VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

DYNAMIC DETECTION AND HEALING OF DATA RACES IN JAVA

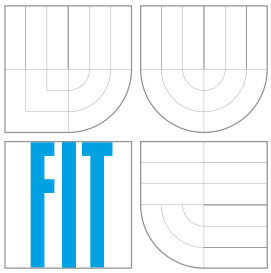
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

ZDENĚK LETKO

BRNO 2008

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

DYNAMICKÁ DETEKCE A LÉČENÍ ČASOVĚ ZÁVISLÝCH CHYB NAD DATY V PROSTŘEDÍ JAVA

DYNAMIC DETECTION AND HEALING OF DATA RACES IN JAVA

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — ZDENĚK LETKO
AUTHOR

VEDOUCÍ PRÁCE — doc. Ing. TOMÁŠ VOJNAR, Ph.D.
SUPERVISOR

BRNO 2008

Abstract

Finding concurrency bugs in complex software is difficult. As a contribution to coping with this problem the thesis proposes an architecture for a fully automated dynamic detection and healing of data races and atomicity violations in Java. Two distinct algorithms for detecting of data races are presented. One of them is a novel algorithm called AtomRace which detects data races as a special case of atomicity violations. The healing is based on suppressing a recurrence of the detected problem and can be performed by introducing an additional synchronization or by legally influencing the Java scheduler. Basically forces certain parts of the code to be executed atomically. The proposed architecture uses bytecode instrumentation to be able to track and influence the execution. The architecture and algorithms were implemented and tested on multiple case studies.

Keywords

formal verification, software testing, dynamic analysis, static analysis, data races, atomicity violation, automatic healing, multi-threaded programs, Java

Abstrakt

Hledání chyb plynoucích ze souběžného zpracovávání výpočtů je obtížné. Proto se tato diplomová práce zabývá detekcí a léčením časově závislých chyb nad daty a chyb plynoucích z nesprávné atomicity operací v prostředí Java. Práce prezentuje dva různé algoritmy pro detekci. Jedním z nich je nový algoritmus nazvaný AtomRace, který detekuje časově závislé chyby nad daty jako speciální případ nesprávné atomicity operací. Následné léčení detekovaných chyb je založeno na potlačení opakování chyby, buď zavedením přídavné synchronizace, nebo legálním ovlivňováním plánovače Javy, za účelem vynucení správné atomicity operací. Navržená architektura, která pracuje souběžně se sledovaným programem, využívá ke sledování a ovlivňování výpočtu techniku instrumentace na úrovni Java bytecode. Architektura a algoritmy byly implementovány a otestovány v několika případových studiích.

Klíčová slova

formální verifikace, testování softwaru, dynamická analýza, statická analýza, časově závislé chyby, chyby v atomicitě operací, automatické léčení, vícevláknové programy, Java

Citace

Zdeněk Letko: Dynamic Detection and Healing of Data Races in Java, diplomová práce, Brno, FIT VUT v Brně, 2008

Dynamic Detection and Healing of Data Races in Java

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana docenta Tomáše Vojnara. Další informace a pomoc mi poskytli Rachel Tzoref, Yarden Nir-Buchbinder a Shmuel Ur z IBM Research Labs Haifa v Izraeli a doktor Bohuslav Křena z Ústavu inteligentních systémů FIT VUT v Brně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Zdeněk Letko
14. května 2008

Poděkování

Děkuji vedoucímu diplomové práce docentu Tomáši Vojnarovi za jeho rady a připomínky. Velmi si vážím hlavně času, který mně a mé práci v uplynulých měsících a týdnech věnoval. Rachel Tzoref děkuji za rady a doporučení při implementaci prototypu.

Tato práce byla podpořena Evropskou unií v rámci FP6-IST projektu SHADOWS (číslo smlouvy IST-035157). Za obsah práce odpovídá pouze její autor. Tato práce nevyjadřuje názor Evropské unie a Evropská unie není odpovědná za užití jakékoliv informace v práci uvedené.

Contents

Chapter 1

Introduction

Concurrent, or multi-threaded, programming has become popular. New technologies like multi-core processors have become widely available and cheap enough to be used even in common computers. True concurrency is thus moving from computing centres to everyday life. However, this fast progress brings also some problems. Common programming languages like Java support concurrency but their design is such that programmers can easily cause concurrency bugs, for example, deadlocks, data races, lost notifications, or atomicity violations. The concurrency bugs are tricky because they usually manifest only rarely and the frequency of their manifestation depends on the conditions given by the used arthichtecture and the surrounding environment. For example, an application which works perfectly on one system starts crashing after a hardware upgrade or when being executed concurrently with some other software. The concurrency bugs can hit the correctness of outputs, performance, or even security of multi-threaded applications.

This thesis is mainly devoted to data races a common concurrency bug. The results presented in the thesis sumarize past two years of work of the author within the European research project SHADOWS[1]—"A Self-Healing Approach to Design Complex Software Systems". The project associates several European universities, technology companies, and research centres. The group at Brno University of Technolgy is involved in the research on healing of concurrency problems. Within this task, the group closely cooperates with IBM Haifa Research laboratories in Israel. Within the SHADOWS project, the working prototype implementing the lockset-based detection and healing algorithms was also tested by other partners, e.g., by Telef'onica Research and Development center in Madrid. Some of the main ideas presented in this thesis has been already published in four papers authored or co-authored by the author of the thesis [19, 23, 21, 22].

A data race arises when multiple threads are accessing a shared memory location (variable), at least one thread is changing it, and there is no synchronization between the accesses preventing them from being simultaneous. If a data race occurs, the memory location can contain a wrong value. That can cause an unpredictable behaviour of the application or even a crash. There are many examples of such problems that appeared in the real, well-tested systems, e.g., the Notrheast blackout on Thursday, August 14, 2003. The Northeast blackout was the largest blackout in North American history which affected an estimated 10 million people and caused financial losses estimated at $6 billion USD. The blackout has been caused by a data race in a grid control system.

Currently, there are several techniques which can be used to detect data races. Software

[1]Project homepage: https://sysrun.haifa.il.ibm.com/shadows/

testing is a broadly used technique but, as was mentioned above, data races show up rarely and therefore can easily be missed by common testing. This situation can be to some degree improved by combining testing with various noise generating techniques that influence the schedluing of the tested program (c.f. [5]). Various dynamic analyses may to improve common testing by not only reporting whether a bug was seen in the given testing run, but also by trying to generalise the witnessed behaviour and reporting that it was in some sense suspicious and could potentially lead to an error (c.f., e.g., [39]). But the ability of dynamic analysis to discover data races manifesting in different executions are still limited. Next, various formal methods are available. Static analyses look for data races by analysing a source code without executing it. Usually, static analyses use some kind of approximation to cope with the complexity of the code. Static, as well as dynamic, analyses often suffer from false positives (warn about data races that are not real) or false negatives (fail to warn about a data race). Yet another possibility is model checking which exhaustively searches the state space of all possible execution scenarios extracted from the application source code. Model checking can avoid false positives as well as negatives if no underapproximating abstraction is taken. On the other hand, model checking has to face the state explosion problem which limits its capabilities to be used for an analysis of an entire complex software. None of the mentioned methods can prove that there is no data race possible in large complex systems. Therefore, such systems have to be guarded against data races in a different way.

One of such possibilities can be a data race healing which can be defined as automatic suppressing of a data race manifestation. The difference between correcting a program and healing is evident. The former needs a developer who analyses the problem and designs a solution. But what if we have no developer available, e.g., because the software is already in the field and running 24/7? Then we need something which can supress the race manifestation till there will be a patch for the problem available. Moreover, what if the application cannot be stopped? Then there is only one possibility, a data race healing performed at run-time which can suppress the manifestaion of data races or at least lower the probability of their recurrence during the execution.

The aim of the thesis is to contribute to the current state of the art in the areas of both dynamic detection of data races (and also atomicity violations) and their self-healing. The thesis introduces an architecture for dynamic detection and healing of data races at run-time. Moreover, the architecture is also able to detect and heal atomicity violation problems. Such a problem occurs if the execution of some block of code is interleaved in an undesirable way with the exectuion of some other block. The arhictecture represents a modular solution for fully automated self-healing of detected data races and atomicity violations.

For the purpose of detecting data races, two algorithms were implemented. One of them was a lockset-based algorithm inspired by the Eraser algorithm [39] which tracks the locking policy used during the execution and warns if there is no consistent lock-based synchronization among accesses to some shared variable. The original algorithm was modified to support Java specific environment, and a support for the Java specific join synchronization was added. This way the number of false alarms was greatly reduced. Nevertheless, false alarms were not elimintated completely. Therefore, a novel, original algorithm named AtomRace has been designed within the thesis. The algorithm does not produce any false alarms and supports all kinds of synchronization. Moreover, it detects not only data races but also atomicity violations.

If a problem is detected, the designed architecture allows fully automated healing. For

this purpose, eight different healing strategies have been designed and implemented. The healing is based on removing conflicts from the execution. This can be done by adding a new synchronization, which can cause another and may be even more devastating problem, e.g., a deadlock, or by legal influencing of the Java scheduler. The second approach is not so effective from the point of view of suppresing the detected error but does not suffer from devastating side effects. For the purpose of healing data races (and also for detecting and healing atomicity violations) the sections of code which are assumed to be executed atomically have to be specified. This can be done manually or automatically. To obtain candidate code sections which are likely to be assumed by the programmer to execute atomically, the two techniques using static analysis has been proposed. To suppress false alarms and wrong healing caused by incorrect atomicity detection, a semi-automated atomicity prunning process has been designed.

A working prototype of the proposed architecture has been tested not only on simple examples but also on real software. The experiences obtained from a comparison of the different implemented algorithms are discussed in the closing part of this thesis.

The thesis is organised as follows. Chapter 2 introduces concurrency bugs and summarises the state of the art in the field of dynamic data race detection and healing. The Java and used tools are also presented in this chapter. The following chapter describes the proposed architecutre, including the way how a detecting and healing algorithms are intercorporated into the self-healing machinery. The used detection algorithms are introduced in Chapter 4, followed by a description of healing algorithms. Chapter 6 describes the approaches for obtaining atomic sections for the application. The prototype implementation is introduced in Chapter 7. The next chapter presents experimental results and discuss the current prototype implementation. Finally, Chapter 9 concludes and presents several ideas for the future work. There are also two appendixes at the end. The first contanins several class diagrams of the prototype implementation and the following appendix presents the commented configuration file demonstrating the possible settings of the prototype.

Chapter 2

An Overview of the Existing Technologies and Approaches

This chapter presents an overview of concurrent bugs, especially data races and atomicity violation, and techniques of detecting them. It also introduces important features of Java programming language and two tools used by the prototype implementation of the architecture for dynamic detection and healing data races and atomicity violations proposed in this thesis. The chapter is organized as follows. As the thesis is focused on problems in Java, the Java concurrency and memory model are introduced first. Next, a taxonomy of concurrent bugs is discussed. Then, data races and atomicity violations are discussed and several existing techniques and algorithms for detecting data races are presented. The next section introduces a methodology for self-healing. Finally, the IBM concurrency testing tool ConTest and the bytecode static analyser FindBugs which are used within the architecture are introduced.

2.1 Java

Java is a modern and popular object oriented programming language [24, 27, 35, 15]. The main advantage of Java is portability provided by the idea of the Java Virtual Machine (JVM)—a software layer which provides a uniform interface to the underlying platform.

For the purpose of this thesis, two aspects of Java are important and are discussed in the next sections. Firstly, the Java memory model is described and then concurrency mechanisms of Java is introduced.

2.1.1 Java Memory Model

There are two general types of variables in Java [24, 27]. *Primitive* variables are basic predefined variables that can be represented by a number (i.e. boolean, char, int). *Reference* variables can be viewed as a set of variables called *fields*. Field can be a primitive or reference variable. This enables Java to build complex structures of data. There are three main memory areas in the JVM which are used to store variables:

- **Heap**. In general, reference variable can be very complex and memory consuming. To avoid copying vast amounts of memory each time a reference variable is passed to some method, only memory address (also called reference) of the reference variable is passed. The reference variables are stored on one place called *heap* and can be

accessed only by their memory addresses. A reference variable can be an object, array, or interface. There can be many references to the same object in the heap and therefore there can be a conflict among threads accessing the object.

- **Stack**. Each thread in Java has its own *stack* which is used for storing data of methods it is executing. The stack is divided into *stack frames* representing the memory area of methods. A stack frame is local to a thread and no method can access any other methods stack frame except through the concept of passing arguments. Local variables of a method are stored in the stack frame of the method and no other threads can access them.

- **Method Area**. The *method area* is used to store code and also static variables. Static variables are the same as other variables except there is only one copy of such a variable in the JVM. This means that two threads accessing the same static variable are accessing the same memory location and therefore there could be a conflict. Static variables are also visible from anywhere in the program.

2.1.2 Concurrency in Java

The Java programming language and JVM support concurrency [35, 24, 15]. Several constructions for the concurrency exist and can be divided into a few categories:

- **Threads**. Threads represent the basic concept used in Java to achieve concurrency. Java threads exist inside the JVM which takes care of them. According to the implementation of the JVM and the environment, threads can be mapped to system threads, lightweight processes, or processes. If a thread is mapped to a system resource, it is not managed by the scheduler implemented in the JVM but by the system scheduler.

  Java provides several methods to manage lifetime cycle of threads—the most important are: `create()` (initialize and run a thread), `join()` (wait for the other thread end), `sleep()` (wait for a predefined time), and `interrupt()` (interrupts an thread). There are also a few deprecated methods which should not be used—`suspend()`, `resume()`, and `stop()`. Other synchronization mechanisms should be used instead of them.

- **Object Locking**. Mutual exclusion is another basic concept needed for concurrency. Java provides monitor based support for mutual exclusion. Each Java object has a monitor that can be used for this purpose. Such mutual exclusion is called *implicit locking*. There are two ways how to obtain a lock. One way is to use the construct `synchronized(){}` where a thread has to obtain the lock given by the object in brackets. Another way is to declare a method as `synchronized`. Then the monitor of the object encapsulating such method will be used for synchronization, and each thread has to acquire it before the method execution.

  Implicit locking requires that the guarded block of code must be sequentially coded (a lock cannot be acquired in one method and released in another). Sometimes this is a problem, and therefore Java 5 introduced a new locking mechanism based on objects provided by the `java.util.concurrent.locks` library. This package defines a more advanced locking mechanism which can also be extended by simple inheritance. The usage of these locks is called *explicit locking*.

- **Thread notification**. Sometimes, it is necessary to synchronize threads by passing a notification message between them. This can be implemented by *wait and notify* construction. A thread which invokes the method `wait()` is suspended till some other thread calls `notify()` over the waiting thread. There is a possibility for a spurious escape from the waiting state caused by a thread interruption. Because of this, another shared state variable has to be used to control the notification mechanism.

  Some other more sophisticated synchronization mechanisms have been developed using thread notification and since Java 5 are available in `java.util.concurrent` package, for instance, barrier synchronization, or synchronized queues.

- **Volatile and immutable objects**. These two minor concepts help Java to cope with some special concurrency related problems. If a variable is declared as *volatile*, Java internally disables all caching functionality for this variable. It means that every access to such a variable has to be done directly in the main memory, more precisely, on the heap. Then, each thread accessing the variable is observing the actual value of the variable. This can be used for sharing a state among threads.

  Immutable objects are objects which have all state variables declared as *final*. This prevents Java from changing the values of these variables, and therefore from changing inner state of objects consisting only from variables declared as *final*. Such objects are thread safe and can be shared or sent to another threads without any synchronization.

Nearly all concurrency mechanisms mentioned above can be used in a wrong way such that the compiler will not issue any warning and the program using them will hang or crash during the run time. The next section will try to give an overview of what and how can go wrong if a wrongly implemented concurrency is used.

2.2 Bugs in Concurrent Programs

2.2.1 Taxonomy of Concurrent Bugs

A concurrent program uses two or more threads, or processes, that cooperate on a common task and communicate using shared memory or message passing. Each thread executes a sequence of statements. A control over the execution of a set of processes and threads is introduced on several layers. At the lowest layer, it is hardware which is optimizing the execution of instructions and taking care of cache coherence. Some hardware solutions have also a native support for threads and therefore contain a kind of scheduler to control execution of threads. The biggest impact on scheduling has the operating system. The scheduler of the operating system schedules processes and threads of all programs running in the system, including threads. Moreover, in some cases, like in Java, there can also be some higher level scheduler like that used in JVM.

All the mentioned schedulers introduce non-determinism into the execution of concurrent programs. To be able to cooperate, threads have to communicate with others and need some kind of synchronization defined by the programmer. If a programmer fails to synchronize the threads properly, a concurrent bug arises. The non-determinism causes that the bug does not necessarily manifest during each run. Often, it is vice versa. The bug manifests in a low percentage of runs where the interleaving of statements executed by threads leads to a wrong behavior of the program. It is common that a wrong value does not cause a problem immediately but instead at some later point in the execution. This point can be far away from the place where the bug arose.

Possible concurrency problems are described and classified, for example, in [25] using a Petri net model of Java concurrency. However the taxonomy of concurrency bugs given in [12] fits more the subject of this work, and therefore is used. For a program P, a set $I(P)$ represents all possible interleavings. Then, $C(P)$ can be defined as the subset of $I(P)$ which contains 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 $E(P) = I(P) - C(P)$. The set $E(P)$ is non-empty if the program P contains one or more concurrency bugs listed in the following list of bug patterns taken from [12]:

1. **Nonatomic operations assumed to be atomic**. An operation seems to be atomic (executed without 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 which can 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.

4. **Double-checked locking**. When an object is initialized, the thread local copy of the object fields are initialized. Moreover not all object fields are necessarily written to the heap. For example, when a static pointer to an object escapes its constructor, any other thread can use it to access the partially initialized object.

5. **Interleaving assumed never to occur**: The programmer wrongly assumes that some interleaving is not possible. There are several examples of such bugs. For example, the programmer puts a `sleep()` statement to the thread t that should be slower than some other thread but the thread t might be quicker even with the sleep statement. Or consider so called *lost notify* bug 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.

6. **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.

7. **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.

8. **Deadlock**. A deadlock happens when there appears a cycle in the graph of resource acquisitions [1]. For example, if threads a and b need to acquire both locks $l1$ and $l2$, and thread a has locked the $l1$ and is waiting for $l2$, and thread b has locked $l2$ and is waiting for $l1$, none of these threads can make any progress.

This thesis is focused on data races which were not explicitly listed in the taxonomy. The reason is discussed in the next section.

2.2.2 Data Races

The notion of data races is orthogonal to the described bug patterns. The definition of *(low-level data race* [39] 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, if there is a data race over a variable, the value of such variable is indeterminable because of the nondeterminism in occurrence of the write operation. This can cause a wrong value being stored to the variable and possibly a wrong behavior of the program.

It is important to note that a low-level data race is not always a bug. Many important synchronization mechanisms tolerate low-level data races. For example, a flag synchronization, barriers, or queues [35]. Programmers can also regard some variable as inconsequential and allow a data race performance reasons [26]. But all these cases should be carefully considered by the programmer.

The orthogonality of data races to the bug patterns taxonomy introduced in the previous section is evident. Patterns 3, 4, and 5 listed in the previous section violate the condition of usage an explicit mechanism to prevent accesses from being simultaneous. Patterns 1, 5, 6, 7, and 8 can also be caused by a low-level data race, but only indirectly when a set of operations are performed over a problematic variable (patterns 1, 5) or when it is used for program branching and thus violates the program logic (6, 7, 8).

2.2.3 Atomicity Violation

The problem of data races is tightly coupled with the problem of *atomicity violation*. Atomicity is a property of several concurrently executed actions. The actions are atomic when their data manipulation effect is equivalent to that of a serial execution of them [44, 26, 43, 47]. The relation of atomicity violations and data races can be explained on the simple `x++;` example. Let there be a data race over the variable x. Then, there is no explicit synchronization among threads preventing the execution of this statement interleaved with another access to the variable x by another thread. The `x++;` statement consists of at least three instructions which load, increment, and store the value of x. If the interleaved operation is a write, the resulting value of x is wrong because it overwrites the value stored by the interleaved thread. The `x++;` statement has to be executed atomically.

The relation of data races and atomicity violations leads to a new definition of data races which utilizes a notion of atomicity violation. The formalized new definition published in [43] is as follows. Let Λ be a set of all locations in the code where a memory is accessed. Then, a subset $L \subseteq \Lambda$ may be designed as atomic. An event is an access to a memory location $l \in \Lambda$ and can be a read which is denoted by writing $R(l)$ or write, $W(l)$. If l denotes a location in L, $L - l$ denotes all the other locations. A *unit of work* u is a sequence of events. Units of work may be nested. In such case, the top most unit of work is considered. The notation $R_u(l)$ denotes a read belonging to u, and similarly $W_u(l)$ for writes. The

	Interleaving scenario	Description
1.	$R_u(l)W_{u'}(l)W_u(l)$	Value read is stale by the time an update is made in u.
2.	$R_u(l)W_{u'}(l)R_u(l)$	Two reads of the same location yield different values in u.
3.	$W_u(l)R_{u'}(l)W_u(l)$	An intermediate state is observed by u'
4.	$W_u(l)W_{u'}(l)R_u(l)$	Value read is not the same as the one written last in u.
5.	$W_u(l)W_{u'}(l)W_u(l)$	Value written by u' is lost.
6.	$W_u(l_1)W_{u'}(l)W_{u'}(L-l)W_u(l_2)$	Memory is left in an inconsistent state.
7.	$W_u(l_1)W_{u'}(l_2)W_u(l_2)W_{u'}(l_1)$	Memory is left in an inconsistent state.
8.	$W_u(l_1)R_{u'}(l)R_{u'}(L-l)W_u(l_2)$	State observed is inconsistent.
9.	$R_u(l_1)W_{u'}(l)W_{u'}(L-l)R_u(l_2)$	State observed is inconsistent.
10.	$R_u(l_1)W_{u'}(l_2)R_u(l_2)W_u(l_1)$	State observed is inconsistent.
11.	$W_u(l_1)R_{u'}(l_2)W_u(l_2)R_{u'}(l_1)$	State observed is inconsistent.

Table 2.1: The problematic interleaving scenarios [43].

$thread(u)$ denotes a thread which is executing the unit of work u. Then, let $l_1, l_2 \in L$, l one of l_1 or l_2, and u and u' two units of work for L, such that $thread(u) \neq thread(u')$. An execution contains a data race if it is in accordance with one of the interleaving scenarios listed in Table 2.1.

Although the work [43] consider such definition as definition of data races, this work will refer to such bugs as a kind of atomicity violation.

2.3 Detection Techniques

There exist multiple approaches to detect bugs in programs including program testing, dynamic analysis, static analysis, and model checking. Let briefly introduce all these 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.

- **Dynamic analysis**. This technique also tries to detect a bug along an execution path and in some cases also in similar paths. 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 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. A wrong assumption or approximation can cause dynamic analysis to produce *false positives* (warn about bugs that are not real), or *false negatives* (miss to warn about a real bug).

- **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. It infers the program behavior from the code and tries to find a bug in this behavior. Usually, it suffers from false positives due to taken approximations. The code coverage is total, and sometimes, the static analysis is even analysing *dead code* which is never used along any possible execution paths.

- **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. It is not only able to find a bug but it also identify the path how to get to the error state and therefore provides enough information to correct the code. Model checking does not produce any false positives or negatives if the surrounding environment is modelled correctly.

This work is focused on dynamic analysis. An overview of existing dynamic analyses used to detect data races is given in the next section.

2.4 Data Race Detection

Finding data races in programs is, as many other interesting questions about programs, an undecidable problem [31]. This work is mostly based on the dynamic approach which simplifies the problem by analysing only one particular execution and therefore there is a stress on this type of detection technique in this section. Firstly, simple low-level data race detection algorithms are presented. Then, high-level data races are described and their detection algorithms mentioned. Next, a step towards a different definition of data races already mentioned in Section 2.2.3 which uses also the notion of atomicity violation is described. Finally, some other race detection methods are briefly mentioned.

The list of data race detection tools and approaches given below is not complete. There are many more tools which were developed to fight concurrency problems in Java or in other platforms. The intention here was to list all the ones on which the results of this thesis are based or which are very close to them.

2.4.1 Detection of Low-level Data Races

The *lockset algorithm* is a dynamic algorithm based on the observation that if every shared variable is protected with a lock, there is no possibility of operations being simultaneous and therefore a race is not possible. The first such algorithm was *Eraser* [39]. The algorithm maintains for each shared variable v, the set $C(v)$ of candidate locks for v. This set contains

those locks that have protected v for the computation so far. That is, a lock l is in $C(v)$ if, in the computation up to that point, every thread that has accessed v was holding l at the moment of the access. When a new variable is initialized, its candidate set $C(v)$ contains all possible locks. When the variable is accessed, Eraser updates $C(v)$ by the intersection of $C(v)$ and the set of locks held by the current thread. If some lock l consistently protects v, it will remain in $C(v)$ till the end of the execution run. 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 takes into account that the following situations will not cause any problem despite they can be determined as data races by the algorithm given above: (1) A shared variable can be initialized without holding a lock if it becomes really shared only after its initialization. (2) A shared variable is written during the initialization only and it is read-only after. (3) Read-write locks allow multiple readers to access a shared variable but allow an access of a single writer only.

Eraser reflects these situations in the following way. As long as a variable has been accessed by a single thread only, reads and writes have no effect on the candidate set $C(v)$. Since simultaneous reads of a shared variable are not races, Eraser reports races only after an initialized variable has become write-shared by more than one thread. These assumptions lead to introducing internal states *Virgin*, *Exclusive*, *Shared*, and *Shared-Modified* for each shared variable with the following meanings:

- **Virgin**. The shared variable has not been initialized yet.
- **Exclusive**. The variable is accessed only by the thread which initialized it.
- **Shared**. The variable is read by multiple threads.
- **Shared-Modified**. The variable is read and written by multiple threads.
- **Race**. A data race on the variable has been detected (due to no or a wrong lock has been used when accessing the variable).

The transitions among the above states are described in Figure 2.1.

Much effort has been devoted to reduce the number of false alarms that Eraser produces when different synchronization mechanisms are used. An ownership model has been introduced in [45]. This 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 has been reflected by inserting a state *exclusive2* which captures the situation when a second thread (different from one which accessed variable as first) is accessing a variable. This helped to decrease the number of false alarms in the object programs but it also introduced inaccuracy because the recording of access information is delayed (the creation and update of locksets is not done until state is *shared*) and therefore races are detected after at least two accesses to the variable.

A further reduction of false alarm has been done by combining with Lamport's *happens before* relation introduced in [20]. The basic idea of the happens-before relation is that a pair of events (e_i, e_j) is related if the communication between processes allows information to be transmitted from e_i to e_j. The relation combines information about execution order and synchronization events to establish a partial temporal ordering on program statements. A data race then occurs only if the lockset is empty and if there is no established temporal ordering between two conflicting memory accesses. Happens-before can be used to enrich lockset algorithms with a support of start/join and wait/notify synchronization. Several

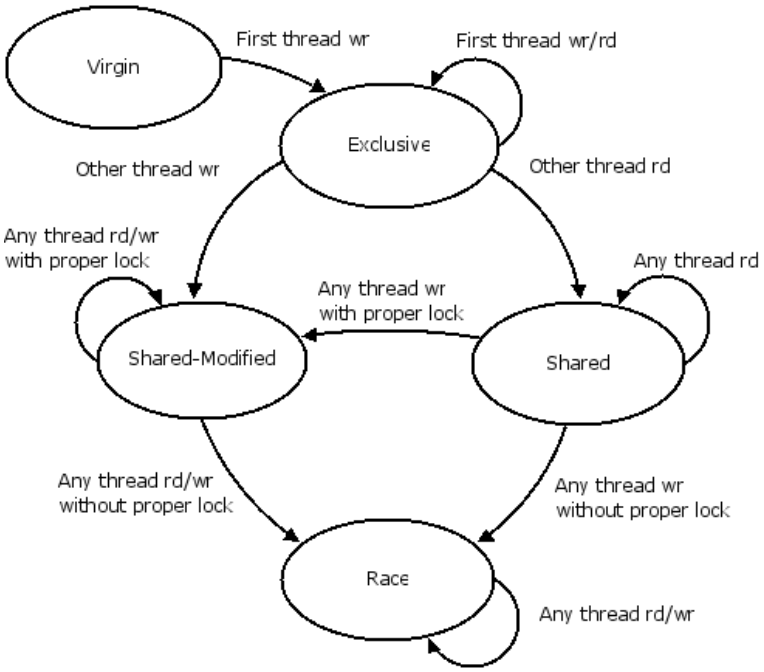


Figure 2.1: Possible states of a shared variable.

tools implemented some combination of a lockset algorithm with some kind of happens before relation [6, 34, 10, 48].

Another approach uses the principle of so-called *vector clocks* introduced in [28]. In [34], each thread maintains a vector clock indexed by thread identifiers. Each entry in the vector clock holds a logical timestamp indicating the last event in a remote thread that could have influenced maintainer of the vector. `Start()`, `join()`, `wait()`, `notify()`, and `notifyall()` are considered as messages which changes the vector clocks of sender and receiver. Then the race detection algorithm warns only about races that are not synchronized by any of the message based policies. The problem of this approach is the cost of maintaining the vectors.

In [48], a notion of a *threadset* was introduced. The threadset 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 memory accesses that are ordered before the current access. Hence the threadset contains labels for accesses that are concurrent, only races caused by these concurrent accesses are reported.

One of the most advanced lockset algorithms called Goldilocks has been presented in [10]. It also combines a lockset algorithm with happens-before relation. The happens-before relation is defined with respect to the Java memory model [27]. It considers the operations `fork`, `join`, `lock acquire`, `lock release`, and access to *volatile* variables as operations that build a happens-before relation among threads. Wait and notify operations are handled implicitly because a lock o has to be acquired before `o.wait()` or `o.notify()` are called. Thus, there is no need to consider them by happens-before rules. Previous algorithms used a state machine for each shared variable in order to handle special cases

such as thread locality, object initialization, etc.. There is no need for state machines with Goldilocks. The computation of basic Goldilocks algorithm is expensive but with introduction of optimizations, mainly *lazy computation* and *short circuit* checks. The short circuit checks handle thread local variables and other cases when there is no need to compute happens-before relation. The lazy computation means that a lockset is build only in the case that there is no happens-before relation between accesses in conflict and is build only additionally. In this way Goldilocks provides the precision of a vector clock algorithm with the overhead of simple Eraser algorithm.

A quite different detection approach has been introduced in TRaDe [7] where a *topological race detection* has been introduced. 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.

2.4.2 Detection of High-level Data Races

All previous algorithms detect only low-level data races. A high-level data races has been introduced in [3]. It is based on the observation that if there is a set of variables which are somehow related, they must be accessed consistently. For example, let have two coordinates x and y. These two variables are related because they identify a point in the space. If some thread operates with both of them separately, it is possible that another thread accessing both of the variables will get an inconsistent pair of the variables x and y. Therefore, high-level data races can be understood as an inconsistency in the granularity of variable sets associated to locks in different threads. They can be detected by *view consistency* which is introduced below.

Five concepts are crucial for definition of the view consistency—a view, a maximal view, a view overlap, a chain property, and a view compatibility. Let define them briefly. A view is generated by a thread t and consists of a set of variables that are operated together (accessed within a single method). E.g., for the previous example, a thread that operates x and y together has view $v = \{x, y\}$, and thread that operates them separately (access each variable in a distinct method) has two views $v_1 = \{x\}$ and $v_2 = \{y\}$. A view v_m generated by a thread t is a *maximal view* iff it is maximal with respect to the set inclusion in the set of all views $V(t)$ generated by the thread t, i.e., $\forall v \in V(t).((v_m \subseteq v) \rightarrow (v_m = v))$. Two views v_i and v_j *overlap* if their intersection is not empty, i.e., $v_i \cap v_j \neq \emptyset$. A set of views $v_1, ...v_n$ *forms a chain* with respect to a view v if for all pairs of views v_i, v_j, of which at least one originates from a thread that is concurrent to the originating thread of v, $((v_i \cap v) \subseteq (v_j \cap v)) \vee ((v_j \cap v) \subseteq (v_i \cap v))$ holds. Finally, a set of views $V(t)$ is *compatible* with the maximal view v_m of another thread iff all overlapping views of t with v_m form a chain, i.e., $compatible(t, v_m)$ iff $\forall v_1, v_2 \in overlap(t, v_m).(((v_1 \subseteq v_2) \vee (v_2 \subseteq v_1))$. Then, the *view consistency* represents the mutual compatibility between all threads is defined as follows: $\forall t_1 \neq t_2, v_m \in M(t_1)[compatible(t2, v_m)]$ where $M(t)$ is the set of all maximal views of a thread t.

The high-level data race is detected in the execution, if an inconsistent view between some threads is found. In other words, a thread is only allowed to use views that are compatible with the maximal views of all other threads to avoid occurrence of a high-level data race. It is evident, that dynamic detection of high-level data races is very costly. Detec-

tion of high-level data races can be also combined with the previous approaches focused on detecting low-level data races but cannot be used instead of low-level data races detection.

2.4.3 Detection of Unserializable Interleavings

The mainstream of data race detection moves towards the problem of *atomicity violation* [44, 26, 43, 47]. It is based on observation that what programmers usually want is not to be free of data races, but to preserve the correct atomicity of program considerations, or even more generally, to maintain data consistently.

A new direction in detection was inspired by previous work on *serializability*. In [14], the atomicity is considered on per method level: "A method is atomic if for every (arbitrarily interleaved) program execution, there is an equivalent execution with the same overall behavior where the atomic method is executed serially, that is, the method's execution is not interleaved with actions of other threads."

The method level is not enough accurate and therefore a notion of *computational units* (CU) was introduced in the SVD tool [47]. A CU is an approximation of a piece of code that should be executed atomically. CU's are inferred from data and control dependencies. 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.

The AVIO tool [26] introduces the so-called *access interleaving invariants* (AI invariant) which reflects 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 containing an atomicity violation and are equivalent to rows 1 to 4 in Table 2.1.

Finally, a more general detection of data races based on serializability notion has been proposed in [43], already described in Section 2.2.3. Although authors of [43] write about a new definition of data races, the author of this thesis considers such bugs as a kind of atomicity violations.

2.4.4 Some More Approaches for Data Race Detection

Of course, there are also other approaches to detect data races and atomicity violations than dynamic analysis. Numerous static analyses have been introduced to detect data races and atomicity violations. To infer violations in the synchronization, they use, e.g., primarily flow insensitive *type-based systems* [13, 36, 38] or mostly flow sensitive *static versions of lockset algorithms* [11, 30, 18]. Further, there also exists works for detecting data races using specialised *model checking* techniques—cf., e.g., [16, 9]. Let briefly look at two static anlyses proposed to detect data races.

RacerX [11] uses flowsensitive, interprocedural static analysis to track the set of locks held at any point of the execution. An automaton similar to the one in Figure 2.1 is tracked for each shared variable. As many other static analyses, RacerX has problem with false positives. This problem is partially suppressed by a subsequent ranking of the discovered races. RacerX needs to enrich the semantics of the program being analysed with information about which functions work with which locks. A user has to manually annotate each function which operates with a lock or interrupts. These annotations are then used for tracking lock and/or interruption related operations in the analysed code.

RCCJava [13] comes with an analysis based on a formal type system that is capable of capturing many common synchronization patterns. The original type system of Java is enriched by additional types and rules describing concurrency. This enriched type system is called *Concurrent Java*. The types are added to the programs by annotations which have to be added by the user, usually, about 20 annotations per 1000 lines of code. The analysis is not complete but the type system is sufficiently expressive to accommodate a majority of synchronization patterns.

2.5 Data Race Healing

Data race healing is a technique that automatically influences the program execution in such way that a detected race does not manifests and the program executes correctly. It can also be seen as hiding data races or as preventing them from introducing a wrong value into the execution of a program. Basically, the data race healing can be accomplished by forcing program execution not to cause incorrect program interleavings. The idea of healing programs on-the-fly is relatively new, and currently, there are only a few works addressing it. Although, the problem similar to healing has been around for a long time under the name of *fault tolerance*, c.f., e.g., [37].

One of the recent works addressing data race healing is ToleRace [29]. ToleRace is a software tool that increases the reliability of multi-threaded programs by "tolerating" some types of data races. ToleRace 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 as it is explained below. 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 algorithm works in three steps:

1. **Prolog**. When the critical section is entered, two copies of the shared data that the lock protects are made.

2. **Body**. Updates to the shared data are done to the local copies. The updates done by the thread that enters the critical section are done to one copy and updates done by any other threads are done to the second copy.

3. **Epilogue**. When the critical section is exited, values of the two copies and the original shared data are compared. The race is detected if the original shared data and the copy updated by other threads differ. ToleRace then can choose which value should be propagated to the shared data. But, the algorithm cannot decide which value should be propagated if all three values differ.

2.6 Self-healing Methodology for Concurrent Bugs

The methodology for self-healing of concurrent bugs has been proposed within the SHADOWS project and published in [19]. The basic loop of any self-healing approach should consist of the following steps:

1. **Problem detection**. Before any self-healing action can be performed, it is necessary to detect that something is wrong with the system. The core of this step is in monitoring the execution of the program.

2. **Problem localization**. When some possibility of an incorrect behavior of the monitored system is detected, it is needed to get to the root cause of the problem. Locating the source of the detected problem is often a hard task even for humans.

3. **Problem healing**. If the cause of the detected problem is found, a healing action is selected from the the list of available healing actions and is applied. The list of healing actions is prepared in advance based on the experiences with bugs.

4. **Healing assurance**. By an application of the self-healing action, the system and its behavior is changed in hope that the problem will be resolved. However, it is desirable to check whether this goal was achieved or not. To solve the problem, a suitable formal verification technique (like model checking or static analysis) can be used.

This work describes the application of the methodology on self-healing data races and atomicity violations. An architecture based on this methodology was implemented and is described in the next chapters of this work.

2.7 Concurrency Testing Tool IBM ConTest

ConTest [5, 8, 33] is an automated tool for testing concurrent software. The main objective of ConTest is to make concurrency related bugs materialize while having a minimal impact on the software performance. ConTest for Java provides three essential services: instrumentation, heuristic noise injection, and a listeners architecture.

The ConTest *instrumentation* works on the level of Java bytecode. An instrumentator adds calls of ConTest methods. Therefore, ConTest is called during an execution of the instrumented software. This solution enables ConTest to observe and influence the behavior of the tested software. Places where ConTest is called are chosen from the perspective of tracking and influencing the execution with respect to concurrency. These places are also called *instrumentation points*:

- **An access to a potentially shared variable**. A variable is potentially shared if it is non-local with respect to methods. The potentially shared variables are *static* variables or *fields*.

- **An access to an array item**. Arrays in Java are managed in a different way than other objects and this has to be reflected. It is not easy to distinguish between a potentially shared array item and an array item local to a method, and so all accesses to array items are instrumented.

- **Method and basic block entry actions**: A method entry is the first location in a method. A basic block is a set of instructions without any branching point in the middle.

- **Method exit actions**. A method can be exited in three ways—by an execution of the last instruction, by an execution of a return instruction, or by throwing unhandled exception.

- **Monitor enter and exit actions**: Monitor enter leads to obtaining an implicit lock and exit leads to releasing an implicit lock.

- **Thread begin, end, interrupt, and join actions**. These operations cover all important events in the thread life cycle.

- **Wait and notify actions**. These operations are used for the message passing synchronization between threads.

Some ConTest instrumentation points are associated with bytecode instructions. For example, monitor enter action is associated with the presence of a `monitorenter` instruction. Whenever the instrumentator finds this instruction in the bytecode, it injects calls of ConTest methods just before the `monitorenter` instruction and also just after it. Some instrumentation points are associated with some method invocation. For example, thread join action is associated with the presence of invoking the `Thread.join()` method.

There is an example of a code instrumented by ConTest in Figure 2.2. It shows that a few lines of the source code Figure 2.2(a) are transformed to several bytecode instructions Figure 2.2(c). In this case, two integer variables are summed and the sum is returned. On the level of bytecode, two integer values (one local loaded by `iload` instruction and one non-local loaded by `getfield` instruction) are placed on the stack and summed as can be seen from Figure 2.2(b). Although, the instrumented bytecode looks complicated (c), it only adds a few calls to ConTest methods with predefined parameters. For example, there is a call of ConTest method `beforeInstanceRead` on lines 11-22. This method takes four parameters. The instance of the class which is on the top of the stack due to call to `aload` instruction at line 11 and three parameters of type `String` represented by references to *Java constant pool*. Each reference is loaded by the `ldc` instruction. In the figure, they are printed with their indexes and values in comments for a better understanding. The first string (`InstrumentExample.var`) is ConTest *identifier* for the variable which is going to be read. The second string (`int`) represents the type of the variable, and the third string (`InstrumentExample.java sum(int) 6 2`) represents the *program location* by a unique identifier of its position in the code. The method returns a reference to the object which points to the instance of the class. Because the method is generic for all types of instances, the returned object has to be casted to the right type by calling `checkcast`. The situation is similar with other ConTest methods. Figure 2.2 also shows that some operations are instrumented only once (e.g., line 0 indicating a method entry) and some in two different places—before and after the operation (e.g., line 25 indicating a read operation of non-local variable). Information received by ConTest methods is also available to applications using ConTest listeners architecture.

Noise injection [5] is a technique that forces different legal interleavings for each execution of a test in order to check that the test continues to perform correctly regardless of the behavior of the scheduler. In a sense, it simulates the behavior of other possible schedulers. When a call of ConTest from the execution of instrumented software is received, the noise heuristic decides, randomly or based on a specific bug-finding technique, if some kind of delay is needed. This implies that noise injection can be done only at instrumentation points. As was mentioned before, concurrency bugs depended on thread interleaving. The noise helps changing the interleaving so that it increases the probability of finding a bug. For example, if there is a data race over shared variable x combined with a statement `x++;` which should be done atomically, the noise injected between read of x and consequent write of the incremented value increases the probability that some other thread which in conflict interleave these two access operations.

Finally, ConTest contains a *unified listeners architecture* that provides an easy to use interface to the execution trace of the instrumented software. Tools of third parties can register Java listeners to specific events. If a listener (or a set of listeners) is registered to an event, the code of the listener (or listeners) is executed when the event happens. Events are given by instrumentation points. Some of these points provide two events—

```
class InstrumentExample{
   private int var;

   public int sum(int a){
      return a~+ var;
   }
   ...
}
```

(a)

```
public int sum(int);
 Code:
   0:   iload_1
   1:   aload_0
   2:   getfield   #2; //Field var:I
   5:   iadd
   6:   ireturn
```

(b)

```
public int sum(int);
  Code:
   0: ldc          #35; //String InstrumentExample.java sum(int) 6 1
   2: invokestatic #26; //Method methodEntry:(String;)V
   5: ldc          #37; //String InstrumentExample.java sum(int) 6-6 1
   7: invokestatic #33; //Method basicBlockEntry:(String;)V
   10: iload_1
   11: aload_0
   12: ldc          #39; //String InstrumentExample.var
   14: ldc          #41; //String int
   16: ldc          #43; //String InstrumentExample.java sum(int) 6 2
   18: invokestatic #48; //Method beforeInstanceRead
                            (Object;String;String;String;)Object;
   21: dup
   22: checkcast     #3; //class InstrumentExample
   25: getfield      #2; //Field var:I
   28: dup_x1
   29: ldc          #39; //String InstrumentExample.var
   31: ldc          #43; //String InstrumentExample.java sum(int) 6 2
   33: invokestatic #53; //Method afterInstanceRead_int:
                            (Object;String;String;)V
   36: iadd
   37: ireturn
```

(c)

Figure 2.2: The source code of a method (a), its bytecode (b), and the ConTest instrumented bytecode (c).

so-called *before* and *after events*. The before action events allow listeners to be executed before an action is performed. The after action events allow listeners to be executed after the action is performed. This allows tools using the ConTest listeners architecture to have enough control over the instrumented software. Some instrumentation points (start and end of threads, method entry and basic block entry) provides only one event because there is technical or performance problem with providing both of them.

The ConTest infrastructure can be used by classical testing tools leading to a higher probability of revealing a concurrent bug. Another possibility is to develop specialised tools for detecting particular classes of concurrent bugs.

2.8 Bytecode Static Analysis Tool FindBugs

FindBugs [17, 4] is an open source static analysis tool that analyzes Java bytecode. It looks for the so called *bug patterns* [2]. Bug patterns in FindBugs are some sequences of operations in the Java bytecode which can cause a wrong behavior of the analysed program.

FindBugs has a plug-in architecture which allows to define a new analysis detecting a chosen pattern. Such a plug-in is called a *detector*. Rather than using a specialized pattern language for describing bugs, FindBugs uses Java. This allows the user to use a variety of techniques to implement a detector. Simple detectors use a *visitor design pattern* over the analysed classfiles, methods, and instructions. Detectors often implement a state machine which detects an occurrence of a bug pattern when the state machine reaches an accepting state. The transitions of such a state machine are based on observing instruction consequences, types, constant values, annotations, or attributes of methods or fields. More sophisticated detectors can also traverse the control flow graph and use information gathered by some previously executed dataflow analyses. Recently, a support for writing new dataflow analyses was added too.

The principle of adding a new detector is simple. Usually, it is enough to extend some provided detector class implementing important interfaces and change the content of several methods. The implementation of a new dataflow analysis is a bit more complicated. A developer has to develop a *fact* which is used as data flow value by the new dataflow analysis. Then, the implementation of the algorithm of the analysis is again based on extending some predefined classes. Finally, the new analysis has to be registered into the so-called *analysis cache* which maintains analyses to be used on the analysed code. This way, FindBugs knows which analysis to execute when an request for the dataflow analysis fact comes from a detector.

Dataflow analyses in FindBugs cannot perform interprocedural context sensitive analysis, yet. However, detectors can use global information and results of previously executed detectors. Therefore, it is possible to make a detector which detects a bug using some summary information.

FindBugs also provides a reporting architecture which allows to uniformly produce warnings concerning the identified bugs. Besides FindBugs, several tools for processing bug reports are available.

A problem of some implemented bug detectors is a presence of both false negatives and positives. FindBugs uses a way of tolerating some false negatives and incorporate a human expert into the process of removing false positives. A graphical user interface for observing the source code and the produced warnings is provided and experts can flag warnings with several kinds of messages, e.g.,*not a bug*.

Chapter 3

An Architecture for Healing Data Races On-the-fly

This chapter presents an architecture for self-healing which is based on the methodology introduced in Section 2.6. The chapter provides an overview of how a different approaches described in the following chapters fit together. Firstly, an overview is given and then each component is described in more detail. Next, an execution monitoring component which is used by the architecture to monitor the execution is introduced.

3.1 A Self-healing Architecture

The proposed architecture is depicted in Figure 3.1. The main part of the architecture consists of three modules implementing the first three steps of the self healing methodology introduced in Section 2.6. The figure also shows an offline part of the architecture denoted by dashed arrows. This part of the architecture is used to preprocess some important information concerning the managed application. The collected information is then available to detection and healing algorithms during the execution.

The *execution monitoring* module watches the managed program and triggers predefined events during the execution. An *event* is defined as an operation performed by the managed application, e.g., an access to some variable. A bytecode instrumentation of the managed application introduced in Section 2.7 is used for this purpose. Some more concrete information concerning an event, e.g., variable instance or thread identification, are not known during the instrumentation and have to be computed on-the-fly during the event processing in the execution monitoring module. In general, it can be said that the module transforms a particular execution of the managed application into streams of events—one stream for each application thread.

Monitored events are then passed to the *analysis engine* which uses a detection algorithm to identify if the incoming event is legal for the predefined rules of a correct program behavior. A problem is detected if some rule was violated. Currently, a two different algorithms for detecting data races and atomicity violations are implemented. They are described in more detail in Chapter 4.

Finally, the *healing logic* can react on a problem detected by the analysis engine and influence the execution with an intention to prevent the problem manifestation or recurrence. The influencing of the execution can be done by a safe but not very efficient influencing of a scheduler or by more effective but potentially dangerous introducing an additional

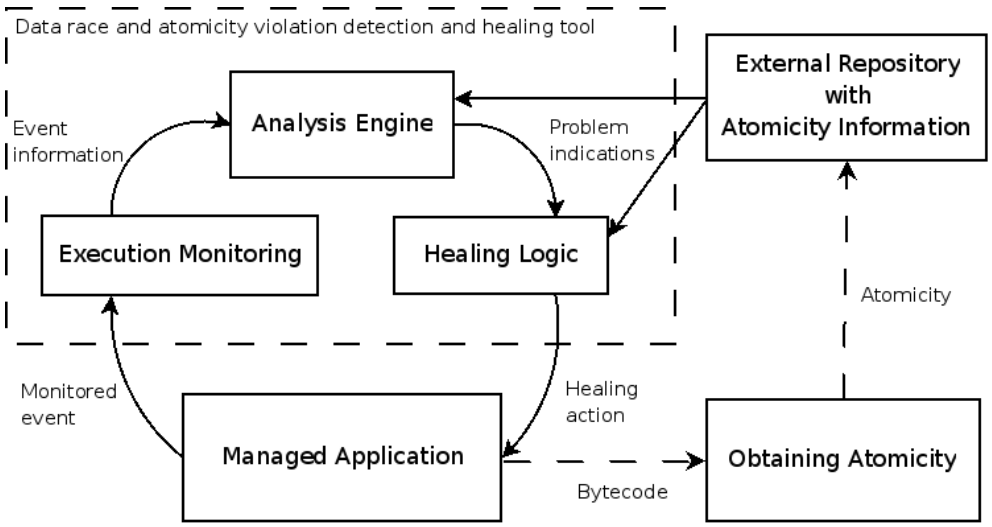


Figure 3.1: The proposed architecture for healing data races on-the-fly.

synchronization. Such influencing is called *healing* and is described in Chapter 5.

In general, an effective healing can not be done without any knowledge about the managed application. Therefore, the healing logic can use the *external repository* to access preprocessed information concerning the managed application. In the case of the proposed architecture, the information which is obtained in advance and stored in the external repository is the *atomicity* of the application introduced in Section 2.2.3.

The atomicity is a property of a source code and can be obtained by some analysis or from the user in advance. The automated obtaining atomicity information is done in the module called *obtaining atomicity*. Because the process of obtaining atomicity information is done before the execution, a more demanding approaches can be used. The *healing assurance* introduced in Section 5.5 can be also performed on this inferred set of atomic sections before the application is executed.

3.2 Monitoring the Java Execution

All dynamic race detection algorithms are based on tracking accesses to variables along the execution path and most of them need to track also synchronization events. The monitoring engine has to provide the analysis engine with all the necessary information needed for the detection and/or healing. The following section introduces identifiers obtained by the monitoring engine and used by the other modules of the proposed architecture. Next, events tracked by the monitoring engine are described. The following chapters introduce algorithms for detection and healing which are based on events defined in the following subsections.

3.2.1 Identifiers of Monitored Objects

The execution of the monitored application is transformed by the monitoring engine into streams of events parametrized by the following identifiers:

- **Thread identification**. In the concurrent environment, it is very important to identify the thread which reaches an event.
- **Object identification**. Java is object oriented language and as such, it uses objects. During an execution, Java creates and destroys many objects and it is necessary to distinguish two different objects which encapsulates two different state information.
- **Variable identification**. The crucial point for race detection and healing is the deterministic identification of variables. In Java, an object which encapsulates a shared variable has to be identified as was mentioned above. Then, a memory location where the shared variable is stored has to be distinguished. In the case of variables declared as `static`, there is no encapsulating object. The situation is also different with array cells which are in Java encapsulated by an array object. In the case of array cells, the deterministic identifier has to contain the object encapsulating the array, the array object, and the cell identifier. Moreover, multidimensional arrays consist of a structure of one-dimensional arrays what lengthen the chain of encapsulating objects which should be part of the identifier.
- **Event identification**. A two groups of events can be identified. The *operation-based* events are determined by the particular bytecode instruction, e.g., the `monitorenter` instruction determines a lock acquiring event. The *location-based* events are determined by a location of the program counter, e.g., program counter pointed to the first instruction in a method determines the method entry event.
- **Location in the code identification**. The detection engine should be able to identify places where a problem was detected in order to point the developer and/or healing logic there. This is a similar problem which solve debuggers and/or compilers when producing warnings.

3.2.2 Monitored Events

Events are triggered at locations where an instrumentation process injected the code necessary for tracking into the original bytecode. They can also be understand as a special instructions which are executed among other bytecode instructions of the managed application. Each event consists of an event identification and can contain also identifiers of objects the event is related to. For the purpose of the proposed architecture, an event identification is determined by the name of an event and by its parameters. There are nine events tracked by the execution monitoring module:

- $beforeAccessEvent(v, loc)$ This event is invoked just before each instruction accessing a shared variable (field in Java) or an shared array cell. The v denotes the deterministic identifier of the variable and the loc denotes the deterministic identifier of the location of instruction accessing v.
- $afterAccessEvent(v, loc)$ This event is invoked just after each instruction accessing a shared variable or an shared array cell. Similarly to previous event, the v denotes the shared variable and the loc denotes the location of instruction accessing v.
- $beforeMonitorEnter(o)$ This event is invoked just before each instruction which tries to obtain a lock. The o denotes the object whose monitor is used for locking.

- $afterMonitorExit(o)$ This event is invoked just after each instruction which releases a lock. The o again denotes the object whose monitor is used for locking.
- $afterThreadBegin$ This event occurs shortly after a new thread started its execution.
- $beforeThreadEnd$ This event occurs shortly before a thread finishes its execution.
- $afterThredJoinEvent(t)$ This event is invoked just after an `Thread.join()` method. The t denotes the thread over which the join operation was called.
- $methodEntryEvent(loc)$ This event occurs before the first instruction of a method is executed. The loc denotes the location of the first instruction in the method.
- $methodExitEvent(loc)$ This event occurs after the last instruction of a method is executed. The loc denotes the location of the last instruction executed in the method or a special value. The special value is used if the method is ending unexpectedly because an unhandled exception was produced within the method.

The identification of the thread executing an event is obtained from the JVM during the execution by the monitoring engine. Such thread is denoted $t_{current}$ in the following chapters. Data race detection also needs information if a shared variable is accessed for reading or writing. For a simpler description of algorithms in the following chapters, there are no different events for reading and writing. Instead, a function $getMode : Loc \rightarrow \{read, write, null\}$ is used. Each access instruction has a unique location identifier $loc \in Loc$ which identifies a bytecode instruction and therefore it is possible to get the instruction access mode from the location loc. A special value $null$ is introduced to be used when $getMode(loc)$ function is called on location which does not correspond to an instruction accessing a shared variable.

Moreover, the monitoring engine should consider some special issues of Java. Events related to variables defined as `volatile` and *final*[35] should not be monitored. Variables defined `volatile` are designed and intended to be used in cases where data races are tolerable. Variables defined `final` cannot be written during the execution. Therefore, there is no possibility for a data race over such variables.

Chapter 4

Data Race Detection

The detection engine represents a core of the proposed architecture. The algorithm used in the detection engine is a key to proper function of a healing mechanism. If the detection algorithm suffers from *false alarms* also called *false positives*, the healing mechanism starts to influent the execution unnecessarily. And if the detection algorithm suffers from *false negatives*, the healing mechanism does not start healing and the race is not suppressed.

This chapter contains a description of two distinct algorithms which can be used in the analysis engine to detect data races during an execution. The first algorithm is a *lockset-based algorithm* modified to work with Java and enriched with a support of *join synchronization* and *omitted lock suggestion* mechanisms. The algorithm is inspired by the *Eraser algorithm* [39] described in Section 2.4. This algorithm is refered as *Eraser+ algorithm* in the following text.

The second algorithm is a novel algorithm specifically designed for the self-healing approach. The algorithm, called *AtomRace*, detects data races as a special kind of atomicity violations. More precisely, data races are detected as a special case of atomicity violations on atomic sections specially defined to span just particular read/write instructions and the transfer of control to and from them. Further, AtomRace can also be applied to detect atomicity violations on more general atomic sections than those used for the data race detection. They can be defined by the user or obtained by some static analysis. Although the algorithm was invented to be used for a self-healing, a proposed method based on combining the algorithm with a specific noise injection technique allows the algorithm to be used also for testing (or bug hunting).

The algorithm is not only capable of detecting data races but also violations of the given atomicity of the managed application. Although the algorithm was invented to be used for a self-healing, a method based on combining the algorithm with a noise injection technique how the algorithm can be used also for testing is also presented. Further, AtomRace can also be applied to detect atomicity violations on more general atomic sections than those used for the data race detection. They can be defined by the user or obtained by some static analysis.

This chapter is organised as follows. The first section describes the Eraser+ algorithm and all the proposed modifications which were applied. The second section introduces the AtomRace algorithm and the way how the algorithm detects data races and atomicity violations.

4.1 Eraser+: Lockset-based Data Race Detector

This section describes a modification of an lockset based algorithm. Firstly, a lockset-based approach is discussed and then a way how the algorithm is incorporated into the proposed architecture is given. Next, the algorithm is described and two extensions are introduced.

Lockset-based algorithms are based on the consideration that every shared variable should be protected by a lock. Since the algorithm has no way of knowing which locks are intended to protect which variables, it must deduce the protection relation from the execution history. In fact, the algorithm does not detect data races but so called *race conditions*. The algorithm issues a warning about a data race whenever it assumes that there is no lock used for synchronization of threads accessing the problematic variable. However, this does not necessarily imply that there is a true race because, for example, an another kind of synchronization can be used to protect the variable. Therefore, the algorithm works fine if a locking based synchronization is used. Whenever another synchronization is involved, it is not able to reflect it and and the algorithm produces false positives if there is no support for a happens-before relation. The fact, that lockset-based algorithms detect race conditions and not directly data races allows the algorithm to detect even those data races which is not seen during the execution. This is valuable when such an algorithm is used during the testing phase where the intention is to find as many problems as possible.

The proposed Eraser+ algorithm is Incorporated into the architecture using event based interface introduced in the previous chapter. The algorithm needs to maintain a set of locks currently hold by a thread. Therefore, a set $Locks(t)$ of locks currently owned by a thread t is maintained. An object o which is used as a lock is added to the $Locks(t)$ if it is acquired by the thread t. Then, the object o is removed from the $Locks(t)$ if it is released by the thread t. The Eraser+ algorithm supports a kind of join synchronization introduced in Section 4.4. Due to this extension, the algorithm has to maintain also a set of join synchronized threads $Join(t)$ for each thread t. Thread t' is added to $Joined(t)$ of thread t after a successful join method called on t' and all threads from $Joined(t')$ are also moved to $Joined(t)$. Therefore, a $Joined(t)$ contains the transitive closure of threads synchronized by the join operation. The intercorporation of the Eraser+ algorithm into the event based approach and the way how the sets introduced above are maintained is shown in Figure 4.1.

The data race detecting algorithm Eraser+ is performed within the function *isRace* depicted in Figure 4.1. As parameters, it takes the current variable v being accessed at location *loc*, a set of locks $Locks(t)$ currently holded by the thread t, and a set of join synchronized thread $Joined(t)$. If the algorithm detect a race condition, the *isRace* function returns *true*. In such a case, the variable v is added to the *RaceDetected* set which can be then used, for example, by a healing logic as it is described in Section 5.2.

4.1.1 Maintaining Locksets

The original Eraser algorithm described in Section 2.4 was proposed for C. In Java, it is needed to capture situations when the variable is initialized by one thread executing a constructor and then used by a different thread. The original Eraser algorithm often produces false alarms in a such scenario. Therefore, the ownership model also mentioned in Section 2.4 was used to prevent such false alarms and a new state *Exclusive2* was added to the finite automaton used by the algorithm. A modified transition system used by the Eraser+ algorithm can be seen in Figure 4.2. The automaton in the figure describes the internal state of each shared variable.

Initialisation:
$\forall t \in Threads : Locks(t) = \emptyset$, $Joined(t) = \emptyset$
$RaceDetected = \emptyset$

Computation:
switch $(Event)$ {
case : beforeAccessEvent(v, loc)
    if $(isRace(v, loc, Locks(t_{current}), Joined(t_{current}))$ **then**
        add v **to** $RaceDetected$ // RACE DETECTED

case : beforeMonitorEnterEvent(o)
    add o **to** $Locks(t_{current})$

case : afterMonitorExitEvent(o)
    remove o **from** $Locks(t_{current})$

case : afterThreadJoinEvent(t)
    add t **to** $Joined(t_{current})$
    addall $Joined(t)$ **to** $Joined(t_{current})$
}

Figure 4.1: An intercorporation of the Eraser+ algorithm into the proposed architecture.

The meaning of shared variable internal states used in Figure 4.2 is as follows:

- **Virgin**. The shared variable has not been initialised yet.
- **Exclusive**. The variable is accessed only by the thread which initialised it. Such a thread is also called the first *owner*.
- **Exclusive2**. The variable has changed its owner and is accessed only by the new owner. From this state, the automaton is similar to the automaton used by the original Eraser.
- **Shared**. The variable is read by multiple threads.
- **Shared-Modified**. The variable is read and written by multiple correctly synchronized threads.
- **Race**. A data race on this variable has been detected (due to no or a wrong lock has been used when accessing the variable).

An algorithm that implements the given automaton is shown in Figure 4.3. The algorithm uses three internal state variables for each shared variable v. The $state(v)$ contains the information in which state the finite automaton for the particular variable v is. The $owner(v)$ contains a thread t that currently own the variable v. Finally, the set $Candidate(v)$ of candidate locks for the variable v. This set contains those locks that have protected v during the computation so far. The algorithm takes as its input the shared variable identifier v, the set of locks $Locks(t)$ currently held by the thread t currently accessing v and the identifier of the current location loc of instruction currently accessing v. The algorithm returns $true$ if the race has been detected otherwise returns $false$.

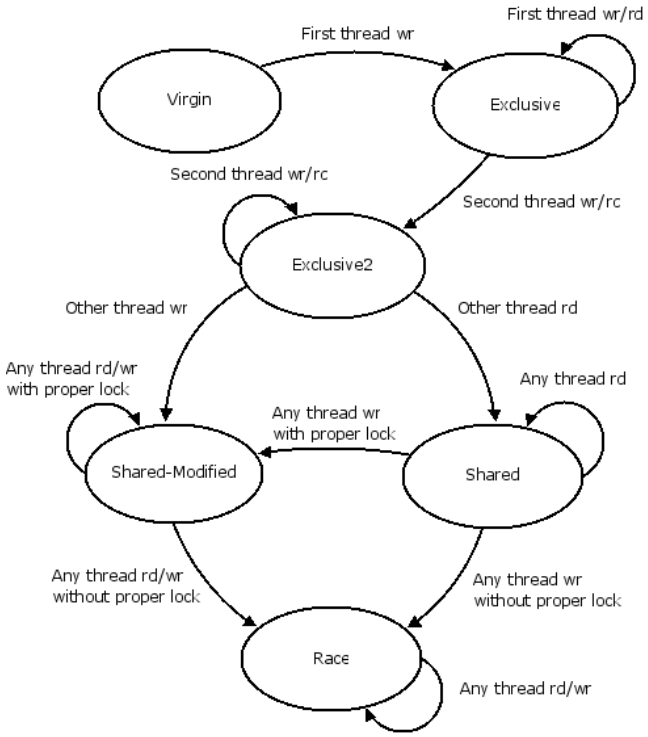


Figure 4.2: Possible states of a shared variable.

4.1.2 Join Synchronization

Further, the original Eraser algorithm has been enriched to support a join synchronization mentioned in Section 2.1. In Java, if a thread t calls the `Thread.join()` method of another thread t', it is ensured that all the events of the thread t' are executed before the events following the `Thread.join()` call in the thread t. This imply that, there is no race possible between successfully join-synchronized threads after a successful join.

The main idea behind the proposed extention is that if there remains only one thread accessing a variable alive and is successfully join synchronized with other threads that accessed variable v, there is no possibility for a data race. In a such case, the state of the variable v is changed to *Exclusive2*. The idea enriches the algorithm listed in Figure 4.3 with operations listed in Figure 4.4.

The implementation of join synchronization needs to construct two more sets. The $ThreadAccessing(v)$ set contains all threads that have accessed a variable v during the computation so far. A thread t is added to the $ThreadAccessing(v)$ if it accesses v. The construction of $Joined(t)$ containing the set of threads already join synchronized with a thread t has been already described in Section 4.1.

If a thread t is accessing a variable v and $Joined(t) \cup \{t\} = ThreadsAccessing(v)$, it implies that the thread t is the last currently existing thread accessing v and all others have been successfully join-synchronized with the thread t. Then, the internal state $status(v)$ of the variable v is changed to `Exclusive2`, its set of candidate locks $Candidate(v)$ becomes empty, and the set $ThreadsAccessing(v)$ contains only the current thread t. This modification helps to rapidly decrease the number of false alarms produced in environments based on loops when during each loop a worker thread is created or in situations where the last

Initialization:
$\forall v \in SharedVariables : status(v) = Virgin, owner(v) =$ `null` $, Candidate(v) = \emptyset$
Input: $v, loc, Locks(t)$
Output: $true$ or $false$

Computation:
switch$(status(v))\{$
case : *Virgin*
    if $getMode(loc) == write$ **then**
        $owner(v) = t_{current}$
        $status(v) = Exclusive1$
    return $false$
case : *Exclusive*
    if $t_{current}$!= $owner(v)$ **then**
        $owner(v) = t_{current}$
        $status(v) = Exclusive2$
    return $false$
case : *Exclusive2*
    if $t_{current}$!= $owner(v)$ **then**
        $Candidate(v) = Locks(t)$
        if $getMode(loc) == read$ **then**
            $status(v) = Shared$
        else
            if $Candidate(v) == \emptyset$ **then**
                $status(v) = Race$
                return $true$
            else
                $status(v) = Shared\_modified$
    return $false$
case : *Shared*
    $Candidate(v) = Candidate(v) \cap Locks(t)$
    if $getMode(loc) == write$ **then**
        if $Candidate(v) == \emptyset$ **then**
            $status(v) = Race$
            return $true$
        else
            $status(v) = Shared\_modified$
    return $false$
case : *Shared_modified*
    $Candidate(v) = Candidate(v) \cap Locks(t)$
    if $Candidate(v) == \emptyset$ **then**
        $status(v) = Race$
        return $true$
    return $false$
case : *Race*
    return $false$
}

Figure 4.3: An Eraser+ algorithm used for detecting data races.

Initialization
$\forall v \in SharedVariables : ThreadsAccessing(v) = \emptyset$

Computation:
switch$(status(v_{current}))\{$
case : *Exclusive2*
add $t_{current}$ **to** $ThreadAccessing(v)$

case : *Shared*, *Shared_modified*
add $t_{current}$ **to** $ThreadAccessing(v)$
if $(Joined(t) \cup \{t\} = ThreadsAccessing(v))$ **then**
$Candidate(v) = \emptyset$
$ThreadAccessing(v) = \{t_{current}\}$
$status(v) = Exclusive2$
}

Figure 4.4: A join synchronization extension of the algorithm in Figure 4.3.

thread finalizes shared global variables.

4.1.3 Lock Suggestion

A simple heuristics that *suggests which lock should have been used* by the thread that caused a data race is proposed in this section. The obtained information can subsequently be provided to the developer as a hint of what lock was probably omitted. The heuristics is inspired by an assumption that if some threads have already used a lock for accessing a shared variable v, then the same lock should be used by all other threads. The set of locks to be used to be suggested to a developer is the set $Candidate(v)$ of candidate locks just before it becomes empty (by an access to v without holding a proper lock).

Of course, the above approach does not guarantee that the suggestion is always correct. For instance, when the first thread accessing a shared variable uses a wrong lock, other threads are suggested to use the same wrong lock too. For that reason, an additional set $Candidate(v,t)$ of candidate locks is maintained for each shared variable $v \in SharedVariables$ and each thread $t \in Threads$ where $SharedVariables$ is the set of all shared variables and $Threads$ is the set of all threads. When a race is detected by the Eraser+ algorithm, the intersection $Candidate'(v)$ of $Candidate(v,t)$ of all threads excluding the threads $Threads_{buggy}$ causing a race is computed in the following way:

$$Candidate'(v) = \bigcap_{t \in Threads \setminus \{Threads_{buggy}\}} Candidate(v,t)$$

In most cases, the set $Candidate'(v)$ contains a lock or locks that should be used when accessing the variable v.

4.2 AtomRace: Data Race and Atomicity Violation Detector

AtomRace is a novel algorithm for detecting data races and atomicity violations at runtime. Data race detection in AtomRace is fully automated and self-contained. For atomicity

violation detection, AtomRace expects the atomic sections that should be monitored to be given to it as a part of its input. As it is discussed in Chapter 6, they can be manually defined by the user or obtained by some further static and/or dynamic analysis. Data race detection is implemented in AtomRace as a special case of atomicity violation detection on atomic sections that are specifically defined for this purpose.

This section is organized as follows. Firstly, the core data race detection algorithm is introduced. Then, an extention of the algorithm supporting more general atomic sections is presented. Finally, in order to significantly increase the probability of detecting data races via atomicity violation, a combination of the algorithm with a specific noise injection heuristics is discussed.

4.2.1 Data Race Detection

A data race is defined as a sequence of two accesses to the same shared variable from different threads where (1) these accesses are not separated by any synchronization, and (2) at least one of them is a write access. In AtomRace, such a situation is detected by looking for a violation of *primitive atomic sections* of the form $beforeAccess(v, loc)$, $read/write(v)$,$afterAccess(v, loc)$ where $read/write(v)$ is an instruction accessing a shared variable v. It is clear that if such a primitive atomic section based on a read instruction is broken by a write instruction, or if a primitive atomic section based on a write instruction is broken by a read or write instruction, a data race happens because there is for sure no synchronization used neither between $beforeAccess(v, loc)$ and $read/write(v)$ nor between $read/write(v)$ and $afterAccess(v, loc)$.

Data race detection based on the above idea can be implemented in a very simple way within $beforeAccessEvent$ and $afterAccessEvent$ introduced in Section 3.2.2 as it is shown in Figure 4.5. For each shared variable v, a variable $Access(v)$ is maintained. It is `null` in the case that v is not being currently accessed by any thread, and contains a couple (t, loc) otherwise. The t represents the thread that is accessing v at the location loc. In order to simplify the description, the $Access(v).t$ and $Access(v).loc$ refer to the thread and location stored in $Access(v)$. The function $getMode$ also introduced in Section 3.2.2 is used to determine if the access is for writing or reading.

The algorithm listed in Figure 4.5 is a bit simplified. It does not cover some special situations which can arise in Java. Next to ignoring `final` and `volatile` shared variables as it was discussed earlier, the algorithm should ignore also operations within a special method `<clinit>`. In Java, the `<clinit>` method is used to assign implicit values to *static variables* when they are used for the first time. This initialisation requires a write access which should, however, not be taken into account when looking for data races. Considering operations in `<clinit>` methods causes AtomRace to produce false alarms in situations when threads read a `static` variable for the first time. In Java, classes are loaded and initialised in the JVM on demand. Therefore, the first thread which tries to access a field or a method from a class which is not present in the JVM, causes JVM *class loader* to load the class and initialize its `static` fields by executing the `<clinit>` method presented in the class. The `<clinit>` method is then executed within the *primitive atomic section* used by AtomRace what causes an overlapping of two primitive atomic sections on the same variable where an initialisation access is for write. Note, that `<clinit>` method is not a constructor. Constructor in Java is also realized as a special method called `<cinit>`. The constructors should be also analysed by the AtomRace algorithm.

Initialisation:
$\forall v \in SharedVariables : Access(v) =$ `null`
$RaceDetected = \emptyset$

Computation:
switch ($AtomRaceEvent$) {
case : beforeAccessEvent(v, loc)
  if ($Access(v)$ == `null`) **then**
    $Access(v) = (t_{current}, loc)$
  else
    if (($getMode(Access(v).loc)$ == `write`) ||
      ($getMode(loc)$ == `write`)) **then**
      add v **to** $RaceDetected$ // RACE DETECTED

case : afterAccessEvent(v, loc)
  if ($Access(v).t == t_{current}$) **then**
    $Access(v) =$ `null`
}

Figure 4.5: A data race detection in AtomRace.

4.2.2 Atomicity Violation Detection

Here, the proposed AtomRace idea will be extended such that it allows to deal with more general *atomic sections* associated with a single shared variable. For a shared variable v, an atomic section is viewed as a code fragment which is delimited by a single entry point and possibly several ending points. The intended meaning of an atomic section over a variable v is that when a thread t starts executing within the atomic section, no other thread should access v before t reaches an end point of the atomic section (with the exception of some kinds of accesses that may be explicitly allowed for the given atomic section). Again the entry point of an atomic section corresponds to some $beforeAccessEvent$ piece of code and the end points to correspond to some $afterAccessEvent$ or a new kind of event called $atomExitEvent$.

The $atomExitEvent$ is a special way how the algorithm can solve the problem of program branching. In the case of primitive atomic sections, there was no program branching within the atomic section. With a more general atomic sections, the branching can happen on intraprocedural or even interprocedural level. The algorithm needs a way how the atomic section can be ended in the case when a branch which does not contain an access to variable v over which the atomic section is defined. Then, there is no $afterAccessEvent(v, loc)$ in such branch. The execution monitoring module offers $methodEntryEvent(loc)$ and $methodExitEvent(loc)$ which was chosen as a compromise between too often or too rarely invoked events. A new $atomExitEvent(v, loc)$ is obtained if a variable v is inside an atomic section ($access(v)! = null$) and a $methodEntryEvent(loc)$ or $methodExitEvent(loc)$ is seen with a location loc equals to loc specified in $atomExitEvent(v, loc)$.

To allow a specification of which accesses from other threads should not be considered to break an atomic section when it is being executed by some tracked thread, a (possibly empty) subset of the set $\{read, write\}$ is associated with each end point of each atomic

section. 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. As discussed in Section 6, this information can be used, e.g., to allow not only checking of pure atomicity, but to allow for handling not purely atomic, but *serializable accesses* (in the sense of [26]) as well.

When dealing with *several atomic sections* associated with the same variable v, it is required that these sections *do not overlap within the thread* in any other way than possibly on their entry and end points. More precisely, the only allowed overlap is that one atomic section has an entry point $beforeAccess(v, loc)$ while the other has an end point $afterAccess(v, loc)$ for the same location loc. Due to this requirement, a thread can only be in one atomic section at a time (with the exception of leaving one section and at the same time entering another).

As shown in Figure 4.6, detection of atomicity violation can be implemented in a simple way within handling the events $beforeAccessEvent(v, loc)$, $afterAccessEvent(v, loc)$, and $atomExitEvent(v, loc)$. For the purpose of describing the algorithm, the set of atomic sections associated with a variable v that are supposed to be tracked and that satisfy the conditions described above is expected to be encoded in a "flattened" way as a set $Atomic(v)$ of triples $(loc_{entry}, loc_{exit}, A)$ where $A \subseteq \{read, write\}$ and loc_{entry}, loc_{exit} are locations corresponding to entry and end points of particular branches of the control flow graph. The notation $Atomic(v).A(loc_1, loc_2)$ is used to refer to the set A in $(loc_1, loc_2, A) \in Atomic(v)$.

For each shared variable v, the set $Access(v)$ is maintained in a similar way as it is in Figure 4.5. Moreover, a set $SuspectAccess(v)$ which contains types of accesses to v that came from other threads than the one whose entered an atomic section over v which is currently monitored is build. The algorithm works in such a way that if a thread t is entering an atomic section over a variable v over which no atomic section is currently being monitored, monitoring of accesses to v from other threads is started, and once t is leaving the atomic section via some end point, a check that no undesirable access to v from a thread other than t is performed.

Note, that the algorithm shown in Figure 4.6 is a bit simplified with regard to the above described functionality. In particular, it has left out the treatment of overlapped atomic sections, which can, however, be added in a straightforward way into the code handling the $beforeAccessEvent(v, loc)$ and $afterAccessEvent(v, loc)$ events, e.g., by adding a $continueWithAtomicSection(v)$ variable.

The algorithm can be also easily extended to cope with *circular atomic sections*, i.e. atomic sections where $loc_{entry} = loc_{exit} = loc$, where the atomic section should not terminate just after firing the statement at the location loc (as it was in the case of primitive atomic sections). In such a case, a section can be tagged and the algorithm does not leave the atomic section during the first occurrence of $afterAccessEvent$ at loc if the atomic section contains such tag.

Finally, the algorithm can also be extended to support recursive atomic sections by counting how many times the section has been entered and left and by terminating the atomic section only when these numbers are equal.

4.2.3 Race and Atomicity Violation Exhibition

The proposed algorithm can be also very useful in bug hunting within the application testing if a suitable *noise injection* technique is used.

The considered atomic sections may be very short and the probability of observing a real

Initialisation:
$\forall v \in SharedVariables$:
$Access(v) = \texttt{null}, SuspectAccess(v) = \emptyset$
$RaceDetected = \emptyset$

Computation:
switch ($AtomRaceEvent$) {
case : beforeAccessEvent(v, loc)
if ($Access(v) == \texttt{null}$) **then**
if ($\exists l_{end}, A \ : (loc, l_{end}, A) \in Atomic(v)$) **then**
$Access(v) = (t_{current}, loc)$
else
if ($Access(v).t$!= $t_{current}$) **then**
add $(t_{current}, loc)$ **to** $SuspectAccess(v)$

case : afterAccessEvent(v, loc), atomExitEvent(v, loc)
if ($Access(v).t == t_{current}$ &&

$\exists A \ : (Access(v).loc, loc, A) \in Atomic(v)$) **then**
if ($SuspectAccess(v)$!= $\emptyset$) **then**
$A = Atomic(v).A(Access(v).loc, loc)$
foreach $(t_s, l_s) \in SuspectAccess(v)$ **do**
if ($getMode(l_s) \notin A$) **then**
add v **to** $RaceDetected$ // ATOMICITY VIOLATION DETECTED
$Access(v) = \texttt{null}$
$SuspectAccess(v) = \emptyset$
}

Figure 4.6: A simplified version of the AtomRace algorithm for detecting atomicity violation.

conflict on them may be very low. However, the probability may be significantly increased by suitably influencing the execution of the program what is exactly the purpose of noise injection. In general, noise injection is a technique that forces different legal interleavings for particular executions of a test in order to increase the concurrent coverage. In fact, it simulates the behavior of various possible schedulers. The noise can be injected at any instrumented point of the tested software. When such a point is reached, the noise heuristics decides—randomly or based on a specific bug-finding technique—if it injects there some kind of delay or other kind of influencing the execution (like a context switch).

The introduction of noise can help the detection of races and/or atomicity violations in two ways: firstly, different legal thread interleavings are enforced. Secondly, randomly chosen atomic sections are executed for a longer time period and therefore the probability that a conflict will occur on them is increased. Both of this helps to see conflicts that would not be seen otherwise. Of course, the probability of seeing a data race and/or atomicity violation can then be rapidly increased also if a true multiprocessor computer is used for testing.

Based on experiences with ConTest noise injection infrastructure [8, 42], three noise injection heuristics for increasing the probability of detecting data races and/or atomicity

violations were proposed. All of them uses injecting call of `Thread.sleep()` inside atomic sections to increase their duration followed by call of `Thread.yield()` to force a thread switch. The probability of noise injection in the particular currently executed location is driven by the parameter that ranges from 0 (=never) to 1000 (=every time)—this parameter is also referred as *noise frequency*. The duration of sleep is given by the number of millisecond that sleep should last. The three heuristics proposed in this thesis are the following:

- **A random heuristics.** This is the simplest heuristics that can be used during a normal testing when there is no suspicion that something wrong is happening in the program. It injects noise to randomly chosen atomic sections.
- **A variable-based heuristics.** This heuristics can be used when some concrete variable is suspected to be accessed with a wrong synchronization. The noise is injected to the sections associated with instances of the suspected variable only.
- **A heuristics based on program locations.** This approach allows the user to identify atomic sections which are suspected to be problematic. The noise is injected to the given program locations only.

The second and third heuristics are more suitable for testing and debugging. If AtomRace or another analysis, e.g., the one based on the proposed Eraser+ algorithm, detects a race or an atomicity violation in one run of the tested software, the developer can use a noise injection focused on the suspected variable or the problematic program locations in order to increase the probability of repeated manifestation of the detected problem.

Chapter 5

Data Race Healing

Data races are usually caused by missing or wrongly used synchronization. Therefore the best way how to correct them is adding or restoring a correct synchronization. In general, healing is based on identifying correct atomicity of the application and when the race is detected, forcing the application to follow the correct atomicity. This chapter introduces two different techniques used for healing. The first is based on introducing an additional synchronization to the program execution when a problem has been detected. If the additional synchronization is not added with respect to already existing synchronization, the healing can cause a worse problem than that which is trying to heal, e.g., deadlock. Therefore, another healing technique based on legal influencing of Java scheduler is introduced. This technique can not cause synchronization problems but, on the other hand, can only make the probability of problem manifestation lower.

This chapter is organized as follows. Firstly, the notion of atomicity patterns is introduced and a way how they can help in the healing process is described. Next, the way how a healing process is Incorporated with the proposed architecture is presented. In Section 5.3, healing based on an additional synchronization is introduced. Finally, healing based on legal influencing of Java scheduler is presented.

5.1 Atomicity Patterns Against Data Races

The definition of data races used in this work expects at least two simultaneous accesses to a variable, one for write, and violated or none synchronization among them. The easiest way how to remove a race is to prevent accesses from being simultaneous. This can be done by making from each access to a problematic variable a *critical section* which allows only one process being in. This can be understood as simulating a transactional memory and has similar effect as if the variable is declared `volatile`. This simple solution can heal data races according to the definition and sometimes, this can remove the problem. However, in many cases, this is not enough. Some blocks of code needs to be executed atomically. Such a blocks of code are refered as *atomic sections* in this thesis. There can be identified some common constructs, that are assumed by the programmer to be executed atomically. Such constructs are called *atomicity patterns*.

Such a common pattern is the *load-and-store atomicity pattern*. It is an assignment statement that is translated into the Java bytecode as a sequence of instructions consisting of one or more load instructions on a shared variable followed by one store instruction on the same variable. An elementary example of this pattern is the statement `x++;` in

Figure 5.1(a). The corresponding bytecode is shown in Figure 5.1(b). At first, the current value of the shared variable is loaded into the local memory of the thread by the instruction at line 2, then the local copy is incremented by instructions at lines 5 and 6, and at the end, the result from the local memory is stored back to the shared memory at line 7.

```
                          2:   getfield  #2
                          5:   iconst_1
x++;                      6:   iadd
                          7:   putfield  #2

  (a)                           (b)
```

Figure 5.1: An example of load-store atomicity pattern: (a) Java code (b) bytecode.

Another atomicity pattern is the *test-and-use atomicity pattern*. The test-and-use atomicity pattern is a conditional statement where the condition is checked at the beginning of the statement and then the result of the test is used inside the statement without making sure that the condition still holds. An example of a code fragment containing the test-and-use bug pattern is in Figure 5.2.

```
                          0:   aload_0
                          1:   getfield  #2
                          4:   ifnull  18
if (p != null) {          7:   aload_0
   p = p.next             8:   aload_0
}                         9:   getfield  #2
                         12:  getfield  #3
                         15:  putfield  #2
                         18:  ...

   (a)                          (b)
```

Figure 5.2: An example of test-and-use atomicity pattern: (a) Java code (b) bytecode.

The value of the shared variable `p` is loaded to the local memory at lines 0 and 1. Then, the condition is tested at line 4. If `p` is not null, then the value of `p` is loaded once again at line 9 followed by loading the value of `p.next` at line 12 and storing it back to `p` at line 15. A similar pattern can be identified in loops. For example if we use `while` statement instead of `if` statement in Figure 5.2.

The healing of such patterns is straightforward. The healing process has to consider such atomicity patterns as a one access to a variable and extend the critical section previously defined for one particular access instruction to cover the whole block of code that should be executed atomically.

5.2 Incorporation of Healing

Modules incorporated into the proposed architecture communicate via events. This section introduces the way, how healing methods can be incorporated to this approach. The healing

logic uses only $beforeAccessEvent$ and $afterAccessEvent$ events. The healing is activated only in the case when there was a data race or an atomicity violation detected on a variable v. Such variable was added by the detection algorithm into the *RaceDetected* set as can be seen, e.g., in Figure 4.1. This set is now used by the healing engine to distinguish events where the healing should be performed. The healing mechanism has to also consider the predefined atomicity of the application available in the external repository via the set $Atomic(v)$ introduced in Section 4.2.2. The incorporation is depicted in Figure 5.3.

Initialisation:
$\forall t \in Threads : Heal(t) = \emptyset$

Computation:
switch$(Event)$ {
case : beforeAccessEvent(v, loc)
  if $(v \in RaceDetected)$ **then**
    if $(v \notin Heal(t_{current}))$ **then**
      if $(\exists l_{end}, A \ : (loc, l_{end}, A) \in Atomic(v))$ **then**
        beforeCriticSection(v)
        add v **to** $Heal(t_{current})$
      else
        beforeNormalAccess(v)

case : afterAccessEvent(v, loc)
  if $(v \in RaceDetected)$ **then**
    if $(\exists l_{start}, A \ : (l_{start}, loc, A) \in Atomic(v))$ **then**
      afterCriticSection(v)
      remove v **from** $Heal(t_{current})$
    else
      if $(v \notin Heal(t_{current}))$ **then**
        afterNormalAccess(v)
}

Figure 5.3: An incorporation of the healing into the proposed architecture.

The algorithm in 5.3 maps access events used by the architecture into healing related events used by the proposed healing techniques. Due to this mapping, the algorithm maintains for each thread $t \in Threads$ a set of variables $Heal(t)$ that are currently healed by the thread t. Note that healing is based on forcing atomic sections to be executed as *critical sections*.

5.3 Lock-based Techniques

The techniques presented in this section are based on an additional synchronization. In particular, the synchronization actions are based on a suitable use of mutexes (locks) or semaphores with one permit available. The difference between locks and semaphores in Java is that semaphores does not have the notion of ownership. This can be helpful when a deadlock induced by the healing logic occurs. Because the proposed architecture is executed within the threads of the managed application, the deadlock prevents the architecture

to release a healing lock. In such cases, a deadlock healing tool external to the proposed architecture can release the semaphore based lock and this way heal the deadlock.

The principle of healing data races and atomicity violations using an additional synchronization is shown in Figure 5.4. The variable v on which a problem was detected is healed by introducing a new lock $lock_v$. This lock is acquired by each thread before any access to the variable. In addition, the lock $lock_v$ must not be released during the entire atomic section forming a critical section.

Initialisation:
$\forall v \in SharedVariables : lock_v$

Computation:
switch($HealingEvent$) {
case : beforeCriticSection(v), beforeNormalAccess(v)
$lock_v$.lock()
case : afterCriticSection(v), afterNormalAccess(v)
$lock_v$.unlock()
}

Figure 5.4: Lock-based healing techniques.

The described healing technique is able to completely remove the detected bug but, on the other hand, this technique can introduce new and even more dangerous bugs, e.g., deadlock. Thus, an application of such a healing action requires healing assurance that is discussed in Section 5.5. Moreover, a frequent locking can cause a significant performance drop in some cases.

5.4 Non-locking Techniques

The techniques presented in this section are based on legal influencing the scheduler. The main idea behind is that data races often occur only for a small subset of all possible program execution schedules. The healing methods presented here try to influence a scheduling of threads in a such way that the detected data race does not manifest. Legal influencing of scheduler means that the technique will use only tools available to any Java program (the scheduler will not be touched), and it also means that the changes to the execution does not change the semantics of the code. It is important to note, that influencing a scheduler is in the proposed way is legal for common applications but can cause problems when used with a real-time software. In such a case, inserting delays or changing the thread priority must be done carefully or not at all. In Java, influencing scheduler can be achieved by the following approaches:

Forcing a context switch. This interference of the execution is equal to executing the program on a different architecture or with a different scheduler setting. Forcing a context switch can be in Java caused by invoking the `Thread.yield()` method, and in some JVM implementations also by invoking the `Thread.sleep(0)` method. The goal of this technique is to execute an atomic section at the beginning of a new scheduling time slice that is given to the thread that forced the context switch after the thread is again scheduled to run. This increases the chance of the atomic section to be finished without an interruption.

Initialisation:

Computation:
switch($HealingEvent$) {
case : beforeCriticSection(v),
Thread.yield()
}

Figure 5.5: Context switch healing.

Another possible use of the forced context switch method for healing is shown in Figure 5.6. In this case, the influenced thread is not the one which is executing an atomic section defined for a variable v, but all other threads which are about to access the variable v. The algorithm maintains for each variable v an indication $Inside(v)$ whether some thread has started but not yet finished an execution of an atomic section defined for the variable v. This technique does not build mutual exclusion and therefore multiple threads can be inside an atomic section defined for a variable v. This imply the use of counter for the indicator $Inside(v)$. The idea behind this technique is to allow the thread executing the atomic section to reach the end without interleaving access to the variable v.

Initialisation:
$\forall v \in RaceVariables : Inside(v) = 0$

Computation:
switch($HealingEvent$) {
case : beforeCriticSection(v)
if ($Inside(v)$!= 0) **then**
Thread.yield()
$Inside(v) = Inside(v) + 1$
case : afterCriticSection(v)
$Inside(v) = Inside(v) - 1$
case : beforeNormalAccess(v)
if ($Inside(v)$!= 0) **then**
Thread.yield()
}

Figure 5.6: A technique where other threads are influenced by context switching.

Adding noise. The influencing of the execution by this technique is equal to a situation when the processor is occupied by an another demanding process for a while. In Java, a noise can be produced by invocation of the `Thread.sleep(x)` or the `Object.wait(x)` methods where x represents the time period for which the actual thread should be suspended. The noise can be also achieved by inserting a code that do nothing, e.g., a loop with some computation that is then discarded.

The noise is produced only in threads that are about to access a variable v when some other thread is executing an atomic section for the variable v. The algorithm was explained above and is shown in Figure 5.6. The idea behind this technique is to give the thread that

is executing an atomic section enough time to finish the atomic section without interleaving.

Changing a thread priority. This technique uses the only way how a Java program can tell the scheduler that some thread should be preferably scheduled to run. In Java, thread priorities are changed by invoking the `Thread.setPriority()` method. The priority of a thread is set to maximal value during the *beforeCriticSection* event and returned back during the *afterCriticSection* event. The idea behind this technique is straightforward. The atomic sections used for healing are executed with the highest priority with the intention to avoid their interleaving.

Utilizing the processor by some "dummy" threads. This technique simulates the situation when a program is executed on a machine with some other demanding processes. Several so-called "dummy" threads with the highest priority are initialized and suspended during the initialization of the technique. The number of threads should be one lower then the number of available processors. The thread executing the *beforeCriticSection* event increases its priority to maximum and runs the "dummy" threads. The "dummy" threads are suspended and the priority is returned back during the execution of the *afterCriticSection* event.

It is important to note that all of the mentioned approaches are software and hardware dependent. Each JVM can implement the used methods in a slightly different way. Sometimes JVM uses a system scheduler and so the techniques are also operating system dependent. Finally, there can be also some support for a thread scheduling in the used hardware.

5.5 Healing Assurance

As was mentioned before, the use of additional healing locks can reliably heal a detected data race or an atomicity violation but it can cause a deadlock. The methods how to avoid such a scenario are out of scope of this thesis and are subject of future work. But let briefly introduce some ideas how the healing assurance can be incorporated into the proposed architecture. Basically, there are two possibilities. Either to use some kind of formal approach applied to atomic sections available in the external repository, e.g., one being developed within the SHADOWS project at FIT [46], or to use dynamic deadlock prevention mechanism, e.g., one being developed within the SHADOWS project at IBM Haifa Research Labs [41].

The formal approach being developed at FIT uses static analysis to check whether there are any synchronization actions within the atomic sections which are available in the external repository. If an atomic section contains a synchronization, a variable to which the atomic section belongs is considered as potentially dangerous for healing via additional locks because a nested synchronization can be introduced into the healed application.

The dynamic analysis approach tracks the locking operations within the managed application and also operations with locks used for healing. If a deadlock detection and prevention tool detects a deadlock (or a possible deadlock) due to a use of a healing lock, it takes actions to release the healing lock and so avoid the deadlock.

Chapter 6

Obtaining Atomicity

In the previous chapters, the atomicity to be used as an input for detection and healing was expected to be given. The atomic sections can be defined either manually by the user or obtained automatically. This chapter introduces techniques that can be used for an automatic construction of the assumed atomic sections. A correct identification of the atomic sections to be monitored and used as input for healing is crucial for detection and healing mechanisms to work properly.

Formally, the atomicity of an application can be seen as a set of sets $Atomic(v)$, associated with each shared variable $v \in SharedVariables$. As it was defined in Section 4.2.2, $Atomic(v)$ contains a set of triples $(loc_{entry}, loc_{exit}, A)$. The locations loc_1 and loc_2 delimit a block of code starting at loc_{entry} and ending at loc_{exit} on one particular branch in the control flow graph (CFG). The set $A \subseteq \{read, write\}$ describes the access mode that is allowed for an interleaved access on the atomic section.

A two kinds of static analyzes of a given program to infer the blocks of code that is likely to be assumed by the programmer to execute atomically are proposed in this chapter. The *pattern-based* static analysis looks for appearances of typical programming constructions that programmers usually expect to execute atomically. The second static analysis is built on the *access interleaving (AI) invariants* with the serializability notion from [26]. Finally, a dynamic analysis that can be used after these static analyzes was proposed. The dynamic analysis is used to identify candidates for atomic sections which are likely not to correspond to real atomic sections and that are thus to be dropped from the set of monitored atomic sections.

As was also mentioned in Chapter 4, the static analyzes should not construct atomic sections over `final` variables whose values do not change, and `volatile` variables which are designed to be used without any further synchronization and therefore should also tolerate atomicity violations. Further, it is not needed to monitor atomic sections laying within the `<clinit>` method, which is guaranteed by the JVM to be executed atomically.

This chapter is organized as follows. Firstly, a the pattern-based static analysis is introduced. Next, the AI invariants-based analysis is described. Finally, the dynamic analysis is presented.

6.1 Pattern-based Static Analysis

The proposed pattern-based static analysis identifies blocks of code that are likely to be intended to execute atomically based on looking for some typical programming constructions,

for which such an assumption is usually done. Two such patterns have been presented in Section 5.1: the load-and-store and the test-and-use atomic patterns.

The *load-and-store* pattern is depicted in Figure 5.1 and represents an assignment statement. The atomic section covering this pattern starts with the first load instruction on line 2 (loc_{entry}) and ends by the store instruction on line 7 (loc_{exit}) of the bytecode in Figure 5.1. This couple of locations can safely be interleaved with read accesses, and hence the A set associated with loc_{entry} and loc_{exit} is $A = \{read\}$. To cover the possibility of an exception before the control reaches loc_{exit}, the atomic section end must be also defined on the all exception paths reachable from the code between loc_{entry} and loc_{exit} in the CFG. For this purpose, possible end at loc_{exit2} is added where loc_{exit2} corresponds to the nearest location of $methodEntryEvent(loc_{exit2})$ or $methodExitEvent(loc_{exit2})$ of the appropriate exception handling branch. Basically, multiple different exceptions can be generated within the atomic section and each can be handled in a distinct branch of the CFG. With the atomic section determined by loc_{entry} and loc_{exit2} the set $A = \{read, write\}$ is associated—meaning that this branch will not be checked for atomicity violation.

The *test-and-use* pattern is depicted in Figure 5.2 and represents a conditional statement. The atomic section in this case starts at the first load instruction on line 0 (loc_{entry}) of the bytecode and ends at the last load instruction on line 9 (loc_{exit}) in the branch of the CFG that is executed if the condition holds. Such an atomic section can be safely interleaved with read accesses, and so $A = \{read\}$. The other branch of the condition and all the exception branches have to be covered by using loc_{exit2} with $A = \{read, write\}$ similarly to the previous pattern.

More similar patterns can, of course, be defined and used but it is kept as a part of the future work.

6.2 AI Invariant-based Static Analysis

Inspired by the notion of *AI invariants* introduced in [26], couples of two immediately consequent accesses to a shared variable v in the interprocedural CFG can be identified as candidates for atomic sections. A dynamic analysis described in Section 6.3 can then be used to remove the candidate sections which do not correspond (or do not seem to correspond) to code sections that should really be executed atomically.

Atomic sections based on identifying couples of consecutive accesses to the same shared variable contain one entry point loc_{entry} and several different end points loc_{exit} due to the possible branching of the code. For each end point, a set A is defined using the notion of *serializability* defined in [26] and listed in Table 6.1. The table lists unserializable scenarios which can be used to determine the set of allowed interleaving access modes A. Hence, for example, if $getMode(loc_{entry}) = read$ and $getMode(loc_{exit}) = read$ for an entry point loc_{entry} and an end point loc_{exit}, the set A will be $A = \{read\}$ according to the first row in the table. Again, a special kind of exit locations is used to cover the branches of CFG which start at loc_{entry} and does not contain any next access to the shared variable, e.g., the exception handling branches. In such a case, a special loc_{exit2} location is used with $A = \{read, write\}$. Because the proposed architecture works on top of events, the location loc_{exit2} is a location of nearest $methodEntryEvent$ or $methodExitEvent$ event tracked by the monitoring engine. Note that the atomic sections defined using the notion of AI invariants *overlap*, i.e., an end location of a previous atomic section is the entry location of the following atomic section.

Interleaving scenario	**Description**
$read_{local}$ $write_{remote}$ $read_{local}$	The interleaving write makes the two reads have different views of the same memory locations.
$write_{local}$ $write_{remote}$ $read_{local}$	The local read does not get the local result it expects.
$write_{local}$ $read_{remote}$ $write_{local}$	Intermediate result that assumed to be invisible to other threads is read by a remote access.
$read_{local}$ $write_{remote}$ $write_{local}$	The local variable relies on a value from the preceding local read that is then overwritten by the remote write.

Table 6.1: Unserializable interleaving scenarios [26].

AI invariants are in [26] defined using a single instruction which constitutes the AI invariant together with the immediately preceding instruction accessing the same shared variable during *one execution run*. This dynamic approach considering each time only a single execution run is not suitable for obtaining atomicity defined in the way used in this thesis. Therefore, a specific dataflow static analysis has been designed to produce an *initial set of AI invariants*. The initial set contains all possible atomic sections based on AI invariants which one can identify in the given application. The concept of dynamic refinement discussed in Section 6.3 is then used to remove from this set such atomic sections which are not appropriate for the application.

The dataflow static analysis that has been proposed iteratively computes in an inter-procedural way control flow sensitive facts for each accessing a shared variable. At the end, the computed values are used to build all tuples of consecutive accesses to a shared variable. The data flow values computed during the analysis consist of a set of tuples (var, loc) which represent the last previous access to the variable $var \in SharedVariables$ at location $loc \in Loc$ for each CFG branch leading to loc.

The dataflow static analysis is defined as a forward dataflow analysis in the traditional way [32]. The dataflow values form a lattice:

$$(2^{SharedVariables \times Loc}, \subseteq, \bigcap, \bigcup, \emptyset, SharedVariables \times Loc)$$

As usual, the analysis uses a Gen function to generate an element and a $Kill$ function to remove an element from the dataflow value associated with a basic block. A *basic block* is defined as a single instruction or a maximal group of sequentially executed instructions with no branching points. The basic block bb related flow functions Gen_{AI} and $Kill_{AI}$ are defined as follows:

$$Gen_{AI}(bb) = \{(var, loc) \mid if\ var\ is\ accessed\ within\ the\ basic\ block\ bb\ and\ loc\ is\ the\ last\ location\ where\ the\ var\ was\ accessed\ inside\ the\ bb\}$$

$$Kill_{AI}(bb) = \{(var, loc') \mid if\ var\ is\ accessed\ within\ the\ basic\ block\ bb\ and\ there\ was\ a\ previous\ access\ to\ var\ at\ location\ loc'\ within\ a\ previous\ basic\ block\ bb'\}$$

Then, the data flow functions $In_{AI}(bb)$ determining the dataflow value at entry point to a basic block bb and $Out_{AI}(bb)$ determining the dataflow value at the exit point of the basic block bb are defined in the standard way as follows:

$$In_{AI}(bb) = \begin{cases} \emptyset & bb \text{ is the entry basic block} \\ \bigcup_{p \in pred(bb)} Out_{AI}(p) & otherwise \end{cases}$$

$$Out_{AI}(bb) = Gen_{AI}(bb) \cup (In_{AI}(bb) - Kill_{AI}(bb))$$

The $pred(bb)$ function returns the set of basic blocks which precede the given bb basic block in the CFG.

The data flow values computed for each basic block are then used for a construction of the sets of atomic sections in the following way. Let first assume that each basic block contains a single access to a shared variable (the more general case will be described later). In the assumed case the dataflow analysis can be viewed to compute a certain dataflow value having the form of a set of tuples (var, loc') for each access to a shared variable var at a location loc. All the tuples (loc', loc, A) can be than added into the set $Atomic(var)$ which is initially empty. In other words, the atomic section is defined using the previous access to the variable var at loc' obtained from the dataflow value and the current access to the variable var at loc. The set A is again determined using Table 6.1. In the case that there are more accesses to the variable within a single basic block, the way described below of constructing the atomic section is used only for the first access to the variable. The next atomic sections within the same basic block are defined, quite naturally, between each two consequent accesses to the variable var.

6.3 Dynamic Refinement of AI-based Atomic Sections

Although the similar refinement algorithm has been presented in [26] to prune the initial set of AI-invariants, the slightly modified algorithm can be used with atomic sections defined by user or obtained by the static analyzes described above. The idea is to remove from the set of assumed atomic sections such sections for which a dynamic analysis find some evidence that they in fact do not seem to be execute atomically.

The dynamic analysis expects a presence of a testing oracle which can distinguish between correct and incorrect executions of the application, e.g., based on assertions, checksums, or some other analysis. Then, the application is run several times and during each run a set of violated atomic sections $ViolatedAtomic(v)$ is collected. If a run is classified by the oracle as correct, the set $Atomic(v)$ is changed for each shared variable v as follows: for each pair of entry and exit locations from the set $ViolatedAtomic(v)$, the set A is changed to $A = \{read, write\}$ which causes the algorithm not to warn about atomicity violation in the given part of the code next time. The pruning ends when the set $Atomic(v)$ is not changed for any shared variable v in any run out of a predefined number of consecutive correct executions. Finally, all the entry-exit pairs which were not seen during the pruning process are removed from the set of effectively monitored atomicity's by setting their set A to $\{read, write\}$. For covering more execution interleavings, the noise injection described in Section 2.7 can be also used during the pruning process as well.

The dynamic refinement has to be careful when removing assumed atomic sections in which the atomicity was broken and yet a correct result of computation was witnessed by the oracle—unless the refinement process is really sure that the section really needs not be

atomic. The problem is that even if the section should really be atomic and its violation comprised a real concurrency error it needs not always reflect in an erroneous run. In such cases, a *threshold* should be used and the violated section should be removed from sections assumed to be atomic only if the atomicity is safely broken more times than what the threshold is. This problem will be illustrated on an example in Section 8.2.

After the pruning, the set $Atomic(v)$ often contains atomic sections which AtomRace can never report as violated because all their end points have the set $A = \{read, write\}$. For performance reasons, such atomic sections may completely be removed from $Atomic(v)$.

Chapter 7

A Prototype Implementation

This chapter describes implementation issues of the architecture for dynamic detection and healing of data races and atomicity violations proposed in this thesis. The implementation follows the principles and algorithms introduced in the previous chapters. The objective of this chapter is to provide a clear, short, and simple picture of the prototype. Not all implementation details are deeply described. Instead, the most important issues are pinpointed.

The chapter is organized as follows. The first section introduces the principle how the tested application is monitored and how the streams of events generated during the monitoring are processed. It also explains how the prototype deals with multithreading and some other problems like Java garbage collection. The next section focuses on the data structures used in the prototype and provides an overview of what concrete information is gathered dynamically during the execution. The third section describes the implementation of the considered detection algorithms, and the fourth section gives a closer description of the implemented healing actions. And at the end, there is a brief overview of how the static analysis used for obtaining atomicity has been implemented.

7.1 Race Detector and Healer

The proposed race detector and healer (or simply *healer*) has been implemented in Java. The healer cooperates with the IBM ConTest tool introduced in Section 2.7 and uses the FindBugs static analyser introduced in Section 2.8. The Java classes implementing the healer are structured into several packages according to their purpose. The packages follows the proposed architecture depicted in Figure 3.1:

- `cz.vutbr.fit.racedetector`. This package contains classes primarily related to detection. It implements the core of the architecture—the *analysis engine*.
- `cz.vutbr.fit.healing`. This package contains classes which implement healing techniques and the *healing logic*.
- `cz.vutbr.fit.atomicity`. This package contains classes related to maintaining atomicity information. It implements the *external repository*.
- `cz.vutbr.fit.findbugs.atom.analysis`. This package contains the dataflow part of static analysis used for *obtaining atomicity*. The dataflow analysis is separated from the other analyses due to some FindBugs requirements.

- `cz.vutbr.fit.findbugs.atom.detect`. This package contains detectors which use the different analyses available in FindBugs to obtain atomicity and other useful information from the given bytecode.

The healer uses the IBM ConTest tool as an interface between the managed application and the healer. So, ConTest is used as an *execution monitoring* engine in the proposed architecure. ConTest instrument the bytecode of the managed application as was discussed in Section 2.7. The active thread of the application then executes among the original code also the code of ConTest. ConTest provides a Java listener architecture which allows other applications to be also called and executed by the active thread of the managed application. So, if the healer registers a listener to an event it wants to monitor, the appropriate listener code is executed where an event occurs. Hence, the execution of the instrumented code is a sequence of actions behaving according to the automata in Figure 7.1 with an input alphabet consisting of Java bytecode instructions.

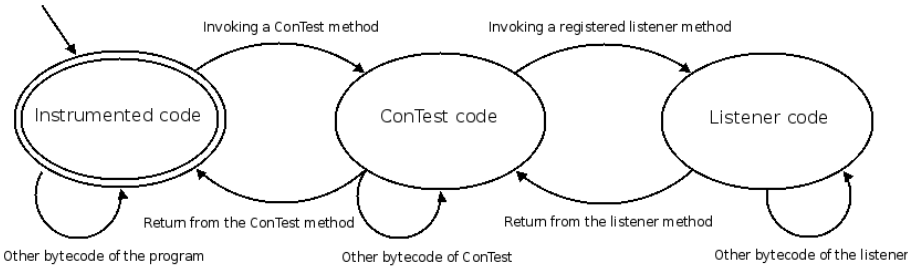


Figure 7.1: A finite automaton describing the execution of ConTest instrumented bytecode with registered listeners.

Note that the healer is called within the threads of the managed application and so it shares the processor time, available memory, and JVM with the managed application. Therefore, the healer can access any variable to which has a pointer and also influences the active thread of the appliacation. The listeners are registred by ConTest according to the configuration file available in the listeners directory in the Java classpath. The content of the file is depicted in Figure 7.2.

```
<?xml version="1.0" encoding="UTF-8"?>
<listeners>
<extension class="cz.vutbr.fit.racedetector.RaceDetector" />
</listeners>
```

Figure 7.2: XML file used by ConTest to find the main class of the healer.

The configuration file in Figure 7.2 tells ConTest to find the `RaceDetector` class and register all listeners that this class implements. So, the `RaceDetector` class represents the entry point to the healer. Its UML class diagram is depicted in Appendix A in Figure A.2. The `RaceDetector` class implements all neccessary ConTest listeners which allow to track all the events listed in Section 3.2.2. It also contains the main state variables of the healer which are set during the initialization. The initialization and finalization process is described in the following subsection.

As can be seen from the class diagram in Figure A.2, the `RaceDetector` class transforms the ConTest listener interfaces into one `Detector` interface which is then used for a communication with the detection algorithms. This transformation is also called a preprocessing and is described in Section 7.2.1. The `RaceDetector` also provides an access to the external repository represented by the `AtomicityColleciton` class. The implementation of the external repository is described in Section 7.5.1. Finally, the `RaceDetector` provides an access to the output file managed by the `Logger` class also described in the next subsection.

7.1.1 Initialization and Finalization

Because the healer is not a normal Java application with a `main` method, both initialization and finalization of its execution differ from the ordinary way. The `RaceDetector` object is created by ConTest before the managed application is started. The constructor of `RaceDetector` is used to load settings from the *configuration text file*, to construct the data structures used by the healer, and to instantiate the detector algorithm. The configuration file is listed in Appendix B.

Although the healer does not need its own thread of execution to be created because it is executed within the threads of the managed application, it creates two auxiliary threads. The `LogProducer$LoggerThread` thread is used to write the output file. This solution has been chosen because the input/output operations are time consuming and could influence the performance of the managed application threads in an undesirable way. The second thread is a finalizer which is used to finalize the execution of the healer.

The finalizer thread is controlled by the JVM. The Java `Runtime.addShutdownHook()` method is used to register this thread. JVM suspends this thread and resumes it when all ordinary threads finish their execution. This mechanism allows the healer to do some computation after the end of the managed application—even if it crashes. Due to this principle, the `LoggerThread` thread has to be defined as a daemon thread because otherwise it will prevent the JVM from terminating.

7.1.2 Minor Implementation Issues

The fact that the healer is executed within the same JVM machine as the managed application brings several problems that have to be addressed. The healer must not hold any pointer to any object of the managed application because otherwise it will prevent the object and all objects transitively referenced by it from being garbage collected. Furthermore, the healer should not execute any method of the managed application such as, for example, the `equals()` or `hashCode()` methods of the application objects. These methods can also be instrumented and so the healer could be called recursively. The mentioned methods are also called if the healer uses a Java collection to maintain the list of application objects, e.g., variables, threads, and locks. All this represents a problem because the healer needs to maintain information related to instances of a shared variable. Therefore, a special type of collection is used. The `WeakIdentityHashMap` collection available in ConTest is a combination of the standard Java `WeakHashMap` which does not prevent objects from being garbage collected and `IdentityHashMap` which does not execute any method of mapped objects.

Another problem for the healer is the multi-threading environment in which it has to run. The healer code has to be thread safe. Commonly, the healer code is executed in parallel and so it has to avoid concurrency problems like deadlocks, data races, and atomicity violations. Furthermore, the healer code should introduce as low additional

synchronization and overhead as possible. The solution of these problems is discussed mainly in Section 7.2.2.

7.2 Monitoring the Execution

In this section, the way how the healer maintains the information obtained from ConTest is presented. This is discussed in the first subsection. The second section describes the structures used by the healer to maintain the information related to the managed application objects.

7.2.1 Information Preprocessing

The healer has to gather information concerning threads, shared variables, events, and program locations. This section describes identifiers are available to the healer and/or how they are obtained.

An identification of the *thread* executing some instrumented point can be obtained simply by call of `Thread.currentThread()` method which returns the reference to the current thread.

The situation with variables is a bit more complicated. As was mentioned in Section 2.1, there are several types of variables in Java. The only variables which can be shared are *fields* and, in the case of array cells, all cells referenced from an array stored in a field. Arrays are handled differently in the bytecode and therefore also by ConTest and the healer—this will be discussed later on.

Each ConTest variable access event contains two identifiers. The first is the *reference* to the object encapsulating the variable, and the second is the variable *name* obtained from the Java signature of the accessing field. This information is available to ConTest methods injected into the original bytecode, as is clear from Figure 2.2. The reference can be `null` if the field is defined as static. To obtain a unique identification of each variable, the healer has to combine these two identificators into one.

The arrays cause a little problem. Shared arrays are fields as others variables and can be handled in the same way. But the ConTest instrumentator cannot distinguish between the access to a shared and local array cells. This problem is solved by instrumenting all accesses to array cells and it is up to the healer to solve this problem. In such a case, ConTest provides a reference of an array object and an index of the cell that is being accessed. The healer maintains a mapping between array references and their encapsulating objects. Local cells are recognized by the healer in such a way that their encapsulating objects are not in the set of shared arrays maintained by the healer. The healer does not currently support tracking of access to multidimensional arrays usually handled as a tree of array objects. The healer then has to combine all three identifiers (encapsulating object, array, index) into one.

An *event* is determined by a Java instruction used in the application. In ConTest, an event is identified by the listener method it invokes in the healer code. The ConTest listeners architecture offers a more detailed identification of some events than it is necessary. For this reason, the healer merges several events into one, for example, $afterRefVarWriteEvent$ and $afterDoubleVarWriteEvent$ into $afterAccessEvent$. The listeners architecture also provides tracking of events that are not important to the healer. Therefore, only listeners correspond to events listed in Section 3.2.2 and two more events are registered by the healer. The two more events are `threadBeginEvent` and `threadEndEvent`

which are used for initialization and finalization of thread related information. This allows the healer to prepare healing and other structures before a thread starts to execute its code.

ConTest currently does not support tracking of other locking mechanisms than the Java implicit locks (monitor based locking used when some block of code is defined as `synchronized`). Therefore, the healer does not support tracking of other kinds of locks, e.g., from the `java.util.concurrent` package. Locks are identified by a reference to the object whose monitor is used.

A very important information provided by ConTest is the *program location*. This string representation of a position in the code is heavily used by the healer as a deterministic key to its various internal structures because it is unique for each instruction of the managed application. ConTest defines this string as *static* and *final*. Therefore, the healer can use the principle of *string internation* [24] and compare the string reference instead of entire strings.

7.2.2 Data Structures for Variables and Threads

As was mentioned above, the healer needs to maintain variaous state related records about the tracked variables and threads. For this purpose a collections of threads and variables state records have to be maintained by the healer. The simplier situation is in the case of threads. Java provides `ThreadLocal` class which is designed just for such situations [35]. This class represents a collection of objects where each object is accessible only from the thread associated with. The collection is maintained within the JVM. The class diagram in Figure A.4 shows how the `ThreadLocal` class is used to manage thread related information in the healer.

The `RDThread` class depicted in Figure A.4 and classes which inherit from this class are used for this purpose. They contain information that is needed by detectors to be maintained for each thread. Typically, it contains cached information which are expensive to obtain. For example, if a variable is identified during the *beforeAccessEvent* the following *afterAccessEvent* from the same thread can use the cached identifier. The `RDThread` class maintains algorithm independent information and inherited `ARThread` and `ERThread` classes maintain algorithm specific information. The `RDThread` class is also used to encapsulate the healing logic as it is described in Section 7.4.

For variables, there is no such mechanism described above for threads. Each variable is determined by a *reference* to encapsulating object and a name (some `String`) obtained by the preprocessing. The composition or other processing of these two values would increase the overhead of the healer. Therefore, a two level mapping arhitecture is used. Firstly, the collection in the `RDVariableCollection` class is used to classify variables according to references. Each encapsulating object is then represented by the `RDObjVarCollection` object which contains a collection of `RDVariable` objects classified by the given string representation. The situation with the `RDVariable` is then similar to `RDThread` class. The `RDVariable` class represents the algorithm independent state information concerning a shared variable. The `ARVariable` and `ERVariable` then represent the algorithm specific state information. In the case of `ERVariable` the state is spread accross several classes as it is described in Section 7.3.2.

ConTest contains a support for partial instrumentation, the healer does not use it yet. Filtering of `final` and `volatile` shared variables is done dynamically. Such variables are identified in advance by a simple static analysis and their ConTest signatures are stored to a file. During the inititalization, the `RaceDetector` construct a `RDVarSet` object and fills

it with stored signatures. The `RDObjVarCollection` then uses this set of variables to filter out such variables from the detection and healing processes.

7.3 Detection

The detection algorithm is intercorporated into the architecture through the `Detector` interface already introduced in the previous section. The interface is used for receiving a preprocessed stream of events generated by ConTest and related to the current execution of the managed application. The implementation of AtomRace detector is depicted in Figure A.1. Although the figure is proposed for the AtomRace detector, the Eraser+ detector follows the same scenario but without `NoiseInjection` class. The detector has access to the currently accessed variable through the `RDVariableCollection` object and to the current thread via the `ThreadLocal` object. The `getNewThread()` and `getNewVariable()` methods are used to create objects of the correct type when a new thread or a new variable is being processed. These objects are then put into described collections.

7.3.1 AtomRace Detection Algorithm

In particular, the current implementation of AtomRace does not cover the whole functionality described in Section 4.2. The atomicity violation detection based on AtomRace has the following limitations. The number of atomic sections a thread can be in is limited to three. The atomic sections are considered to be intraprocedural and to span only over two consequent accesses to the variable they are associated with. Finally, the set A (as defined in 4.2.2) is not with the atomic sections instead, the violated accesses are checked against Table 6.1. All these limitations stem from the current state of the obtaining atomicity described in Section 7.5. So, in general, only intraprocedural pattern-based or AI invariant-based atomic sections are supported. However, the implementation can easily be extended to support more general atomicity in the future.

The AtomRace class diagram is depicted in Figure A.1. The core class which implements the `Detector` interface is called `AtomRace`. AtomRace does not need to track monitor related events and so these methods contain an empty implementation. It also does not need to monitor thread-related events and so only `threadBeginEvent` contains an initialization of the thread-related object using `getNewThread()` method, and other thread-related events contain an empty implementation. As can be seen from the figures 4.5 and 4.6, the core of the algorithm is handling the *beforeAccessEvent* and *afterAccessEvent* events. These events are combined with the other functionality offered by the proposed architecture in the following way.

The `beforeAccessEvent()` method of the `AtomRace` class contains the following operations. Firstly, the actual thread and vairable being accessed are identified and corresponding data structures are obtained. Then, a check if some atomic section starts at the given location is performed. Based on this information, three operations are performed in the following order. At first, healing is started if there was a problem detected on the variable. Then, the *beforeAccessEvent* part of the algorithm is performed. This order helps to check the efficiency of the used healing techniques. Finally, the noise injection, if it is enabled, is performed. This order ensures, that the noise is inserted within the *atomic section* and so there is a higher probability to spot a data race.

The `afterAccessEvent()` method works in a way in some sense opposite to the above. The thread, variable, and atomic section information are obtained. Then, based on the

obtained information, the following operations are performed. Firstly, the noise injection is performed which causes the thread stay longer in the code section assumed to be atomic, hence increasing the probability that if the assumption is wrong, this will show up. Then, the *afterAccessEvent* part of the algorithm is performed. Finally, if necessary, the healing operation is performed. The `beforeMethodEvent()` and `afterMethodEvent()` methods which are used for a special *atomExitEvent* event handling contain the same code as `afterAccessEvent()` but without performing of the noise injection because a noise injection does not make sense here.

The `ARVariable` class maintains the state of the corresponding shared variable. It also offers methods `beforeAccess()` and `afterAccess()` which change the variable state given by $inside(v)$ and $SuspectAccess(v)$ values. These values are changed according to the *beforeAccessEvent*, *afterAccessEvents*, and *atomExitEvent* cases described by the algorithms in Figures 4.5 and 4.6. The `ARViolation` class is used to handle the $SuspectAccess(v)$ set of interleaved accesses to the variable.

Warnings produced by the AtomRace algorithm contain an identification of a problematic variable, conflicting threads, and locations where the violation has been detected. An example of such a warning can be seen in Figure 7.3. In the case of an atomicity violation, the warning contains also a description of the violated atomic section and a list of all accesses that interleaved the atomic section in a problematic access mode. An example of such a warning can be seen in Figure 7.4.

```
Race possible for variable 'Airlines@9931f5->Airlines.soldSeats'.
   The variable was accessed simultaneously by:
    - Thread: Thread-6(Thread@19efb05) at 'Airlines.java 90' (WRITE)
    - Thread: Thread-4(Thread@b89838) at 'Airlines.java 84' (READ)
  The variable should be declared as volatile or
  a proper synchronization should be added.
```

Figure 7.3: A warning produced by AtomRace about a detected data race.

```
Race possible for variable 'Airlines@118f375->Airlines.soldSeats'.
  The atomic section:
   From: Airlines.java 84
   To: Airlines.java 84, Airlines.java -2
  Executed by thread 'Thread-6(Thread@1e0be38)'
  was violated by the following accesses:
   Thread-18(Thread@1975b59) at 'Airlines.java 89' (WRITE)
   Thread-4(Thread@506411) at 'Airlines.java 91' (WRITE)
```

Figure 7.4: A warning produced by AtomRace about a detected atomicity violation.

The probability of seeing a conflict increases when the execution of the watched atomic section takes a longer time. Therefore, the three noise injecting techniques introduced in Section 4.2.3 have been implemented and combined with the AtomRace algorithm.

The noise is produced by the `NoiseInjection` class depicted in Figure A.1. There are two methods used for injecting a noise. The `randInjectNoise()` method injects noise with the probability given by the noise frequency specified in the configuration file listed in Ap-

pendix B. The `progLocInjectNoise()` method injects the noise everytime the current program location is listed in the special input text file `noiselocations`. The methods are called during the processing of *beforeAccessEvent* and *afterAccessEvent* in the `AtomRace` class. The noise should be added inside the primitive atomic section and therefore the noise producing methods are called after the detection algorithm starts to watch the variable in the `beforeAccessEvent()` method and before it stops watching it in the `afterAccessEvent()` method. The Eraser+ algorithm described below does not need noise to be injected to some specific places and therefore, it uses only noise injected by the ConTest tool only.

7.3.2 Eraser+ Detection Algorithm

The Eraser+ algorithm described in Section 4.1 is incorporated into the architecture in the same way as the AtomRace algorithm. The core class implementing the `Detector` interface is called `Eraser` in this case. The Eraser+ algorithm needs to maintain not only the state of monitored variables but also the state of threads.

The state of a thread is maintained in `ERThread` and related `ERJoinSync` objects according to the algorithm listed in Figure 4.1. The `monitorEnterEvent()`, `monitorExitEvent()`, and `threadJoinEvent()` methods are used for this purpose.

The state of a variable is maintained in the corresponding `ERVariable` object according to the algorithm listed in Figure 4.3. The algorithm constructs the history of accesses to all variables. This history is maintained by the `ERThreadInfoCollection` class. For each thread t accessing a variable v, the access information $accessInfo(v,t)$ is maintained using the `ERThreadInfo` class. The information includes the last access location and information whether the thread at least once accessed the variable for writing. Moreover, the $Candidate(v,t)$ set defined in Section 4.1.3 is also maintained by the `ERThreadInfo` class.

Warnings produced by the lockset based algorithm contain the identification of the problematic variable, the current thread, and a list of accesses to the variable by other threads. Optionally, the warning can also contain a list of candidate locks for each thread and the suggestion of a lock which should be used during a problematic access. This suggestion is based on the algorithm presented in Section 4.1.3. An example of such warning is shown in Figure 7.5.

```
Race possible for variable 'Airlines@867e89->Airlines.soldSeats'.
  Race caused by thread : 'Thread-6(Thread@15c7850)'.
  Variable accessed by threads (mode):
   * Thread 'Thread-4(Thread@506411)' (WRITE) at 'Airlines.java 94'
      - Thread candidate locks: none.
   * Thread 'Thread-6(Thread@15c7850)' (WRITE) at 'Airlines.java 84'
      - Thread candidate locks: none.
You probably should use a lock, we suggest to make a new lock to protect
this variable or use one of the threads candidate locks printed above.
```

Figure 7.5: A warning produced by the Eraser+ algorithm.

7.4 Healing

This section presents the implementation of the healing approaches introduced in Chapter 5. At first, the healing characteristics and implementation are mentioned and then all the implemented techniques are described.

As explained in Section 5.1, the healing is based on influencing the execution of threads of the managed application with the intention to force the execution to follow the predefined atomic sections in the code of the managed application. The main disadvantage of the current implementation is that it is not able to heal the first occurence of the race. The problem with healing even the first occurence is caused by the fact that the dynamic detection algorithms usually detect a data race or atomicity violation occurence when the threads are already in conflict.

Because healing influences threads, `RDThread` class is used to access to healing algorithms as can be seen in the class diagram in Figure A.4. Eight different healing methods have been implemented using the principles introduced in Chapter 5. Most of them are based on influencing the scheduler a technique described in Section 5.4, and only one introduces an additional synchronization as was showed in Section 5.3.

The core healing algorithm depicted in Figure 5.3 is implemented in the `Healing` class using the `startHealing()` and `stopHealing()` methods. These methods transform the detector events into healing events (*before/afterNormalAccess*, *before/afterCriticSection*). The class also contains the `initHealing()` and finalHealing() methods used for an initialization and finalization of the healing mechanisms—if it is needed. All classes which implement particular healing techniques are inherited from the `Healing` class. A factory method `getHealingMethod()` is used to initialize the concrete healing method for each `RDThread` object. Thus, each thread can use a different healing method, if necessary. The following subsection presents all the implemented healing methods.

7.4.1 Implemented Healing Techniques

A list of all implemented healing techniques begins with techniques which are based on influencing the JVM scheduler. At the end the only technique introducing a new synchronization is listed. The technique based on semaphors also introduced in Section 5.3 was not yet implemented. However, its implementation is very similar to the technique based on explicit locks and therefore can easily be added.

- **Yield**. The `HealingYield` class implements the simplest method of influencing the execution. The technique has already been described in Section 5.4 and is depicted in Figure 5.5.

- **Priority**. The `HealingPriority` class implements method based on changing thread priorities introduced in Section 5.4. It uses only the *beforeCriticSection* and *afterCriticSection* events. Before a thread enters an atomic section, its current priority is saved in particular `HealingPriority` object. Then, the new priority is set to the maximum by calling the `Thread.setPriority()` method. When the thread leaves the atomic section, the original priority is restored.

- **YieldPriority**. The `HealingYieldPriority` class implements the technique that combines both previously described solutions. In the *beforeCriticSection* event, the priority of the thread is increased first and then a `Thread.yield()` is called. The

idea is to receive a new time slice from the scheduler and continue with the highest priority. The priority is set back at the *afterCriticSection*.

- **OTYield**. The `HealingOTYield` class implements the technique depicted in Figure 5.6. The `Thread.yield()` method is called in situations when some other thread is inside an atomic section associated with the samve variable. The $Counter(v)$ introduced in Figure 5.6 is implemented using the `AtomicInteger` class from the `java.util.concurrent` package. Such a counter provides an atomic incrementation and decrementation of its value. The counter is stored in `RDVariable` class because it must be unique for each instance of the shared variable.

- **OTWait**. The `HealingOTWait` class implements similar technique to the previous one. It uses call of `Thread.wait(0, 10)` instead of `Thread.yield()`. This causes the thread to wait for a ten nanoseconds. The delay has been chosen with respect to the performance of the healing technique. Of course, bigger atomic sections should have this delay longer and short atomic section shorter. In general, the delay should be long enough to let the thread which is in an atomic section to leave it.

- **Threads**. The `HealingThreads` class implements the technique introduced in Section 5.4 which uses several high priority "dummy" threads to utilize all processors but one. The `HealingDummyThread` class contains such a thread which does nothing useful and can not be optimalized by the compiler or JVM. The technique can activate them by calling the `HealingDummyThread.goAll()` method and passivate them by calling `HealingDummyThread.endAll()`. The healing technique activates the "dummy" threads and increases the priority of the current thread to maximum in *beforeCriticSection* event. Threads are passivated and the priority is turned back in *afterCriticSection* event.

- **YieldThreads**. The `HealingYieldThreads` class slightly modifies the previous technique. Because the bunch of "dummy" threads needs to be activated by the scheduler, the current thread helps it by calling `Thread.yield()` method after it call `goAll()`. This should give the scheduler oppurtunity to start the "dummy" threads before the thread increases its priority and enters the atomic section.

- **Newmutex**. The `HealingNewMutex` class implements the technique based on introducing an additional synchronization as was described in Section 5.3. The technique is depicted in Figure 5.4.

7.5 Obtaining Atomic Sections

FindBugs introduced in Section 2.8 is used to implement a static analysis for obtaining *atomic sections*. An atomic section is block of code that is likely to be assumed by the programmer to exectute atomically. FindBugs has been chosen because it is free, open source, and often used during the development phase. A very brief and sometimes not up-to-date documentation of FindBugs significantly complicates the development of a new analysis. Moreover, FindBugs has only a very limited support for interprocedural analysis. All these reasons caused that the current implementation of obtaining atomic sections is limited. The pattern-based static analysis detects only load-and-store atomicity patterns only if it is written on one line in the code, and the AI invariant detection is implemented using an intraprocedural data flow analysis.

The static analyses work with Java bytecode instrumented by ConTest shown in Figure 2.2. The instrumented bytecode has been chosen because it already contains the code that triggers the events monitored by ConTest. This instrumented code includes the ConTest identifiers of program locations and ConTest identifiers of shared variables which are then used as identifiers in structures mapping an event to the related `RDVariable` object.

The rest of this section is organized as follows. Firstly, the external repository implementation is described. Then, AI invariant-based analysis implementation is presented, followed by the pattern-based analysis which uses the results of the AI invariant-based analysis. Finally, the dynamic refinement implementation is introduced.

7.5.1 Implementation of the External Repository

The external repository is heavily used by the healing process for identifying blocks of code that should be executed atomically and, in the case of atomicity violations, also during the detection process. Therefore, the atomic sections has to be present in the memory during the execution. For this purpose, the `AtomicityCollection` class has been designed. It provides access to a set of atomic sections which are identified in advance by the static analysis. Also, a static analysis uses this class to maintain the atomic sections that it has identified. The class provides the `storeAtomicityToFile()` and `restoreAtomicityFromFile()` methods which produce or read a file with atomic sections in the XML (eXtensible Markup Language). The format of the XML file is defined by the DTD (Document Type Declaration) listed in Figure 7.6.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT ATOMICITY (SIMPLEATOM | DOUBLEATOM | TRIPLEATOM | MULTIATOM)*>
<!ELEMENT SIMPLEATOM (BEGIN, END)>
<!ELEMENT DOUBLEATOM (BEGIN, END, END)>
<!ELEMENT TRIPLEATOM (BEGIN, END, END, END)>
<!ELEMENT MULTIATOM (BEGIN, END+)>
<!ELEMENT BEGIN EMPTY>
<!ELEMENT END EMPTY>
<!ATTLIST BEGIN loc CDATA~"">
<!ATTLIST BEGIN mode (read | write) "read">
<!ATTLIST END loc CDATA~"">
<!ATTLIST END mode (read | write) "read">
```

Figure 7.6: The DTD defining the format of XML files containing atomic sections.

The atmoic sections which are from the implementation point of view represented by the tuple $(loc_{entry}, Loc_{ends})$. The loc_entry is the entry point to the atomic section and Loc_{ends} is a set of locations where the atomic seciton ends. The Loc_{ends} locations has to cover all possible CFG branches starting from loc_{entry}. This design has been chosen for better caching ends of atomic sections. Whenever an atomic section is entered, the Loc_{ends} set is cached. The following stream of locations in the code (places where the following events are triggered) are then compared only against the cached sets. To speed up the comparison, a several types of atomic sections have been implemented. They differ only in handling of the Loc_{ends} set. They are implemented in the `SimpleAtomicity`, `DoubleAtomicity`, `TripleAtomicity` and `MultiAtomicity` classes. All these classes implements the `Atomicity` interface which allows for a consistent access to them.

7.5.2 AI Invariant-based Static Analysis

The AI Invariant-based static analysis has been introduced in Section 6.2. The implementation of this analysis can be divided into two parts. The first part contains a dataflow analysis producing data flow values for each basic block. The second part contains an analysis (also called *detector*) which uses the data flow facts computed by the first analysis to produce atomic sections in the form $(loc_{entry}, Loc_{exit})$.

Let start with the first part of the analysis. FindBugs provides a dataflow analysis engine which is utilized by the AI invariant-based dataflow analysis. The new dataflow analysis has to register itself into the set of analyses that FindBugs will execute over the code. This is not an easy step and requires to implement several classes which do the job. The set of analyses is maintained in the so-called *AnalysisCache*. The `EngineRegistrar`, `AtomDataflowEngine`, and `AtomDataflowEngine` classes register the new analysis into the *AnalysisCache* and define the interface between FindBugs and the new analysis. The new dataflow analysis is represented by the `AtomDataFlow` class inherited from the FindBugs `DataFlow` class and is parametrized with two classes—similarly as Java collections are parametrized. The first parameter is the `AtomFact` class which implements the *data flow value*. The second parameter is the `AtomAnalysis` class which is ihnerited from the FindBugs `ForwardDataflowAnalysis` class and implements the *flow functions*.

The lattice encapsulated by the `AtomFact` class has to be extended by a new maximum labeled as *TOP* which represents a special value of the fact before it is initialized during the first pass of the analysis, and a new minimum labeled as *BOTTOM* which represents an indication that the fact contains a wrong value because an error occured during the analysis. The flow functions then have to be modified so that *BOTTOM* value is spread to all places which computes with this value.

The analysis encapsulated with the `AtomAnalysis` class is based on two main methods. The `meetInto()` method implements the $In_{AI}(bb)$ function introduced in Section 6.2, and the `transferInstruction()` method implements the $Out_{AI}(bb)$ function.

In fact, the analysis does not track accesses to variables but only calls of ConTest instructions representing *beforeAccessEvent*s injected into the code during the instrumentation. This helps the analysis to produce only information related to events tracked by the healer. The `CTMethods` class and mainly its method `processVarConTestMethod()` is used to derive ConTest program location string and ConTest variable identification string from the parameters of the processing ConTest instrumentation point.

The second part of the analysis is then implemented in the `FindAtomicity` class. This class inherits the FindBugs `CFGDetector` class which does the work of traversing the code. For each method, the FindBugs analysis environment invokes the `visitMethodCFG()` method of the `FindAtomicity` class. This method processes each basic block of the method and based on the available `AtomFact` objects, constructs the atomic sections that appear in the method. Finally, at the end of the analysis, the `finnishPass()` method is called and the detected atomic sections are stored into the XML file.

7.5.3 Pattern-based Static Analysis

The pattern-based static analysis introduced in Section 6.1 has been implemented on top of the AI invariant-based analysis. This solution has been chosen because it was the easiest way how to obtain atomic sections covering the given patterns and containing also the exit event program locations. Currently, only the load-and-store pattern is supported. Moreover, the detection is limited to statements written into one line of the source code.

The analysis is implemented in the `FindPatterns` class. This class does the same work as the `FindAtomicity` class, and, at the end, it filters out atomic sections that span more than one line of source code. Finally, if there are several AI invariants which form one instance of the load-and-store pattern, they are joined into one atomic section starting at the first read access to the shared variable and ending at the last write access on the same line of the source code. The exit events are collected from all joint atomic sections.

7.5.4 Dynamic Refinement

As was explained in Section 6.3, the dynamic refinement based on the AtomRace algorithm is used to prune the set of atomic sections obtained either from the user or by one of previously described analyses. The dynamic refinement assumes a presence of some oracle which can distinguish correct and problematic executions. There is no such generic oracle designed yet and the classification has to be done by some third party tool or by the user. Because of this, the described dynamic refinement is not tightly interconnected with the rest of the implemented architecure. It works as follows.

If the *atomRaceLearn* option is specified in the healer configuration file, the AtomRace algorithm produces an XML output file at the end of the execution. This file contains all atomic sections violated during the execution. A simple Java program called `RemoveAtomicity` which uses the `AtomicityCollection` class loads two XML files where the first represents the set of atomic sections assumed to be guaranteed in the application and the second represents the set of atomic sections that were violated. Then, when an oracle or the user says that the run did not fail, the program can be used to remove violated atomic sections from the set of atomic sections of the application. This solution allows to write a third party oracle which is then able to execute the detection and the obtained results use for refinement of atomic sections. The threshold introduced Section 6.3 was not yet implemented.

Chapter 8

Experiments

This chapter summarizes experimental results obtained from the prototype implementation of the proposed techniques described in the previous chapters. The chapter shows both strong and also some weak features of the algorithms and their implementation. Three test cases have been chosen to prove the concepts. They are used to examine the basic behavior of the proposed solutions and to experiment with dependencies of their behavior on the way their parameters are set. The simplest one is used to show the basic behavior. The two more real life examples are used to demonstrate the usability of the prototype. The chapter is organized as follows. Firstly, the environment in which the described experiments carried out is introduced, and the considered test cases are described. After that, practical experiences with different implemented approaches for obtaining atomic sections are presented followed by an comparison of proposed algorithms. Finally, different healing techniques are discussed.

8.1 Experimental Environment and Test Cases

As was already mentioned in the introduction, the execution of a multi-threaded application highly depends on the multi-threading capabilities of the used hardware and software. Therefore, all tests have been done on several architectures which vary in the number of available processors: one processor Intel Celeron 2,4 GHz, two processors 2xIntel Xeon 1,7GHz, four processors 2xDual Core AMD Opteron 2220, and eight processors 2xQuad Core Intel Xeon 5355. All machines run Linux with the 2.6 kernel, Sun JVM version 1.5, and ConTest version 2.6.5.3.

Three different test cases have been chosen to examine behavior of the described approaches. The test cases differ in size and in the circumstances under which an error can occur within them. To better understand dependencies between the behavior of the proposed techniques an the way their parameters are set, each test case was run with 62 different settings of the implemented prototype. The tests studied both proposed bug detection algorithms, the influence of different noise injecting techniques on their efficiency, and the fruitfulness of all implemented healing techniques. Finally, a few more tests focused on scalability were performed.

A simple bank account simulating program. This considered test case with an improper synchronization is the simplest test case. The program contains two classes and simulates bank accounts where multiple account threads perform simple changes to the particular accounts and the total balance of the bank without a proper synchronization

```
public static void Service(int id, int sum) {
   accounts[id].Balance += sum;                // thread safe
   BankTotal += sum;                           // data race
}
```

Figure 8.1: A problematic method in a bank account simulating program.

over the global balance variable `BankTotal`. The problematic method is depicted in Figure 8.1. The method is called from all account threads and simulates an operation with account balances. In the method, two variables are changed in the same way. The `Balance` variable is unique for each account thread, and hence, there is no race possible if a correct thread `id` is used as the parameter. The `BankTotal` variable is shared among all threads and there is a load-and-store bug pattern on it. When no operation with `BankTotal` is interrupted, the final balance corresponds to the performed operations. However, if some operation with `BankTotal` is interrupted, the final balance may get wrong. The problematic method is called many times during the execution of the test case. In the tests, a 2 threads calling the `Service()` method are considered and between each two calls a loop with 15000 iterations simulating some demanding work is performed. The problematic method is called 300 times from each thread during each execution.

The IBM web crawler algorithm. This algorithm has been embedded in an IBM product called WebSphere. The skeleton of the algorithm has 19 classes and 1200 lines of code. The program creates a set of threads waiting for a connection. If a connection simulated by a testing environment is established, a worker thread serves it. After that, the worker is again ready to serve another connection. The problematic method is shown in Figure 8.2. This method is called when the crawler is being shut down. If some worker thread is just serving a connection (`connection != null`), it is only notified not to serve any further connection. This notification is done within the `finish()` method from Figure 8.2 by a thread performing the shutdown process. A problem occurs if the `connection` variable is set to `null` by a worker thread (a connection was served) between the check for `null` and a `setStopFlag()` method invocation. This case corresponds to a test-and-use bug pattern and such a situation causes an unhandled `NullPointerException`. Contrary to the previous race example, this race manifests in a minimal number of executions. More precisely, in 4 out of 6200 executions done with all predefined options on each machines. The problem manifests only as a result of heavy influence of injected noise focused on the problematic variable. Each test execution runs 10 worker threads serving approximately 10 connections.

Such a low probability of the problem manifestation does not allow to test healing methods because it would be hard to distinguish if the problem does not occur because of healing technique or because a conflict situation does not occur. Therefore the propability of a conflict was increased by inserting a loop with 10000 iteration between the check for `null` and the `setStopFlag()` method invocation. Even with this modification, the problem occurs only in a relatively low percentage of test executions as can be see in Table 8.3 (column SA). Note, that by this modification a different atomic section than that in the previous test case was created. In the bank test case, the block of code that should be executed atomically is very short. Here, the block of code is much longer what is used to demonstrate the influence of an atomic section size on healing techniques.

The Java TIDorb project. This system developed by Telefónica Spain has more than 1400 classes and represents the biggest test case. Java TIDorb is a CORBA-compliant ORB

```
public void finish() {
    cont = false;
    Debug.print("worker finishing");
    if (connection != null) connection.setStopFlag();    // data race
    if (workerThread != null) workerThread.interrupt();
}
```

Figure 8.2: A problematic method in the IBM web crawler program.

(Object Request Broker) product that is a part of the MORFEO Community Middleware Platform [40]. Several tests created by the developers are available in the project repository[1]. The basic *echo concurrent* test has been chosen for tests presented here. The test starts a server process for handling incoming requests and a client process which constructs several client threads each sending several requests to the server. The server constructs several threads which serve the requests. If there are no enough server threads available, the client threads produce a timeout exception and retry later. The test runs 10 client threads each sending 40 requests to the server. The server runs 10 threads to serve the clients requests.

Contrary to the previous test cases, TIDorb does not contain any previously known data race or atomicity violation—and as is described in Section 8.3.3, several data races has been identified by the prototype. On the other side, the TIDorb has not been tested for data races before.

8.2 Obtaining Atomicity

Atomicity of the managed application is important for a correct healing and atomicity violation detection. Obtaining atomicity is so far the weakest point of the architecture. This was clear in advance because the inferring of atomic sections is a real problem addressed by ongoing research. Therefore, all listed approaches or implementations have limitations.

The pattern-based analysis introduced in Section 6.1 works fine, despite, the current implementation supports only load-and-store atomicity pattern. The analysis correctly detects both patterns in bank test case depicted in Figure 8.1, 2 patterns in crawler test case, and 155 patterns in TIDorb. 1 pattern detected in bank test case and 10 patterns detected in TIDorb test case were related to variables on which a data race has been detected. Therefore, these atomic sections could be used for healing and atomicity violation detection. If the implementation of the pattern-based analysis would support also the test-and-use bug pattern, all identified data races could be healed.

The dataflow static analysis producing the initial set of AI invariants and proposed in Section 6.2 works also well, despite, it currently supports only detection of intraprocedural atomic sections. This limitation did not influent the fact, that all atomic sections related to identified data races in all test cases have been detected correctly because the identified atomicity problems on problematic variables were all also intraprocedural.

The inferring mechanism proposed in Section 6.3 did not work because the proposed threshold was not implemented. The experiments with dynamic refinement showed that the current implementation detects only correctly synchronized atomic sections (those which refer to blocks of code correctly synchronized). But, for atomic sections which refer to blocks

[1] https://svn.forge.morfeo-project.org/svn/tidorbj

of code that should be atomic but they are not, the refinement did not work. In some cases, AtomRace correctly identifies atomicity violation but the oracle based on handling an exception or checking the output of the managed application did not detect a problem. For example, if the atomic section covering the test-and-use bug pattern in crawler test case is interleaved between the start of checking for atomicity violation in *beforeAccessEvent* and an execution of the instruction which reads the `connection` variable, the `nullPointerException` is not produced, despite, there is a problem. In such cases, the atomic section which should stay in the set was removed. This problem can be lowered by using the proposed threshold. But note, that even the proper setting of the threshold might be a hard task.

The problem described in the previous paragraph was solved in the way that refinement was done by hand. Only atomic sections related to identified problems were left in the set of atomic sections and all other atomic sections were removed. This way a set used during the tests described in the following sections to heal a problem and detect atomicity violations was constructed.

8.3 A Comparison of Detection Algorithms

This section presents the experimental comparison of the detection algorithms that have been proposed and implemented. Firstly, the results obtained from our experimental bank and crawler test cases are discussed. Then, the detection efficiency of the algorithms in TIDorb test case is compared.

Experiences obtained from all test cases showed, that the best way of using the implemented architecture for detecting data races and atomicity violations, e.g., during a debugging process, is to use a combination of both implemented detection algorithms. Firstly, the potentially problematic variables are pinpointed by the Eraser+ algorithm. Then the AtomRace algorithm with predefined proper atomic sections is used in combination with the AtomRace specific noise injection focused on variables pinpointed by the Eraser+ algorithm. This way, the possible false alarms produced by the Eraser+ algorithm are cut out and a more contributing warning produced by AtomRace is obtained. Moreover, in some cases, the lock suggestion mechanism implemented in Eraser+ can help to identify which lock could be used to remove the detected problem.

8.3.1 The Bank Test Case

The Eraser+ algorithm correctly detects that the variable `BankTotal` is not guarded by any lock and therefore there can be a data race. The algorithm did not produce any false alarm, although the original Eraser algorithm which does not support join-synchronization would produce false alarms for each `Thread.balance` instance. The AtomRace algorithm also correctly detected the bug without any false alarm.

The results of data race detection in bank test case are shown in Table 8.1. The table contains 4 rows, each for a different testing machine. The first column determines the machine by the number of available processors. The following columns contain two values for each cell in a form x/y. The x value represents the ratio of runs in which a race occurred and the y value represents the ratio of a successful problem detection. The *SA* column contain only x values because no detection algorithm was active. The other columns in the table presents results for the following configurations of the healer:

- **SA**. This column shows a ratio of a race manifestation (a bad result was obtained) if the application was run without the healer attached.
- **ER+**. This column contains values obtained if the Eraser+ algorithm was used without any additional noise.
- **AR-data**. This column shows the efficiency of AtomRace algorithm if there are no atomic sections available. In this setting, the AtomRace detects only data races.
- **AR+atom**. This column shows the efficiency of AtomRace if proper atomic sections are specified.
- **AR+noise**. This column shows the efficiency of AtomRace algorithm with proper atomic sections specified and ConTest noise injection activated with a noise frequency 100 (out of 1000—means to every instrumented location).
- **AR+rand**. This column contains values obtained if the AtomRace algorithm with proper atomic sections was used and AtomRace specific random noise introduced in Section 4.2.3 noise injection is activated with the noise frequency 100 and noise duration 5 nanoseconds.
- **AR+var**. This column shows the efficiency of AtomRace algorithm with proper atomic sections and AtomRace specific noise focused on the problematic variable `BankTotal` with noise frequency 100 and noise duration 5 nanoseconds.
- **AR+loc**. This column contains values obtained if AtomRace algorithm with proper atomic sections and AtomRace specific noise focused on the problematic locations (both in load-and-store pattern) is activated with the noise duration 5 nanoseconds (frequency for this kind of noise is every time 1000).

As can be seen from Table 8.1, the race in bank test case was manifested only in 1 of 100 executions on an 8 processor machine. This low ratio of race manifestation increases when the healer is attached. This is caused by the fact that the code of the healer is also executed within the atomic section and therefore there is a higher probability for interleaving by some another thread. The Eraser+ algorithm, in the ER+ column, detects the problem in all executions because none lock was used. The *AR+atom* column shows that with a given proper atomic section the AtomRace correctly detects all manifestations of a bug. A slightly higher ratio of detected bugs than of race manifestation in this column shows that AtomRace correctly detects atomicity violations which does not cause a wrong result detected by the used oracle (the result was correct). The reason for this result was described in the previous chapter in the previous section in part of dynamic inferring. The next columns demonstrates dramatic influence of noise injection techniques on race manifestation and related detection efficiency of the AtomRace algorithm. The noise injection techniques can be compared also with respect to amount of noise injected into the execution. This can be seen from the following table.

Table 8.2 shows the performance degradation caused by the detection techniques presented in Table 8.1. The values in this table were obtained when an average execution time of all threads was divided by the average execution time of all threads of an execution without any detector nor ConTest attached. Therefore, the quotient in the column *SA* is equal to 1. It shows that slow down caused by the Eraser+ algorithm (column *ER+*) is slightly higher than the slow down caused by the AtomRace algorithm. This is caused

proc	SA	ER+	AR-atom	AR+atom	AR+noise	AR+rand	AR+var	AR+loc
1	0	0,70 / 1,00	0,10 / 0,04	0,07 / 0,09	1,00 / 1,00	1,00 / 1,00	1,00 / 1,00	0,07 / 0,11
2	0	0,20 / 1,00	0,20 / 0,12	0,33 / 0,39	0,96 / 0,99	1,00 / 1,00	1,00 / 1,00	0,25 / 0,35
4	0	0,28 / 1,00	1,00 / 0,84	0,47 / 0,57	1,00 / 1,00	1,00 / 1,00	1,00 / 1,00	0,42 / 0,50
8	0,01	0,56 / 1,00	0,78 / 0,16	0,69 / 0,70	1,00 / 1,00	1,00 / 1,00	1,00 / 1,00	0,44 / 0,78

Table 8.1: Detection efficiency of implemented algorithms in the bank test case.

proc	SA	LB	AR-atom	AR+atom	AR+noise	AR+rand	AR+var	AR+loc
1	1	1,27	1,22	1,23	1,94	1,36	1,25	1,23
2	1	1,08	1,07	1,07	1,38	1,37	1,14	1,08
4	1	1,01	1,01	1,01	1,14	1,51	1,08	1,01
8	1	1,15	1,14	1,15	4,57	5,97	2,08	1,15

Table 8.2: Performance degradation caused by different detection techniques.

mainly by the fact that there is no need for computing locksets in AtomRace. Both of them cause a moderate performance degradation (ranging from 1% to 22% depending on the architecture) to the test case execution. The table also demonstrates that AtomRace injection mechanisms, mainly the one focused on problematic variable (*AR+var* column) does not slow down the application as much as others. ConTest injects noise to more locations with the intention to see as many different interleavings as possible and therefore the used frequency 100 per mille injects more noise in the application.

It is important to note, that although the bank test case is used here to illustrate the detection capabilities it is proposed mainly for testing of healing. Real data races or atomicity violation occurs much less often as can be demonstrated on the following examples.

8.3.2 The Web Crawler Test Case

In the case of the web crawler, the Eraser+ algorithm produces a warning for each instance of the problematic variable `Worker.connection` which was used during the execution. But, all warnings are in fact false positives. The problem here is that a wait-notify synchronization is used to control accesses to `connection` variable. A connection is firstly assigned to a worker and then the worker is notified. This kind of synchronization is not supported by Eraser+. On the other hand, the lock suggestion mechanism suggests a correct lock which was used with the variable during the initialization.

The very low probability of race manifestation in non-modified crawler test case causes that also AtomRace does not detect the problem. More precisely, it detects only all four executions in which the problem manifested. All four occurrences were seen when an AtomRace algorithm in combination with the AtomRace specific noise injection focused on the `Worker.connection` variable was used (frequency 800 per mile and strength 15 ns). Three of them were detected on 1 processor machine and one on 4 processors machine.

Table 8.3 shows the detection efficiency in a modified version of crawler test case. The

proc	SA	ER+	AR+atom	AR+noise	AR+rand	AR+var	AR+loc
1	0,11	0,08 / 2,03	0,42 / 0,43	0,06 / 0,10	0,04 / 0,08	0,24 / 0,38	0,13 / 0,13
2	0,08	0,13 / 2,27	0,70 / 0,70	0,43 / 0,43	0,00 / 0,00	0,08 / 0,13	0,00 / 0,00
4	0,35	0,44 / 2,12	0,70 / 0,71	0,07 / 0,08	0,01 / 0,03	0,29 / 0,40	0,00 / 0,00
8	0,06	0,24 / 2,17	0,74 / 0,76	0,01 / 0,02	0,02 / 0,02	0,19 / 0,33	0,00 / 0,00

Table 8.3: Detection efficiency of implemented algorithms in the crawler test case.

rows and columns describe the same configurations and metrics as in previously described Table 8.1. This time, the race detection and manifestation is highly dependent on the execution trace. Even the lockset based algorithm does not detect data race on all ten instances of the problematic variable as can be seen in the *ER+* column. AtomRace with correctly specified atomicity section span the test-and-use atomicity pattern again correctly detects all occurrences of the problem and also some others. But this time, any injected noise did not increased the probability of the race manifestation. Even, the rule that more noise inejected into the program causes more race manifestations does not hold. Sometimes, the noise worked for some machine (usually one processor machine) and did not work for others. This behavior confirmed the tricky nature of concurrency problems which manifests only under some circumstances.

8.3.3 The TIDorb Test Case

In this test case, there were no previously known bugs and so this test case poses a real challenge to the implemented detectors. This section firstly describes how the detection algorithms helped to find some data races and then the identified data races are described. At the end, the performance and scalability of the algorithms is discussed.

At first, the Eraser+ algorithm has been used. During several executions with different levels of noise, the detector identified 15 *suspected variables*—variables which were identified by the algorithm as variables which were operated in the way that a data race can manifest. Although the warnings contain all possible information available in the Eraser+ algorithm, reasoning about possible races was sometimes very hard and time consuming, especially, when the problematic variable was accessed from several classes, c.f., Figure 8.3.

Then, the AtomRace detection algorithm has been used. Even without any introduced noise, the algorithm identified 2 suspected variables. When combined with additional random noise, the number of suspected variables increased to 3. Then, a set of variables identified by the Eraser+ algorithm was given to AtomRace and the variable oriented noise injection was used. This way, the number of identified suspected variables increased to 5.

In the case, the reasoning about possible races was much easier than in the case of the previous approach. AtomRace pinpointed two places in the code, sometimes far away each from other, and gave information which access from the two pinpointed was changing the value. Such information lead to a nearly immediate understanding of what was going on.

The results of detecting data races in the TIDorb test case are summarized in Table 8.4. Eraser+ identified 15 suspected variables and AtomRace 5 of them. All data races detected

```
class TypeCodeImpl{
   boolean m_exhaustive_equal = false;
   ...
   public void setExhaustiveEqual(boolean value){
      if (m_exhaustive_equal != value) // To avoid warning about race
         m_exhaustive_equal = value;
   }
}
class ValueTypeCode{  // also the same for other *TypeCode classes
    ...
    public boolean equal(org.omg.CORBA.TypeCode tc){
        if (!m_exhaustive_equal)
            return true;
        ...
    }
}
```

Figure 8.3: A data race in TIDorb Java.

by the AtomRace algorithm (and also by Eraser+) were true races. An unsupported kind of synchronization was identified for 2 variables identified by the Eraser+ algorithm. The threads accepting connections could not access a variable (and cause a true race) before the server start to listen for incoming connections. For 8 suspected variables identified only by Eraser+, there was not identified whether they are true races (some kind of oracle which can detect the buggy execution can be constructed) or false alarms (there is for sure, no possible data race).

For all five true races, oracles which can detect if the data race manifests during the execution were constructed. These oracles were used for demonstrating that the races are true races and also for evaluating of the healing capabilities of the proposed architecture in the next section. For example, if the data race manifestation leads to a `NullPointerException`, the problematic block of code was enclosed by the `try`–`catch` construction and the exception was handled.

	Suspected	True Races	False Alarms	Unclear
Eraser+	15	5	2	8
AtomRace	5	5	0	0

Table 8.4: Detected data races in the Java TIDorb test case.

The following paragraphs describe some interesting races identified by the algorithms.

The first data race pattern is depicted in Figure 8.3. The variable `m_exhaustive_equal` from the `TypeCodeImpl` class is used as a flag but is not defined `volatile`. The wrong value of `m_exhaustive_equal` causes in some cases a wrong result of the `equal` methods, e.g., in the `ValueTypeCode` class. Note, the construction in `setExhaustiveEqual`. The `if` statement and its comment have been introduced by the developers of TIDorb after their experiments with Eraser+ data race detection algorithm—in that time Eraser+ also warns about volatile and final variables and so they were overburden with false alarms. This construction lower the probability that the algorithm detects the race but does not remove

```
class IIOPCommunicationDelegate extends CommunicationDelegate{
   ...
   public void invoke(RequestImpl request) {
      try {
         if ( this.forwardReference == null ) {
            ...
         } else {
            this.forwardReference.invoke(request);
         }
      } catch (org.omg.CORBA.COMM_FAILURE cf) {
         this.forwardReference = null;
         throw cf;
      }
   }
}
```

Figure 8.4: The fourth data race in TIDorb Java.

the race from the program. Instead, a test-and-use bug pattern has been introduced and the race become more tricky because it will manifest more rarely.

The second data race has been identified in the `IIOPCommLayer` class on `recover_count` variable. This variable is used to count the number of retry actions if a communication related exception is caught. The `recover_count--;` statement followed by `if (recover_count < 1)` statement is used to control the number of retry actions. The data race manifestation causes that threads do a retry operation more times than it was specified. This data race has nearly no impact on the correct behavior of the application, and thus, could be in some cases tolerated.

The third data race has been identified in the `IIOPProfile` class on `m_listen_point` variable. The variable is tested for `null` value at first and then is set to a new value within the method defined as `synchronized`. But, because a test for the `null` value is out of the synchronized method, the value can be set by one thread to a value which is then overridden by a value from an another thread which passed the test for `null` before the first thread managed to change the value.

The fourth data race is depicted in Figure 8.4. The problematic variable `forwardReference` can be by one thread set to `null` in the `catch` branch while an another thread going to invoke a method on `forwardReference`. Such a situation causes the `NullPointerException`. This race is again very rarely manifested because it rely on exception produced within the `try–catch` block.

The noise influence on data race detection in TIDorb example was similar to the results presented for the previous test cases. The TIDorb test case is suitable to compare the performance overhead and scalability of both detection algorithms because it is relatively big, complex, and real test case.

The performance overhead and scalability of the implemented algorithms can be seen from the graph in Figure 8.5. The graph shows dependency of the number of TIDorb client threads (each sending 40 requests) and average time of a single client thread execution. The numbers of client threads are on the x-axis and average time on the y-axis. Both detection algorithms are compared with an execution of instrumented bytecode when only ConTest is active. The Eraser+ algorithm was ran with lock suggestion mechanism disabled because

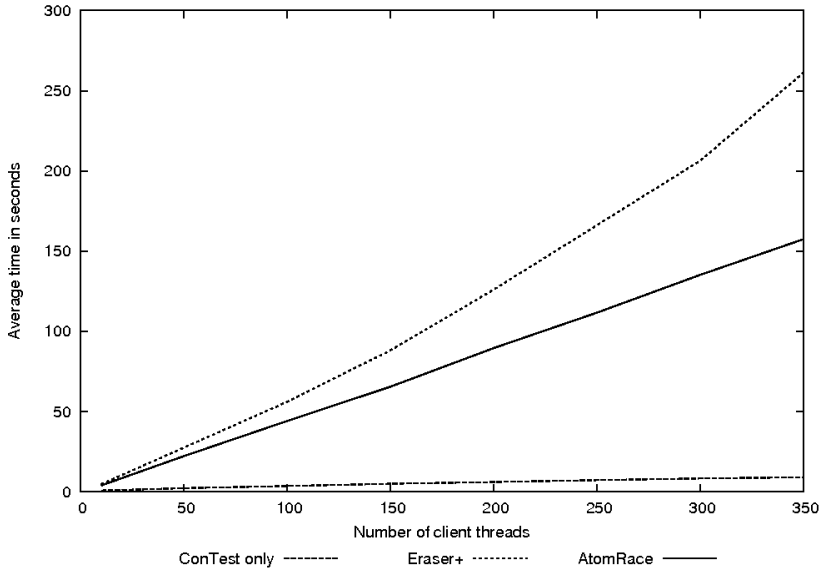


Figure 8.5: The performance overhead and scalability of the algorithms.

otherwise the experiment ends with an `OutOfMemoryException` for number of client threads higher than 200. It can be seen that AtomRace algorithm has lower overhead and slightly better scalability than Eraser+ algorithm even when the Eraser+ algorithm worked on the level of the original Eraser. It is important to note, that the current implementation can be further optimized as it is discussed in the future work. For example, utilization of partial instrumentation could dramatically decrease the overhead caused by the algorithms.

8.4 A Comparison of Healing Techniques

This section discuss different implemented healing techniques. Healing abilities are discussed on data races described in the previous sections. The healing efficiency is tested using assertions (oracles) introduced into the original code of test cases. This assertions already used in the previous section allows to detect if the detected data race manifested, e.g., in the case of web crawler test case if a `NullPointerException` is threw within the problematic block of code. This section is structured similarly to the previous one. Firstly, the results obtained with the bank test case are introduced, followed by the results of web crawler and TIDorb test cases.

8.4.1 The Bank Test Case

The data race in the bank example occurs very often. This makes the healing more complicated. On the other hand, the problematic block of code is relatively short what can be an advantage for some healing techniques. Table 8.5 shows how the implemented healing techniques fight with the problem. For a better understanding the *ER+* column taken from Table 8.1 is repeated with the intention to show the ratio of race manifestation before a healing is enabled. This column contains percentage of runs where the race manifested if the lockset based algorithm was used to detect data races. Other columns contain results of different healing techniques introduced in Chapter 5 and described in Section 7.4.1:

proc	ER+	Yield	Priority	YieldPrior	NewMut	OTYield	OTWait	Threads	YieldThr
1	0,62	0,17	0,14	0,07	0,01	1,00	1,00	0,14	0,09
2	0,20	0,20	0,28	0,32	0,09	0,23	0,01	0,32	0,28
4	0,28	0,31	0,97	0,29	0,05	0,94	0,52	0,25	0,99
8	0,56	0,48	0,54	0,40	0,15	0,46	1,00	0,53	0,50

Table 8.5: Efficiency of healing techniques in the bank test case.

- **Yield**. The simplest method using call to `Thread.yield()` of the thread entering the atomic section. The method is depicted in Figure 5.5.
- **Priority**. The method based on changing thread priorities introduced in Section 5.4.
- **YieldPrior**. The technique that combines both previously described solutions.
- **NewMut**. The technique based on introducing an additional synchronization which was described in Section 5.3.
- **OTYield**. The method uses call to `Thread.yield()` if some other thread is inside a critical section. The method is depicted in Figure 5.6.
- **OTWait**. The method uses call to `Thread.wait(0, 10)` if some other thread is inside a critical section.
- **Threads**. The technique uses several high priority "dummy" threads to utilize all processors but one.
- **YieldThr**. The technique uses several high priority "dummy" threads to utilize all processors but one and call to `Thread.yield()` method.

As can be seen from Table 8.5, in most cases the probability of race manifestation has been decreased in comparison to an execution with healer attached (*ER+* column). Of course, the healer code inside the problematic block of code increased the probability of conflict occurrence. Therefore these probabilities are higher in comparison to execution without the healer attached (see Table 8.1). Note that even the technique based on introducing an additional synchronization listed in column *NewMut* did not heal the race totally. This is caused by the fact that the race is detected when it occurs and threads are already in conflict. Therefore, the healing strategy based on encapsulating the problematic block of code can not be used because the begin of such code has already been executed. The healing even first occurrence of data races and atomicity violations is the object of future work.

It can also be seen that some methods did not heal the race. For example, *OTYield* and *OTWait* based on calling `wait()` or `yield()` in other threads then those which are in the atomic section. This results from the repetitive execution of the problematic code. The interleaving thread is delayed only once and then it can continue which can result in conflict with another iteration through the problematic section. The only exception is two processor machine in *OTYield* column. The quite different result is caused by the fact that injected noise by chance fits perfectly the time which the thread executing atomic section needs to escaping it. There are many more things that influences the fruitfulness of healing. For example, it can be seen from the *priority* column that AMD processors differently works with threads.

proc	CT+	Yield	Priority	YieldPrior	NewMut	OTYield	OTWait	Threads	YieldThr
1	0,08	0,01	0	0	0	0	0	0	0
2	0,13	0	0,01	0,01	0	0	0	0	0
4	0,44	0,44	0,42	0,46	0	0,25	0	0,41	0,62
8	0,24	0,24	0,18	0,2	0	0,24	0	0,31	0,27

Table 8.6: Healing techniques efficiency in the web crawler test case.

proc	CT+	Yield	Priority	YieldPrior	NewMut	OTYield	OTWait	Threads	YieldThr
4	0,3	0	0,2	0,15	0	0,05	0	0,15	0,35
8	0,35	0	0,15	0,1	0	0,05	0	0,4	0,3

Table 8.7: Efficiency of healing techniques in the TIDorb test case.

8.4.2 The Web Crawler Test Case

Modified web crawler test case represents a quite different challenge for the healing techniques. The bug occurs only once for each instance of the problematic variable during the execution. This makes the healing more easy. But on the other hand, the healer is not able to heal the first occurrence of the race what is actually the only situation when the healing is needed. In this particular case, the false alarm produced by the Eraser+ algorithm on the problematic variable luckily cause that even the first occurrence of the race can be healed because detect problem before it occurs, as was described in Section 8.3.2.

Table 8.6 shows the healing efficiency in such a case when even the first occurrence of the race is healed. Again, the column *ER+* represents the probability which should be decreased by the healing techniques. As can be seen in other columns, the all the healing techniques works on one and two processors machines. Machines with more processor shows weaknesses of methods based on yields or priority changing when a long atomic section is to be healed. Only *OTWait* method which stops the execution of the interleaving thread for a while helped. And of course, the method based on introducing a new lock-based synchronization also works perfectly.

8.4.3 The TIDorb Test Case

The results of healing the data race depicted in Figure 8.3 have been chosen to demonstrate the healing capabilities of the proposed architecture. The if statement added by developers to suppress data race warnings of the Eraser+ algorithm caused that both algorithms detect the race when the threads are already in conflict. Therefore, healing did not work. The architecture allows to specify variables which are healed from the beginning of the execution. When the problematic variable `m_exhaustive_equal` was added to the set of variables that should be healed during the whole execution, the healing works as can be seen in Table 8.7.

As can be seen from the table 8.7, the healing methods based on calling `Thread.yield()` are successful. This is probably caused by the nature of the bug which manifests only if a thread has cached the value of the variable and then used this cached value in the `if` statement. The reason why *YieldThr* method does not work for this case is currently not known. Healing methods based on an additional synchronization *NewMut* and suspending threads that are going to enter an atomic section if another thread is inside *OTWait* worked perfectly.

Chapter 9

Conclusions and Future Work

An architecture for dynamic self-healing of data races and atomicity violations was presented in this thesis. The architecture is able to detect a problem during the execution of a concurrent program and suppress its recurrence within the same execution and also in any future executions. A two algorithms for detecting data races and atomicity violations were introduced. A lockset-based algorithm and a novel algorithm—called AtomRace. The lockset-based algorithm is a modification of the Eraser [39] algorithm enriched with a support for join synchronization and lock suggestion mechanism. This algorithm can detect only data races. The AtomRace algorithm is based on detecting data races as a special case of atomicity violations on atomic sections specially defined to span just particular read/write instructions. The algorithm also supports more general atomic sections and without any modifications can be used to detect atomicity violations. Although other algorithms for detecting data races and/or atomicity violations are constructed with the intention to find as many (potential) bugs as possible, this focuses on announcing just the bugs which are for sure real. More precisely, the algorithm does not procude any false alarms in the case of data race detection nor in the case of atomicity violation detection—in the latter case relative to the definition of atomic sections the algorithm is given as its input. The absence of false alarms is very important especially for healing because otherwise the architecture starts to heal a problem which is not real. AtomRace also brings several furher advantages as support of any kind of synchronization and a lower overhead than other comparable algorithms.

Next, several techniques for on-the-fly suppressing of detected problems were introduced and compared. Most of them are based on legal influencing of the Java scheduler. The influencing is done by inserting a noise, changing of threads priorities, and occuping the processor. The experiments showed that techniques based on influencing the Java scheduler do not provide a sufficient efficiency in suppressing of detected problems but can help in cases where the technique based on additional synchronization cannot be used. The healing technique based on introducing an additional synchronization can heal the detected problem if atomic sections of the application are correctly inferred. Both, detection and healing techniques were examined on several examples ranging from a simple one to real complex software.

The presented architecture is ready to be integrated with other self-healing technologies being developed by other partners within the SHADOWS project. This way, a complex self-healing technology which is able to heal many different kinds of bugs in concurrency, functionality, and performance. Software self-healing sounds to most of people like science-fiction and most of them do not believe much in its success. However, this thesis hopefully

demonstrated by example that there could be a working solution. The presented experiments also show that the proposed solution has some weaknesses which should be addressed by the future work.

One of the problems of the proposed implementation is its high overhead. One of the way to decrease the overhead is to use a partial instrumentation of the managed code. The main idea of the partial instrumentation is to inject call of the healer to as few program locations as possible. Next, there is also a possibility to use specifically tuned synchronization and data structures for storing the data that the detecting and healing algorithms use.

The practical experiences demonstrated that the key for correct healing and detection is a precise inference of the blocks of code that are likely to be assumed by the programmer to be executed atomically. The techniques for a fully automated obtaining of these blocks of code that were implemented within the thesis did not work too well. In the future, techniques being developed, for example, by Cormac Flanagan et al.[1] could be exploited and used as a basis for future research on the subject.

Of course, the healing should not cause worse problems than those which are healed. Therefore, the architecture should be extended by some kind of formal analysis cappable proving that using additional synchronization on the inferred atomic sections can not cause a problem. This toppic is currently studied by Pavel Vyvial and Vendula Hrub'a from the SHADOWS team at FIT. Pavel Vyvial tries to exploit static analysis to detect a nested synchronization within inferred atomic sections. If a nested synchronization is identified, such section should not be healed by the technique using an additional synchronization. Vendula Hrubá is experimenting with a bounded model checking technique which should give an answer to the question whether it is safe to use additional synchronization to protect the atomic section.

The challenge for the future work on healing data races and atomicity violations is the healing even the first occurrence of the problem. This approach needs to identify situations which can possibly lead to a problem and influence the execution in such a way that the problem does not occur even once. After that, the architecture for self-healing can really help in situations common for real data races and atomicity violations which occurs very rarely.

[1] http://www.soe.ucsc.edu/ cormac/atom.html

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] Eric Allen. *Bug Patterns in Java*. APress L. P., 2002.

[3] 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.

[4] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Using findbugs on production software. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion*, pages 805–806, New York, NY, USA, 2007. ACM.

[5] Marina Biberstein, Eitan Farchi, and Shmuel Ur. Choosing among alternative pasts. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 289.1, Washington, DC, USA, 2003. IEEE Computer Society.

[6] 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.

[7] 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.

[8] 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.

[9] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Precise data-race detection and efficient model checking using locksets. Technical Report MSR-TR-2005-118, Microsoft Research, March 2006.

[10] 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.

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

[12] 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.

[13] 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.

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

[15] Peter Haggar. *Practical Java: Programming Language Guide*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[16] Tom Henzinger, Ranjit Jhala, Rupak Majumdar, and Shaz Qadeer. Thread-modular abstraction refinement. In *Proceedings of the 15th International Conference on Computer-Aided Verification (CAV)*, pages 262–274. Lecture Notes in Computer Science 2725, Springer-Verlag, January 2003.

[17] David H. Hovemeyer. *Simple and effective static analysis to find bugs*. PhD thesis, University of Maryland, College Park, MD, USA, 2005. Director-William W. Pugh.

[18] 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.

[19] 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.

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

[21] Zdeněk Letko. Dynamic detection and healing of low level data races. In *Proceedings of the 13th Conference STUDENT EEICT 2007*, Volume 2, pages 257–259. Brno University of Technology, 2007.

[22] Zdeněk Letko. An architecture for sefl-healing of data races and atomicity violations for java. In *Proceedings of the 14th Conference STUDENT EEICT 2008*, Volume 2, pages 256–258. Brno University of Technology, 2008.

[23] 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.

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

[31] Robert H. B. Netzer. Race condition detection for debugging shared-memory parallel programs. Technical Report CS-TR-1991-1039, Technion - Israel Institute of Technology, 1991.

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

[33] 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.

[34] 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.

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

[36] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. Locksmith: context-sensitive correlation analysis for race detection. *SIGPLAN Not.*, 41(6):320–331, 2006.

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

[38] Amit Sasturkar, Rahul Agarwal, Liqiang Wang, and Scott D. Stoller. Automated type-based analysis of data races and atomicity. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 83–94, New York, NY, USA, 2005. ACM.

[39] 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.

[40] Javier Soriano, Miguel Jimenez, Jose M. Cantera, and Juan J. Hierro. Delivering mobile enterprise services on morfeo's mc open source platform. In *MDM '06: Proceedings of the 7th International Conference on Mobile Data Management*, page 139, Washington, DC, USA, 2006. IEEE Computer Society.

[41] Rachel Tzoref, Shmuel Ur, and Yarden Nir. Deadlocks: from exhibiting to healing. In *8th International Workshop on Runtime Verification, (satellite event of ETAPS'08)*, to appear 2008.

[42] 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.

[43] 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.

[44] C. von Praun and T. Gross. Static detection of atomicity violations in object-oriented programs. In *In Proc. Formal Techniques for Java-like Programs*, volume 408 of *Technical Reports from ETH Zurich*. ETH Zurich, 2003.

[45] 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.

[46] Pavel Vyvial. Healing assurance in java programs. In *Proceedings of the 14th conference Student EEICT 2008*, volume 1, pages 204–206. Brno University of Technology, 2008.

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

[48] 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.

Appendix A

Class Diagrams

This appendix contains several class diagrams of the implemented prototype.

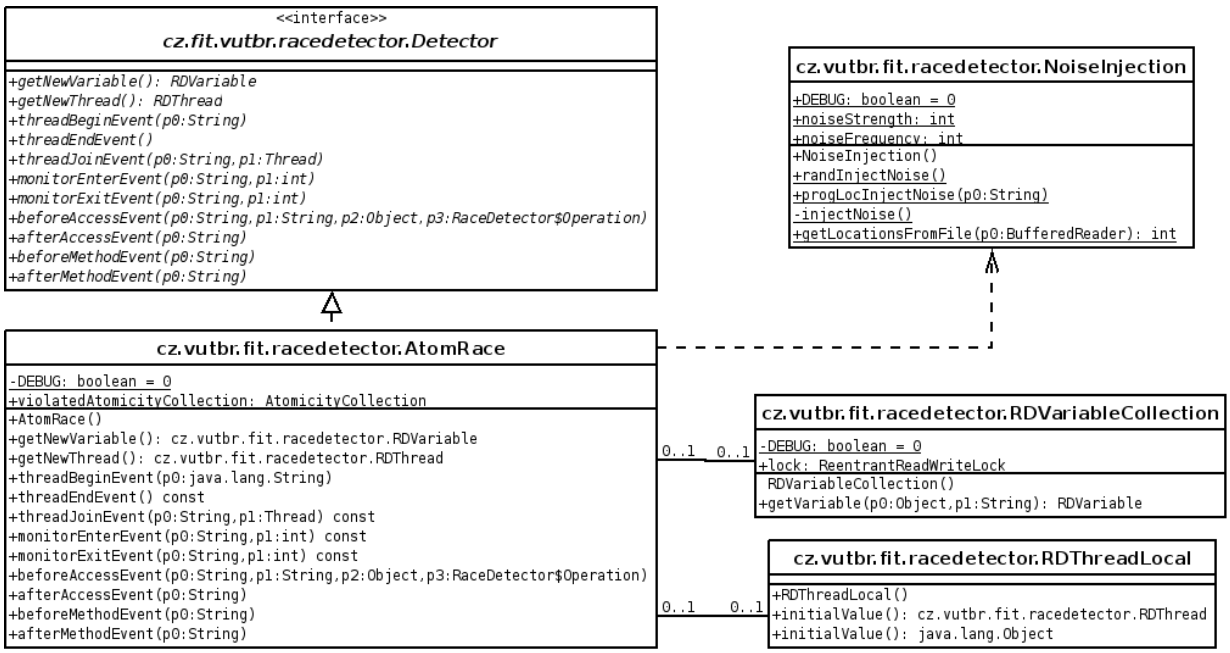


Figure A.1: The AtomRace class diagram.

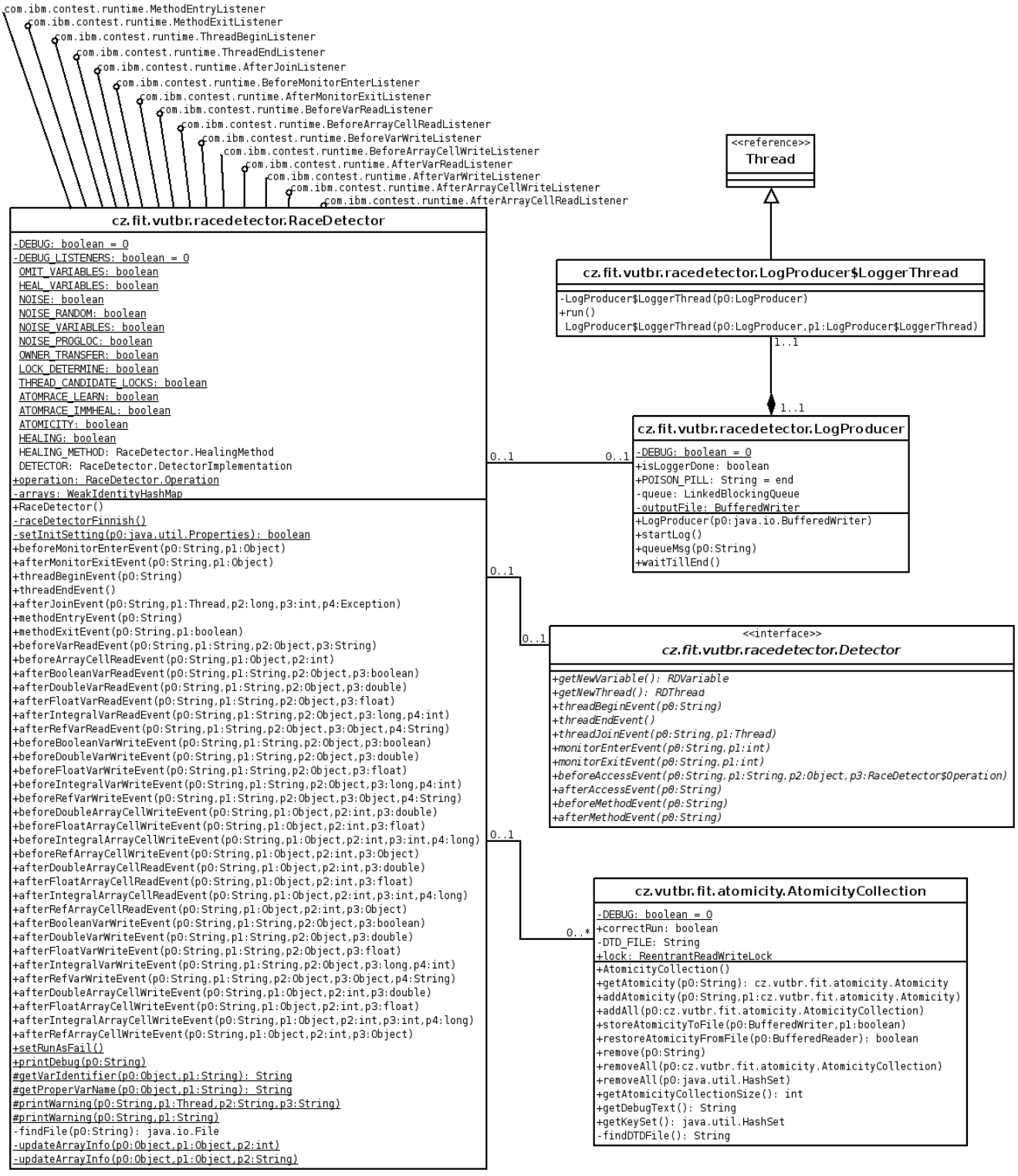


Figure A.2: The RaceDetector class diagram.

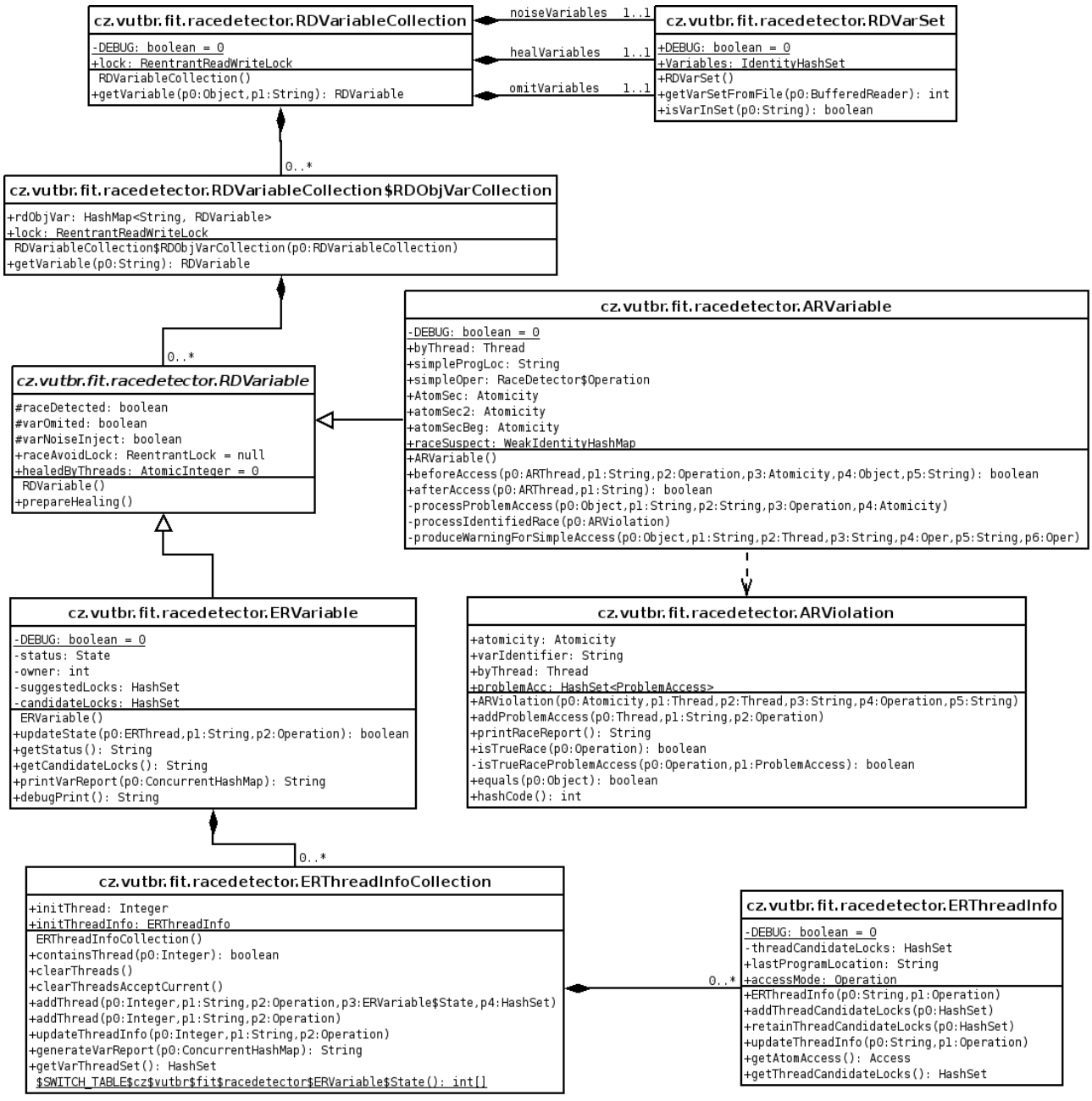


Figure A.3: The RDVariableCollection class diagram.

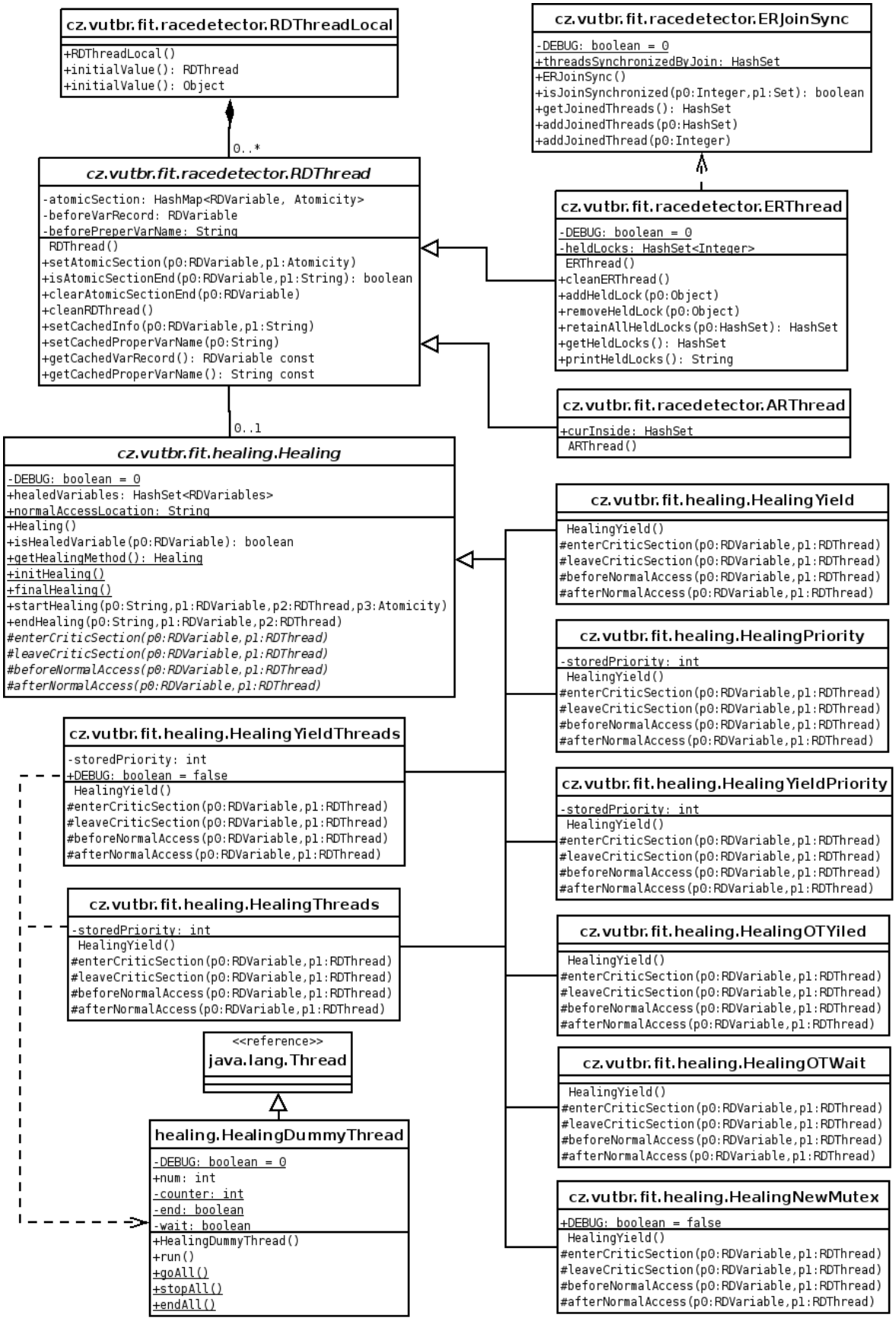


Figure A.4: The RDThreadLocal class diagram.

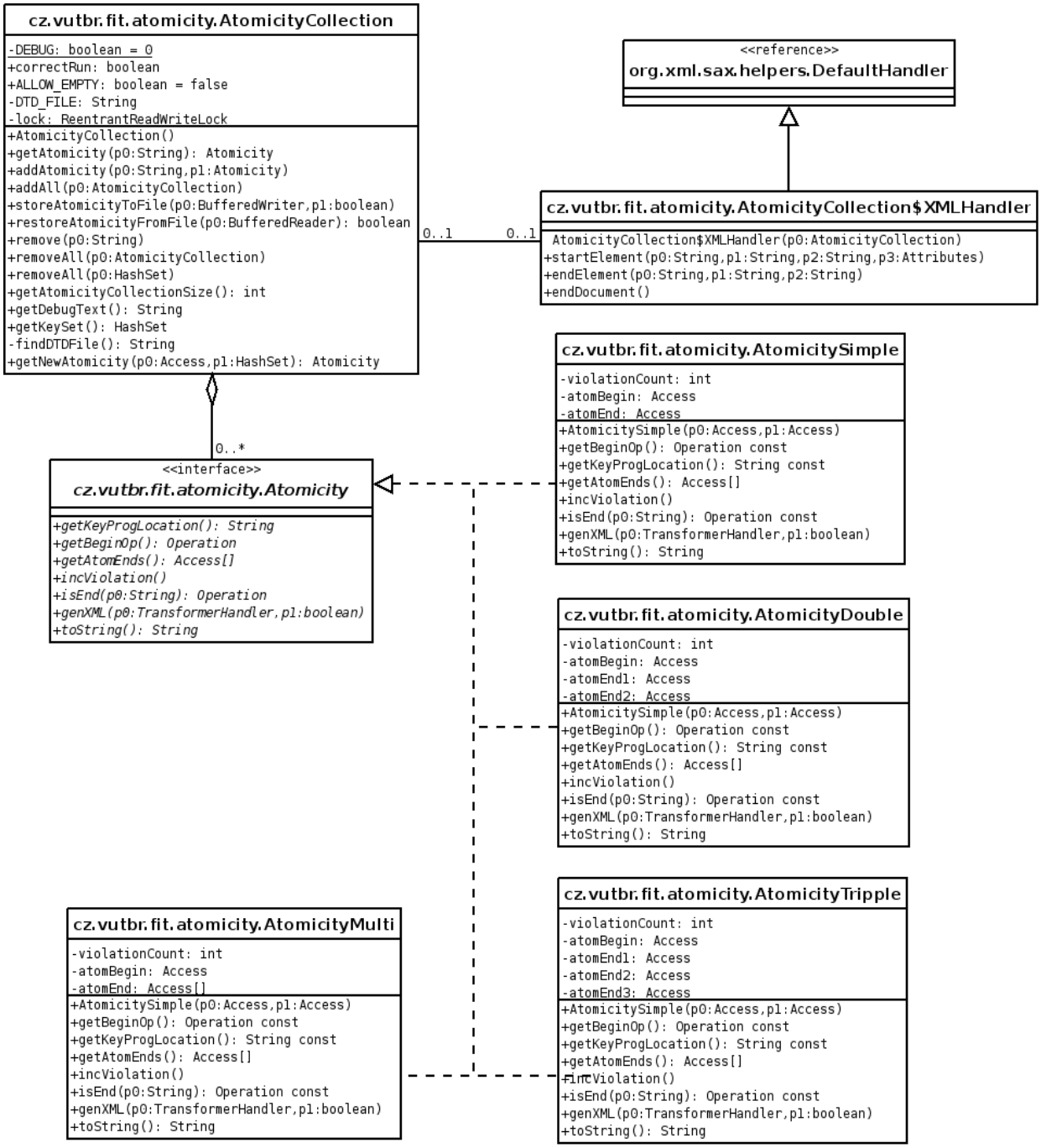


Figure A.5: The AtomicityCollection class diagram.

Appendix B

Settings of the Implemented Prototype

Below, there is a file with settings of the implemented prototype. Each option is shortly described and the default value for each option is given.

```
#RACEDETECTOR PROPERTIES
#========================
#Set the directory used by RaceDetector to store its output file.
#Given directory must be written with the last slash or backslash.
# Linux example: /tmp/
# windows example: C:/Temp/
# default: (nothing)
rdOutputDirectory =

#Enables the usage of a list of variables that should not be analyzed.
#The intention is to specify variables which produced false alarms in the
#previous executions. The alarms for this variables are not produced any
#more. It expects the presence of "omitvariables" text file in the input
#directory of RaceDetector.
# default: false
rdOmitVariables = false

#Enables the usage of a list of variables that should be healed from the
#beginning. All the instances of variables listed in a file will be
#healed as the race has been already detected on them. This can be used
#to run the buggy application till there will be a patch correcting
#the race. It expects the presence of "healvariables" text file
#in the input directory of RaceDetector.
# default: false
rdHealVariables = false

# === RACE CAUSING
#Enables the randomly situated noise injection with the intention
#to increase the probability of race manifestation.
# default: false
```

```
rdNoiseRandom = false

#Enables the usage of a list of variables that should be focused
#by noise injecting mechanism. In such case, a noise will be inserted
#in the middle of sections that should be atomic with the intention
#to increase the probability of race manifestation. It expects
#the presence of "noisevariables" text file in the input directory.
# default: false
rdNoiseVariables = false

#Enables the usage of a list of program locations that will be
#focused by noise injecting mechanism. In such case, a noise is
#inserted around the given program location with the intention
#to increase the probability of race manifestation. It expects
#the presence of "noiselocations" text file in the input directory.
# default: false
rdNoiseProgLoc = false

#The strength (duration) of a noise injected into the execution in case
#that rdNoiseVariables is enabled. Then for every specified variable a
#noise of this strength is produced. The strength should be
#between 0 and 1000.
# default: 70
rdNoiseStrength = 70

#This parameter influences the amount of noise which is used.
#The frequency should be between 0 and 1000.
# default: 0
rdNoiseFrequency = 0

# === RACE DETECTING ALGORITHM
#Algorithm used for race detecting. There are currently two algorithms:
#Eraser - lockset based algorithm which looks at locking policy and is
# able to detect wrong locking policy usually causing a data race.
#Atomrace - atomicity based algorithm which watch accesses
#to the variables. Possibilities here: "eraser" or "atomrace".
# default: eraser
rdDetector = eraser

# --- Eraser setting
  #It is common that initialization and use of objects are logically
  #separated. In such cases second owner should not be burnered with
  #access checks.
  # default: true
  rdOwnerTransfer = true

  #Turn on the lock suggestion mechanism.
  # default: true
```

```
  rdLockDetermine = false

# --- AtomRace setting
#"atomicity.xml" file has to be available in the input directory.
  #AtomRace has two modes:
  #1) Making atomicity set more accurate by removing atomicity's that
  #are not true (means that can be violated in a normal execution
  #of the program). This mode is called pruning or learning.
  #2) When the atomicity set is accurate enough, it can be used for
  #detecting atomicity violation and also for healing.
  #Algorithm works in the first phase if true is selected here.
  #Otherwise it works in detection phase.
  # default: false
  rdAtomRaceLearn = false

# === HEALING  (Directory with atomicity must be present)
#Enable the healing.
# default: false
rdHealing = false

#Specify the healing method. There are the following possibilities:
#
#  YIELD: This method is based on the idea that if the thread
#release the CPU (by calling yield), next time when is scheduled gets
# full CPU time slot.
#  PRIORITY: This method increase the priority of the thread and so
#decrease the probability of interruption.
#  YIELDPRIORITY: The combination of previous two.
#  NEWMUTEX: This method protects the problematic variable by a new
#explicit lock. This lock is used every time the variable is accessed
#If there is atomic section defined for the variable, the lock is
#not released in between. This approach can heal the race
#but can cause deadlock.
#  OTYIELD: This method force every thread trying to access
#the problematic variable to release the CPU, if another thread is
#inside the section which should be executed atomically.
#  OTWAIT: This method force every thread trying to access
#the problematic variable to wait for a while, if another thread is
#inside the section which should be executed atomically.
#  THREADS: This method runs a NUM_CPU-1  working dummy threads
#to utilize CPU and so simulates a one processor machine.
#  YIELDTHREADS: This method works as YIELD method + it runs
#a NUM_CPU-1 working dummy threads to utilize CPU.

# default: newmutex
rdHealingMethod = newmutex
```