks and use these patterns for detection of possible deadlocks at runtime. Patterns are described as tuple (P_A, P_W) where P_A represents a set of places in the code where threads acquire a problematic lock l and P_W represents set of places where threads are waiting for the lock l. When a deadlock pattern is detected, Java exception handling subsystem is used to analyze it and deduce related pattern. During the next run of the program, deadlock can be avoided either by adding a ghost-lock guarding problematic code regions or by dynamic changing of order in which problematic locks are acquired.

Similar pattern-based solution (patterns are called *templates*) was presented in [30] where detected deadlock patterns are healed using waiting before acquiring a problematic locks. Such healing technique avoids deadlocks but can cause *livelock*. Livelock is situation similar to deadlock, except that the states of the involved threads constantly change but no thread is progressing. Proposed algorithm is able to detect such situation and handle it similarly to deadlock—yields are put to another places in the code.

Many of mentioned approaches can similarly to previously described techniques produce false alarms and start to heal bugs that are not real. Healing assurance is in most of cases done by a prove that healing works and only some works also checks whether healing does not introduce a new bug.

2.5 Testing of Concurrent Software

Nondeterminism induced by concurrent computation is hard to test as was mentioned in the introduction. Therefore, researchers focused mainly on dynamic analysis (some authors also call it testing) and other detection techniques mentioned above. For all kinds of testing concurrency and also for dynamic analysis, it is very important to see as many different (and still legal) interleavings of program threads as possible. Approaches addressing this challenge are described in the following subsection. The second subsection briefly introduces techniques

for systematic testing of programs.

2.5.1 Increasing Coverage of Concurrent Behavior

All techniques mentioned in this subsection are based on repeated execution of the same test with the same inputs. During each execution, algorithms try to affect the scheduler with intention to see as many different interleavings as possible. Basically, there are two techniques which can achieve this goal: (1) modifying the scheduler and (2) noise injection techniques.

Modifying the scheduler. The classic Java scheduler is by design non-deterministic [59]. But if a deterministic scheduler is used, one can control interleavings among threads. An example of this approach is the tool called RaceFuzzer [66]. RaceFuzzer uses imprecise hybrid dynamic race detector to detect pairs of statements (s_1, s_2) that are probably in a race (can happen concurrently and access the same variable). The program is then reexecuted with a deterministic scheduler. The scheduler blocks the thread which reach statement s_1 and tries to let other threads to reach the corresponding statement s_2. If both s_1 and s_2 statements are reached in different threads, both statements can be executed concurrently and therefore a true race is detected. Here, the modified scheduler is used to discover interleaving that cause a true race.

A more advanced solution has been presented in the tool called Chess [49]. The tool records already seen interleavings in all previous executions and based on model checking techniques [5] choose suitable interleaving for the current execution. Because there are too many possible interleavings, the tool limits the number of thread switches that are examined only to a few most likely causing a bug.

Above mentioned techniques can be very efficient but require a modified version of the scheduler and in general the whole execution environment. This is a big disadvantage of these methods.

Noise injection. Noise injection techniques do not require any modification of the execution environment. The technique influences scheduling indirectly by injecting so-called *noise* into execution. Noise is usually generated by instructions like `yield()` and `wait()` [71]. An example of this approach is a tool called Concurrency Testing Tool (ConTest) [15, 55]. ConTest randomly or based on a chosen heuristics puts noise into the execution. The noise increases the probability of spotting a different thread interleaving. This stochastic solution combined with dynamic detection techniques works quite well even for complex systems as was shown, e.g., by us in [33, 38].

A similar approach can be found in several other recent works. For instance, in deadlock detectors presented in [29, 54], a noise is injected into code with intention to achieve interleaving leading to situation where a previously detected deadlock manifests. This way, the proposed algorithm checks whether a detected deadlock is real (similarly as RaceFuzzer checks whether the race is real). Noise injection is a cheap and promising technique for testing concurrency of complex software systems.

2.5.2 Systematic Testing

Systematic testing [14] is a style of testing which is complete under some chosen metrics. Systematic testing is opposite to incomplete or random forms of testing.

Metrics used for systematic testing are called *coverage metrics*. A test either examines a particular part of the tested system (the particular part is covered by the test) or not. Systematic testing tries to cover the whole (search) space induced by the chosen metrics.

There are several types of testing scenarios where systematic testing has been successfully applied [48]: (1) structural (white-box) testing which derives tests from the internal structure of the software under test, (2) functional (black-box) testing which tests the logical behavior of the system, (3) hybrid (gray-box) testing which combines the previous two approaches, and (4) Non-functional testing which concentrates on other properties of the tested system, e.g., on the best-case and worst-case execution times of real-time systems.

In all these areas of systematic testing, testers have to provide tests and their inputs (and in some cases expected outputs). This is a nontrivial and time consuming work. Therefore, several methods for automatic test generation have been developed, c.f., e.g., [58, 69, 68]. The Randoop approach presented in [58] constructs unit test inputs (in the form of sequences) iteratively for randomly selected methods. Unlike a pure random approach, Randoop generates inputs for the next test based on a feedback obtained from previous tests. Another approach generating unit tests has been presented in [68] where static analysis is used to mine so-called *sequences* that model sequences of method calls of the particular class under test. These sequences are then used for generating unit tests that cover all methods of the tested class.

Because the state space of possible test inputs is usually huge, different heuristics for choosing suitable inputs were proposed [48]. The heuristics can be divided into two groups: (1) search-based heuristics which get use algorithms searching for best place in the specified state space of test inputs (e.g., simulated annealing algorithm), and (2) evolutionary algorithms which use simulated evolution as a search strategy (e.g., genetic algorithms).

The state of the art in applying of different kinds of heuristics to different types of program testing has been mapped and moved ahead by recently finished European project called Evolutionary Testing for Complex Systems (EvoTest)[1]. Within this project, a new framework for Evolutionary white-box software testing [25] has been developed. The framework uses genetic algorithms to generate suitable inputs for structural tests with intention to achieve a maximal code coverage.

According to my knowledge, there are no approaches applying systematic testing and heuristics mentioned above to the field of testing of concurrent systems.

[1] http://www.evotest.eu

Chapter 3

Goals of the Thesis

The primary goal of my thesis is to propose a technique that detects concurrency bugs in complex software systems with effort similar to testing. The key idea is in suitable incorporation of dynamic and static analyzes into systematic testing approach. The primary goal can be divided into two aims: (1) to improve algorithms for detection of concurrency bugs and (2) to modify and/or develop algorithms for systematic testing of concurrency in complex software systems.

Regarding the primary aim to detect concurrency bugs, given goals can be concretized as follows:

- Analyze possible bugs in synchronization of complex software systems. Mainly focus on so far less studied synchronization mechanisms (barriers, wait-notify constructs, etc.), library constructs specified in the Java specification request 166[1] (synchronized queues, executor interface, etc.), and their usage patterns in complex software systems.
- Develop detectors of order-violation bugs which are not supported by existing approaches.
- Design a suitable combination of dynamic and static techniques for detection of chosen bugs.

Concerning the secondary aim to develop systematic testing of concurrent software, given goals can be concretized as follows:

- Design a suitable incorporation of dynamic and static analyzes of concurrency bugs into the testing process.
- Propose application of systematic testing techniques for testing of concurrency in complex software systems.
- Develop new heuristics based on artificial intelligence algorithms and/or statistics for controlling systematic testing of complex concurrent software.

Besides the primary goal of my thesis, I would like to propose new self-healing methods for bugs that can be effectively detected by dynamic analysis and hardly detected using static analysis.

[1] http://www.jcp.org/en/jsr/detail?id=166

Chapter 4

So-far Achieved Results

The results that we have achieved concerns the following three topics: (1) dynamic detection of concurrency bugs, (2) dynamic healing of detected concurrency bugs, and (3) an infrastructure for experimenting with combining artificial intelligence, dynamic analysis, and systematic testing techniques. All three results are shortly described in the following sections.

The results within the first two topics were achieved during my participation in the European research project called SHADOWS[1]. Within this project our research group cooperated with the research group led by Dr. Shmuel Ur from IBM Research Laboratories in Haifa, Israel. The cooperation continues even after the end of the SHADOWS project and the third topic is also product of our joint work.

4.1 Dynamic Detection of Concurrent Bugs

We have developed dynamic detection techniques for finding data races, atomicity violations, and mishandled notify bugs. Detectors of data races and atomicity violations were implemented and published on reputable international workshop PADTAD [34, 38]. A tool which uses proposed algorithms were released in the form of authorized software[2] and presented in the tool paper [33]. Our work on mishandled notify is still ongoing and we plan to publish it this year.

Finding data races. We have proposed and implemented two algorithms for detection of data races. The first is a modification of the well known lockset-based algorithm Eraser [65] and we call it *Eraser+*. We enrich the original algorithm with an ownership model [20, 73] described in Section 2.1 and we add a support for Java join synchronization.

In Java, if a thread t_1 calls the `join()` method of another thread t_2, it ensures that all the events of the thread t_2 are executed before the events following the `join()` call in the thread t_1. That, in fact, introduce a happens-before relation among instructions executed in t_2 and instructions executed in t_1 after successful call to `join()`. We reflect that by introducing a set $S(t)$ for each thread t such that a terminated thread t_1 and all threads in its set $S(t_1)$ are added into the set $S(t)$ after a successful join synchronization of t_1 with the thread t. Each variable

[1] http://sysrun.haifa.il.ibm.com/shadows/
[2] http://www.fit.vutbr.cz/research/groups/verifit/tools/racedetect/

v maintains a set of threads $T(v)$. A thread t is added to the set $T(v)$ when t accesses v. If a thread t is accessing a variable v and $S(t) \cup \{t\} \supseteq T(v)$, we know that the thread t is the last currently existing thread accessing v and all others have been successfully join synchronized with t. Then, the variable v changes its status back to `Exclusive2`, its $C(v)$ is set to contain all possible locks, and $T(v)$ contains only the current thread t. This way, we rapidly decreased number of false alarms produced by the original algorithm.

The second algorithm for detection data races which we call *AtomRace* detects data races as a special case of atomicity violations. As for detecting data races, it is based *directly* on the definition of a (low-level) data race which says that a data race occurs if two or more threads access a shared variable and at least one access is for writing and there is no explicit synchronization which prevent these accesses from being simultaneous. Thus, a data race can be detected by finding a situation when such an access scenario occurs.

In AtomRace, this is detected as a special case of an *atomicity violation* when atomic sections are defined simply as sequences of instructions *BeforeAccessEvent*, i, *AfterAccessEvent* where i is a read or write instruction on shared data and *BeforeAccessEvent*/*AfterAccessEvent* are special instructions that are added by instrumentation before/after i. Of course, a data race happens only when at least one of two colliding atomic sections is based on a write instruction. The probability of spotting a collision of this kind in a regular program is low, however, we exploit *noise injection* techniques [71, 15] that significantly increase this probability.

Finding atomicity violations. The atomic sections monitored by AtomRace may be extended to span more subsequent instructions on a shared variable v. Then, *BeforeAccessEvent* (entry point) and *AfterAccessEvent* (end point) can be used to determine a part of code that should be executed atomically with respect to accesses to v—atomic section as_v related to variable v. Then, we associate a (possibly empty) subset of the set $\{read, write\}$ with each atomic section as_v. This subset indicates which kind of operations *can be* performed by other threads on v while a tracked thread is running between the entry point of a given atomic section and a given end point of this atomic section.

Efficiency of AtomRace in detection of atomicity violations depends on accuracy of atomicity sections which are given as input to AtomRace. In [38], we proposed three methods which can be used to automatically infer correct and accurate atomicity sections. The so-called *pattern-based* static analysis looks for appearances of typical programming constructions that programmers usually expect to execute atomically. Another static analysis builds on the access interleaving (AI) invariants with the serializability notion from [43] which identifies triplets of accesses to v from two different threads which are not serializable and therefore problematic. We also tried to dynamically learn atomicity sections. Initially, we claim each two subsequent accesses to a variable v to construct atomic section. Then, by repeated execution of the application with AtomRace attached, we identify those atomic sections which were violated and remove them from the initial set.

Applying noise injection technique. Both algorithms (Eraser+ and AtomRace) for dynamic detection of concurrent bugs were implemented using IBM tool called ConTest [15] which supports noise injection technique introduced in Section 2.5.1. Our experiments show that suitable heuristics for noise injection increases efficiency of dynamic detection tools dramatically. There-

fore, we proposed heuristics that insert noise only into atomic sections used by AtomRace. This increases efficiency of AtomRace and keep its overhead on a reasonable level.

Detecting mishandled notify. Our work on mishandled notification is still in early stage. We proposed an algorithm which link together conditions (bool expressions) on which threads are waiting and monitors used for a wait-notify synchronization. This information helps us to identify problems in programs which use multiple conditions on a single monitor. For instance, the algorithm is able to detect situation when a thread waiting for a different condition is notified instead of another which really waits for a condition that has arisen.

4.2 Healing of Concurrent Software

We proposed and evaluated several techniques for dynamic healing of detected data races and atomicity violations. This work was published in several workshop papers [34, 38, 33]. Our techniques can be divided into two classes.

Healing by affecting the scheduler. The first class of healing techniques is based on legal affecting the scheduler. This technique is motivated by the idea of noise injection technique. Before executing a problematic part of code where a problem was detected, the currently running thread invokes for instance the `Thread.yield()` method, which causes a context switch. Next time, the thread gets an entire time window from the scheduler and so it can pass the problematic code section without an interruption with a much higher probability. This technique works only on computers having just one core CPU where interleavings of threads is dependent on context switches on CPU.

The technique can also be used in an opposite way. If some thread t is accessing a shared variable or is executing some atomic section, all other threads can detect this situation and call `Thread.yield()` or `Thread.sleep()` and allow t to finish the problematic piece of code. This technique provides much better results. However, all healing techniques based on legal influencing the scheduling do not guarantee that a detected problem will really be completely removed, but they can decrease the probability of its manifestation. These techniques do not work well for instance if the section whose atomicity is to be enforced is longer. On the other hand, due to the nature of the approach, the healing is safe from the point of view that it does not cause new, perhaps even more serious problems (such as deadlocks).

Healing by additional synchronization. The second class of self-healing techniques injects additional healing locks to the application. Every time a critical variable on which a possibility of a data race or atomicity violation was detected is accessed, the accessing thread must first lock a specially introduced lock. Such an approach guarantees that the detected problem cannot manifest anymore. However, introducing a new lock can lead to a deadlock, which can be even more dangerous for the application than the original problem. Moreover, a frequent locking can cause a significant performance drop in some cases.

On-the-fly healing. We proposed not only techniques that are able to heal already detected problems (detected in previous executions) but also a modification of AtomRace algorithm which blocks thread that is about to interleave an atomicity section in an unallowed manner. This way, we still detect an atom-

icity violation or a data race but accesses to a problematic variable are serialized and therefore problem is suppressed on-the-fly.

4.3 Systematic Testing of Concurrent Software

We have designed and partially implemented prototype of infrastructure for systematic testing which can be later used for experiments with applying search techniques and evolutionary algorithms in the field of software testing. The infrastructure is similar to one presented in [25]. This framework has been developed within European research project EvoTest and is written in C. The framework focus only on application of evolutionary algorithms in the field of structural (white-box) testing. Our framework is more general. We expect that our framework could be used for both search and evolutionary techniques and applied to any kind of testing include testing of concurrency. We also plan to interconnect the framework with other tools used for testing of Java programs (e.g., JUnit) and libraries providing data mining algorithms. We plan to publish the infrastructure this year.

Shortly, the infrastructure takes as input a set of tests of an application, a description of valid parameters of tests, and a search algorithm used for choosing a test and its parameters. The infrastructure works iteratively. Each iteration has three steps: (1) The (possibly big) state space made of tests and possible parameters is analyzed using search algorithm which identifies test and parameters to be used. (2) The chosen test is executed (dynamic and/or static analysis can be also performed) and obtained results are stored. (3) The results are analyzed and transformed to a knowledge which can be used either by search algorithm or by dynamic and/or static analysis during the next iteration.

The infrastructure has multiple scenarios of utilization in our further research. It can be used for evaluation of various search algorithms, finding tests and parameters which can be used for optimal testing of a given application with respect to various constraints, etc. I plan to utilize this infrastructure in achieving my research goal. Currently, three research groups are interested in the development of the infrastructure—our group , group of Dr. Shmuel, and people from the Center of Research on Evolution Search & Testing (CREST) at King's College London. This group was also involved in the EvoTest European project.

Chapter 5

Future Work

In accordance with the set-up goals, my future research concerns development of methods for detection of concurrency bugs and their suitable incorporation into the systematic software testing process.

In the area of dynamic detection, I am going to focus on less studied synchronization constructs and on a fairly new library `java.utli.concurrent` which provides new synchronization primitives and tools to Java developers. I would also like to address the problem of missing detection algorithm for order-violation bugs and propose new self-healing methods for chosen bugs. A work on dynamic detection of problems with mishandled-notify construction has already started and we plan to publish our results this year.

A new framework for systematic testing, which is also going to be published this year, allows me to experiment with different search heuristics, testing methods, and data analysis algorithms. I plan to continue in development of this framework and research new heuristics which can be used for efficient systematic testing of complex concurrent systems. At first, I plan to get use of recently published *concurrency coverage* metric [70] and develop a suitable heuristic for efficient insertion of noise into execution of tests with intention to get maximal concurrency coverage during the testing phase.

Later, I would like to design a new heuristics based on work done by research group of evolvable hardware at our faculty led by Dr. Sekanina (genetic algorithms) and Center of Research on Evolution Search & Testing (CREST) group at King's College London (search techniques).

I plan to perform experiments with my prototypes on complex software systems provided to me by IBM (within cooperation of our research groups) and by RedHat (within cooperation of RedHat and our faculty where I am also involved).

Bibliography

[1] Rahul Agarwal and Scott D. Stoller. Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables. In *PADTAD '06: Proceeding of the 2006 workshop on Parallel and distributed systems: testing and debugging*, pages 51–60, New York, NY, USA, 2006. ACM Press.

[2] Rahul Agarwal, Liqiang Wang, and Scott D. Stoller. Detecting potential deadlocks with static analysis and runtime monitoring. In *In Proceedings of the Parallel and Distributed Systems: Testing and Debugging (PADTAD) Track of the 2005 IBM Verification Conference*, pages 191–207. Springer-Verlag, 2005.

[3] Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte. Interval tree clocks. In *OPODIS '08: Proceedings of the 12th International Conference on Principles of Distributed Systems*, pages 259–274, Berlin, Heidelberg, 2008. Springer-Verlag.

[4] C. Artho, K. Havelund, and A. Biere. High-level data races. In *VVEIS'03: The First International Workshop on Verification and Validation of Enterprise Information Systems*, France, 2003. Angers.

[5] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series).* The MIT Press, 2008.

[6] Rajkishore Barik. Efficient computation of may-happen-in-parallel information for concurrent java programs. In *Languages and Compilers for Parallel Computing*, pages 152–169, Berlin, Heidelberg, 2006. Springer-Verlag.

[7] Saddek Bensalem and Klaus Havelund. Dynamic deadlock analysis of multithreaded programs. In *In Proceedings of the Parallel and Distributed Systems: Testing and Debugging (PADTAD) Track of the 2005 IBM Verification Conference*, pages 208–223. Springer-Verlag, 2005.

[8] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[9] Eric Bodden and Klaus Havelund. Racer: effective race detection using aspectj. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, pages 155–166, New York, NY, USA, 2008. ACM.

[10] Jun Chen and Steve MacDonald. Towards a better collaboration of static and dynamic analyses for testing concurrent programs. In *PADTAD '08: Proceedings of the 6th workshop on Parallel and distributed systems*, pages 1–9, New York, NY, USA, 2008. ACM.

[11] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. 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*, pages 258–269, New York, NY, USA, 2002. ACM Press.

[12] Mark Christiaens and Koenraad De Bosschere. Trade: Data race detection for java. In *ICCS '01: Proceedings of the International Conference on Computational Science-Part II*, pages 761–770, London, UK, 2001. Springer-Verlag.

[13] Ravi Chugh, Jan W. Voung, Ranjit Jhala, and Sorin Lerner. Dataflow analysis for concurrent programs using datarace detection. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 316–326, New York, NY, USA, 2008. ACM.

[14] Rick D. Craig and Stefan P. Jaskiel. *Systematic Software Testing*. Artech House, Inc., Norwood, MA, USA, 2002.

[15] Orit Edelstein, Eitan Farchi, Evgeny Goldin, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Framework for testing multi-threaded java programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, 2003.

[16] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: a race and transaction-aware java runtime. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 245–255, New York, NY, USA, 2007. ACM Press.

[17] Dawson Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. *SIGOPS Oper. Syst. Rev.*, 37(5):237–252, 2003.

[18] Eitan Farchi, Yarden Nir, and Shmuel Ur. Concurrent bug patterns and how to test them. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 286.2, Washington, DC, USA, 2003. IEEE Computer Society.

[19] Cormac Flanagan and Stephen N. Freund. Type-based race detection for java. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 219–232, New York, NY, USA, 2000. ACM Press.

[20] Cormac Flanagan and Stephen N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. *SIGPLAN Not.*, 39(1):256–267, 2004.

[21] Cormac Flanagan and Stephen N. Freund. Type inference against races. *Sci. Comput. Program.*, 64(1):140–165, 2007.

[22] Cormac Flanagan and Stephen N. Freund. Fasttrack: efficient and precise dynamic race detection. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 121–133, New York, NY, USA, 2009. ACM.

[23] Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. *SIGPLAN Not.*, 43(6):293–303, 2008.

[24] Eric Goubault. Geometry and concurrency: a user's guide. *Mathematical. Structures in Comp. Sci.*, 10(4):411–425, 2000.

[25] Hamilton Gross, Peter M. Kruse, Joachim Wegener, and Tanja Vos. Evolutionary white-box software test with the evotest framework: A progress report. In *ICSTW '09: Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops*, pages 111–120, Washington, DC, USA, 2009. IEEE Computer Society.

[26] Christian Hammer, Julian Dolby, Mandana Vaziri, and Frank Tip. Dynamic detection of atomic-set-serializability violations. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 231–240, New York, NY, USA, 2008. ACM.

[27] Klaus Havelund. 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*, pages 245–264, London, UK, 2000. Springer-Verlag.

[28] David Hovemeyer and William Pugh. 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.

[29] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. 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*, pages 110–120, New York, NY, USA, 2009. ACM.

[30] Horatiu Jula and George Candea. A scalable, sound, eventually-complete algorithm for deadlock immunity. pages 119–136, 2008.

[31] Vineet Kahlon, Yu Yang, Sriram Sankaranarayanan, and Aarti Gupta. Fast and accurate static data-race detection for concurrent programs. In *CAV*, pages 226–239, 2007.

[32] Israel Koren and C. Mani Krishna. *Fault Tolerant Systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[33] Bohuslav Krena, Zdenek Letko, Yarden Nir-Buchbinder, Rachel Tzoref-Brill, Shmuel Ur, and Tomás Vojnar. A concurrency testing tool and its plug-ins for dynamic analysis and runtime healing. In *RV*, pages 101–114, 2009.

[34] Bohuslav Křena, Zdeněk Letko, Rachel Tzoref, Shmuel Ur, and Tomáš Vojnar. Healing data races on-the-fly. In *PADTAD '07: Proceedings of the 2007 ACM workshop on Parallel and distributed systems: testing and debugging*, pages 54–64, New York, NY, USA, 2007. ACM.

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

[36] Jim Larus and Ravi Rajwar. *Transactional Memory (Synthesis Lectures on Computer Architecture)*. Morgan & Claypool Publishers, 2007.

[37] Zdeněk Letko. Dynamic detection and healing of data races in java. Master's thesis, FIT BUT Brno, 2008.

[38] Zdeněk Letko, Tomáš Vojnar, and Bohuslav Křena. Atomrace: Data race and atomicity violation detector and healer. In *PADTAD '08: Proceedings of the 2008 ACM workshop on Parallel and distributed systems: testing and debugging*, to appear in 2008.

[39] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[40] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[41] Brad Long and Paul Strooper. A classification of concurrency failures in java components. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 287.1, Washington, DC, USA, 2003. IEEE Computer Society.

[42] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 329–339, New York, NY, USA, 2008. ACM.

[43] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. Avio: detecting atomicity violations via access interleaving invariants. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 37–48, New York, NY, USA, 2006. ACM Press.

[44] Brandon Lucia, Joseph Devietti, Luis Ceze, and Karin Strauss. Atom-aid: Detecting and surviving atomicity violations. *IEEE Micro*, 29:73–83, 2009.

[45] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 378–391, New York, NY, USA, 2005. ACM Press.

[46] Stephen P. Masticola and Barbara G. Ryder. Non-concurrency analysis. In *PPOPP '93: Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 129–138, New York, NY, USA, 1993. ACM.

[47] Friedemann Mattern. 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.

[48] Phil McMinn. Search-based software test data generation: a survey: Research articles. *Softw. Test. Verif. Reliab.*, 14(2):105–156, 2004.

[49] M. Musuvathi, S. Qadeer, and T. Ball. Chess: A systematic testing tool for concurrent software. Technical Report MSR-TR-2007-149, Microsoft Research, 2007.

[50] Rahul Nagpaly, Karthik Pattabiramanz, Darko Kirovski, and Benjamin Zorn. Position paper - tolerace: Tolerating and detecting races. In *STMCS: Second Workshop on Software Tools for Multi-Core Systems (STMCS)*, 2007.

[51] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. *SIGPLAN Not.*, 41(6):308–319, 2006.

[52] Gleb Naumovich and George S. Avrunin. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. In *SIGSOFT '98/FSE-6: Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 24–34, New York, NY, USA, 1998. ACM.

[53] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[54] Yarden Nir-Buchbinder, Rachel Tzoref, and Shmuel Ur. Deadlocks: From exhibiting to healing. pages 104–118, 2008.

[55] Yarden Nir-Buchbinder and Shmuel Ur. Contest listeners: a concurrency-oriented infrastructure for java test and heal tools. In *SOQUA '07: Fourth international workshop on Software quality assurance*, pages 9–16, New York, NY, USA, 2007. ACM.

[56] Yusuka Nonaka, Kazuo Ushijima, Hibiki Serizawa, Shigeru Murata, and Jingde Cheng. A run-time deadlock detector for concurrent java programs. In *APSEC '01: Proceedings of the Eighth Asia-Pacific on Software Engineering Conference*, page 45, Washington, DC, USA, 2001. IEEE Computer Society.

[57] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 167–178, New York, NY, USA, 2003. ACM Press.

[58] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, Minneapolis, MN, USA, 2007. IEEE Computer Society.

[59] Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.

[60] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[61] Eli Pozniansky and Assaf Schuster. Efficient on-the-fly data race detection in multithreaded c++ programs. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 179–190, New York, NY, USA, 2003. ACM.

[62] Eli Pozniansky and Assaf Schuster. Multirace: efficient on-the-fly data race detection in multithreaded c++ programs: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(3):327–340, 2007.

[63] B. Randell, P. Lee, and P. C. Treleaven. Reliability issues in computing system design. *ACM Comput. Surv.*, 10(2):123–165, 1978.

[64] Paruj Ratanaworabhan, Martin Burtscher, Darko Kirovski, Benjamin Zorn, Rahul Nagpal, and Karthik Pattabiraman. Detecting and tolerating asymmetric races. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 173–184, New York, NY, USA, 2009. ACM.

[65] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multi-threaded programs. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 27–37, New York, NY, USA, 1997. ACM Press.

[66] Koushik Sen. Race directed random testing of concurrent programs. *SIGPLAN Not.*, 43(6):11–21, 2008.

[67] R E Strom and S Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986.

[68] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. Mseqgen: object-oriented unit-test generation via mining source code. In *ESEC/FSE '09: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 193–202, New York, NY, USA, 2009. ACM.

[69] Nikolai Tillmann and Jonathan de Halleux. Pex-white box test generation for .net. In *TAP*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer, 2008.

[70] Ehud Trainin, Yarden Nir-Buchbinder, Rachel Tzoref-Brill, Aviad Zlotnick, Shmuel Ur, and Eitan Farchi. Forcing small models of conditions on program interleaving for detection of concurrent bugs. In *PADTAD '09: Proceedings of the 7th Workshop on Parallel and Distributed Systems*, pages 1–6, New York, NY, USA, 2009. ACM.

[71] Rachel Tzoref, Shmuel Ur, and Elad Yom-Tov. Instrumenting where it hurts: an automatic concurrent debugging technique. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 27–38, New York, NY, USA, 2007. ACM.

[72] Mandana Vaziri, Frank Tip, and Julian Dolby. 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*, pages 334–345, New York, NY, USA, 2006. ACM Press.

[73] Christoph von Praun and Thomas R. Gross. Object race detection. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 70–82, New York, NY, USA, 2001. ACM Press.

[74] Christoph von Praun and Thomas R. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 115–128, New York, NY, USA, 2003. ACM.

[75] Christoph von Praun and Thomas R. Gross. Static detection of atomicity violations in object-oriented programs. *Journal of Object Technology*, 3(6):103–122, 2004.

[76] Liqiang Wang and Scott D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Trans. Softw. Eng.*, 32(2):93–110, 2006.

[77] Min Xu, Rastislav Bodík, and Mark D. Hill. A serializability violation detector for shared-memory server programs. *SIGPLAN Not.*, 40(6):1–14, 2005.

[78] 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.

[79] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. *SIGOPS Oper. Syst. Rev.*, 39(5):221–234, 2005.

[80] Fancong Zeng. Pattern-driven deadlock avoidance. In *PADTAD '09: Proceedings of the 7th Workshop on Parallel and Distributed Systems*, pages 1–8, New York, NY, USA, 2009. ACM.

[81] Fancong Zeng and Richard P. Martin. Ghost locks: Deadlock prevention for java. In *MASPLAS'04 Mid-Atlantic Student Workshop on Programming Languages and Systems*, april 2004.