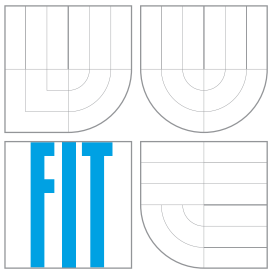
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

PRAKTICKÉ METODY AUTOMATIZOVANÉ VERIFIKACE PARALELNÍCH PROGRAMŮ

PRACTICAL METHODS OF AUTOMATED VERIFICATION OF CONCURRENT PROGRAMS

DISERTAČNÍ PRÁCE
PHD THESIS

AUTOR PRÁCE
AUTHOR

Ing. JAN FIEDOR

ŠKOLITEL
SUPERVISOR

Prof. Ing. TOMÁŠ VOJNAR, Ph.D.

BRNO 2016

Abstrakt

V dnešní době jsou vícevláknové programy běžné a s nimi i chyby v souběžnosti. Během posledních let bylo vytvořeno mnoho technik pro detekci takovýchto chyb, a i přesto mají vývojáři softwaru problém nalézt správné nástroje pro analýzu svých programů. Důvod je jednoduchý, fungující neznamená vždy praktický. Hodně nástrojů implementujících detekční techniky je obtížně použitelných, přizpůsobených pro konkrétní typy programů nebo synchronizace, nebo špatně škálují, aby zvládly analyzovat rozsáhlý software. Pro některé typy chyb v souběžnosti dokonce ani neexistují nástroje pro jejich detekci, i přesto že vývojáři softwaru na tyto chyby často narážejí ve svých programech. Hlavním cílem této práce je navrhnout nové techniky pro detekci chyb ve vícevláknových programech. Tyto techniky by měly být schopny analyzovat rozsáhlé programy, umožnit detekci méně studovaných typů chyb v souběžnosti, a podporovat širokou škálu programů s ohledem na to, jaké programové konstrukce používají.

Abstract

Nowadays, multi-threaded programs are quite common and so are concurrency errors. Over the years, many techniques were developed to detect such errors, yet software developers still struggle to find the right tools to analyse their programs. The reason is simple, working does not always mean practical. Many tools implementing the detection techniques are hard to use, tailored for a specific kind of programs or synchronisation, or do not scale well to handle large software. For some types of concurrency errors, no tools even exist, yet many software developers encounter such errors in their programs. The main goal of this thesis is to develop new techniques for detecting errors in multi-threaded programs. These techniques should be able to handle complex programs, allow one to detect some of the less studied types of concurrency errors, and support a broad variety of programs.

Klíčová slova

dynamická analýza, souběžnost, vkládání šumu, testování, transakční paměť, kontrakty

Keywords

dynamic analysis, concurrency, noise injection, testing, transactional memory, contracts

Citace

Jan Fiedor: Practical Methods of Automated Verification of Concurrent Programs, disertační práce, Brno, FIT VUT v Brně, 2016

Practical Methods of Automated Verification of Concurrent Programs

Prohlášení

Prohlašuji, že jsem tuto disertační práci vypracoval samostatně pod vedením Prof. Ing. Tomáš Vojnara, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jan Fiedor
August 31, 2016

Poděkování

V první řadě bych rád poděkoval svému školiteli Tomáši Vojnarovi, že mě přivedl do světa vědy, kde jsem mohl strávit tolik krásných let řešením zajímavých problémů. Dále bych rád poděkoval svým rodičům a bratrovi, kteří vždy stáli při mně a podporovali mne. Díky také patří všem kolegům z výzkumné skupiny, kteří každý den vytvářeli přijemné pracovní prostředí. Nakonec bych rád poděkoval také zahraničním kolegům, hlavně Joao Lourencovi, se kterými byla vždy radost spolupracovat a hodně mně toho naučili.

Contents

Chapter 1

Introduction

From the beginning of the era of computers, the most common way of increasing the performance of a computer (and programs running on it) was to increase the clock rate of its processor. A higher clock rate allowed the processor to execute more instructions in the same time interval, thus increasing the performance of any code ran by it by simply executing it faster. However, about ten years ago, the processors reached the 3 GHz clock rate, and it turned out that going above this threshold is difficult because of problems like heat or gate delays.

When increasing the clock rate of a processor stopped to be the best way to increase the performance of a computer, the manufacturers started to look for other ways to further increase the performance. The idea how to proceed came from servers. As even the most powerful processor was not efficient enough to accommodate the demands of servers, these computers were usually using several processors to have enough computational power to handle their workload. The drawback of using multi-processor computers was that they needed specialised motherboards supporting more than one processor, and such motherboards were quite expensive, large, and often required more sophisticated cooling. Generally speaking, they were not fit to be used in standard desktop computers.

Increasing the number of processors a desktop computer can utilise was the right way to go, however, the question was how to do it in a way that is compatible with the hardware commonly used in desktop computers. The problem was solved by IBM as early as in the year 2001 when they created the first dual-core processor POWER4. This was the first (general-purpose) processor that contained two independent processing units (called *cores*), each of them able to run program instructions separately. From the point of view of a computer, it was the same as having two separate processors, however, physically, it was a single processor compatible with the hardware used in desktop computers. Still, it took another four years for this kind of processors to be used more widely. In 2005, both Intel and AMD released their first dual-core processors, and so began the era of multi-core processors.

When multi-core processors became a common part of desktop computers, it became clear that more cores do not always mean better performance. The reason was one fundamental difference between multi-processor servers and multi-core desktop computers. Servers usually handle a huge amount of requests, however, each of these requests may be processed separately, i.e., by different programs or different instances of the same program. The important thing here is that there exists a large amount of (often small) programs and each can be run on any of the available processors. Even if each of these programs can utilise only a single processor, it does not matter as there are many other programs to be

ran on the remaining processors to fully utilise them. The more processors there is, the more programs is able to run at the same time, and the more work the server can handle during a given amount of time.

However, in case of desktop computers, the situation is different. The main difference is the disparity of workloads. Desktop computers usually handle the requests of a single user and the user usually does one thing at a time, i.e., uses one program at a time. If this program cannot utilise more than a single processor, then adding more processors (cores) does not increase its performance. This simple fact led to a need to write multi-threaded programs, i.e., programs which split its work into threads (lightweight processes) where each thread may be executed by a different processor (core).

Up until the era of multi-cores, multi-threaded programs were needed mainly for high performance computing. Such programs were developed by specialists having deep knowledge about multi-threaded computation and problems related to it. Nowadays, every moderately complex program is usually multi-threaded in order to provide the best performance possible. The problem is that writing multi-threaded programs is significantly harder than writing single-threaded programs. It is not sufficient to just split the computation into several parts and execute each part in a separate thread. Some parts may be dependant on other parts, requiring them to be executed first, some parts may access the same resources and cannot do so simultaneously, etc. In the end, the programmer must properly synchronise these threads. Failing to do so may lead to errors. On the other hand, oversynchronising the program, i.e., not allowing the other threads to run even when it is safe to do so, may prevent the errors, but may also degrade the performance of the program.

As multi-threaded programs are usually used to achieve maximum performance, programmers rarely end up oversynchronising them. This also means that they usually tend to synchronise only the parts which they think needs it. Unfortunately, many programmers are unable to correctly identify the parts that needs to be synchronised. This may lead to various concurrency errors caused by a missing synchronisation. Such errors usually manifest very rarely, and thus it is very hard to detect and localise them. Because of that, techniques for detecting such errors are needed to help the programmers to fix them. Indeed, reports [34, 88, 136] emphasize that it often takes more than a month to fix a concurrency-related error and that nearly 70 % of the fixes are buggy when first released. The topic of detecting errors in multi-threaded programs is both interesting and challenging, and also very relevant nowadays as the multi-core processors are not only present in desktop computers now, but also in devices like tablets, mobile phones, etc.

1.1 Detecting Errors in Multi-threaded Programs

Detecting errors in multi-threaded programs is much harder than in sequential programs because they may manifest only under very rare interleavings of actions executed by the different threads. Such interleavings are not very likely to be spot during classical testing, but they can occur in the production where the software is run for a much longer time, on different machines, under different load, and in different environment settings. This situation in turn stimulates research efforts devoted to all sorts of advanced methods for testing, analysis, and verification of multi-threaded programs.

Formal methods of verification, such as, e.g., model checking [14, 31], may potentially be able to precisely analyze a given program. Unfortunately, these precise approaches do not scale well for complex software systems. The size of the state space to be analyzed in such systems is simply too big to be handled by the precise approaches despite various

optimizations that are used in advanced formal verification techniques. Therefore, more lightweight approaches such as static and dynamic analyses or intelligent testing are often used. These approaches use approximations of the analyzed programs to cope with the complexity of the systems, which can pay off in the number of detected errors despite such approaches can both miss errors as well as produce false alarms [8].

Static analyses, such as [76], usually focus on searching for purely syntactic error patterns (possibly slightly refined, e.g., by using some information on the behavior of the verified programs pre-computed by suitable dataflow or type analyses). Such analyses scale well to even large code bases and may provide valuable information to the developer [86], but they often cannot discover concurrency-related errors because they do not model threads and their interactions [76]. Of course, there also exist static analyses which do consider concurrent threads, such as, e.g., [9, 123]. These analyses are able to detect concurrency-related errors, but they often produce many false alarms due to the abstractions they work with.

Testing [42, 75, 119, 163] relies on (possibly repeated) execution of a given program. It can precisely analyze all aspects of concurrent behavior, but it can only consider the witnessed execution paths and thread interactions. To increase the chances that testing will find concurrency-related errors, one can use (1) dynamic analysis [19, 45] techniques extrapolating the witnessed behaviour and/or (2) techniques allowing one to increase the number of different interleavings witnessed in repeated test runs (such as systematic testing [75, 119, 163] or noise injection [42]). As this thesis concentrates mainly on dynamic analysis and testing, we discuss these approaches in more detail below.

Stress testing. Many discussions on various forums suggest to use stress testing for discovering concurrency-related errors by simply executing a large number of threads competing for shared resources. This approach increases the possibility of spotting concurrency errors a little, and it can help to reveal some concurrency errors—usually those which manifest quite often. This may lead developers to a false conviction that the program is tested enough [131].

Noise injection. Noise injection inserts delays into the execution of selected threads with the aim of forcing new (legal) interleavings, which have so far not been witnessed and tested. This approach allows one to test more interleavings of synchronization-sensitive actions in shorter time because the system is not that much overloaded by other actions. Noise injection is also able to test legal interleavings of actions which are far away from each other in terms of execution time and in terms of the number of concurrency-relevant events [42] between those actions during average executions provided that strong enough noise is injected into some of the threads. In a sense, the approach is similar to running the program inside a model checker such as JPF [151] with a random exploration algorithm enabled. However, model checkers such as JPF are often limited in the programming constructs they natively support. Moreover, making purely random scheduling decisions may be less efficient than using some of the noise heuristics which influence the scheduling at some carefully selected places important from the point of view of synchronization only. The approach of noise injection is mature enough to be used for testing of real-life software, and it is supported by industrial-strength tools, such as IBM Java Concurrency Testing Tool (ConTest) [42] or the Microsoft Driver Verifier where the technique is called delay fuzzing [1]. Within IBM, ConTest allowed many bugs to be discovered, and as far as we can say, it is still in industrial use.

Systematic testing. Systematic testing, c.f., e.g., [75,119,120,158,163] has become quite popular recently. The technique uses a deterministic control over the scheduling of threads. A deterministic scheduler is sometimes implemented using intense noise injection keeping all threads blocked except the one chosen for making a progress. Often, other threads which do not execute synchronization-relevant instructions or which do not access shared memory are also allowed to make progress concurrently.

The systematic testing approach can be seen as execution-based model checking which systematically tests as many thread interleaving scenarios as possible. Before execution of each instruction which is considered as relevant from the point of view of detecting concurrency-related errors, the technique computes all possible scheduler decisions. The concrete set of instructions considered as concurrency-relevant depends on the particular implementation of the technique (often, shared memory accesses and synchronization relevant instructions are considered as concurrency relevant). Each such decision point is considered a state in the state space of the system under test, and each possible decision is considered an enabled transition at that state. The decisions that are explored from each state are recorded in the form of a partially ordered happens-before graph [119], totally ordered list of synchronization events [158], or simply in the form of a set of explored decisions [75, 163]. During the next execution of the program, the recorded scheduling decisions can be enforced again when doing a replay or changed when testing with the aim of enforcing a new interleaving scenario.

As the number of possible scheduling decisions is high for complex programs, several optimizations and heuristics reducing the number of decisions to explore have been proposed. The *locality hypothesis* [119] says that most concurrency-related errors can be exposed using a small number of preemptions. This hypothesis is exploited in the CHESS tool [119] which limits the number of context switches taking place in the execution (iteratively increasing the bound on the allowed number of context switches). Moreover, the tool also utilizes a partial-order reduction algorithm blocking exploration of states equal to the already explored states (based on an equivalence defined on happens-before graphs). The Maple tool [163] limits the number of context switches to two and additionally gets use of the *value-independence hypothesis* which states that exposing a concurrency error does not depend on data values. Moreover, the Maple tool does not consider interleavings where two related actions executed in different threads are too far away from each other. The distance of such actions is computed by counting actions in one of the threads, and the threshold is referred to as a *vulnerability window* [163].

However, despite a great impact of the above mentioned reductions, the number of thread interleavings to be explored remains big for real-life programs and therefore the approach provides great benefit mainly in the area of unit testing [75, 119, 163]. The systematic testing approach is not as expensive as full model checking, but it is still quite costly because one needs to track which scheduling scenarios of possibly very long runs have been witnessed and systematically force new ones. The approach makes it easy to replay an execution where an error was detected, but it has problems with handling various external sources of non-determinism (e.g., input events).

Systematic testing offers several important benefits over noise injection. Its full control over the scheduler allows systematic testing to precisely navigate the execution of the program under test, to explore different interleavings in each run, and to also replay interesting runs (if other sources of nondeterminism, such as input values, are handled). It allows the user to get information about what fraction of (discovered) scheduling decisions has already been covered by the testing process. However, the approach does also suffer from various

problems. The approach has problems to deal with external sources of non-determinism (user actions in GUI, client requests) as well as with continuously running programs where its ability to reuse already collected information is limited. In all those problematic cases, noise injection can be successfully used. Moreover, the performance degradation introduced by noise injection is significantly lower.

Dynamic analysis. Another way to improve traditional concurrency testing is to use dynamic analysis which collects various pieces of information along the executed path and tries to extrapolate the witnessed behavior in order to find errors which are in the program but did not necessarily occur during the execution. Many problem-specific dynamic analyses have been proposed for detecting special classes of errors, such as data races [45,97,133,137], atomicity violations [109], or deadlocks [2,19,83]. These techniques may find more bugs in fewer executions than classical testing. Some of the techniques, e.g., [45], are even sound (i.e., do not miss an error) and precise (i.e., do not suffer from false alarms) with respect to the observed execution path. However, most of the approaches are unsound and typically produce many false alarms.

Efficiency of dynamic analysis can be increased when a different execution path is analyzed during each execution of the test. A combination of noise injection or systematic testing and dynamic analysis can thus lead to a synergy effect. However, monitoring of the program behavior by a dynamic analysis algorithm typically introduces further synchronization among threads and represents a form of noise affecting thread scheduling, which may be important to take into account when applying regular noise injection heuristics.

Combined techniques. Finally, there are tools and techniques that combine various approaches to test multi-threaded programs. For instance, multiple techniques get use of information obtained by static and/or dynamic analysis in navigating systematic testing tools. An example of such a technique is the recently published *active testing* approach, targeting certain types of errors, such as data races [140], atomicity violations [130], and deadlocks [83]. The technique uses results of approximate static and/or dynamic analyses to hint systematic testing where a potential error can be found. The technique works in two stages. During the first *prediction phase*, a static and/or dynamic analysis is performed and warnings about specific concurrency errors are collected. In the second *validation phase*, the test is repeatedly executed with a deterministic scheduler. The scheduler behaves as a random scheduler until some thread reaches an action discovered during the prediction phase. If such an action is spotted, all threads that are about to execute this action are stopped. Whenever more threads are stopped, the scheduler enforces all possible interleavings.

1.2 Goals of the Thesis

The main goal of the thesis is to develop new techniques for detecting concurrency errors. This goal is naturally very broad as it is next to impossible to create a technique that would be able to detect any kind of error in any given program. While the existing techniques can detect various kinds of concurrency errors in different classes of programs, they certainly do not cover everything, leaving a lot of space for new techniques. Moreover, many of the existing techniques also have trouble handling larger programs or require complex configuration in order to provide reasonable results, making them hard to use in practice.

Hence, this thesis focuses in particular on four main aspects (sub-goals) that the invented techniques should accomplish:

1. Increase efficiency of the current approaches.
2. Be practical, i.e., easily usable in practice.
3. Support a broader variety of programs, i.e., more program constructions.
4. Support more properties to be checked, i.e., detect less commonly studied kinds of errors.

The first and most important sub-goal is to develop techniques that are able to analyse complex real world multi-threaded programs and find errors in them. While many of the existing techniques aim at this goal, they are still limited in either their efficiency or precision. For example, model checking can possibly detect any kind of error, as it searches the whole state space of a program, but, it will take forever to search the state space of a larger program. Utilising various state space reduction or abstraction techniques may allow the model checking to be used for larger programs, but, on one hand, introduce various sources of imprecision and, on the other hand, the cost of the resulting analysis will still stay quite high. In contrast, dynamic analysis can handle programs with millions of lines of code, but, it analyses a concrete execution of a program (a small part of its state space) only, and so it may miss many errors. To minimise the amount of errors missed, one usually performs some extrapolation of the execution, but, this may introduce false alarms. This thesis mainly focuses on (1) increasing the efficiency of current dynamic analysis techniques by combining them with other approaches like noise injection or bounded model checking to exploit their strengths and suppress their weaknesses, and (2) developing new dynamic analysis techniques utilising precise yet effective extrapolations.

The second sub-goal ensures that the techniques are usable in practice, i.e., can be used by companies and other software developers to analyse any program they have. There exist many techniques which look good on a paper, but their practical usability is very limited. These techniques often work only for small programs, do not support various language features, or require a large amount of work to make the analysis of a program possible (e.g., model the environment, write conditions to be checked, provide program invariants, etc.). All of this makes these techniques difficult to use in practice as the software developers cannot spend months to get their programs to a state in which they can be analysed. The created techniques should be as automated as possible, i.e., the users should be able to use them out of the box, without any input or work needed from their side. Ideally, the users should take their program, choose the analysis to perform, run the analysis and get some results. Naturally, analyses are more precise and/or effective if the users provide some additional information about the program. While it is good to allow the users to do such a thing, this information should never be required in order to use the technique.

The last two sub-goals are partially related to the second one, the practical usage of the techniques. When software developers want to analyse their programs, they need to find a technique which (1) is able to analyse the class of programs they have, e.g., the implementation of the technique supports the language the programs are written in, and (2) is able to detect the errors which these programs may contain (usually, they will first try the techniques which detect the most common types of errors and then try to use other techniques which deal with less common types of errors). While there exists a lot of different techniques for detection of the most common types of errors, their implementation

often supports only a concrete class of programs. Even when one can say that the other classes of programs are less common and so techniques for analysing them are not needed that much, we hear quite often from software developers that the only reason they are not analysing their programs is because there is no implementation of a given technique which can handle their class of programs. A similar problem is the less common kinds of errors. Again, while one may argue that these errors are rare and it is not worth the effort to invent techniques to detect them, we found out that many companies in fact encounter these kinds of errors in their software. So it is not that there is no demand for techniques for detecting such errors, it is just hard to develop techniques for detecting them. The main problem is that these errors are often tied to the semantics of the program an thus it is next to impossible to create a technique that would work for any program. This thesis addresses these problems by (1) building a framework that allows one to easily create custom dynamic analyses for analysing multi-threaded C/C++ programs, a common class of programs which is surprisingly often not supported by the implementations of various existing techniques, and (2) inventing new techniques for detecting some of the less-studied kinds of concurrency errors such as order violations.

1.3 Plan of the Thesis and Achieved Results

This section describes the organisation of the whole thesis and also links the following chapters to the sub-goals mentioned in the previous section. Each of the chapters is based on one or more published papers.

The thesis is organised chronologically based on how the research was conducted and can be roughly divided into three parts. The first part serves as a general overview of the types of concurrency errors and techniques used for their detection. The second part focuses on improving existing approaches for error detection and laying foundations for developing new techniques and/or analysing new classes of programs. The third part then builds on this foundation and focuses on analysing programs which use new means of synchronisation and developing techniques to detect some of the less studied types of concurrency errors. The rest of this section contains a more detailed summary of each of the thesis chapters.

Chapter 2 serves as an introduction to the world of concurrency errors. It describes various kinds of errors one may encounter in a multi-threaded program and presents existing techniques used to detect them. It also provides an overview of the state-of-the-art research in this area, showing which concurrency errors are well-studied and which barely have any techniques for their detection. The content of this chapter targets mainly sub-goals 3 and 4 as it shows which kinds of programs and errors are less studied and thus interesting for further research. This chapter is based on a paper presented at Eurocast 2011 and a technical report associated with it.

Chapter 3 presents our proposal of improving the error detection by combining imprecise yet fast dynamic analysis with slow but precise (bounded) model checking to increase the chances to detect errors. The main idea is to use the imprecise information provided by the dynamic analysis to guide the bounded model checker to the part of the program's state where a potential error was detected by the dynamic analysis. This greatly improves the performance of the bounded model checker as it does not need to search the whole state space, just the interesting parts of it. While the principles of the technique are general, the current implementation targets Java programs. The content of this chapter targets mainly the first two sub-goals as the resulting technique is fully automated and can handle large programs. This chapter is based on a paper presented at RV 2011 and a technical report

associated with it.

Chapter 4 describes our proposal of the ANaConDA framework. This framework allows one to easily implement new (noise-based) techniques for analysing C/C++ programs on the binary level. The motivation to create this framework came from the industry. While there exist many tools for Java implementing various analyses proposed in the literature, many companies cannot use them because they are developing C/C++ programs which these tools are unable to analyse. Although it is possible to reimplement these tools to support C/C++ programs, it is not an easy task without some utilities that would ease the process. While for Java, one may find various frameworks that would simplify this task, there are barely any such tools for C/C++. To address this problem, our research shifted from analysing Java programs to C/C++ programs. It was decided that the framework would perform the analysis on the binary level as it allows one to perform an analysis without the need to have the source code of the program (or a library) available. Such analysis is also more precise as it sees all the changes and optimisations done by the compiler to the original source code. However, working on the binary level also brings some new challenges to be solved, so this chapter also describe various caveats of monitoring the execution of a program on a binary level. The content of this chapter targets mainly sub-goal 3 as the framework allows one to analyse one of the less studied classes of programs. This chapter is based on parts of two papers presented at PADTAD 2012 and RV 2012 (awarded the best tool paper award).

Chapter 5 discusses several new noise injection techniques which increase the chances to detect errors. These techniques are based on injecting various delays into the execution of a multi-threaded program in order to influence the scheduling of its threads and alter its usual execution. In many cases, critical errors occur only in a very specific and quite rare executions, and the noise injection techniques try to increase the chances that such executions will occur even within a limited number of test runs to help the detection techniques to spot errors. Note that noise injection techniques are quite general and can be used in conjunction with many different error detection techniques. The content of this chapter targets mainly the first two sub-goals as the invented noise injection techniques improve the efficiency of many detection techniques, and while tailoring the noise configuration may help to achieve higher efficiency, even a random noise usually makes a big difference. This chapter is based on parts of papers presented at PADTAD 2012 and published in the STVR journal.

Chapter 6 deals with analysing programs which use transactional memory, a new kind of simple to use yet effective synchronisation. This new form of synchronisation is lately finding its way into the industry and monitoring and debugging this kind of programs proved to be very problematic. This chapter discusses various approaches to monitor these programs as well as how these approaches influence the behaviour of the monitored program which may be important, e.g., for performance analyses. The content of this chapter targets mainly sub-goal 3 as it allows one to monitor and analyse programs using a new type of synchronisation. This chapter is based on a paper presented at MEMICS 2014.

Chapter 7 concludes the research done within the thesis by introducing a new technique for detecting concurrency errors using contracts. A contract is a rather generic concept that allows one to handle not only some of the less studied concurrency errors such as order violations or missed signals, but also errors specific for a given context, e.g., errors tied to the semantic of a concrete program. This technique belongs among the extrapolating dynamic analyses that are capable of detecting errors even if they did not occur in the execution of a program. We instantiate the principle of extrapolating dynamic analysis in

a novel way for the given context, and, moreover, we show that it can be combined with noise injection too. This technique was already used to detect several errors in industry code which were then fixed using the information provided by it. The content of this chapter targets mainly sub-goals 1 and 4 as the invented technique can effectively detect some of the a less studied types of concurrency errors even in complex industry code. This chapter is based on two papers, one presented at Eurocast 2015 and one submitted to ICST 2017, and a technical report associated with the second paper.

Chapter 8 summarises the work described in this thesis and discusses possible directions for future work.

Chapter 2

Concurrency Errors

This chapter presents an overview of errors one may encounter in multi-threaded programs and also discusses various techniques for their detection. It pinpoints the kinds of errors and also classes of programs which are less studied and thus interesting for further research.

2.1 Introduction

Many works devoted to detection of concurrency errors have been published in recent years and many of them presented definitions of concurrency errors that the proposed algorithms are able to handle. These definitions are usually expressed in different terms suitable for a description of the particular considered algorithms, and they surprisingly often differ from each other in the meaning they assign to particular errors. To help understanding the errors and developing techniques for detecting them, this chapter strives to provide a uniform taxonomy of concurrency errors common in current programs, together with a brief overview of techniques so far proposed for detecting such errors. Note that while many of the techniques mentioned in this chapter are implemented to target Java programs, most of them can be applied to other programming languages as well.

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

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

Related work. Of course, there have been several attempts to provide a taxonomy of concurrency errors in the past decades, c.f., e.g., [23, 104, 108]. In [23], authors focus on concrete bug patterns bound to concrete synchronisation constructs in Java like, e.g., the `sleep()` command. In [108], a kind of taxonomy of bug patterns can also be found. The authors report results of analysis of concurrency errors in several real-life programs. A detailed description of all possible concurrency errors that can occur when the `synchronised` construct is used in Java is provided in [104] where a Petri net model of the synchronisation construct is analysed. In comparison to these works, our aim is to provide uniform definitions of common concurrency errors that are not based on some specific set of programs or some specific synchronisation means, and we always stress the generic skeleton of the definitions and the notions parameterising it. We do not rely on concrete bug patterns because they are always incomplete, characterising only some specific ways how a certain type of error can arise.

Plan of the chapter. Our discussion of common concurrency errors is divided into two main sections. (1) Section 2.2 covers various errors in *safety*, i.e., errors that cause something bad to happen. In particular, data races, atomicity violation, order violation, deadlocks, and missed signals are discussed. (2) Section 2.3 covers various errors in *liveness*, i.e., errors that prevent something good from happening, as well as errors mixing liveness and safety. Concretely, starvation, livelocks, non-progress behaviour, and blocked threads are discussed. In all cases, the particular error is first defined, trying to stress the main concept of the error and the notions by which it is parameterised, followed by a brief discussion of various known techniques for detecting the particular error. Finally, in Section 2.4, a short conclusion is given.

2.2 Safety Errors

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

2.2.1 Data Races

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

Definition 1. *A program execution contains a* data race *iff it contains two unsynchronised accesses to a shared variable and at least one of them is a write access.*

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

Detection of Data Races

Data races are a well studied concurrency problem and therefore there exist many different techniques for their detection. Dynamic techniques which analyse one particular execution of a program are usually based on computing the so-called locksets and/or happens-before

relations along the witnessed execution. Static techniques often either look for concrete code patterns that are likely to cause a data race or they compute locksets and/or happens-before relations over all executions considered feasible by the static analyser. There also exist static detection techniques that use type systems to detect data races. We discuss the basic principles of some of these techniques in the rest of this subsection.

The techniques based on *locksets* [137] build on the idea that all accesses to a shared variable should be guarded by a lock. The lockset is defined as a set of locks that guard all accesses to a given variable. Detectors then use an observation that if the lockset associated with a certain shared variable is non-empty, then there is at least one lock such that every access to the shared variable from any thread is protected by this lock, and hence there is no possibility of simultaneous accesses, and so a data race is not possible.

The happens-before-based techniques exploit the so-called *happens-before relation* [94] (denoted $\rightarrow$) which is defined as the least strict partial order that includes every pair of causally ordered events. For instance, if an event x occurs before an event y in the same thread, then $x \rightarrow y$. Also, when x is an event creating some thread and y is an event in that thread, then $x \rightarrow y$. Similarly, if some synchronisation or communication means is used that requires an event x to precede an event y, then $x \rightarrow y$. All notions of synchronisation and communication, such as sending and receiving a message, locking and unlocking a lock, sending and receiving a notification, etc., are to be considered. Detectors build (or approximate) the happens-before relation among accesses to shared variables and check that no two accesses (out of which at least one is for writing) can happen simultaneously, i.e., without the happens-before relation between them.

Type systems provide a syntactic method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute [132]. A formal type system provides a powerful and efficient checker of correctness of the code mainly if the programming language is strongly and statically typed (i.e., each variable has a deterministic type in each point of computation, and the types can be inferred without executing the code). Detection of concurrency bugs is usually done by extending the initial type system with a number of additional types that handle concurrency. These additional types are usually expressed by code annotations. A type system then searches for violations of rules defined over the newly defined types. Sometimes, the notion of *typestates*, introduced in [145], is used. Typestates extend the ordinary types that do not change through the lifetime of an object by allowing them to change during the course of the computation. A typestate property can be captured by a finite state machine where the nodes represent states of the type and the arcs correspond to operations that lead to state transitions.

Lockset-based algorithms. The first algorithm which used the idea of locksets was Eraser [137]. The algorithm maintains for each shared variable v, a set $C(v)$ of candidate locks for v. When a new variable is initialised, its candidate set $C(v)$ contains all possible locks. Eraser updates $C(v)$ by intersecting $C(v)$ and the set $L(t)$ of locks held by the current thread whenever v is accessed. Eraser warns about a data race if $C(v)$ becomes empty for some shared variable v along the execution being analysed. In order to reduce the number of false alarms, Eraser introduces an internal state $s(v)$ for each shared variable v used to identify whether v is used exclusively by one thread, v is read by multiple threads, or multiple threads change the value of v. The lockset $C(v)$ is then modified only when the variable is shared and a data race is reported only if $C(v)$ becomes empty and v is in the state denoting the situation when multiple threads access v for writing.

The original Eraser algorithm designed for C programs was then modified for programs

written in object-oriented languages, c.f., e.g., [21, 29, 153, 164]. The main modification (usually called as the *ownership model*) is inspired by the common idiom used in object-oriented programs where a creator of an object is actually not the owner of the object. Then one should take into account that the creator always accesses the object first and no explicit synchronisation with the owner is needed because the synchronisation is implicitly taken care by the Java virtual machine. This idea is reflected by inserting a new internal state of the shared variables. The modification introduces a small possibility of having false negatives [90, 153] but greatly reduces the number of false alarms caused by this object-oriented programming idiom.

A static data race detector that approximates locksets using an interprocedural data-flow analysis has been presented in [46]. The detection has three phases: (1) The control flow of each procedure is obtained and a call graph of the whole system is constructed. (2) A top-down data-flow analysis based on locksets is performed over the constructed graphs. A data race is suspected if an access to a shared variable is not guarded by a lock that is present in the lockset constructed for the variable along a path being analysed. (3) A final inspection algorithm then ranks each detected possible bug and reports only those that are real with a higher probability. However, the approach still produces many false alarms and it also has high memory requirements.

Better results were obtained, e.g., in [85, 122] where two more static analyses were incorporated into the machinery. An *alias analysis* [127] identifies a set of variables that refer to the same memory location, and an *escape analysis* [127] identifies a set of objects that are accessible in more than one thread. The first analysis enables the method to join locksets of different variables referencing the same data, and the escape analysis allows the method to limit the computation of locksets for variables that could be shared only.

A computation of locksets has also been implemented in the Java PathFinder (JPF) model checker [151]. JPF performs explicit state model checking, and therefore the *RaceDetector* checker computes a lockset for each shared variable and each state of the state space.

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

Happens-before-based algorithms. Most happens-before-based algorithms use the so-called *vector clocks* introduced in [115]. The idea of vector clocks for a message passing system is as follows. Each thread has a vector of clocks T_{vc} indexed by thread identifiers. One position in T_{vc} represents the own clock of t. The other entries in T_{vc} hold logical timestamps indicating the last event in a remote thread that is known to be in the happens-before relation with the current operation of t. Vector clocks are partially-ordered in a pointwise manner ($\sqsubseteq$) with an associated join operation ($\sqcup$) and the minimal element (0). The vector clocks of threads are managed as follows: (1) Initially, all clocks are set to 0. (2) Each time a thread t sends a message, it sends also its T_{vc} and then t increments its own logical clock in its T_{vc} by one. (3) Each time a thread receives a message, it increments its own logical clock by one and further updates its T_{vc} according to the received vector T'_{vc} to $T_{vc} = T_{vc} \sqcup T'_{vc}$.

Recently, a new variation of the vector-clock algorithm has been published [4]. The modification allows a distributed computation of vector clocks. Each thread maintains not a vector but a tree structure holding values of vector clocks. However, according to the best of our knowledge, there is no detector that uses this new algorithm yet.

Algorithms [133, 134] detect data races in systems with locks via maintaining a vector

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

Maintaining such a big number of vector clocks as above generates a considerable overhead. Therefore, in [60], the vector clocks of variables from above were mostly replaced by the so-called *epochs* associated with each variable v that are represented as tuples (t, c) where t identifies the thread that last accessed v and c represents the value of its clock. The idea behind this optimisation is that, in most cases, a data race occurs between two subsequent accesses to a variable. In such cases, epochs are sufficient to detect unsynchronised accesses. However, in cases where a write operation needs to be synchronised with multiple preceding read operations, epochs are not sufficient, and the algorithm has to build an analogy of vector clocks for sequences of read operations.

The happens-before relation can also be approximated statically. A very expensive and quite precise static analysis has been introduced in [114]. The algorithm proposed in this paper aims at computing the so-called *may-happen-in-parallel* relation (MHP) which is the complement of the happens-before relation. MHP is approximated in two steps. Initially, any pair of instructions is assumed to be able to happen in parallel (the MHP is total). Then, the initial set of pairs is pruned such that only those which cannot be seen to happen in succession remain in the set. A data race is reported if two accesses to a shared variable are in the MHP relation.

The original algorithm for computing MHP relations is very inefficient and is able to handle very small programs only. Therefore, several modifications have been proposed. The approach presented in [125] tries to compute MHP relations using a data-flow framework. The data-flow analysis is performed over the so-called *parallel execution graph*. This graph combines control-flow graphs of all threads that could be started during the execution with special edges induced by synchronisation actions in the code. The size of the graph increases exponentially with the size of the program and with the number of threads. Further, the data-flow computation of MHP relations has been slightly improved in [15] by the so-called *thread creation trees* (TCT) that help to compute a rough over-approximation of MHP relations before the data-flow evaluation is used. However, the obtained static analysis is still quite expensive.

A somewhat similar approach has also been used in the JPF model checker [151]. The *PreciseRaceDetector* detects data races by checking whether two accesses to the same variable may happen in parallel. In every state that JPF visits, the algorithm checks all actions that can be performed next. If this collection of actions contains at least two accesses to the same variable from different threads, then a data race is reported. Compared to the static analyses presented above, the detection in JPF is completely precise: It announces a data race only if it can really happen.

A bit different detection approach has been introduced in TRaDe [30] where a *topological race detection* [66] is used. This technique is based on an exact identification of objects which are reachable from a thread. This is accomplished by observing manipulations with references which alter the interconnection graph of the objects used in a program—hence the name topological. Then, vector clocks are used to identify possibly concurrently executed segments of code, called *parallel segments*. If an object is reachable from two parallel segments, a race has been detected. A disadvantage of this solution is a considerable

overhead.

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

Hybrid algorithms. Hybrid algorithms such as [45, 57, 129, 164] combine the two approaches described above.

In RaceTrack [164], a notion of a *threadset* was introduced. The threadset is maintained for each shared variable and contains information concerning threads currently working with the variable. The method works as follows. Each time a thread performs a memory access on a variable, it forms a label consisting of the thread identifier and its current private clock value. The label is then added to the threadset of the variable. The thread also uses its vector clock to identify and remove from the threadset labels that correspond to accesses that are ordered before the current access. Hence the threadset contains solely labels for accesses that are concurrent. At the same time, locksets are used to track locking of variables, which is not tracked by the used approximation of the happens-before relation. Intersections on locksets are applied if the approximated happens-before relation is not able to assure an ordered access to shared variables. If an ordered access to a shared variable is assured by the approximated happens-before relation, the lockset of the variable is reset to the lockset of the thread that currently accesses it.

One of the most advanced lockset-based algorithms that also uses the happens-before relation is Goldilocks presented in [45]. The main idea of this algorithm is that locksets can contain not only locks but also volatile variables (i.e., variables with atomic access that may also be used for synchronisation) and, most importantly, also threads. An appearance of a thread t in a lockset of a shared variable means that t is properly synchronised for using the given variable. The information about threads synchronised for using certain variables is then used to maintain the transitive closure of the happens-before relation via the locksets. An advantage of Goldilocks is that it allows locksets to grow during a computation when the happens-before relation is established between operations over v. The basic Goldilocks algorithm is relatively expensive but can be optimised by *short circuiting the lockset computation* (three cheaper checks that are sufficient for race freedom between the two last accesses on a variable are used) and using *a lazy computation of the locksets* (the locksets are computed only if the previous optimisation is not able to detect that some events are in the happens-before relation). The optimised algorithm has a considerably lower overhead approaching in some cases pure lockset-based algorithms.

A similar approach to Goldilocks but for the JPF model checker has been presented in [87]. This algorithm does not map variables to locksets containing threads and synchronisation elements (such as locks) but threads and synchronisation elements to sets of variables. This modification is motivated by the fact that the number of threads and locks is usually much lower than the number of shared variables. The modification can be done because model checking allows the method to modify structures associated with different threads at once. In a dynamic analysis, this cannot be done and locksets must be maintained in a distributed manner.

In [154], an abstract interpretation over abstract heaps is performed. The algorithm maintains the so-called *object use graphs* (OUG) capturing accesses from different threads to particular objects. Nodes of an OUG represent events performed with the object for which the OUG is built, and edges denote approximated happens-before relations between

the events. The data race detection algorithm performed after the OUGs are built then does not need to analyse the entire program but only relatively small OUGs. A data race is detected if there exist two events such that: (1) There is no ordering between the events, (2) the events originate from different threads, (3) at least one event is a write action, and (4) the events are not done under a common lock protection.

Type-based algorithms. In [56, 59], a clone of classic Java called *ConcurrentJava* has been proposed. The papers presented several annotations of using synchronisation primitives and of accessing shared variables allowing race-freeness be checked by an extended typing system of ConcurrentJava. For instance, each definition of a shared variable can be annotated with an annotation `guarded-by` l which requires each access to the particular variable to be guarded by a lock l.

A combination of type-based and data flow analysis has been presented in [160]. The proposed algorithm uses typestates to handle locking states for each variable and does not need any annotation provided by the user. Typestates are computed using an intraprocedural data-flow analysis and symbolic path simulation. The technique is able to handle large programs but still produces false alarms mainly due to unsupported synchronisation mechanisms (only locks are supported).

2.2.2 Atomicity Violation

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

Definition 2. *A program execution violates atomicity iff it is not equivalent to any other execution in which all code blocks which are assumed to be atomic are executed serially.*

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

Detection of Atomicity Violation

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

As for the blocks to be assumed to execute atomically, some authors expect the programmers to annotate their code to delimit such code blocks [63]. Some other works come with predefined patterns of code which should typically execute atomically [70, 109, 150]. Still other authors try to infer blocks to be assumed to execute atomically, e.g., by analysing data and control dependencies between program statements [159], where dependent program statements form a block which should be executed atomically, or by finding out access correlations between shared variables [107], where a set of accesses to correlated shared variables should be executed atomically (together with all statements between them).

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

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

In [159], blocks of instructions which are assumed to execute atomically are approximated by the so called *computational units* (CUs). CUs are inferred automatically from a single program trace by analysing data and control dependencies between instructions. First, a dependency graph is created which contains control and read-after-write dependencies between all instructions. Then the algorithm tries to partition this dependency graph to obtain a set of distinct subgraphs which are the CUs. The partitioning works in such a way that each CU is the largest group of instructions where all instructions are control or read-after-write dependent, but no instructions which access shared variables are read-after-write dependent, i.e., no read-after-write dependencies are allowed between shared variables in the same computational unit. Since these conditions are not sufficient to partition the dependency graph to distinct subgraphs, additional heuristics are used. Atomicity violations are then detected by checking if the strict 2-phase locking (2PL) discipline [47] is violated in a program trace. Violating the strict 2PL discipline means that some CU has written or accessed a shared variable which another CU is currently reading from or writing to, respectively (i.e., some CU accessed a shared variable and before its execution is finished, another CU accesses this shared variables). If the strict 2PL discipline is violated, the program trace is not identical to any serial execution, and so seen as violating atomicity. Checking if the strict 2PL discipline is violated is done dynamically during a program execution in case of the online version of the algorithm, or a program trace is first recorded and then analysed using the off-line version of the algorithm.

A much simpler approach of discovering atomicity violations was presented in [109]. Here, any two consecutive accesses from one thread to the same shared variable are considered an atomic section, i.e., a block which should be executed atomically. Such blocks can be categorised into four classes according to the types of the two accesses (read or write) to the shared variable. Serialisability is then defined based on an analysis of what can happen when a block b of each of the possible classes is interleaved with some read or write access from another thread to the same shared variable which is accessed in b. Out of the eight total cases arising in this way, four (namely, r/w/r, w/w/r, w/r/w, r/w/w) are considered to lead to an unserialisable execution. However, the detection algorithm does not consider all the unserialisable executions as errors. Detection of atomicity violations is done dynamically in two steps. First, the algorithm analyses a set of correct (training) runs in which it tries to detect atomic sections which are never unserialisably interleaved. These atomic sections are called *access interleaving invariants* (AI invariants). Then the algorithm checks if any of the obtained AI invariants is violated in a monitored run, i.e., if there is an AI invariant which is unserialisably interleaved by an access from another thread to a shared variable which the AI invariant (atomic section) accesses. While the second step of checking AI invariants violation is really simple and can be done in a quite efficient way, the training step to get the AI invariants can lead to a considerable slow down of the monitored application.

A more complicated approach was introduced in [57, 156], where atomicity violations

are sought using the Lipton's reduction theorem [102]. The approach is in particular based on checking whether a given run can be transformed (reduced) to a serial one using commutativity of certain instructions (or, in other words, by moving certain instructions left or right in the execution). Both [57] and [156] use procedures as atomic blocks by default, but users can annotate blocks of code which they assume to execute atomically to provide a more precise specification of atomic sections for the algorithm. For the reduction used to detect atomicity violations, all instructions are classified, according to their commutativity properties, into 4 groups: (1) *Right-mover* instructions R which can be swapped with immediately following instructions. (2) *Left-mover* instructions L which can be swapped with immediately preceding instructions. (3) *Both-mover* instructions B which can be swapped with preceding or following instructions. (4) *Non-mover* instructions N which are not known to be left or right mover instructions. Classification of instructions to these classes is based on their relation to synchronisation operations, e.g., lock acquire instructions are right-movers, lock release instructions are left-movers, and race free accesses to variables are both-movers (a lockset-based dynamic detection algorithm is used for checking race freeness). An execution is then serialisable if it is deadlock-free and each atomic section in this execution can be reduced to a form $R^*N^?L^*$ by moving the instructions in the execution in the allowed directions. Here, $N^?$ represents a single or no non-mover instruction and both-mover instructions B can be taken as right-mover instructions R or left-mover instructions L. Algorithms in both [57] and [156] use dynamic analysis to detect atomicity violation using the reduction algorithm described above.

Other approaches using the Lipton's reduction theorem [102] can be found in [62, 155] where type systems based on this theorem are used to deal with atomicity violations.

In [62], atomicity is analysed in programs written in a language called AtomicJava, a subset of Java with a type system for atomicity. The type system works with atomicity-related types denoting an expression as compound, atomic, mover (in the sense of Lipton), etc. These types may, moreover, be conditional upon the locks held. The user may annotate methods by the atomicity types and he/she can also annotate variables by locks the variables are supposed to be guarded with. The type inference rules proposed in the paper then automatically derive type constraints whose solution (if any) provides atomicity types for particular methods. If the methods were annotated by the user, conformance of the automatically derived and manually provided atomicity types is checked. The use of the conditional atomicity types makes the analysis more precise than the previous approaches.

In [155], a more simple type system for programs with non-blocking synchronisation is used which operates with five atomicity-related types: atomic, non-atomic, right-, left-, and both-mover expressions are distinguished. Again, procedures are considered as the main unit of atomicity. In this case, no annotations are provided, the technique just informs on which methods it considers atomic and which not, which is a bit restricting. Methods which are not executed atomically may, but need not violate atomicity assumptions of the programmer.

Atomicity over multiple variables. The above mentioned algorithms consider atomicity of multiple accesses to the same variable only. However, there are situations where we need to check atomicity over multiple variables, e.g., when a program modifies three different variables representing a point in a three-dimensional space. Even if we ensure that every consecutive read and write accesses to each of these variables are executed atomically, the program can still have an unserialisable execution. This is because the three atomic blocks guarding each pair of accesses to each of these variables can be interleaved with

other atomic blocks operating with these variables. Some of these variables can then end up modified by a different thread than the others which cannot happen in a serial execution. Nevertheless, the above discussed detectors would not detect any atomicity violation here.

In [7], the problem of violation of atomicity of operations over multiple variables is referred to as a *high-level data race*. In the work, all synchronised blocks (i.e., blocks of code guarded by the `synchronised` statement) are considered to form atomic sections. The proposed detection of atomicity violations is based on checking the so-called *view consistency*. For each thread a set of views is generated. A view is a set of fields (variables) which are accessed by a thread within a single synchronised block. From this set of views, a set of maximal views (maximal according to set inclusion) is computed for the thread. An execution is then serialisable if each thread is only using views which are compatible, i.e., form a chain according to set inclusion, with all maximal views of other threads. Hence, the detection algorithm uses a dynamic analysis to check whether all views are compatible within a given program trace. Since the algorithm has to operate with a big number of sets (each view is a set), it suffers from a big overhead.

A different approach is associated with the Velodrome detector [63]. Here, atomic sections (called transactions) are given as methods annotated by the user. Detection of atomicity violations is based on constructing a graph of the *transactional happens-before relation* (the happens-before relation among transactions). An execution is serialisable if the graph does not contain a cycle. The detection algorithm uses a dynamic analysis to create the graph from a program trace and then checks if it contains a cycle. If yes, the program contains an atomicity violation. Since creating the graph for an entire execution is inconvenient, nodes that cannot be involved in a cycle are garbage collected or not created at all. Like the previous algorithm, Velodrome too may suffer from a considerable overhead in some cases.

The simple idea of *AI invariants* described in [109] has been generalised for checking atomicity over pairs of variables in [70, 150], where 11 or 14, respectively, problematic interleaving scenarios were identified. The user is assumed to provide the so-called *atomic sets* that are sets of variables which should be operated atomically. In [150] there is proposed an algorithm which infers which procedure bodies should be the so-called *units of work* w.r.t. the given atomic sets. This is done statically using a dataflow analysis. An execution is then considered serialisable if it does not correspond to any of the problematic interleavings of the detected units of work. An algorithm capable of checking unserialisability of execution of units of work (called atomic-set-serialisability violations) is described in [70], based on a dynamic analysis of program traces. The algorithm introduces the so-called *race automata* which are simple finite state automata used to detect the problematic interleaving scenarios.

There are also attempts to enhance well-known approaches for data race detection to be able to detect atomicity violations over multiple variables. One method can be found in [107] where data mining techniques are used to determine *access correlations* among an arbitrary number of variables. This information is then used in modified lockset-based and happens-before-based detectors. Since data race detectors do not directly work with the notion of atomicity, blocks of code accessing correlated variables are used to play the role of atomic sections. Access correlations are inferred statically using a correlation analysis. The correlation analysis is based on mining association rules [3] from frequent itemsets, where items in these sets are accesses to variables. The obtained association rules are then pruned to allow only the rules satisfying the minimal support and minimal confidence constraints [3]. The resulting rules determine access correlations between various variables. Using this information, the two mentioned data race detector types can then be modified

to detect atomicity violations over multiple variables as follows. Lockset-based algorithms must check for every pair of accesses to a shared variable that the shared variable and all variables correlated with this variable are protected by at least one common lock. Happens-before-based algorithms must compare the logical timestamps not only with accesses to the same variables, but also with accesses to the correlated variables. The detection can be done statically or dynamically, depends on the data race detector which is used.

2.2.3 Order Violations

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

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

Detection of Order Violations

Like in the case of atomicity violations, a prerequisite for detecting order violations is to know which order restrictions are assumed. These can be specified manually, generic order requirements may be used (e.g., an object must be first initialised and only then used), or some restrictions may be automatically inferred. The order restrictions considered in current approaches are often quite simple, frequently considering only pairs of instructions. The actual discovery of order violations can in theory be done dynamically as well as statically. Currently, however, there are not many works dealing with order violation detection as has been pinpointed in [108].

In [162], authors introduce, for each memory operation o, a set of memory operations $PSet(o)$ which o depends upon and which can safely occur before o. These sets are extracted from a set of correct executions of the analysed program. Then, order violations are sought in further runs by looking for a memory operation o such that the previous memory operation dependent upon o is not in $PSet(o)$.

The ConMem tool [165] detects several behavioural patterns corresponding to order violations that can lead to a program crash. For each test input, ConMem monitors one execution of the given program. It uses a dynamic analysis to first identify parts of executions (denoted as ingredients) that may lead to a crash if ordered differently than in the given execution (e.g., assignments of `null` to a shared pointer and dereferences of this shared pointer from different treads). Then, ConMem analyses synchronisation around these potentially problematic constructions to see whether fatal interleavings exist to trigger an error (e.g., an interleaving where a thread t_1 assigns `null` to a shared variable v, and subsequently, a thread t_2 dereferences v). The paper describes four problematic patterns consisting of ingredients and timing conditions that lead to an error and that ConMem is able to detect. One example is the *Con-NULL* pattern with ingredients rp—a thread t_1 reads a pointer ptr, wp—a thread t_2 writes `null` to ptr, and timing conditions requiring wp to execute before rp with no write operation on ptr happening in between of wp and rp. Another example is the *Con-UnInit* pattern with ingredients r—a thread t_1 reads a variable v without previously

writing to v, w—a thread t_2 initialises v, and the timing condition requiring r to execute before w.

2.2.4 Deadlocks

Deadlocks are a class of safety errors which is quite often studied in the literature. However, despite that, the understanding of deadlocks still varies in different works. We stick here to the meaning common, e.g., in the classical literature on operating systems. To define deadlocks in a general way, we assume that given any state of a program, (1) one can identify threads that are blocked and waiting for some event to happen and (2) for any waiting thread t, one can identify threads that could generate an event that would unblock t.

Definition 4. *A program state contains a set S of deadlocked threads iff each thread in S is blocked and waiting for some event that could unblock it, but such an event could only be generated by a thread from S.*

Most works consider a special case of deadlocks, namely, the so-called *Coffman deadlock* [32]. A Coffman deadlock happens in a state in which four conditions are met: (1) Processes have an exclusive access to the resources granted to them, (2) processes hold some resources and are waiting for additional resources, (3) resources cannot be forcibly removed from the tasks holding them until the resources are used to completion (no preemption on the resources), and (4) a circular chain of tasks exists in which each task holds one or more resources that are being requested by the next task in the chain. Such a definition perfectly fits deadlocks caused by blocking lock operations but does not cover deadlocks caused by message passing (e.g., a thread t_1 can wait for a message that could only be sent by a thread t_2, but t_2 is waiting for a message that could only be sent by t_1).

Detection of Deadlocks

Detection of deadlocks usually involves various graph algorithms as it is, for instance, in the case of the algorithm introduced in [128] where a *thread-wait-for graph* is dynamically constructed and analysed for a presence of cycles. Here, a thread-wait-for graph is an arc-labelled digraph $G = (V, E)$ where vertices V are threads and locks, and edges E represent waiting arcs which are classified (labelled) according to the synchronisation mechanism used (join synchronisation, notification, finalisation, and waiting on a monitor). A cycle in this graph involving at least two threads represents a deadlock. A disadvantage of this algorithm is that it is able to detect only deadlocks that actually happen. The following works can detect also potential deadlocks that could happen but did not actually happen during the witnessed execution.

In [72], a different algorithm called GoodLock for detecting deadlocks was presented. The algorithm constructs the so-called *runtime lock trees* and uses a depth-first search to detect cycles in it. Here, a runtime lock tree $T_t = (V, E)$ for a thread t is a tree where vertices V are locks acquired by t and there is an edge from $v_1 \in V$ to $v_2 \in V$ when v_1 represents the most recently acquired lock that t holds when acquiring v_2. A path in such a tree represents a nested use of locks. When a program terminates, the algorithm analyses lock trees for each pair of threads. The algorithm issues a warning about a possible deadlock if the order of obtaining the same locks (i.e., their nesting) in two analysed trees differs and no "gate" lock guarding this inconsistency has been detected.

The original GoodLock algorithm is able to detect deadlocks between two threads only. Later works, e.g., [2,19] improve the algorithm to detect deadlocks among multiple threads. In [2], a support for semaphores and wait-notify synchronisation was added. The recent work [84] modified the original algorithm so that runtime lock trees are not constructed. Instead, the algorithm uses a stack to handle the so-called *lock dependency relation*. The algorithm computes the transitive closure of the lock dependency relation instead of performing a depth first search in a graph. The modified algorithm uses more memory but the computation is much faster.

A purely data-flow-based interprocedural static detector of deadlocks called RacerX has been presented in [46]. The detection it implements has two phases: (1) The control flow graph of each procedure is obtained and the complete control flow graph of the whole system is constructed. (2) A data-flow analysis is performed over the constructed graph and the order in which locks are nested is analysed. A deadlock is reported when locks are not obtained every time in the same order. In [157], a bottom-up data-flow static analysis is used to detect deadlocks. The algorithm traverses the call graph bottom-up and builds a lock-order graph per method. Each node of the lock-order graph represents a set of objects that may be aliased and an edge in the graph indicates nested locking of objects along some code path. If the obtained graph contains cycles, a possible deadlock is reported. Both algorithms produce many false alarms due to the approximations they use.

The algorithm presented in [124] reduces the number of false alarms obtained by a data-flow interprocedural analysis described in the previous paragraph using six conditions. The first four represent results of reachability, alias, escape, and approximated may-happen-in-parallel analyses. The next two conditions handle special cases of using reentrant locks and a guarding lock. The algorithm filters all potential deadlocks through these conditions and reports only those which fulfil all the conditions.

A combination of symbolic execution, static analysis, and SMT solving is used in [36] to automatically derive the so-called *method contracts* guaranteeing deadlock free executions. The algorithm does not focus on detection of deadlock if the whole program is available. Instead, the algorithm analyse pieces of code (usually libraries) and automatically infers conditions that must be fulfilled when these libraries are used in order to avoid deadlocks.

2.2.5 Missed Signals

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

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

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

Detection of Missed Signals

There are not many works focusing specially on missed signals. Usually, the problem is studied as a part of detecting other concurrency problems. In [2], a lost notification error is reported if there is a notify event e in a trace tr and there exists a trace that is a *feasible permutation* of tr in which e wakes up fewer threads than it does in tr. Such a situation is

possible when the wait event of one of the threads woken in *tr* is not constrained to happen before the event *e*.

In [76], several code patterns that might lead to a lost notification are listed. For instance, each call of `wait()` must be enclosed by a loop checking some external condition. A pattern-based static analysis is then used to detect such bug patterns.

2.3 Liveness and Mixed Errors

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

Before we start discussing more concrete notions of liveness and mixed errors, let us first introduce the very general notion of *starvation* [147].

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

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

2.3.1 Livelocks and Non-progress Behaviour

There are again various different definitions of a livelock in the literature. Often, the works consider some kind of a *progress* notion for expressing that a thread is making some useful work, i.e., doing something what the programmer intended to be done. Then they see a livelock as a problem when a thread is not blocked but is not making any progress. However, by analogy with deadlocks, we feel it more appropriate to restrict the notion of livelocks to the case when threads are looping in a useless way while trying to synchronise (which is a notion common, e.g., in various works on operating systems). That is why, we first define a general notion of non-progress behaviour and then we specialise it to livelocks.

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

A non-progress behaviour is a special case of starvation within an infinite behaviour. On the other hand, starvation may exhibit even in finite behaviours and also in infinite progress behaviours in which a thread is for a while waiting for an event that is not guaranteed to happen. As we have said already above, livelocks may be seen as a special case of non-progress behaviour [147].

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

As was mentioned before, there are many, often inconsistent, definitions of a livelock. Moreover, many works do not distinguish between livelocks and a non-progress behaviour [20,73,100,143,146]. Other papers [113,118] take a livelock to be a situation where a task has such a low priority that it does not run (it is not allowed to make any progress) because there are many other, higher priority, tasks which run instead. We do not consider such a situation a livelock and not even a non-progress behaviour but a form of starvation. There are even works [5] for which a thread is in a livelock whenever it is executing an infinite loop, regardless of what the program does within the loop. However, there are many reactive programs which run intentionally in an infinite loop, e.g., controllers, operating systems and their components, etc., and it is not appropriate to considered them to be in a livelock.

Detection of Livelocks and Non-progress Behaviour

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

The first issue the non-progress detection methods have to deal with is getting to know what is to be considered a non-progress behaviour. In the literature there are used various different notions of progress. Some works, e.g., [143], define progress by looking at the *communication* among two or more cooperating threads. If a thread communicates, it is progressing. In [65], progress is associated with operations on the so-called *communication objects* (such as shared variables, semaphores, FIFO buffers, etc.). In [100,146], progress is defined by reaching a so-called *progress action* or *progress statement*, respectively (e.g., delivering output, responding to the environment, etc.). In [73], progress is expressed by a so-called *liveness signature*, a set of state predicates and temporal rules, specifying which application states determine whether a program is making a progress when they are repeatedly reached.

To enable a non-progress detection, the general notions of progress from above have to be concretised for a particular program. This is mostly expected to be done by the user, e.g., by specifying progress actions [100], labelling statements as progress statements [74], annotating the code [73], calling actions of the so-called observer [20], etc. However, some works do not require any user input and use a fixed notion of progress [65].

One of the most common approaches for detecting non-progress behaviour in finite-state programs is to use model checking [65,74] and search for non-progress cycles [49]. In [65], no non-progress cycles are being sought, instead bounded model checking is used to find a path where no progress is being made for a user-specified period of time. Such an approach can lead to false alarms, but can be used for large and infinite-state programs. Recently, there has also appeared various (infinite-state) program analyses, based, e.g., on transition predicate abstraction and ranking function synthesis, for proving (non-)termination. These can also be used for verifying liveness properties [33], but these approaches, albeit useful, for instance, for verifying drivers, are still not applicable to large software systems.

Another often used approach is to use a dynamic analysis in the form of *dynamic monitoring* [6, 73]. In [73], a liveness signature, a set of state predicates and temporal rules, is used to determine whether a program is making progress. In other words, the liveness signature determines which program states are significant in determining whether a program is making a progress. If a program does not reach any of these program states for some time, the program is at a so-called *standstill*, i.e., is not making progress. In [6], the so-called *Q-Learning* is used to navigate a program execution to follow paths which most likely lead to an execution trace in which the program is not making progress. In case of finite-state programs, the tool searches for an execution trace containing a non-progress loop, in case of infinite-state programs, it tries to find an execution trace of a user-specified length. A problem with dynamic monitoring techniques is that they cannot distinguish between a non-progress behaviour and starvation because they are checking bounded liveness properties [139], which are in fact safety properties, not liveness properties.

In [20], a static analysis is used to detect a non-progress behaviour in programs written in Ada. Progress is defined here as a *communication with an external observer*. Each program is represented by a control flow graph. To detect a non-progress behaviour, the method searches the control flow graph for an infinite loop in which no communication with the external observer is performed.

There are also works which attempt to use approaches often used for detection of other concurrency errors. For example, in [69], the authors define *antipatterns* which may lead to a non-progress behaviour and use static and dynamic analyses to locate these antipatterns in programs.

In [100], the authors define progress as an execution of a progress action and a progress cycle as a cycle which contains at least one progress action. Then they translate a necessary condition of an existence of a non-progress behaviour, i.e., that there exists an infinite run in which progress cycles are repeated only a finite number of times, into a homogeneous integer programming problem. Subsequently, they try to find a solution to this problem. If the problem has no solution, then the program surely does not contain a non-progress behaviour. A downside of this method (apart from its cost) is that it is incomplete, so if the problem has a solution there may or may not be a non-progress behaviour.

2.3.2 Blocked Threads

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

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

An absence of some unblocking event which leaves some thread blocked may have various reasons. A common reason is that a thread, which should have unblocked some other thread, ended unexpectedly, leaving the other thread in a blocked state. In such a case, one often speaks about the so-called *orphaned threads* [50]. Another reason may be that a thread is waiting for a livelocked or deadlocked thread.

Detection of Blocked Threads

We are not aware of any works specialising in this kind of errors. Of course, the most simple solution to deal with this error is to check that a thread is waiting for more than some time to be unblocked. This is, however, a very crude approach. In fact, a similar approach is used in MySQL to detect deadlocks and it was shown [108] that it is quite inaccurate detection method which often leads to false alarms and, in case of MySQL, to unnecessary restarts.

Like with all previous errors, another possibility is to use model checking [14], limited by its high price and problems with some program operations like input and output. In theory, one could also use, e.g., static analysis for detection of some undesirable code patterns that could cause permanent blocking, similar to deadlock antipatterns [69] in case of a deadlock. However, detection of this kind of errors remains mostly an open issue.

2.4 Conclusion

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

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

Chapter 3

Combining Dynamic Analysis and Model Checking

This chapter presents the DA-BMC tool chain that we proposed to allow one to combine dynamic analysis and bounded model checking for finding synchronisation errors in concurrent Java programs. The idea is to use suitable dynamic analyses to identify executions that are suspected to contain synchronisation errors, reproduce these executions in a model checker, and perform bounded model checking in a vicinity of the replayed execution to confirm whether there are some real errors.

3.1 Introduction

The previous chapter described various approaches for detecting concurrency errors. Many of these approaches were based on dynamic analysis. The advantage of dynamic analysis is that it scales well and thus can handle very large programs. The disadvantage is that it analyses only a concrete execution of a program and can detect only errors encountered in it. To improve on this restriction, dynamic analyses usually extrapolate the behaviour of a program to detect also errors that may happen, yet did not occur in the execution. The price for such an ability to detect errors not seen in the execution is the precision. Extrapolation often over-approximates the behaviour of a program, assuming existence of executions that are not feasible in reality. Detecting errors in such infeasible executions then leads to false positives. On the other hand, techniques like model checking are precise and can detect all errors in a program without producing false alarms. However, to do so, model checking must search the whole state space of a program (or a significant portion of it), which may be impossible for larger programs. In this chapter, we describe a tool chain denoted as *DA-BMC*[1] that tries to combine advantages of both dynamic analysis and (bounded) model checking.

In our tool chain, implementing the approach proposed in [78], we use the infrastructure offered by the *Contest* tool [43] to implement suitable dynamic analyses over Java programs and to record selected points of the executions of the programs that are suspected to contain errors. We then use the *Java PathFinder (JPF)* model checker [151] to replay the partially recorded executions, using JPF's capabilities of state space generation to heuristically navigate among the recorded points. In order to allow the navigation, the JPF's state space search strategy, including its use of partial order reduction to reduce the

[1] `http://www.fit.vutbr.cz/research/groups/verifit/tools/da-bmc`

searched state space, is suitably modified. Bounded model checking is then performed in the vicinity of the replayed executions, trying to confirm that there is really some error in the program and/or to debug the recorded suspicious behaviour.

We illustrate capabilities of DA-BMC on several case studies, showing that it really allows one to benefit from advantages of both dynamic analysis and model checking.

3.2 Recording Suspicious Executions

The first step when using DA-BMC is to use a suitable *dynamic analysis* to identify executions suspected to contain an error and to record some information about them—recording the entire executions would typically be too costly. In DA-BMC, this phase is implemented on top of the *Contest tool* [43]. Contest provides a listener architecture (implemented via Java byte-code instrumentation) on top of which it is easy to implement various dynamic analyses. We further refer to two such analyses, namely, Eraser+ and AtomRace intended for detection of data races (and, in the second case, also atomicity violations), which have been implemented as Contest plugins in [89]. Further analyses can, of course, be added. Contest also provides a noise injection mechanism which increases the probability of manifestation of concurrency-related errors.

In order to record executions, we have implemented another specialised listener on top of Contest. We record information about an execution in the form of a *trace* which is a sequence of *monitored events* that contains partial information about some of the events that happen during the execution. In particular, Contest allows us to monitor the following events: *thread-related* events (thread creation, thread termination), *memory-access-related* events (before integer read, after integer read, before float write, after float write, etc.), *synchronisation-related* events (after monitor enter, before monitor exit, join, wait, notify, etc.), and some *control-related* events (basic block entry, method enter, and method exit). The user can choose only some of such events to be monitored. As shown in our case studies, one should mainly consider synchronisation-related and memory-access-related events, which help the most when dealing with the inherent non-determinism of concurrent executions.

Each monitored event contains information about the source-code location from which it was generated (class and method name, line and instruction number) and the thread which generated it. The recorded trace also contains information produced by the applied dynamic analysis which labels some of the monitored events as suspicious from the point of view of causing an error.

3.3 Replaying Recorded Traces

The second step when using DA-BMC is to reproduce suspicious executions recorded as traces of monitored events in a model checker. More precisely, there is no guarantee that the same execution as the one from which the given trace was recorded will be reproduced. The tool will simply try to generate some execution whose underlying trace corresponds with the recorded trace. It is also possible to let the model checker generate more executions with the same trace.

In DA-BMC, we, in particular, use the *Java PathFinder (JPF)* model checker [151]. JPF provides several state space search strategies, but also allows one to add new user-specific search strategies. Moreover, it provides a listener mechanism which is useful for

performing various analyses of the searched state space and/or for guiding the search strategies to a specific part of the state space. JPF uses several state space reduction techniques, including partial order reduction (POR), which out of several transitions that lead from a certain state may explore only some [13].

A recorded trace is replayed by navigating JPF through the state space of a program such that the monitored events encountered on the search path correspond with the ones in the recorded trace. The states being explored during the search are stored in a priority queue. The priority of the inserted states depends on the chosen search strategy (DFS and BFS are supported). In each step, the next parent state to be processed is obtained from the queue. After that, all relevant children of the parent state are generated. Here, we should note that, in JPF, a transition between a parent and child states represents, in fact, a sequence of events happening in a running program. This sequence is chosen by the POR to represent all equivalent paths between the two states. Into the priority queue, we only save the child states that may appear on a path corresponding to the recorded trace. In other words, each program event encountered within the JPF's transition between the parent and child states must either be an event which is not monitored (and hence ignored), or an event which corresponds with the one stored in the recorded trace at an appropriate position. This correspondence is checked during the generation of a transition in JPF.

Sometimes, it is also necessary to influence the POR used by JPF. That happens when the POR decides to consider another permutation of the events than the one actually present in the trace. Then, the POR is forced to use the needed permutation as follows. If the generation of the sequence of events that the POR wants to compose into a single transition encounters some monitored event, and this event differs from the one expected in the recorded trace, then we force JPF to finish the generation of the sequence of events to be put under a single transition and to create a new state. The navigation algorithm then searches the transitions enabled in this state that correspond with the recorded trace (if there is none, the search backtracks).

Since the replaying is driven by a sequence of monitored events generated from the Contest's instrumentation of the given program, we run the instrumented byte-code in JPF. We, however, make JPF skip all the code that is a part of Contest in order not to increase the size of the state space being searched. Moreover, Contest not only adds some instructions into the code, but also replaces some original byte-code instructions. This applies, e.g., for the instructions `wait`, `notify`, `join`, etc. In this case, when such an instruction is detected in JPF, we dynamically replace it with the original instruction.

As the JPF's implementation of `sleep()` ignores interruption of sleeping threads, we provide a modified implementation of the `interrupt()` and `sleep()` methods which correctly generate an exception if a thread is interrupted by another thread when sleeping. For that to work correctly, the possibility of branching of the execution after `sleep()` must be enabled in JPF.

Still, it might not be possible to replay a trace if the program depends on input or random data or if it uses some specific dynamic data structures like hash tables where, e.g., objects might be iterated in a different order in each run of the program. In these cases, it is necessary to modify the source code of the analysed program, e.g., by adding JPF data choice generators to eliminate these problems.

Table 3.1: Overhead generated by trace recording

	Execution time with Eraser in seconds	Slowdown due to recording (number of events in the trace)	
Bank	1.62	44% (2 035)	34% (1 211)
Airlines	0.85	77% (969)	58% (609)
Crawler	4.45	31% (3 288)	23% (1 859)
DinPhil	0.49	43% (110 035)	25% (55 257)

3.4 Bounded Model Checking

As we have already said above, the trace recorded from a suspicious execution does not identify the execution from which it was generated in a unique way. Moreover, even the original suspicious execution based on which the applied dynamic analysis generated a warning about the possibility of some error needs not contain an actual occurrence of the error (even if the error is real). To cope with such situations, apart from possibly exploring several paths through the state space corresponding with the recorded trace, we use bounded model checking that starts from the states from which an event that is marked as suspicious is enabled, or from some of its predecessors. The latter is motivated by the fact that once a suspicious event is reached, it may already be too late for a real error to manifest.

To be able to use bounded model checking to see whether an error really appears in the program, it is expected that the user supplies a JPF listener capable of identifying occurrences of the error (in our experiments, which concentrate on data races, we, e.g., use a slight modification of the `PreciseRaceDetector` listener available in JPF). The listeners looking for occurrences of errors may be activated either at the very beginning of replaying of a trace, or they may be activated at the beginning of each application of bounded model checking. The user is allowed to control both the depth of the bounded model checking as well as the number of backward steps to be taken from a suspicious event before starting bounded model checking.

3.5 Experiments

To demonstrate capabilities of DA-BMC, we consider four case studies. The first two, *BankAccount* and *Airlines*, are simple programs (with 2 or 8 classes, respectively) in which a data race over a single shared variable can happen. The *DiningPhilosophers* case study is a simple program (3 classes) implementing the problem of dining philosophers with a possibility of a deadlock. Finally, our last case study, *Crawler*, is a part of an older version of an IBM production software (containing 19 classes) with a data race manifesting more rarely and further in the execution. All the tests were performed on a machine with 2 Dual-Core AMD Opteron processors at 2.8GHz.

First, we measured the slowdown of program executions when recording various types of events. An analysis of the overhead associated with trace recording is presented in Table 3.1. The first column of the table gives the average time needed for executing the particular case studies while they are monitored by the Eraser dynamic analysis implemented as a Contest plugin. The second column gives the average slowdown when recording all the events that can be monitored by Contest and hence DA-BMC (cf. Section 3.2). Finally,

Table 3.2: Finding real errors in traces produced by Eraser

No. of traces	Error discovery ratio (traces found / BMC runs)				Time/memory consumption (sec/MB)			
	DFS		BFS		DFS		BFS	
	1	5	1	5	1	5	1	5
Bank	46%(1/1)	49%(2/2)	46%(1/1)	46%(2/2)	2/517	4/633	3/522	5/659
Airlines	100%(1/1)	100%(1/1)	100%(1/1)	100%(1/1)	1/482	1/482	1/482	1/482
DinPhil	100%(1/1)	100%(1/1)	100%(1/1)	100%(1/1)	11/417	20/411	20/414	22/413
Crawler	7%(0.8/15)	7%(1.8/34)	2%(0.5/49)	2%(1.2/50)	122/1312	268/1479	311/2857	321/3020

Table 3.3: Efficiency of finding errors using DFS in traces of Crawler produced by AtomRace

Traces searched \ No. of backtracked states / Max depth of BMC	3/10	5/15	10/30	15/45	20/60
1	66%	71%	78%	84%	90%
5	71%	73%	80%	85%	90%

States \ Max depth of bounded model checking	30	40	50	60	70	80
10	78%(0)	78%(1)	78%(3)	-	-	-
20	-	90%(0)	-	90%(2)	90%(2)	90%(2)
30	-	-	92%(2)	-	94%(0)	-
40	-	-	-	-	-	89%(5)

the third column gives the average slowdown when the thread and memory-access events are recorded only.

To sum up, when recording all the possible types of events mentioned above, the slowdown was about 30-40 %. When recording only thread and memory-access-related events, the slowdown was just about 20-30 % but the number of corresponding paths found by JPF increased by about 50 %.

Note, however, that the overhead differs quite significantly from one example to another. Therefore, the types of events to be recorded should be chosen depending on the program being analysed, taking into account which kind of events and how often it can generate. For instance, since the DiningPhilosophers case study is a program which frequently switches threads, but minimally accesses shared variables, it is sufficient to record only thread-related events in order to precisely navigate through the state space.

Next, we performed a series of tests in which we measured how often a real error is identified when replaying a trace and performing bounded model checking (BMC) in its vicinity. We let JPF to always backtrack 3 states from the state before a suspicious event and to use the maximum BMC depth of 10. The results are shown in Table 3.2. We distinguish whether 1 or up to 5 paths corresponding to the recorded trace were explored, using either DFS or BFS. For each of these settings and each case study, the left part of Table 3.2 gives the percentage of recorded traces based on which a real error was found. Further, in brackets, it is shown how many corresponding paths were on average found by JPF for a single trace, and how many times BMC was on average applied when analysing a single trace. The right part of Table 3.2 then gives the corresponding time and memory consumption. Clearly, BFS has higher time and memory requirements than DFS (mainly because it performs significantly more runs of BMC). It is also less successful in finding an error if the error manifests later in the execution (like in Crawler). It can also be seen that the number of corresponding paths searched has a little contribution to the overall success of finding a real error.

The low percentage of real errors found in traces of Crawler is mostly due to the number of false alarms produced by Eraser that were eliminated by DA-BMC, which nicely illustrates one of the main advantages of using DA-BMC. Further, note that classical model

checking as offered by JPF did not find any error in this case since it ran of our deadline of 8 hours (DFS) or ran out of the 24GB of memory available to JPF (BFS). To analyse how successful DA-BMC is in finding real errors in traces recorded in Crawler and how the success ratio depends on the various settings of DA-BMC, we have then done experiments with traces recorded using the AtomRace analysis, which does not produce any false alarms. The results can be seen in Table 3.3. Its left part shows how the percentage of real errors found depends on the number of explored paths corresponding to the recorded trace, the number of states backtracked from the state before a suspicious event, and the maximum BMC depth. The right part analyses in more detail how the percentage depends on the number of backtracked states and the maximum BMC depth (a single path corresponding to a recorded trace is analysed). The numbers in brackets express the percentage of replays which reached a 10 minute timeout. We can see that while increasing the number of searched corresponding paths has some influence on the error detection, it is evident that the BMC settings have a much greater impact. Moreover, the number of backtracked states increases the chances to find an error much more than the increased maximum depth of BMC.

3.6 Conclusion

We have presented DA-BMC—a tool chain combining dynamic analysis and bounded model checking for finding errors in concurrent Java programs (and also for debugging them). We have demonstrated on several case studies that DA-BMC allows one to benefit from the precision of model checking while not having to accept its full computational price. Said from a different perspective, one benefits from the relatively cheap dynamic analyses while the number of false alarms they produce is reduced via a use of bounded model checking.

Chapter 4

The ANaConDA Framework

This chapter presents the ANaConDA framework that allows one to easily create dynamic analysers for analysing multi-threaded C/C++ programs on the binary level. As the analysis is performed on the binary level, it does not require the source code of the program, and it also sees all the optimisations done by the compiler, allowing it to be more precise. The framework is quite general and can be instantiated for dealing with programs using various thread models. It also supports noise injection.

4.1 Introduction

Most of the techniques mentioned in Chapter 2 were implemented (and optimised) for Java. This does not mean that their principles could not be applied for C/C++ and other programming languages too. However, re-implementing an analysis once implemented for a different language environment is a tedious endeavour. Likewise, when there appears an idea for a new analysis, the journey to obtaining its fully functional implementation is usually rather long. While there are many frameworks simplifying this task for Java (which is one of the reasons why the implementations often target Java), there are only a handful of such tools for C/C++.

To address the lack of tools for C/C++, this chapter presents the ANaConDA framework which is a framework for adaptable native-code concurrency-focused dynamic analysis built on top of PIN [111]. The goal of the framework is to simplify the creation of dynamic analysers for analysing multi-threaded C/C++ programs on the binary level. In order to perform a dynamic analysis, one first needs to monitor the execution of a program. However, monitoring the execution of a program can be quite challenging and programmers might spend more time writing the monitoring code than by writing the analysis code itself. That is why the framework provides a monitoring layer offering notification about important events, such as thread synchronisation or memory accesses, so that developers of dynamic analysers can focus solely on writing the analysis code. In addition, the framework also supports noise injection which will be discussed in more detail in the next chapter.

In order to be able to monitor the execution of a program and perform some dynamic analysis as well as to insert some noise into the execution of the program, a need to execute some additional code in some places of the execution of the original program arises. There are several levels at which one can insert such additional code to the program—namely, at the source code level, at the level of the intermediate code, or at the binary level.

Inserting the code at the binary level has one big advantage over the other approaches

in that it does not need to have the source files of the program being analysed, which is particularly important when dealing with libraries whose source files might not be available even for the developers of the program under test. Another advantage might be that this kind of instrumentation is more precise in that we can insert the code exactly where we want it to be executed, and the placement is not affected by any optimisations possibly made by the compiler. These advantages of course come at the cost of that we may possibly lose access to various pieces of high-level information about the program (names of variables, etc.). However, even such information can be available if we have the debugging information present in the program, and moreover, we can also get access to some low-level information, like register allocations, which might be important for some analyses.

In this chapter, we identify several typical problems that arise when trying to monitor the execution of multi-threaded C/C++ programs at the binary level (such as monitoring function execution, retrieving information about the executed code, dealing with atomic, conditional, or repeated instructions, as well as supporting different C/C++ multithreading libraries), and we discuss their possible solutions. The problems are first discussed on a general level, a more detailed description is given in the following sections, where a concrete implementation of the chosen solutions is described. Then we focus more on how to write an analyser using the framework, how to get some useful information which may help the user in locating an error, and how to use the tool. As the framework can be instantiated to support various multithreading libraries, we also describe some concrete instantiations, in particular, the instantiation for `pthreads` and Win32 API. Finally, we discuss several real-life experiments done with the framework.

Related Work. There exist many frameworks which may be used to simplify the creation of dynamic analysers for Java programs. The closest to ANaConDA is IBM ConTest [44] which inspired some parts of the design of ANaConDA. RoadRunner [61] is another framework very similar to ANaConDA. Both of these frameworks can monitor the execution of multi-threaded Java programs and provide notification about important events in their execution to dynamic analysers built on top of the frameworks. CalFuzzer [82] is an extensible framework for testing concurrent programs which can also be used to create new static and dynamic analysers and to combine them. Chord [121] is another extensible framework which might be used to productively design and implement a broad variety of static and dynamic analyses for Java programs. When dealing with C/C++ programs, the options are much poorer. One tool somewhat related to ANaConDA is Fjalar [68] which is a framework for creating dynamic analysers for C/C++ programs. However, Fjalar is primarily designed to simplify access to various compile-time and memory information. It does not provide any concurrency-related information. Moreover, it is build on top of Valgrind [126], which brings several disadvantages as discussed in Section 4.3.

As for binary instrumentation frameworks, many were developed over the years. Most of them, such as PIN [111] or Valgrind [126], control the whole execution of a program, instrumenting its code just before it is executed (using just-in-time compilation) or when it is loaded into the memory. Other frameworks, e.g., PEBIL [95], instrument the program's binary file in advance and do not participate in its execution in any way. There are also frameworks which combine these two approaches—e.g., VMAD [81] inserts several versions of the instrumented code into the program's binary file and then chooses at run-time which version will be executed.

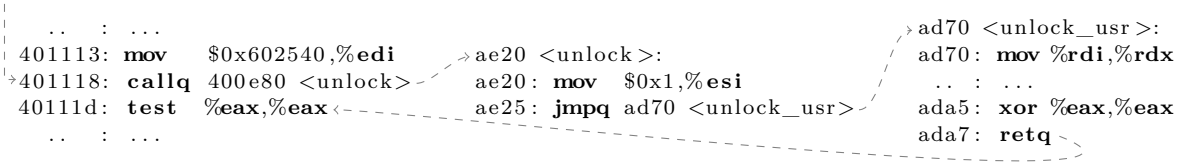


Figure 4.1: An example of an execution not triggering an after-function notification

Plan of the chapter. The rest of the chapter is organised as follows. In the next section, various typical problems of monitoring the execution of C/C++ programs at the binary level are discussed with their possible solutions. Section 4.3 then describes the concrete implementation of the proposed solutions in the ANaConDA framework. Section 4.4 shows how to configure the framework to support different multithreading libraries and Section 4.5 provides the basics of how to use the framework to analyse a multi-threaded C/C++ program. In Section 4.6, an experimental evaluation of the framework is given. Section 4.7 then concludes the chapter and discusses some of the interesting directions for future work.

4.2 Monitoring at the Binary Level

In this section, we discuss several typical problems that arise when trying to monitor and analyse the execution of a multi-threaded C/C++ program compiled to a binary form. In particular, after a brief introduction of the types of binary instrumentation frameworks that one can use to insert execution-monitoring code, we discuss the problems of monitoring function execution, retrieving information about executed instructions, handling atomic, conditional, and repeatable instructions, and abstracting concrete synchronisation primitives for the analysers to be used. For these problems, we analyse possible solutions, trying to stay on a rather general level. In Section 4.3, we will then present some further implementation details concerning the use of the proposed solutions in the ANaConDA framework.

4.2.1 Instrumentation Frameworks

There exist several frameworks for binary instrumentation which can be used to insert execution-monitoring code to a program. They might be divided into two groups—static and dynamic binary instrumentation frameworks.

Static binary instrumentation frameworks, like, e.g., PEBIL [95], insert execution-monitoring code to a program by rewriting the object or executable code of the program before the program is executed, thus modifying the content of the program's binary file. *Dynamic binary instrumentation frameworks*, like, e.g., PIN [111] or Valgrind [126], insert execution-monitoring code to a program at run-time, leaving the program's binary file untouched.

An advantage of static binary instrumentation is that it does not suffer from the overhead of instrumenting the code of a program every time it is executed. On the other hand, it cannot handle constructions like self-modifying or self-generating code, which is not known before the program actually executes. On the contrary, dynamic binary instrumentation is slower, but it can cover all the code that is executed by a program. Furthermore, since the binary file of the program is not modified in any way, the instrumentation is more transparent to the user who can run some (possibly lengthy) analysis on the program and, at the same time, use the program as usual. This possibility is also very important when

analysing libraries as it allows the user to analyse a library when used by a program being analysed and simultaneously allow other programs to use the same library as usual. This is not possible without maintaining two separate versions of the library and coping with problems with paths to these versions when the code of the library is rewritten before its execution (usage) in case of static instrumentation.

However, regardless of which of the two types of the instrumentation approaches is used, there are some issues that need to be dealt with when analysing multi-threaded programs at the binary level. These issues are discussed below.

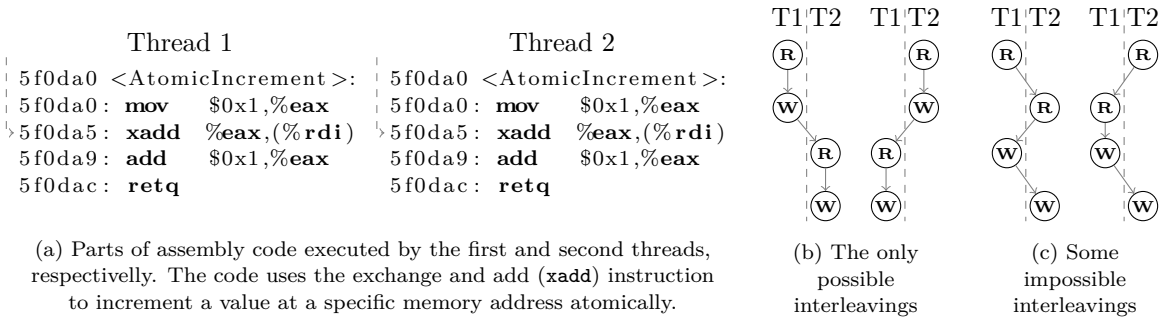
4.2.2 Monitoring Execution of Functions

The first problem to deal with is how to properly monitor the invocation and termination of functions. This is very important when analysing multi-threaded C/C++ programs as thread management, synchronisation of threads, and other thread-related actions are typically implemented by calling specific library functions. For instance, if an analyser needs to know that the monitored program acquired a lock, the best time to issue a notification about this event is after the function performing the lock acquisition is finished. However, since `call` instructions are not *fall-through instructions* (i.e., there is no guarantee that the instruction by which the program will continue after the invoked function finishes will be the instruction right after the given `call` instruction), one cannot place the code notifying an analyser about the event after the `call` instruction itself.

One way to solve the above problem could be to wrap the function to be monitored in another function and call everywhere in the program the wrapper function instead of the original one. The wrapper function could then internally call the original function, but also execute some code before and after it is called. This solution, however, suffers from two problems. First, the framework used for the binary instrumentation would have to support function wrapping (replacement), and second, calling the original function from the wrapper function might be quite time consuming and may lead to a significant slowdown of the whole analysis. Moreover, to wrap a function, we need to have a wrapper function with the same signature as the original function, containing the required monitoring code, so it might also be problematic to use this approach when we do not know in advance which functions we will be wrapping.

Another way to solve the problem could be to insert the monitoring code before and after the code of the monitored function itself, e.g., by inserting the code before the first instruction of the function and before each return instruction in the code of the function. So, instead of issuing the notification after the function returns, the analyser would be notified right before the function returns, which would be practically the same from the point of view of the analyser. An additional advantage of this approach would be that it would also decrease the instrumentation overhead as instead of analysing all `call` instructions (to see whether they could invoke the function to be monitored) and instrumenting many of them, the code of the functions to be monitored would only be instrumented. Nevertheless, this approach has one critical pitfall—namely, at the binary level, it is possible to return from a function even from code not belonging to that function!

A concrete example where instrumenting all return instructions of a function will not trigger the code that should be executed after the function returns can be seen in Figure 4.1. The figure shows three pieces of code. On the left there is a part of the binary code generated from a simple C++ program which uses the pthread library to guard a critical section through the `pthread_mutex_lock` and `pthread_mutex_unlock` functions. The other



(a) Parts of assembly code executed by the first and second threads, respectivelly. The code uses the exchange and add (`xadd`) instruction to increment a value at a specific memory address atomically.

(b) The only possible interleavings

(c) Some impossible interleavings

Figure 4.2: An example illustrating problems concerning atomic instructions

two code snippets are parts of the code of the pthread's `__pthread_mutex_unlock` and `__pthread_mutex_unlock_usercnt` functions. Once the execution of the program reaches the `call` instruction at address `401118`, the program calls `__pthread_mutex_unlock`[1] (the function `pthread_mutex_unlock` is in fact just an alias of `__pthread_mutex_unlock`). The execution then continues in the `__pthread_mutex_unlock` function which adds the second argument of the `__pthread_mutex_unlock_usercnt` function and then jumps to it. The program starts executing the `__pthread_mutex_unlock_usercnt` function, and after a while, it returns, but not to the `__pthread_mutex_unlock` function (because this function did not call it, it jumped to it), but to the function that called `__pthread_mutex_unlock`. Now, it should be clear where the problem is: If we instrument the `__pthread_mutex_unlock` function, there will never be a notification that a thread released a lock because there will be no code inserted before the return instruction which is executed to return from the call to the `__pthread_mutex_unlock` function. Moreover, when we try to insert the monitoring code before the return instruction that is really executed in the given example, we have to be still careful. The reason is that we do not know whether some optimisation did not make more functions jump to the given code. If so, it could, for instance, happen that the `pthread_mutex_lock` function jumps to this part of code too, which would lead to issuing a notification that a thread released a lock while it instead acquired it.

Nevertheless, there is a solution to the above described problem. Namely, we can use the fact that no library function can jump outside of the code of the library itself. This is because when the library is compiled, the compiler can insert jumps just to the parts of the code it knows, and it knows only the code of the library itself. So, if we insert some monitoring code before every return instruction in the library, we must be able to detect that the program is returning from a call to some of the library's functions. The only issue that is then left is to find out from which function the program is actually returning. Probably the most efficient way in doing so is to partially monitor the call stack, i.e., when a function whose execution should be monitored is called, some monitoring code inserted before the first instruction of the function can be triggered, and in this code, we can save the current state of the thread's call stack (the value of the stack pointer is quite sufficient). Then, when a return instruction is executed in the library, we can check if the current stack pointer matches the previous stored one, and if yes, issue a notification that a certain library function has been executed.

[1] For the sake of simplicity, we ignore here the fact that the function is not called directly, but through the jump table stored in the `.plt` section of the program's binary.

4.2.3 Retrieving Required Information

A further problem to be solved is how to provide analysers with sufficient information about an instruction or function whose execution has just finished. This is because a lot of information is lost when an instruction or function is finished. For example, we often do not know which memory an instruction has accessed after the instruction is executed because the memory address might have been computed from the values of some registers, and those values might have been changed when the instruction was executed. Similarly, when a function is executing, we can access its parameters easily, but when its execution finishes, the arguments might not be easily obtainable anymore.

To provide an analyser all the information it needs, it is thus often necessary to preserve some of the information obtained before executing some instructions and to reuse this information in the monitoring code executed later. In case of multi-threaded programs, such information must be tracked for each thread separately, and since it might be accessed quite frequently, the efficiency of storing it is of a high importance. As the best way to deal with this problem, we see a utilisation of some kind of thread local storage, which is lock-free, i.e., it does not require any synchronisation between the threads. Fortunately, some binary instrumentation frameworks, such as PIN [111], provide a support of thread local storage.

In fact, according to our experience, it is often useful to store also information which can be computed even after executing some instructions. This is, in particular, the case of source-code information about the code being executed such as the names of variables accessed. Such information may be available, e.g., through the debugging information present in the program's binary, but accessing it is often slow. If we get this information in some monitoring code and know that some other monitoring code executed after a while will need this information too, it is better to store the extracted information and reuse it later than to extract it again.

4.2.4 Atomic Instructions

Another problem which may arise when analysing programs at the binary level is the need to properly handle atomic instructions that access the memory more than once. Indeed, when a single instruction accesses the memory multiple times, the monitoring code should notify the analyser about that, which is typically done by generating the appropriate number of memory access events. If the instruction is not atomic, this is perfectly fine, but when the instruction is atomic, some analysers need to be informed about the atomicity of the appropriate sequence of memory accesses, or else they might produce false alarms.

In particular, some of the detectors which may have troubles with atomic instructions are data race detectors, such as Eraser [138] or AtomRace [97], and atomicity violation detectors, such as AVIO [110]. Both these kinds of detectors analyse possible interleavings of accesses to particular memory addresses and report an error if there are two unsynchronised memory accesses to the same memory address and at least one of the accesses is a write access, or there is an interleaving of the memory accesses which is unserialisable. Clearly, if such detectors are not informed that some sequence of memory accesses is guaranteed to execute atomically, they can produce false alarms. A concrete illustration of such a scenario can be seen in Figure 4.2 which shows a situation when two threads are executing the code of the `AtomicIncrement` function. This function uses the exchange and add (`xadd`) instruction to atomically increment a value at a given memory address. The `xadd` instruction first reads a value at a given memory address, then adds some value to it, and then stores the modified

value back at the same memory address. If the monitoring code notifies the analysers that the program read a value from some memory address and then wrote to the same memory address without the information that these two accesses happened atomically, the analyser will assume that all the interleavings shown in Figures 4.2b and 4.2c are possible in the program because the threads are not synchronised in any way. However, in fact, the interleavings in Figure 4.2b are the only ones that can happen in the program.

One possible solution to this problem is to extend the access notifications with additional information saying that an access that has just happened occurred atomically wrt. some previous access. Pairing such accesses in the analyser may, however, be problematic (e.g., leading to backtracking in the analysis). In our opinion, a better way is to introduce a new type of notification which tells the analyser that one instruction performed multiple accesses at once, and let it decide if it wants to react to this situation and how (the analysers must of course be ready to receive such notifications).

Note that in some higher programming languages like Java, there is no need to solve this kind of problems when performing analysis at the binary, or more precisely byte-code, level as there are no byte-code instructions which access the memory more than once [101]. Atomic updates are implemented here as a block of byte-code instructions placed between the `monitorenter` and `monitorexit` instructions which lock the memory address securing that no other thread will access it until the update is completed. If all accesses to the memory address are guarded by the same lock, the analysers will see that there are no possible interleavings leading to an error on this memory address and will not produce false alarms. The same solution can be used in C/C++ programs, but using atomic instructions can be much more efficient, and hence it is often used.

4.2.5 Conditional Instructions and Loops

Beside the atomic instructions there are a few other kinds of instructions which must be carefully handled in order not to confuse various existing analysers. This is, in particular, the case of the conditional instructions and the repeat instructions. While the conditional instructions might not be executed at all even when the control reaches them, the repeat instructions, on the contrary, may be executed more than once as though they were placed in a loop. For instance, the `rep`-prefixed instructions, designed for manipulating continuous sequences of memory locations (e.g., within string operations), are both conditional and repeat instructions since they may be executed a fixed number of times, until some condition is met, or sometimes not executed at all (if the loop they involve should be executed zero times).

Since many of these kinds of instructions access memory, we need to be sure that the access notifications are sent correctly, i.e., that the analyser is notified only when the instruction was really executed or every time the instruction was executed in a loop. When using dynamic binary instrumentation frameworks, where the instructions are executed by some kind of a virtual machine, the virtual machine can usually handle these things for us. On the other hand, in case of the static binary instrumentation, where we might not know if the instruction will be executed or how many times it will be executed, the situation can become unsolvable without some approximation.

4.2.6 Abstraction of Synchronisation Primitives

Since thread management and synchronisation in C/C++ programs is usually done by calling suitable library functions as we have already mentioned above, and since there exist

many different libraries which can be used for this purpose, a further question is how to support analysis of programs using any of these libraries with a minimum additional effort (at least when no highly non-standard synchronisation means are used).

In order to allow for an easy support of multiple libraries, the dynamic analysers themselves should clearly be separated from low-level details of using the libraries. For example, an analyser should not know that a lock is represented by a `pthread_mutex_t` structure or a Windows `HANDLE`, it should just be able to say whether two locks are the same or not. More generally, according to our experience, in order to satisfy the needs of common analysers, one needs to allow them (1) to suitably identify program threads in order to be able to distinguish which thread behaves in which way, (2) to recognise which functions are used for various standard synchronisation operations, and (3) to suitably identify the synchronisation resources used (such as locks or conditions) in order to be able to say which of them are used when.

Since one can hardly find a fully automatic solution of the above needs, we find as appropriate to provide users with a support allowing them to solve these issues in an easy way manually. In particular, for a given library, the users should be able to easily specify:

- Which functions in the library are performing common thread-related actions such as acquiring a lock, waiting on a condition, etc.
- Which arguments of these functions represent the synchronisation resources they work with.
- How to transform the concrete representations of synchronisation resources and threads to their abstract identifications.

In Section 4.3, we will describe in more detail the concrete implementation of this approach as used in our framework.

4.3 Implementation

To validate the solutions proposed in the previous section, we have implemented a prototype tool which can monitor the execution of a multi-threaded C/C++ program, insert noise into it, and provide analysers that can be written on top of it with various important pieces of information that are typically needed when detecting errors in concurrency. In this section, we will discuss how the above discussed general ideas have been concretised in the implementation.

We have based our implementation, called the ANaConDA[2] framework, on top of the PIN binary instrumentation framework [111]. There are several reasons motivating the use of PIN as a binary instrumentation backend. First, PIN performs dynamic instrumentation, i.e., it instruments a program in the memory before it is executed. This means that the binary files of the program are left untouched. This is especially important when dealing with libraries as it allows one to transparently use an instrumented version of a library and simultaneously use the library as usual in other programs. PIN can also be used on both Linux and Windows, compared to Valgrind which is Linux-only, which allows a much wider range of programs to be analysed. Of course, PIN is primarily developed for use with Intel binaries. However, if the binary code does not contain any special AMD-only instructions, PIN works fine even for AMD binaries. Another advantage of PIN is that it preserves the

[2] `http://www.fit.vutbr.cz/research/groups/verifit/tools/anaconda`

parallel execution of threads of the analysed multi-threaded program. Valgrind, on the contrary, serialises thread execution [126], which may unnecessarily slow down the program and also the analysis as the analysis code usually runs in these threads too.

Synchronisation. To allow the tool to be easily used for developing dynamic analysers capable of running over various libraries for thread management and synchronisation, we followed the propositions made in Section 4.2.6 and implemented a system which allows the user to easily abstract the information needed by typical analysers from the concrete form used in a given library into a common format available for the analysers. In particular, for each important type of synchronisation functions, e.g., functions for acquiring and releasing locks or for signaling conditions and waiting on them, the user may specify the names of the functions implementing these operations. For an abstract identification of the synchronisation resources used (e.g., locks or conditions), we introduced special `Mapper` objects. When the user defines the names of synchronisation functions, he also specifies the indices of their arguments holding the synchronisations resources used by these functions as well as the the `Mapper` object which should be used to translate these resources to their abstract names. The translation is then done through the `map()` method which takes a pointer to an address where the appropriate argument of the encountered synchronisation function is stored and returns a number abstractly identifying the appropriate synchronisation resource. For example, for the case of locks and conditions from the pthread library used to synchronise threads, the mapper objects can use the fact that the locks and conditions are objects of the `pthread_mutex_t` and `pthread_cond_t` structures existing in the same logical memory space shared by the threads. Since objects existing in the same memory space are uniquely identified by their addresses, one can devise a `Mapper` object which simply uses the addresses of the objects as their identifiers in this case (hashed to 32 bits as is usual also for various other identifiers in PIN). Finally, as for an abstract identification of threads, we use the fact that the PIN framework already provides some thread abstraction, and so we reuse it for a unique thread identification.

As the synchronisation functions to be monitored are assumed to be specified by the users, and hence we do not know them and cannot prepare wrappers in advance, we cannot use the function wrapping approach for monitoring function executions (not to mention that calling the original function from a wrapper function is often really slow). Therefore, we use the approach for monitoring function executions proposed at the end of Section 4.2.2. Namely, we insert a monitoring code before the first instruction of every synchronisation function specified by the user and also before all the return instructions in the libraries containing at least one of these functions. When some synchronisation function is about to be executed, the monitoring code stores the current value of the stack pointer together with a pointer to the notification function which should be called after the monitored function is executed to a separate shadow stack. Once a return instruction is to be executed, the monitoring code compares the current value of the stack pointer with the one stored at the top of the shadow stack, and if there is a match, it will notify the analyser that a synchronisation function was executed by calling the notification function stored at the top of the shadow stack.

Special Instructions. We have implemented the support for notification about several atomic accesses to memory from an atomic instruction as discussed in Section 4.2.4 (so far for accesses to the same memory address only, which is sufficient for detecting data races on which we currently concentrated). To handle conditional instructions discussed

in Section 4.2.5, we used the PIN's `INS_InsertPredicatedCall()` function to put the monitoring code around these instructions. Inserting the code through this function ensures us that PIN will check if the instruction will really be executed and invoke the monitoring code in this case only. However, unfortunately, using this function to insert the monitoring code around the `rep`-prefixed instructions led to a very strange behaviour in which the monitoring code was sometimes not invoked even when the instruction was executed. To fix this problem, we had to use the `INS_InsertCall()` function as code inserted using this function is always called, and we then check ourselves if the instruction will or will not be executed. This implementation is a little less efficient as letting the PIN do the checks is quicker, but it behaves correctly, and considering that the amount of `rep`-prefixed instructions is not so large, it is a quite negligible slowdown.

Backtraces. A problem that has not been considered in Section 4.2.3 is that the information needed for analysis is not the only information useful for the users. When the analyser detects an error, it should provide the users as much information as possible to help them localise the error. Retrieving information about the executed code, such as names of variables or locations in the source code, can give the users some information about the error. However, this information is often not sufficient since it may be difficult to know how the program got to the variable or location where the error was detected. A much better help to the user is a backtrace to the erroneous part of the program.

ANaConDA currently supports backtraces equivalent to the ones given by the Linux `backtrace()` function, which contain the return addresses of the currently active function calls. The return addresses are stored on the call stack in the corresponding stack frames. The top stack frame's address can be obtained from the base pointer register, and each stack frame also contains the previous value of the base pointer, referring to the previous stack frame. By following the chain of base pointers, we can extract the return addresses and create a backtrace although we have to be careful when processing the stack frames as sometimes (e.g., during the initialisation of the program) the base pointer register may be used for other purposes and might point somewhere else than to a stack frame. The advantage of this approach is that we do not need to monitor every function call in the program and update the backtrace constantly. We are constructing the backtrace on demand, i.e., only when the analyser explicitly requests it, and we only need to know the value of the base pointer register, which can be retrieved with a negligible overhead. The only drawback is that the program must properly create the stack frames, which may sometimes not be true if some optimisations are used.

4.4 Instantiation

The ANaConDA framework abstracts analysers built on top of it from the specific multithreading library used, but it of course cannot do that without any information about the library. As explained in more detail in the previous section, the user must specify: (1) the names of the functions performing various thread-related operations, (2) the indices of parameters holding the synchronisation primitives the functions operate with, and (3) the `Mapper` objects used to abstract the synchronisation primitives to numbers uniquely identifying them. Abstraction of synchronisation primitives is necessary because their representation varies across various libraries, but analysers need to work with them in a uniform way.

```
pthread_mutex_lock 1 addr()
pthread_mutex_trylock 1 addr()
```

(a) Lock acquisitions (`lock` file)

```
__pthread_mutex_unlock_usercnt 1 addr()
```

(b) Lock releases (`unlock` file)

Figure 4.3: An example of the configuration of monitoring lock operations in the `pthread` library

Pthreads. For example, if we use the `pthread` library and want to get notifications about lock acquisitions and releases, we have to specify that the `pthread_mutex_lock` and `pthread_mutex_trylock` functions are performing the lock acquisitions and that the `__pthread_mutex_unlock_usercnt` function handles the lock releases. This is done by adding the names of these functions to the `lock` and `unlock` configuration files, respectively. All of these functions are taking the lock as the first parameter, and because locks are objects of the `pthread_mutex_t` structure, we can use the ANaConDA framework's built-in mapper object `addr` to convert the addresses of these objects into numbers uniquely representing them. To give this information to ANaConDA, we have to specify the index and the name of the `Mapper` object right after the name of the corresponding monitored function as can be seen in Fig. 4.3. The instantiation for signaling conditions and waiting on them is similar, we just have to instruct the framework to monitor the `pthread_cond_signal`, `pthread_cond_broadcast`, and `pthread_cond_wait` functions by inserting the appropriate information to the `signal` and `wait` configuration files.

Win32 API. As for the Win32 API, there is no function that performs purely lock acquisitions. Instead, the `WaitForSingleObject` function is used taking a generic `HANDLE` as the first parameter and performing a lock acquisition only if the `HANDLE` represents a lock (it may also represent, e.g., a thread or an event). In this case, we have an alternate way to tell ANaConDA when a function performs a lock acquisition. We can specify that the `WaitForSingleObject` function is a generic wait function whose behaviour depends on the type of the synchronisation primitive passed to it and then name a function which creates or initialises new locks. The framework then remembers which synchronisation primitives are locks because they were created by the user-identified lock creation/initialisation function. Subsequently, when a generic wait function (like `WaitForSingleObject`) is called, it will first determine what kind of synchronisation primitive its parameter represents. If it is a lock, it will properly trigger the lock acquisition notifications. In particular, in Win32 API, locks are created by the `CreateMutex` function which returns a `HANDLE` representing the lock. Configuring lock releases is much simpler as they are performed by a dedicated `ReleaseMutex` function which takes the lock (`HANDLE`) as the first parameter. As the `HANDLE` is in fact a generic pointer, we can also use the `addr` mapper object here to transform it into a unique number.

The Win32 API has no functions for signaling conditions and waiting on them. If such operations are needed, the users usually implement the operations themselves or use some libraries like `pthread-win32` implementing them. However, as ensuring that the functions performing these operations will trigger the corresponding ANaConDA notifications is as easy as adding a few lines to the appropriate configuration files, the framework does not have any problems with the users using their own custom functions for these operations, which illustrates the generality of the framework.

Another problem with the Win32 API is that some of the functions that need to be monitored are jumping at the beginning of other monitored functions. In this case, PIN executes the monitoring code inserted before such functions, and if no special care was

taken, the analyser would get a notification about a single event multiple times. The solution could seem to be easy as one could, e.g., think of simply specifying that one of the functions should not be monitored. However, the functions often have exactly the same names, so one cannot so easily differentiate between them. The framework solves this problem by checking if the stack pointer changed when a monitored function is about to be executed, and it does not issue a notification if its value remained the same as that means that nobody called the function, and the control must have jumped to it.

4.5 Usage

To analyse a multi-threaded C/C++ program using ANaConDA, one first has to write (or get) an analyser to be used. The analyser must have the form of a shared object (in Linux) or a dynamic library (in Windows) which contains a set of functions that ANaConDA should call when a specific event, such as a lock acquisition, occurs in the program being analysed. The analyser has to register the callback functions for the events it needs to be notified about. This is done by calling the appropriate registration functions (provided by ANaConDA) in the `init()` function of the analyser, which ANaConDA executes once the analyser is loaded. For example, to be notified about lock acquisitions and releases, the analyser has to register its callback functions using the `SYNC_AfterLockAcquire` and `SYNC_BeforeLockRelease` functions, respectively.

Performing the actual analysis is then quite simple. One just needs to execute the PIN framework with ANaConDA as the *pintool*[3] to be used and specify the analyser which should perform the desired analysis together with the program which should be analysed. Noise injection can be enabled and configured in the `noise` section of the `anaconda.conf` configuration file. Currently, only the *sleep* and *yield* noise is supported, but the user may use different noise injection settings for the read and write accesses and also for each of the monitored functions. The slowdown of the execution of the analysed program is similar to Fjalar, i.e., around 100 times. Note, however, that the slowdown is mainly due to PIN and depends on many factors such as the amount of instrumentation inserted, the amount of information requested by the analyser, the amount of noise injected into the program, etc.

4.6 Experiments

To test whether ANaConDA can handle really large and complex programs, we have used it to analyse the `Firefox` browser (more than 3 million lines of code) which uses the `pthread` library. We did not find any severe or unknown errors. We did, however, find several data races which are left in the code since they are considered harmless. Considering the size of the program, the fact that it is thoroughly checked for data races regularly, and also that we used a very simple data race detector and performed only a very limited set of tests since we did not have any automatic test suite to use, we consider these results to still be quite promising.

We further analysed the `unicap` libraries for video processing, which also use the `pthread` library and are considerably smaller (about 40k lines of code) which allowed us to perform a larger number of tests. We have found several (previously unknown) data races in the `libunicap` and `libunicapgtk` libraries. Two of the data races can be considered

[3] A pintool can be thought of as a PIN plugin that can modify the code generation process inside PIN, i.e., it determines which code should be executed and where in the monitored program.

severe as they may cause a crash of the program which uses these libraries. In both cases, one thread may reset a pointer to a callback function (i.e., set it to `NULL`) in between of the times when another thread checks the validity of this pointer and calls the function referenced by it, which can cause an immediate segmentation fault. We are currently preparing to report these errors to the developers using the ANaConDA's recently added backtrace support that can provide a rather detailed information where and why the error occurred.

Finally, we also successfully tested the framework on several Windows toy programs (100–500 lines of code). An application to larger programs is planned for the near future.

4.7 Conclusion

In this chapter, we have presented ANaConDA—a framework simplifying the creation of dynamic analysers for analysing multi-threaded C/C++ programs on the binary level. We have discussed several typical problems which arise when monitoring multi-threaded C/C++ programs at the binary level in order to allow for their dynamic analysis, and we have proposed solutions to these problems. We have shown how to instantiate it for several widely used multithreading libraries and demonstrated on several case studies that it can handle even large real-life programs. With the help of the framework, we were able to write a simple analyser in a day and successively find several errors with it, which shows the usefulness of the framework.

For the future, there are several interesting directions that can be taken. First, we would like to improve our implementation of the proposed ideas, extend it by a support of more C/C++ concurrency libraries, and test the resulting tool on larger concurrent C/C++ programs. We would also like to implement more dynamic analyses on top of our framework and include their evaluation into the further experiments.

Chapter 5

Improved Noise Injection

This chapter describes how to utilise noise injection to increase the efficiency of dynamic analysis and testing. Beside proposing how the noise injection can be improved and devising several new noise injection techniques, this chapter also includes a detailed comparison of both the new and the existing noise injection techniques and gives some general suggestions on how to use the noise injection in particular scenarios.

5.1 Introduction

The result of both testing and dynamic analysis greatly depends on the witnessed execution of a multi-threaded program. Unfortunately, there is a huge amount of executions one may encounter, and only a small fraction of them usually cause an error. It is often harder to find the rare executions containing the error than to detect the error within the execution that contains it. One way to deal with this problem was described in Chapter 3 where an extrapolating dynamic analysis was used to find suspicious executions that are highly likely to contain an error. Such executions were then replayed in a (bounded) model checker which determined if they indeed contain an error or not. Despite the optimisation offered by the approach, replaying complex executions may not be feasible due to their size or presence of operations that cannot be replayed, such as input/output operations. A much simpler, cheaper, yet effective approach that addresses the same problem is *noise injection* which influences the scheduling of threads so that different interleavings of concurrent actions are witnessed, and one may more likely see the executions that contain an error. The biggest advantage of noise injection is its generality. It works well with techniques designed for both testing and dynamic analysis, and these techniques do not need to be changed in any way to work with each other.

In this chapter, multiple results achieved recently in the area of noise-based testing and dynamic analysis are presented. In particular, multiple heuristics for solving the *noise placement problem* (i.e., where and when to generate noise) as well as the *noise seeding problem* (i.e., how to generate the noise) are introduced and experimentally evaluated.

Moreover, we propose an improvement to the typical usage of noise injection which allows one to use a fine-grained combination of several noise placement and noise seeding heuristics within a single program. To demonstrate how the fine-grained combination of noise can be used, we create a new (fine-grained) noise injection technique tailored for improving the chances to detect data races, the most common type of concurrency errors. We implemented this new noise injection technique using the ANaConDA framework and tested

it on a set of smaller C/C++ projects. The obtained results show that using fine-grained noise injection techniques can lead to a further increase of chances to spot a concurrency errors. We also discuss the possible influence of various noise injection settings on dynamic analysis of multi-threaded C/C++ programs.

Apart from the technique mentioned above, we also present two additional noise injection heuristics—in particular, a new noise placement heuristic based on access patterns of shared variables and a new noise seeding heuristic which blocks all threads but one. Both of these heuristics target common atomicity violation scenarios, and the newly proposed noise seeding heuristic might also help in order violation scenarios. The newly proposed heuristics are compared with a selection of already existing heuristics which provided promising results in the previous experimental comparisons [54, 99].

The presented set of 8 Java benchmark programs and 4 C benchmark programs is the so-far largest comparison of noise-injection-based testing techniques. The comparison shows that the different heuristics can indeed significantly improve the efficiency of testing. However, they also show that there is no single best noise injection technique among the many noise injection heuristics, requiring a careful selection of the noise injection technique(s) to be used in a given scenario.

Finally, the chapter also provides a unified overview of multiple results from the area of noise-based testing and dynamic analysis that were published in the past few years [54, 55, 77, 92, 93, 96, 99]. In particular, various noise injection heuristics and their influence on error detection and ability to increase concurrency-related coverage [54, 96, 99] (absolutely and relatively, i.e., taking into account the longer test execution time caused by noise injection) are studied. The obtained experiences with noise injection on different levels (namely, Java bytecode and C/C++ binaries) are summarized into suggestions that can make further applications of noise-based testing and dynamic analysis easier.

Plan of the chapter. The rest of the chapter is organized as follows. Section 5.2 describes the current heuristics used to solve the noise placement and noise seeding problems. Then it presents our proposal of improving the typical usage of noise injection by combining different noise placement and noise seeding heuristics. Finally, it introduces several new noise injection heuristics. Section 5.3 evaluates the improved noise injection proposed in the previous section, showing that combining different noise placement and noise seeding heuristics can further increase the efficiency of both testing and dynamic analysis. Section 5.4 summarises the previously published comparisons and then focuses on evaluating the newly proposed heuristics. It provides a deep comparison of both the new and existing heuristics, showing that the newly proposed heuristics are useful in many cases, but none of the heuristics included in the comparison is best in every situation. It also discusses various technical aspects of noise injection on binary and byte-code levels and provides a few suggestions for noise-based testing and dynamic analysis. Section 5.5 then concludes the chapter and mentions several possible future directions in the area of noise-based testing and dynamic analysis.

5.2 Noise Injection

Noise injection techniques [44] aim at increasing the number of different interleavings witnessed in the executions of a multi-threaded program by disturbing the scheduling of its threads. This is achieved by inserting certain noise generating code at some locations of

the program whose goal is to force the program to switch threads at times when it would normally seldom do it. This way, rare executions may be forced to appear, possibly leading to an occurrence of an error (or to a behaviour that can be claimed suspicious by a dynamic analyser).

The effectiveness of noise injection depends on a satisfactory solution to the *noise placement* and *noise seeding* problems. The noise placement problem addresses the question where, i.e., at which program locations, and when, i.e., at which executions of these locations, to cause a noise. The noise seeding problem then determines how to cause the noise, i.e., which type of noise generating mechanism should be used, and how long it should last. The problems are, of course, not independent, and so, a suitable *combination* of noise placement and noise seeding heuristics (and of suitable values of their many parameters) is to be sought in practice.

In this section, we first discuss the existing approaches for solving the noise placement and noise seeding problems. Then, we argue that these approaches can be improved by a careful fine-grained combination of different noise injection techniques. Finally, we propose several new noise injection techniques.

5.2.1 Noise Placement Heuristics

Noise placement heuristics determine where, i.e., at which program locations, and when, i.e., at which executions of these locations, should a noise be injected.

It is discussed in several papers [41, 48, 144] that putting a noise at every possible program location (*ploc* [41]) is inefficient. This approach significantly increases the incurred overhead, and it does not help much in increasing chances to find bugs since only a few relevant context switches are critical for a concurrency error to manifest. Also, it turns out that putting a noise at a certain program location can help to spot the concurrency error, but it can also mask it completely.

The IBM ConTest tool [42] allows one to inject a noise only before and/or after concurrency-related events (namely, accesses to class member variables, static variables, and arrays stored in the JVM heap, calls of `wait()`, `interrupt()`, `notify()`, `monitorenter`, and `monitorexit` routines). Since the tool has no information which member fields and arrays are really shared (i.e., accessed by multiple threads), all instructions operating with the heap are considered. Moreover, motivated by a coding anti-pattern in which developers use calls of `wait()` instead of proper synchronization, ConTest is able to intercept calls to the `wait()` and `sleep()` routines too.

The *rstest* tool [144] considers as possibly interesting only those locations that appear before concurrency-related events. Moreover, rstest uses a simple escape analysis and a lockset-based algorithm to identify the *unprotected accesses* to shared variables. An unprotected access reads or writes a variable which is visible to multiple threads without holding an appropriate lock. This optimization reduces the number of program locations where the noise can be put but suppresses the ability to detect some concurrency errors, e.g., high-level data races or deadlocks where all accesses to problematic variables are correctly guarded by a lock.

Moreover, the number of accesses to shared memory and calls of synchronization elements is still high in multi-threaded programs. Therefore, several heuristics for determining more concretely where and when to put a noise were proposed [18, 41, 144, 148].

The simplest heuristic is based on a random number generator [41, 144]. This *random* heuristic puts a noise before an executed program location with a given probability, where

the probability is the same for all program locations considered. Most other heuristics extend this heuristic in a way that they reduce the number of possible program locations before which the noise might be injected. When considering all possible program locations in a program, this heuristic is called *random-all* below to distinguish it from the other heuristics that can be seen as modifications of the *random* heuristic.

It was shown [18] that restricting the number of program locations only to those accessing shared variables or a specific shared variable when applying the *random-all* heuristic increases the probability of spotting an error. These two modifications of the *random-all* heuristic are denoted here as *sharedVar-all* and *sharedVar-one*, respectively. When the *sharedVar-one* heuristic is used, the shared variable is usually chosen randomly from a list of known shared variables.

Several heuristics based on concurrency coverage models have been published. Coverage-directed generation of interleavings [41] considers two coverage models. The first model determines whether the execution of each method was interrupted by a context switch. The second model determines whether a method execution was interrupted by any other method. The level of methods used here can be in most of the cases too coarse. In [148], a coverage model considers, for each synchronization primitive, various distinctive situations that can occur when the primitive is executed (e.g., in the case of a synchronized block defined using the Java keyword `synchronized`, the tasks are: *synchronization visited*, *synchronization blocking* some other thread, and *synchronization blocked* by some other thread). The approach then injects a noise at corresponding synchronization primitive program locations to increase the coverage. None of these two heuristics focuses on accesses to shared variables which can limit their ability to discover some concurrency errors, e.g., data races.

A coverage-based noise placement heuristic [99] (referred to as *coverage* further on) targets both accesses to shared variables as well as the use of synchronization primitives, and so it can be used to discover lock-based deadlocks as well as data-related concurrency errors, such as data races and atomicity violations. The heuristic considers only program locations that appear before concurrency-related events as suitable for noise injection. The technique detects subsequent accesses to shared variables and monitors whether these accesses originated in different threads. Such couples of subsequent accesses are considered as interesting to be influenced by noise. The noise in particular tries to test the opposite orderings of recorded events in each couple. Therefore, a noise is put before the first access recorded in a couple with a hope that the access which was recorded as the subsequent occurs earlier. If both accesses are guarded by the same lock, the described approach would inject a noise into a shared critical section which would not change the ordering of the recorded events. In such a case, the heuristic injects the noise before the appropriate locking operation where the common lock was obtained. Additionally, this heuristic monitors the frequency of a program location execution during a test and puts a noise at the given program location with a probability biased wrt. this frequency—the more often a program location is executed the lower probability is used.

5.2.2 Noise Seeding Heuristics

Noise seeding heuristics determine how to cause a noise, i.e., which type of noise generating mechanism should be used, and how long should it last, i.e., how strong the noise should be.

As the primary purpose of injecting a noise is to disturb the usual scheduling of threads, the noise generating mechanism should influence the scheduler in some way. There exist

several ways how a scheduler decision can be affected in Java [41]. The *priority* heuristic changes priorities of threads which allows chosen threads to make progress more often than threads with a lower priority. The *yield* heuristic injects one or more calls of the `yield()` method which causes a context switch. The *sleep* heuristic injects one call of `sleep()`, and the *wait* heuristic injects a call of `wait()`. The `sleep()` and `wait()` methods take a timeout for which to block a thread as their parameter. In case of the *wait* heuristic the concerned thread must first obtain a special shared monitor, then call `wait()` with a timeout on it, and finally, release the monitor. Using the monitor makes the current thread flush all local data to shared memory and make them visible for other threads. Likewise, the *synchYield* heuristic combines the *yield* heuristic with obtaining a monitor. The *busyWait* heuristic does not obtain a monitor, but instead loops for some time.

The *haltOneThread* heuristic [148] occasionally stops one thread until all remaining threads cannot make any further progress. Finally, the *timeoutTamper* heuristic randomly reduces the time-out used when calling the `sleep()` and `wait()` methods in the tested program. This allows one to test that the delay inserted by these methods is not used instead of proper synchronization.

All the noise seeding heuristics mentioned above are parametrised by the *strength of noise*. In case of the *sleep*, *wait* and *busyWait* heuristics, the strength gives the time to wait or loop. In the case of the *yield* heuristic, the strength says how many times the `yield()` routine should be called. Finally, in the case of the *priority* heuristic, the strength determines how much the thread priority changes.

In [18], a *barrier* scheduling heuristic based on semaphores is presented. Each shared variable is assigned a specific semaphore in such a way that a thread is made to wait just before the particular shared variable is accessed. When more than one thread is waiting at the same monitor (and thus for access to the same variable), then the `notifyAll()` method is used to simultaneously advance the waiting threads in hope to spot a data race. To prevent deadlocks, the waiting of threads on the injected semaphores is timed.

While the above works discuss noise seeding heuristics for Java, many of them are also applicable for other languages such as C/C++. For example, the ANaConDA framework, introduced in the previous chapter, supports the *yield*, *sleep*, and *busyWait* heuristics.

5.2.3 Fine-Grained Combinations of Noise

In our opinion, the previously used approaches to noise injection have one drawback–namely, they allow the user to specify the noise seeding heuristic at a global level only, meaning that the same type of noise with the same parameters will be used everywhere in the program (an exception is the random setting of ConTest when a random type of noise with random parameters is used every time some noise is to be generated). However, in our opinion, supported by the later presented experimental data, it is sometimes beneficial to use different noise seeding heuristics for different locations in the program (e.g., for the read accesses, write accesses, each of the thread synchronisation functions, etc.) and to do it in a systematic rather than random way.

To illustrate our idea, take, e.g., the case of data races. Considering that a data race arises when there are two unsynchronised accesses to the same memory address and at least one of the accesses is a write access, it might appear to be better to use the sleep noise than the yield noise. This is because when we encounter some access to the memory address of interest, the best we can do is to search the other threads for the second (conflicting) access. This means that as many of the remaining threads should go through as many memory

accesses as possible. Forcing the program to switch threads several times using the yield noise of a commonly used strength will help us to search only a small part of the executions of the other threads. The sleep noise will block the execution of the thread performing the first access giving us considerably more time to detect the second (conflicting) access in one of the remaining threads.[1] However, a problem is that if we use the same sleep noise for all accesses, we will block not only the thread performing the first access, but also many of the remaining threads, which will unnecessarily lower the number of memory accesses they will perform. Hence, what we want to achieve is to lower the amount of noise injected into the remaining threads which we search for the second conflicting access.

The above situation is where the more fine-grained noise injection configuration proposed below can help. In particular, we can use different noise placement heuristics for different, possibly conflicting types of accesses to hold the threads performing one of the types of accesses more than the threads performing the other type of accesses. There are two ways to do that. One possibility is to use the sleep noise only, but with a bigger strength for one of the access types and considerably lower for the other. However, an even better way is to force the threads performing the second type of accesses to give up the CPU (using the yield noise) as this will help more threads to perform more memory accesses. As we will show in the experiments section, this approach really gives us better results than using a single global noise seeding heuristic, and, in addition, it often slows the execution of the program much less.

A question left open in the previous paragraph is which accesses should be hold more and which less. In our opinion, supported by later experiments, this mainly depends on which of the conflicting accesses happens more rarely. Clearly, if one of the conflicting accesses happens more rarely (e.g., a write access if there are few possibilities to write and many possibilities to read), it is better to hold the thread performing the rare access using a sleep noise and search for the more common accesses than doing it conversely.

5.2.4 New Noise Heuristics

In this section, we propose several new noise placement and noise seeding heuristics, including the one based on the fine-grained combinations of noise that was already discussed in the previous section and which we now develop into a concrete noise injection heuristic. While these heuristics are tailored to help with the detection of specific kinds of concurrency errors, they are often useful in other situations as well.

Read/write noise. The *read/write* noise placement heuristic uses different noise settings for the shared variable read accesses and write accesses. The settings might differ in the frequency which controls how often a noise is generated before a particular class of accesses or in the chosen noise seeding heuristic. The heuristic is motivated by the common data race scenarios where there are two unsynchronized accesses to a shared variable and at least one of these accesses is a write access. So, when a memory access is encountered, the best thing one can do is to search the other threads for the second (conflicting) access. In order to lower the noise injected to the other threads, the fact that one of the accesses causing a data race is typically a write access while the other is a read access is exploited. Based on this observation, a stronger noise before one type of memory accesses and a weaker noise before the other is injected.

[1] A similar effect could be reached by using a much stronger yield noise, which would, however, involve actively waiting threads that would in turn unnecessarily slow down the entire system.

Pattern noise. The *pattern* noise placement heuristic injects a noise before accesses to variables which were already accessed before within the same method or function. The motivation here is to create a noise placement heuristic that would help in discovering atomicity violation scenarios. An atomicity violation occurs when two accesses to a shared variable, which should be performed atomically, are interleaved by another access to this variable. The idea is to inject a noise before the second (or any further) access to a shared variable from the same thread within a logical unit (a program method in this case) so other threads have more time to access this variable in between the accesses, causing an atomicity violation. It makes sense to inject such noise even inside a block intended by the programmer to be executed atomically (e.g., a block defined by the `synchronized` keyword in Java) and test whether the synchronization is implemented correctly.

Inverse noise. Some kinds of concurrency errors manifest in situations where a thread executes an action earlier that it should, e.g., sends a notification before someone starts waiting, accesses a variable before it is initialized, etc. The *inverseNoise* noise seeding heuristic does the opposite of the *haltOneThread* heuristic. That is, it stops all but one thread and allows this one thread to get as far as possible in its execution. This increases chances that the thread will trigger an action which it should perform only after some of the blocked threads do something, e.g., start waiting, initialize a variable, etc. Moreover, the other threads are stopped at the nearest instrumentation point which is suitable for noise injection. Therefore, the current thread has the opportunity to execute instructions which trigger an atomicity violation if some of the blocked threads are blocked within an improperly guarded atomic section.

5.3 Evaluating Improved Noise Injection in C/C++

In this section, we present the results of our experiments of utilising noise injection when testing and dynamically analysing C/C++ programs. Both the previously existing and the newly proposed approach discussed in the previous section are validated. We use these results to justify the proposed improvement and also as a basis for a discussion of the possible influence of various noise injection settings on testing and dynamic analysis of multi-threaded C/C++ programs.

5.3.1 Experimental Setup

For our experiments, we used 116 multi-threaded programs implementing a simple ticket algorithm on top of the pthread library. These programs were created by students of an advanced operating systems course. Note that most of them were rated full points as the test script and a brief code review did not find any errors. We were, however, able to find various errors in many of these programs using dynamic analysis in conjunction with noise injection or even just the noise injection alone.

Algorithm 1 describes the general idea behind the ticket algorithm that each of the programs implements. The goal is to synchronise all threads of a program in doing some mutually exclusive work (modelled by calling `doWork()`). When a thread wants to do the work, it is assigned a ticket number and waits for its turn. The `getticket()` function assigns a thread the first free ticket (i.e., a ticket with the lowest ticket number not assigned to any other thread yet). This is done using a shared variable `next_ticket` holding the number of the next free ticket. All accesses to this variable are done in a critical section

Algorithm 1: Ticket algorithm

```
1 foreach thread do
2     while (ticket = getticket()) < M do
3         sleep(random);
4         await(ticket);
5         doWork();
6         advance();
7         sleep(random);
```

guarded by the `tmutex` lock. The accesses to the part where the work is done are then guarded by a monitor entered by calling the `await()` function and left by calling the `advance()` function. These two functions work with a shared variable `curr_ticket` which determines the ticket number needed to enter the monitor. The `await()` function reads the `curr_ticket` variable and forces the thread to wait if it is not its turn (i.e., if it does not have a ticket with the `curr_ticket` number). The `advance()` function then increments the `curr_ticket` variable allowing another thread to enter the part guarded by the monitor. Accesses to `curr_ticket` in each of the functions are done in critical sections guarded by the `mmutex` lock. If the work is done M times, the threads finish their execution.

5.3.2 Detecting Data Races

To look for data races in the considered programs, we used the simple detector called AtomRace [97]. AtomRace tracks which memory addresses are being accessed by the particular threads by monitoring the before and after memory access notifications. If it discovers that two threads can concurrently access the same memory address (at least one of them for writing), which is detected by the appropriate pairs of before and after memory accesses being overlapped, it informs about a data race. As it is normally not very probable to see two concurrent memory accesses, AtomRace uses noise injection to disturb the usual scheduling of threads in order to witness executions in which such memory accesses happen (if that is allowed by the program under test). Clearly, when AtomRace announces a data race, it is a real data race, not a false alarm (of course, provided that it takes into account the possible appearance of atomic instructions).

Using AtomRace implemented on top of the ANaConDA framework, we were able to find data races in 23 of the 116 considered programs, all of them having various kinds of negative impacts on the expected behaviour of the programs. We find this quite satisfactory taking into account that three quarters of these programs were rated full points because they passed all the standard tests.

To further analyse how the noise injection settings influence the overall success of the detector to find data races, we selected 13 of the 23 programs in which we were able to find bugs and performed a large number of tests on them. The remaining programs were not included in the tests as they contained deadlocks in addition to data races, which made them difficult to compare to the rest of the programs. The results are shown in Table 5.1, each column representing one of the tested programs and each row one of the configurations of the noise injection in the following format: First, a base configuration of noise generation used at memory accesses and synchronisation functions is given, consisting of the type of noise, its frequency, and its strength. The `rs` prefix put before the type of noise indicates

Table 5.1: Success ratio of the AtomRace detector for various configurations of the noise injection (the values represent the percentage of runs, out of 500, in which a data race was found)

Noise configuration \ Program	t01	t02	t03	t04	t05	t06	t07	t08	t09	t10	t11	t12	t13
instrumented, no sleep or yield noise	2.4	11.8	0.2	1.2	0.0	1.0	1.6	2.2	0.4	0.0	0.0	0.0	32.2
sleep 500 10	69.2	46.6	100.0	100.0	1.2	53.6	69.4	98.6	0.6	0.2	0.8	0.2	100.0
yield 500 10	3.8	35.4	1.4	10.8	0.0	1.4	5.4	11.6	0.6	1.8	0.0	0.2	84.4
rs-sleep 500 10	**96.4**	**87.8**	97.0	86.2	0.6	31.0	79.0	**99.2**	9.2	10.0	**2.4**	0.2	100.0
rs-yield 500 10	6.0	17.0	0.2	0.6	0.0	0.6	6.2	9.0	0.4	1.6	0.0	0.4	71.2
sleep 100 10	64.0	69.2	80.2	56.0	5.4	40.2	70.4	81.0	31.2	31.4	0.6	**42.4**	98.4
yield 100 10	0.8	17.0	0.4	3.0	0.0	0.0	4.2	6.0	0.6	0.8	0.0	0.0	42.8
rs-sleep 100 10	21.4	55.8	23.0	11.8	0.0	18.2	60.4	71.8	**34.4**	**37.0**	0.8	27.6	92.8
rs-yield 100 10	1.8	9.0	0.6	1.0	0.0	0.4	3.2	3.4	0.6	2.0	0.0	0.0	34.8
sleep 500 20	34.6	48.2	100.0	100.0	0.6	31.8	79.0	97.8	0.0	0.4	0.2	0.6	100.0
yield 500 20	14.2	56.4	4.4	9.4	0.0	2.2	8.6	17.0	1.0	0.8	0.2	0.0	94.0
sleep 500 5	24.2	68.2	100.0	69.2	6.4	26.8	79.8	90.2	1.8	2.2	1.4	0.0	100.0
yield 500 5	2.8	22.2	3.0	13.4	0.0	3.6	5.8	5.2	0.6	1.2	0.0	0.0	63.0
sleep 100 20	59.2	30.4	78.6	6.0	5.6	61.0	67.2	86.4	32.2	33.6	0.8	**42.6**	98.8
yield 100 20	1.2	19.0	0.6	3.0	0.0	1.8	6.4	6.8	1.4	0.8	0.0	0.2	54.2
sleep 100 5	52.4	73.6	78.4	74.4	14.2	18.4	61.0	74.0	29.8	30.6	0.2	38.0	98.2
yield 100 5	1.4	13.0	0.2	1.4	0.0	0.6	3.4	4.2	1.8	1.6	0.0	0.2	38.8
sleep 500 10 / read 20 / write 5	**64.8**	**89.4**	99.0	80.8	0.0	17.0	28.6	91.2	0.4	2.2	0.4	0.0	100.0
sleep 500 10 / read 5 / write 20	33.4	57.2	100.0	91.6	43.0	**92.6**	**96.2**	**99.8**	1.2	0.6	1.4	0.0	100.0
yield 500 10 / read 20 / write 5	3.8	31.4	3.8	9.0	0.0	1.4	3.6	9.2	1.8	1.8	0.4	0.0	86.0
yield 500 10 / read 5 / write 20	5.0	59.6	2.0	8.2	0.0	1.6	9.2	19.8	1.8	0.2	0.0	0.0	88.0
sleep 100 10 / read 20 / write 5	**73.4**	44.4	67.4	2.2	0.0	79.0	51.2	73.0	**34.4**	**37.0**	0.8	0.0	99.2
sleep 100 10 / read 5 / write 20	31.4	27.6	70.8	69.2	4.8	**89.2**	79.6	68.8	30.0	29.6	1.2	0.0	99.2
yield 100 10 / read 20 / write 5	0.8	13.2	0.2	2.2	0.0	0.4	4.6	4.2	1.0	0.8	0.0	0.0	49.0
yield 100 10 / read 5 / write 20	1.2	23.0	0.4	2.6	0.0	1.4	6.6	5.8	0.6	0.8	0.0	0.0	48.8
sleep 500 10 / read sleep / write yield	51.2	61.2	59.2	100.0	0.0	2.4	0.8	83.8	0.2	2.6	0.2	0.0	100.0
sleep 500 10 / read yield / write sleep	18.6	38.4	99.4	100.0	**50.6**	80.8	**95.6**	**97.0**	7.4	4.6	1.2	0.0	100.0
yield 500 10 / read sleep / write yield	32.6	64.6	63.8	100.0	0.0	4.6	0.0	68.6	0.2	3.8	0.2	0.0	100.0
yield 500 10 / read yield / write sleep	10.0	52.2	98.0	100.0	**51.0**	**95.0**	**99.6**	**98.8**	6.0	4.2	**2.0**	0.0	100.0
sleep 100 10 / read sleep / write yield	34.2	**81.0**	44.0	7.4	**62.4**	0.0	2.2	55.0	28.8	**37.6**	0.8	0.0	87.4
sleep 100 10 / read yield / write sleep	9.4	35.6	52.4	96.8	9.6	37.6	78.6	62.0	1.0	2.4	0.0	0.0	88.2
yield 100 10 / read sleep / write yield	25.8	46.4	51.2	6.2	**64.4**	0.2	4.4	69.6	**43.0**	**43.4**	0.2	0.0	85.6
yield 100 10 / read yield / write sleep	16.6	33.6	43.6	94.4	7.2	35.6	80.2	61.2	1.4	1.6	0.2	0.0	90.6

that the strength is not implemented as constant, but as random with the given value being the maximum possible strength. The values of the frequency say how probable it is that some noise will be generated every time the given location is reached on the scale from 0 to 1000, i.e., 500 means 50 %, 100 means 10 % etc. The values of the strength say how many times a yield should be called at the given location of the given thread or how many milliseconds the thread should wait in case of the sleep noise. Then, if applicable, differences from the base configuration in the strength and possibly also type of noise are given behind a slash separately for the read and write accesses. The values in the body of the table then express the percentage of runs (out of 500) in which the data race detector found a data race, i.e., the percentage of executions which actually led to an error. The first noise configuration in the table corresponds to runs where no sleep nor yield noise is generated, but the noise generation code is inserted, together with the code notifying the AtomRace detector about the execution. However, even such instrumentation is already generating some small noise which can help manifestation of errors as we will see in Section 5.3.3.

The selected programs contain various kinds of errors that all lead to data races in the end. In two of the programs (`t01` and `t02`), the data race is on the `next_ticket` variable. In the first program (`t01`), the variable is updated in a critical section, but then read outside of it. Since the `getticket()` function performing these accesses is frequently called from

many of the threads, the data race manifests quite often[2]. In the second program (`t02`), the accesses to the variable are not guarded at all, so the data race manifests even more often. The next program (`t03`) contains a data race on a shared variable used to assign IDs to each of the threads. This variable is updated and read without any synchronisation, however, all of these accesses happen when the threads are started one immediately after another, so the data race may only occur during this short time. Program `t04` uses a shared structure to store the thread IDs and their current tickets. Accesses to all the members of this structure are not synchronised, leading to data races on each of them, multiplying the probability that a data race appears. Program `t05` has a rarely occurring data race on individual items of a shared array where each item may be accessed by the main thread, and one of the other threads simultaneously just before the main thread starts to wait for the second thread to end (join). Programs `t06`, `t07`, and `t08` contain data races on a `timespec` structure, shared among all threads, used to randomly generate the number of milliseconds a thread should sleep before and after entering the monitor. Some of these programs access the structure more often than the others, so the frequency of encountering a data race vary between them. The next two programs (`t09` and `t10`) read the `curr_ticket` variable outside a critical section at one place. All other accesses are, however, performed in the critical section, so it is not very likely that a data race would occur. Program `t11` uses the same lock for guarding the critical section in the `getticket()` function as well as the critical sections in the monitor functions `await()` and `advance()` leading to an extremely rare situation where two threads enter the critical section in the `getticket()` function and access the `next_ticket` variable (because the code does not check if the `tmutex` lock was acquired successfully and just continues). A similar situation happens in program `t12`, which initialises the `tmutex` and `mmutex` locks in each of the created threads, which resets the locks' ownership information, status, and other fields. Changing the ownership information often leads to assertion errors as we will see in Section 5.3.3. On the other hand, resetting the lock status leads to data races since it allows more threads to enter the critical sections guarded by these locks. However, these data races can manifest only if the noise simultaneously prevents all assertion errors which may otherwise show up before the data races. The last program (`t13`) contains a data race on a shared variable used to store the return codes of pthread library's functions. Since this variable is accessed at so many places in the program, the data race occurs very often here.

Evaluation of the Results

As can be seen from Table 5.1, the sleep noise is clearly superior in helping AtomRace in finding data races when the same strength is used. However, using the sleep noise too much can sometimes have quite the contrary effect—hiding the data races instead of helping to find them. Take, for example, programs `t01` and `t02`. They contain a similar error of not guarding the accesses to the `next_ticket` variable, but while `t01` is not guarding only some of the read accesses, `t02` is not guarding any of the accesses, so the possibility of encountering a data race in an execution should be higher here than in case of `t01`. However, when using the sleep noise with frequency 500 (50 %) and strength 10, data races are detected in only 47 % of `t02` runs, while in case of `t01` it was nearly 70 %. After decreasing the frequency to 100 (10 %) or strength to 5, the success ratio of data race

[2]This is the case when the program is run with the instrumentation needed for AtomRace and with the instrumentation for noise generation which is just not generating any sleep nor yield noise. The same holds for the discussion of the other case studies too.

detection increases to nearly 70 %. The problem here is that if we put all the threads to sleep for about the same time (which is the more probable the higher the frequency is), the scheduling of threads will remain the same as without the noise, not introducing the uncommon executions we wanted to witness. In many cases, lowering the frequency helps, likewise using a random strength instead of the fixed one. Sometimes even using a smaller fixed strength might help as we do not block the threads for too much time which increases the chances that there will be some threads which we might search when we start sleeping (there are often none if we use a too strong strength—e.g., when using strength 20 in `t02`, we detect data races in only 30 % runs even if we use frequency 100). Similar problems happen when using noise injection in Java programs [17].

The results also support our opinion from Section 5.2.3 that using different noise injection settings for different locations can give better results. This approach can, e.g., help in solving the problem discussed in the previous paragraph. Indeed, in case of `t02`, instead of using a random strength, it is better to use fixed strengths 20 for reads and 5 for writes, which leads to detection of a data race in nearly 90 % of runs. Moreover, the approach also helps in many other cases, e.g., in case of `t06`, `t07`, and `t08` where using sleep noise with strength 5 for reads and strength 20 for writes gives similar or better results than using strength 10 for all accesses. In some cases where lowering the frequency decreases the probability of detecting a data race, like in `t04`, we can use yields for reads and sleeps for writes with the frequency being 100 only and still keep the success ratio as for frequency 500, which speeds up the execution of the program considerably. Sometimes combining various strengths or different types of noise is the only way to detect some data races in a fair number of testing runs as, e.g., in case of `t05`.

Further, we also verified our assumptions from Section 5.2.3 that blocking the more rare types of accesses should give us better results. For example, `t04` contains data races on members of a structure which are written to several times, but read from frequently. The results are very good when using the sleep noise with a high frequency and strength, but they are considerably worse when the amount of noise is lower. However, using frequency 100 with the sleep noise for the rare writes and yield noise for the common reads gives us nearly the same results as using the strong noise. On the other hand, when we use the opposite combination of noise for the different accesses, the results are very poor. In cases where the read and write accesses to the variable on which a data race is found are equally common, like in case of `t06` and `t07`, it is still better to use the sleep noise for writes and yield noise for reads. This is due to when considering all accesses to all variables in the program, the read accesses are typically more frequent, and so we will not block the remaining threads too much. On the other hand, consider the `t09` and `t10` programs. They contain a data race on a shared variable which is accessed mostly in a critical section except several reads. So all the writes are guarded and there are only several reads that may access the shared variable when it is written to. If we block the threads performing the write, it is highly improbable that we will find the rare unguarded read in some of the remaining threads, but blocking the threads performing the rare reads allows us to find the data race in a fair number of runs.

The statement that blocking the rarer accesses gives us better results seems not to hold in case of `t05` where with frequency 500, using the sleep noise for reads and yield noise for writes is clearly better, but with frequency 100, it is just the opposite. The problem here is that the write accesses are so rare that if we use a small frequency, we will not block the execution of the thread performing the write access to find the conflicting read in the other threads in the meantime. So when using a high frequency, the probability to inject the noise

before some write is relatively high, and it is better to use the sleep noise for writes. On the other hand, the read accesses are performed in a loop, and so there is a large number of them. Causing too much noise before the reads does not help here at all as it often hides the data races, but if we inject the noise before the reads with a low frequency, we will still have a good chance to encounter the rare writes in the other threads.

5.3.3 Detecting Assertion Errors

Apart from increasing chances to see an error or to produce a warning from a dynamic analyser, noise injection can also be used in conjunction with the mechanism of C/C++ assertions. The assertions explicitly guard the execution of a program against situations which, from the perspective of the programmer, should never happen, but if they happen, the programmer should be notified about them. When dealing with the naturally non-deterministic execution of multi-threaded programs, it is hard to say, even for a skilled programmer, whether some situation is really excluded from happening, and so the number of assertion checks tends to be higher in multi-threaded C/C++ programs. The noise injection can then be conveniently used to increase the chances to see the executions which violate the assertions present in the code alerting the programmer about situations he/she did not expect to occur.

We have also tested whether the noise injection can help us in detecting wrong usages of the pthread library such as cases when a thread releases a lock which it does not own and the like. Such scenarios are detected directly by assertions built into the pthread library. We used the same set of 116 programs as before and checked their output for assertion errors originating from the pthread library. Among the 116 programs, we found 3 that break the built-in assertions.

Again, we studied how the noise injection settings influence the overall success of detecting the wrong usages of the pthread library. The results are shown in Table 5.2, each column representing one of the tested programs and each row one of the configurations of the noise injection in the same format as in Table 5.1. The values in the body of the table then express the percentage of runs (out of 500) which ended with an assertion error.

The first two programs (`t02` and `t12`) are both initialising the `tmutex` lock in each of the created threads, resetting the lock's ownership information and status when a new thread is started. This allows more than one thread to acquire the lock as one thread may acquire the lock, and then another thread may start, reset the status of the lock to *not acquired* and afterwards acquire the lock itself, hence becoming the owner of the lock. If the program now switches to the first thread, and this thread will release the lock, the pthread library will raise an assertion error saying that some thread is trying to release a lock which it does not own. This also leads to data races as more than one thread may access the critical section guarding the `next_ticket` variable and access it. The third program (`t14`) releases the `mmutex` lock twice in the `advance()` function, instead of acquiring it and then releasing it. Due to this, while some thread is inside the critical section of `advance()`, another thread can acquire the not locked `mmutex`, and then the former thread may release it despite it never acquired it, which causes the same assertion error as in case of the first two programs. Note, however, that this situation is quite rare as the threads may usually acquire the lock only a moment before the problematic release is done.

As can be seen from Table 5.2, running the instrumented program without any noise gives us already good results when trying to detect a wrong usage of the pthread library. However, this is because we are in fact injecting a very weak noise into the execution as

Table 5.2: Success ratio of finding assertion errors for various configurations of the noise injection (the values represent the percentage of runs which ended with an assertion error)

Noise configuration \ Program	t02	t12	t14
normal run	0.0	0.0	0.0
instrumented, no sleep or yield noise	48.0	50.8	8.0
sleep 500 10	0.0	0.0	1.2
yield 500 10	**62.4**	**51.0**	**8.8**
rs-sleep 500 10	3.2	1.0	3.8
rs-yield 500 10	41.2	48.8	8.0
sleep 100 10	2.0	27.8	7.2
yield 100 10	49.6	**51.2**	6.6
rs-sleep 100 10	16.4	32.8	6.8
rs-yield 100 10	44.2	**56.2**	**8.8**
sleep 500 20	0.0	0.0	2.6
yield 500 20	**64.6**	**55.2**	6.6
sleep 500 5	0.0	0.0	3.2
yield 500 5	58.6	48.4	8.2
sleep 100 20	4.2	26.4	2.0
yield 100 20	56.6	47.4	7.4
sleep 100 5	21.2	25.6	4.6
yield 100 5	51.0	49.4	8.0
sleep 500 10 / read 20 / write 5	0.0	0.0	5.2
sleep 500 10 / read 5 / write 20	0.0	0.0	6.0
yield 500 10 / read 20 / write 5	**62.4**	0.0	7.6
yield 500 10 / read 5 / write 20	**64.0**	0.0	**10.4**
sleep 100 10 / read 20 / write 5	7.4	0.0	5.4
sleep 100 10 / read 5 / write 20	9.2	0.0	4.6
yield 100 10 / read 20 / write 5	49.6	0.0	6.2
yield 100 10 / read 5 / write 20	54.6	0.0	7.0
sleep 500 10 / read sleep / write yield	2.2	0.0	3.4
sleep 500 10 / read yield / write sleep	0.0	0.0	3.0
yield 500 10 / read sleep / write yield	0.2	0.0	2.8
yield 500 10 / read yield / write sleep	0.0	0.0	1.6
sleep 100 10 / read sleep / write yield	50.2	0.0	6.4
sleep 100 10 / read yield / write sleep	22.4	0.0	4.4
yield 100 10 / read sleep / write yield	60.6	0.0	**9.4**
yield 100 10 / read yield / write sleep	47.4	0.0	3.4

we let the framework execute the noise injection code (although it inserts no noise) which by itself disturbs the scheduling of the threads. In runs with no instrumentation (even when the program is running within the PIN framework), the success ratio is practically zero. The yield noise may sometimes help us to achieve slightly better results while the sleep noise is mostly rather hiding the errors. In case of `t14`, the success of encountering an assertion error is very dependent on when and where the noise is injected and not so much on the noise settings used, since the wrong usage manifests only in a very specific situations. So if we manage to inject the noise in the right place and at the right time, even weak noise will help.

5.3.4 Testing C/C++ vs. Java Programs

We also tried to compare how much the various types of noise help in case of C/C++ programs compared to Java programs. Since implementing the same program in two different programming languages in a way that the implementation is as close as possible is not an easy task, we have chosen a simple `bank` program for the tests, which is one of the typical and often used case studies [91]. This program contains a data race on a shared

Table 5.3: Success ratio of the AtomRace detector for various configurations of the noise injection (the values represent the percentage of runs in which a data race was found)

Noise Type	rs-sleep					
Frequency	500			100		
Strength	20	10	5	20	10	5
C++	98.8	99.0	100.0	91.6	91.8	91.2
Java	100.0	100.0	99.6	99.4	99.8	99.0

Noise Type	rs-yield					
Frequency	500			100		
Strength	20	10	5	20	10	5
C++	67.6	63.8	58.8	52.0	41.8	52.4
Java	36.2	29.0	24.2	22.4	21.8	28.6

variable accessed usually in a critical section, but sometimes also outside of it.

To test the Java version of the program, we used ConTest [44] together with a Java implementation of AtomRace [97]. Since the ConTest tool uses random strengths, we used the `rs-sleep` and `rs-yield` noise in the C++ version of the program to be able to compare them to the corresponding ConTest's types of noise. The results are shown in Table 5.3. The values express the percentage of runs (out of 500) in which the data race detector found a data race, i.e., the percentage of executions which actually lead to an error.

We can see that while the sleep noise is a little more helpful in case of the Java version of the program compared to the C/C++ version, giving the nearly 100% success ratio even for lower frequencies, the yield noise is clearly better in case of the C++ version, helping to find a data race in twice as many runs as in the Java version. A question that remains is whether the differences are caused by the programming language itself or whether they depend on the concrete thread management and synchronisation library used.

5.4 Comparison of Noise Injection Techniques

While the main focus of the previous section was to evaluate the improved noise injection, which allows one to combine different noise placement and noise seeding heuristics, and compare it with the existing approaches to noise injection, which are restricted to a single pair of noise placement and noise seeding heuristics, this section focuses more on the noise injection heuristics themselves. In particular, this section evaluates the newly proposed heuristics and compare them with the best currently known heuristics.

This section first presents a selection of results of previously published comparisons [54, 96, 99] of the older noise injection heuristics. Based on the results, the most promising heuristics and their parameters are pinpointed and used in a new comparison of the old and newly proposed noise injection heuristics on a set of C and Java benchmarks. The used set of benchmarks is the so-far biggest set of benchmarks used for evaluating noise-injection-based testing. The mentioned selection of the most promising heuristics that we have been done allowed us to test the selected promising noise configurations much more thoroughly (which would not be possible with all of the possible noise configurations due to the high time requirements of the experiments). The obtained results are discussed separately for C/C++ and Java because of the different noise injection infrastructures used for testing programs written in these languages—in particular, for C/C++ code, instrumentation on the binary level was used, whereas for Java, instrumentation on the bytecode level was used.

The differences between the noise-based testing of C/C++ programs on the binary level and of Java programs on the bytecode level are presented next. Despite the differences, the commonalities and dissimilarities of the obtained results are discussed. Finally, some hints on how to effectively use noise-based testing are presented.

5.4.1 Results of Previously Performed Comparisons

In this section, the most important aspects of previously published comparisons of noise injection heuristics [54, 96, 99] are highlighted. These results were used to set up an environment for the comparisons presented in this article. In particular, the results lead to a choice among the many possible configurations of noise placement and noise seeding heuristics—those which provide good results in the comparisons presented in this section.

Comparison of Existing Heuristics

An extensive and systematic comparison of results of various existing noise placement and noise seeding heuristics including the coverage-based noise placement heuristics and the related noise seeding heuristics introduced above for Java has been published in [96]. The heuristics were compared according to their efficiency to improve detection of concurrency errors, to improve the concurrency-related coverage metrics HBPair* and Avio* in the considered test cases, and to affect the execution time of the considered test cases. The HBPair* and Avio* metrics described in [52] have been chosen due to their very good ratio of providing satisfactory results from the point of view of suitability for saturation-based or search-based testing and a relatively low overhead of measuring the achieved coverage (and hence their suitability for performing many tests with an acceptable interference with the tested programs). The SearchBestie platform [93] was used to set up and execute the needed tests with IBM ConTest [42]. The heuristics were evaluated on a set of 4 test cases (namely, Airlines, Crawler, FTPServer, and TIDOorbJ test cases) described in [52].

First, there was done a comparison of several noise seeding heuristics denoted as basic below (namely, the *yield*, *synchYield*, *wait*, *busyWait*, and *sleep*) and the IBM ConTest *mixed* noise seeding heuristic which randomly chooses one of the basic noise seeding heuristics at each call of the noise injection routine. Then, the improvement which can be achieved by combining basic noise seeding heuristics with the *haltOneThread* and *timeoutTampering* heuristics was studied. All heuristics were used with the *random-all* noise placement heuristic enabled.

The results indicate that there is no optimal configuration, i.e., for each test case and each testing goal (improvement of coverage, error manifestation, or overhead minimization), one needs to choose different noise seeding heuristic [96]. Moreover, in some cases, the noise injection heuristics improved the obtained results considerably while, in some other cases, the noise seeding configurations used with the *random-all* noise placement heuristic actually provided considerably worse—demonstrating the ability of noise injection techniques to mask concurrency errors [90]. The *timeoutTamper* heuristic provided a considerable improvement for the *crawler* test case. As already said, this test case is a skeleton of an IBM software product. When developers extracted the skeleton, they modeled its environment using timed routines. The *timeoutTamper* heuristic influences these timeouts in a way leading to a significantly better results.

Next, a comparison of different noise placement heuristics has been published by Letko [96] as well. Mainly, the *random-all*, *sharedVar*, and *coverage* heuristics were considered. Additionally, a heuristic which randomly sets up noise settings before each test execution was

considered in the comparison too. The noise placement heuristics were again compared according to the ability to detect concurrency errors and to provide a high coverage. Then, a comparison of the heuristics using relative results was provided as well. In this comparison, the total number of covered tasks or detected errors was divided by the execution time (in seconds) the heuristics needed to achieve the results.

Again, none of the heuristics achieved best results in the comparisons for all the considered test cases. Overall good results were obtained by different versions of the *sharedVar* heuristic which focuses noise to shared variables only. There was no winner among the two versions of the heuristic: *sharedVar-all* which targets all accesses to shared variables and *sharedVar-one* which targets accesses to a single randomly chosen shared variable in each test execution. The heuristic using random settings for each test execution achieved on average good results too. This was because accumulated results from multiple runs (namely, 20 and 50 times) were used for the comparison—some of the randomly chosen settings therefore provided very good results regardless of the test case which turned in the overall results. The *coverage* heuristic achieved good results in some cases as well.

Finally, the best relative improvement achieved by noise-based testing in the considered test cases was presented by Letko [96]. Table 5.4 shows the results obtained when evaluating the best relative improvement (denoted as *Impr.*) in the experiments for the considered metrics and test cases. The improvement is computed as a relative improvement compared to the configuration without noise injection (note that collection of coverage information and the instrumentation itself already introduce a certain amount of noise). The next three columns (denoted as *nFreq*, *Seeding heur.*, and *Placement heur.*) present the noise frequency, noise seeding heuristic, and noise placement heuristic used. Combinations of the basic noise seeding heuristics with the *timeoutTampering* heuristic (denoted as *tt*) and *haltOneThread* heuristic (denoted as *ht*) were also allowed and evaluated.

Table 5.4: The best relative improvement achieved by noise heuristics

Test	Metric	Impr.	nFreq.	Seeding heur.	Placement heur.
Airlines	Error	5.93	150	yield + tt	sharedVar-one
	Avio*	1.99	–	–	no noise
	HBPair*	1.90	–	–	no noise
Crawler	Error	★	–	busyWait	coverage
	Avio*	8.20	50	mixed + tt + ht	sharedVar-all
	HBPair*	3.55	200	mixed + tt + ht	sharedVar-all
FTPServer	Error	1.09	50	sleep	sharedVar-one
	Avio*	1.26	50	wait + tt + ht	sharedVar-all
	HBPair*	1.55	150	busyWait + ht	sharedVar-all
TIDOrbJ	Error	–			
	Avio*	1.12	200	busyWait + tt + ht	sharedVar-one
	HBPair*	1.23	200	busyWait + tt + ht	sharedVar-one

The improvement of the error manifestation ratio (denoted as *Error*) in the TIDOorbJ test case is not present because the version of the test case we used contains no error. The ★ symbol in the error manifestation ratio of the Crawler test case means that the improvement cannot be computed because in the experiments, the error does not manifest when the noise was disabled. The best value, which was achieved by the *coverage* heuristic,

Table 5.5: Success ratio of the AtomRace detector for various configurations of the noise injection (the values represent the percentage of runs, out of 500, in which a data race was found)

Noise injection configuration					Test case		
ConfID	Placement heur.	Seeding heur.	Freq.	Strength	t05	t06	t07
instrumented, no sleep or yield noise					0.0	1.0	1.6
1	random-all	sleep	500	10	1.2	53.6	69.4
2	random-all	sleep	500	0–10	0.6	31.0	79.0
3	read/write	sleep / sleep / sleep	500	10 / 5 / 20	**43.0**	**92.6**	**96.2**
4	read/write	yield / yield / sleep	500	10 / 10 / 10	**51.0**	**95.0**	**99.6**

reached 2 % of error manifestation in this test case (on average 1 error manifestation per 50 executions).

In some cases (e.g., in the Airlines test case), the improvement of the error detection is high, reaching several hundreds percents. The lowest improvement was achieved in the FTPServer test case. This is mainly because the error manifestation ratio is quite high even without the noise injection and by the fact that any performance degradation in effect makes the code containing the error execute less often. Overall, the table presents the positive effect of relatively cheap and easy to use noise injection technique in the process of testing concurrent programs. Again, one cannot claim a clear winner among the noise placement and noise seeding heuristics. However, the *sharedVar* noise placement heuristic achieved very good overall results in this evaluation.

Comparison of Existing and Improved Noise Injection

Next, a comparison of the *read/write* noise placement heuristic with the *random-all* heuristic on a set of 14 C programs implementing a simple ticket algorithm using the pthreads library is presented in Section 5.3. These programs were created by students of an advanced operating systems course and all contain data races. They are referred as test cases `t01` to `t14`. The ANaConDA framework [55] was used to perform the tests. The framework uses the Intel PIN framework [111] for dynamic binary instrumentation to insert the code implementing the noise injection heuristics into a C/C++ program binary. As the framework cannot provide concurrent coverage information yet, an evaluation of the successfully detected data races in each test run was performed. For the detection of data races, a C++ implementation of the AtomRace dynamic detector [98] was used.

Results obtained for some selected noise injection configurations and test cases are shown in Tables 5.5 and 5.6. Each configuration is defined by a noise placement and noise seeding heuristics together with the values of frequency and strength used (denoted as *Placement heur.*, *Seeding heur.*, *Freq.*, and *Strength*, respectively). If the *read/write* noise placement heuristic is used, the *Seeding heur.* and *Strength* columns then contain 3 values. These are the values used for the synchronization operations, read accesses and write accesses, respectively. In case of the *Seeding heur.* column, the values represent the noise seeding heuristic used, and in case of the *Strength* column, the value of strength used. If the value of strength is an interval, the particular value was taken randomly from the interval each time the noise was injected.

The *read/write* noise placement heuristic allows to use different noise seeding heuristics and their parameters for different types of memory accesses. Of course, there are many pos-

Table 5.6: Success ratio of the AtomRace detector for various configurations of the noise injection (the values represent the percentage of runs, out of 500, in which a data race was found)

Noise injection configuration					Test case	
ConfID	Placement heur.	Seeding heur.	Freq.	Strength	t04	t05
instrumented, no sleep or yield noise					1.2	0.0
5	read/write	sleep / sleep / yield	100	10 / 10 / 10	7.4	**62.4**
6	read/write	sleep / yield / sleep	100	10 / 10 / 10	**96.8**	9.6
7	read/write	yield / sleep / yield	100	10 / 10 / 10	6.2	**64.4**
8	read/write	yield / yield / sleep	100	10 / 10 / 10	**94.4**	7.2

sibilities how to combine them, so the two most promising combinations were focused. First, the same noise seeding heuristics have been used, but parametrized them with different values of strength, i.e., a bigger strength for one type of memory accesses and a considerably lower for the second one was applied. The goal was to lower the amount of noise injected to the threads that are intended to be search through when detecting data races. As the results in Table 5.5 show, such configurations (Configuration no. 3) achieved better results than the configurations using the *random-all* heuristic (Configurations no. 1 and 2).

Configurations which use different noise seeding heuristics for different memory accesses were also used. More precisely, the *sleep* heuristic for one type of memory accesses and the *yield* heuristic for the second one were studied. Their values of strength were left the same. The goal was not only to lower the amount of noise injected to the threads to be searched through, but also to allow the threads to perform as many memory accesses as possible. While the sleep noise is blocking the thread performing the first access, the yield noise is forcing the program to quickly switch threads so the threads will be running more often and hence perform more memory accesses. As the results in Table 5.5 show, such configurations (Configuration no. 4) achieved even better results than the ones combining different values of strength (Configuration no. 3).

The tests also proved that it is important to choose the right type of memory accesses before which the stronger noise is injected. When there are only a few unprotected write accesses which might cause a data race, the stronger noise should be put before these accesses. This is because it is far more probable that one will encounter the more common read accesses in the other threads which are being searched than the rare write accesses. If the situation is opposite, the stronger noise should be put before the read accesses. Table 5.6 shows the difference in results for two programs which mainly differ in how a data race might manifest. As the `t04` test case contains only a few unprotected write accesses which might cause a data race and many unprotected read accesses, the configurations injecting a stronger noise before the write accesses (Configurations no. 6 and 8) give far superior results than configurations injecting a stronger noise before the read accesses (Configurations no. 5 and 7). In case of the `t05` test case which contains only a few unprotected read accesses and many unprotected write accesses, the results are completely opposite.

5.4.2 A Comparison of Noise Injection Techniques in C/C++

In this section, new experiments that were performed with C programs and noise injection heuristics selected according to the experience from older experiments described in the

previous section are presented (new experiments with Java programs will be described in the following section). First, a description of the testing environment and experiment settings which are used to compare selected noise injection heuristics, including the newly proposed heuristics, is given. Then, the obtained results of these heuristics on four C programs are provided.

Testing Environment

The ANaConDA framework [55] already mentioned in Section 5.4.1 was used to perform the tests. For each execution of a test, the framework collects information about test duration and about the fact whether an error has manifested. In contrast to the previous comparisons where each noise configuration was given an equal number of test executions, in this comparison, each considered configuration of noise heuristics was given 20 minutes of real time to test the program and average results were computed. Therefore, the configurations with higher impact on the performance were provided with lower number of executions of the test. This allows to demonstrate efficiency of the heuristics in practical testing scenarios where the time and other resources for testing are usually limited.

Heuristics. As there are many possible combinations of various noise placement and noise seeding heuristics and as each of these heuristics might be parametrised in many different ways, there exists a large number of configurations that might be used. In order to keep the number of considered configurations on a reasonable level, the focus is devoted to heuristics and their parameters which provided good results in the previous comparisons and also on the new heuristics introduced in the previous sections.

In case of the noise placement heuristics, the following ones are considered: the *random-all* heuristic which is used as a base-line, the *sharedVar-all* and *sharedVar-one* heuristics which provided good results in the evaluation of noise placement heuristics for testing Java programs, the *read/write* heuristic which turned out to be efficient in the previous experiments with noise injection in C/C++, and the newly proposed *pattern* heuristic. All these heuristics decide whether to inject a noise based on the *frequency* parameter which controls how often the noise is injected at the selected place. The frequency parameter was set such that the noise was generated either in 15 % or 30 % of situations. These values were also inspired by the results of the previous comparisons.

As for the noise seeding heuristics, the *sleep*, *yield*, and *busyWait* heuristics were considered because they provided good results in some cases in the previous comparisons. Moreover, the newly proposed *inverseNoise* heuristic was added. The noise seeding heuristics are parametrised by the *strength* parameter. This parameter was set to 2 and 20 milliseconds in the case of *sleep* and *busyWait* heuristics and to 10 and 100 executions of the `yield()` function in the case of the *yield* heuristic. In the case of the *read/write* heuristic, the strength parameter for writes and reads was set in the mutually complementary way. That is, if a higher value for writes (e.g., 20 ms) was used, the lower value for reads (i.e., 2 ms) was applied, and vice versa. As for the newly proposed *inverseNoise*, the parameter was set to 2 and 20 operations executed by the current thread while other threads are blocked. The higher values were chosen based on the results of the previous comparisons where a stronger noise often helped more than a weaker one. The lower values were used primarily because of the *read/write* heuristic, where combining strong and a much weaker noise led to the best results. Also, as the *yield* heuristic disturbs the usual scheduling of threads far less than the other noise seeding heuristics, higher values of strength were used for it. In case of

the *read/write* noise placement heuristic, configurations combining the *sleep* and *yield* noise seeding heuristics with fixed values of strength were also used (10 for the *sleep* heuristic and 50 for the *yield* heuristic).

The combinations of heuristics described above give 81 noise configurations (5×2 noise placement heuristics, 4×2 noise seeding heuristics, and 1 configuration without noise—referred to as *nonoise* below). Note that the previous comparisons did not contain the *sharedVar-all*, *sharedVar-one* and the newly proposed *pattern* noise placement heuristics. Also, the *busyWait* and the newly proposed *inverseNoise* noise seeding heuristics were not considered. They are thus examined for C/C++ programs for the first time.

Test cases. For the experiments, 4 simple C programs (about 200 to 500 lines of code) implementing a simple ticket algorithm using the pthreads library were used. These programs were chosen from a set of programs which were already mentioned in Section 5.4.1 and previously used in experiments presented in Section 5.3. The chosen programs are referred to as test cases `t01`, `t03`, `t05` and `t06`. The main reason to use only a subset of programs was that some of the newly tested noise placement heuristics need information about the variables which are accessed. In case of C/C++ programs, these information need to be extracted from the debugging information of the program. However, the ANaConDA framework has only a partial support for extracting this kind of information, and for many of these programs the compiler generated debugging information which the framework was not able to process. So in order to test these new heuristics, availability of this information is required. As the framework imposes a huge slowdown on the execution of the tested program, bigger programs were not considered for the tests because one would be able to perform only a few testing runs in the given 20 minute time slot. All the programs were executed on an 4-core Intel Xeon X5355 2.66GHz machine with the Hyper-threading support (up to 8 threads might run simultaneously) and 64GB memory running Linux with the 2.6.32 kernel.

The selected programs contain various kinds of errors that all lead to data races in the end. In `t01`, the data race is on a shared variable holding the number of a ticket allowed to enter a critical section. The variable is updated in a critical section, but then read outside of it. The next program (`t03`) contains a data race on a shared variable used to assign IDs to each of the threads. This variable is updated and read without any synchronization, however, all of these accesses happen when the threads are started one immediately after another, so the data race may only occur during this short time. Program `t05` has a rarely occurring data race on individual items of a shared array where each item may be accessed by the main thread and one of the other threads simultaneously just before the main thread starts to wait for the second thread to end (join). Program `t06` contains a data race on a `timespec` structure, shared among all threads, used to randomly generate the number of milliseconds a thread should sleep before and after entering the monitor.

Experimental Results

In this section, a comparison of the efficiency of detecting concurrency-related errors using various noise injection configurations is described first. Then, focus is devoted to the results obtained by the newly proposed heuristics.

Since all of the test cases contain a data race and the consequences of these data races are not always externally visible, a dynamic analysis using the AtomRace dynamic detector [98] was performed in order to find these errors. Like the noise injection, the dynamic

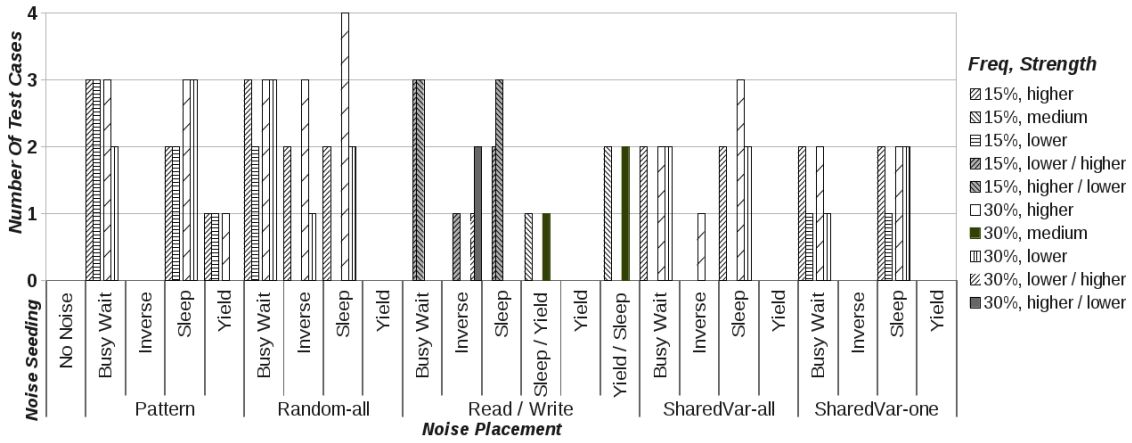


Figure 5.1: A comparison of configurations across all of the considered C test cases

analysis requires the program to be instrumented, so it is problematic to compare the results obtained with and without dynamic analysis. However, note that the tests that were originally used to evaluate the considered student projects from which the test cases are derived did not found any errors. The instrumentation of a program usually increases the probability of finding an error, even when no noise is injected, as the execution of the instrumented code itself causes a sort of a very weak noise which might help a little with the error detection. So, even with the *nonoise* configuration, it was possible to detect some errors in the 20 minute time slot in most of the test cases (namely, `t01`, `t03`, and `t06`).

To compare the efficiency of each configuration, their general success across all of the test cases executed was measured. The results are summarized in Figure 5.1. The x-axis shows the noise configurations grouped by the noise placement and noise seeding heuristics with the values of noise frequency and strength represented by the different hatch of the bars. The y-axis then shows the number of test cases (out of 4) for which the respective configuration was among the best 30 % of the configurations (i.e., among the best 24 configurations in the case). Here, the best configurations were chosen according to the percentage of runs in which a data race was detected. The other noise configurations were, in fact, capable of detecting an error in most of the test cases too, but in less test runs.

The graph shows that even when the test cases are very similar and contain the same type of concurrency errors, most of the configurations work only for some of the test cases. Of course, one can see that some of the configurations were more successful than the others. In general, configurations using the *sleep* and *busyWait* heuristics were the most successful ones. The most successful approach was to combine these heuristics with the *random-all*, *read/write*, or *pattern* heuristics.

A further analysis of the results has also shown that choosing the right combination of noise placement and noise seeding is important, but tweaking the values of noise frequency and strength may also significantly influence the results. Many configurations provided very different results when the values of frequency or strength were changed.

As for the newly proposed heuristics, configurations using the *pattern* heuristic proved to be very useful in most of the test cases (namely, the `t01`, `t03`, and `t06` test cases). On the other hand, the *inverseNoise* heuristic helped only a little and only when combined with the

random-all heuristic. As for the heuristics tested for the first time in C programs, namely the *sharedVar-all* and *sharedVar-one* heuristics, these heuristics achieved good results for some test cases, but they were not so good overall compared to the other noise placement heuristics.

5.4.3 A Comparison of Noise Injection Techniques in Java

In this section, new experiments with noise injection techniques in Java are presented. Similarly to the the results for C presented above, description of the testing environment and test cases is given first. Subsequently, a summary of the obtained results is presented.

Testing Environment

The code instrumentation and noise injection was done using the IBM ConTest framework [42] executed with plug-ins implementing the noise heuristics and collecting selected coverage information. Automatic test instrumentation, execution, and evaluation was orchestrated by the SearchBestie framework [93]. In contrast to the experiments with noise injection to C programs, the infrastructure collected not only information related to execution time and error manifestation but also coverage under two selected coverage metrics, namely, HBPair* and Avio*, which were already used in the previous comparisons mentioned above and which were chosen due to their very good ratio of providing satisfactory results in the experiments with saturation-based testing described in [52] and a relatively low overhead of measuring the achieved coverage.

Heuristics. In the comparison below, all the configurations previously described in the comparison of noise heuristics for testing C/C++ programs were used, together with several more configurations based on the coverage-based heuristic [99] introduced in Section 5.2.1. Hence, the following noise placement heuristics are considered: *random-all*, *sharedVar-all*, *sharedVar-one*, *pattern*, *read/write*, and the coverage-based *coverage* heuristic which is exclusive to the comparison of Java programs. The reason why the *coverage* heuristic is studied only for the Java programs is the fact that the ANaConDA framework, used in the experiments with C programs, is not currently able to provide any coverage information and thus cannot support any coverage-based heuristics. All the heuristics were parametrized by the frequency parameter set to 15 % or 30 %. Note that this is the first time the *read/write* and the newly proposed *pattern* heuristic are evaluated on Java test cases.

As for the noise seeding heuristics, the same heuristics as in the C comparison above are considered, namely, *yield*, *sleep*, *busyWait*, and the newly proposed *inverNoise* heuristics. Again, two levels of noise strength for each of the heuristics were used: 2 and 20 milliseconds for the *sleep* and *busyWait* heuristics, 2 and 20 instructions for the *inverNoise* heuristic, and 10 and 100 executions of `yield()` for the *yield* heuristic. Finally, experiments with the configuration which injects no noise into the execution but which instruments the code and collects coverage information were evaluated as well (referred to as *nonoise* below). Recall, that execution of any injected code in fact influences performance and scheduling of threads.

The above described combinations of heuristics give 97 noise configurations (6×2 noise placement heuristics, 4×2 noise seeding heuristics, and 1 *nonoise* configuration). Similarly to the comparison for C programs, each configuration was given a 20 minutes time slot to test the considered program.

Test cases. The above described configurations of noise injection techniques were evaluated on 8 Java test cases based on 6 Java programs of various size. The *Airlines*, *Crawler*, and *FtpServer* test cases are described in [52]. The *Animator* test case is based on a simple graphic application for algorithm animation called *XtangoAnimator*. The test case creates a window and draws a picture according to a given batch file. The test case consists of 31 classes and contains a data race that leads to `NullPointerException`.

The *Rover* test case is a Java version of the NASA Ames K9 Rover Executive [64]. The test case, consisting of 83 classes, executes a selected high-level plan or plans—programs written in a language that specifies actions and constraints on the movement, experimental apparatus, and other resources of the rover. The test case contains a deadlock and a data race in the testing environment during exchanging of two consecutive high-level plans. Both errors make the test hang. Similarly to the *Crawler* test case, the probability of spotting the errors is extremely low without the use of a noise injection.

The *Elevator* test case is a simple real-time discrete event simulator [153] which contains atomicity violation leading to `NullPointerException`. Elevators are modeled as individual threads that poll directives from a central control board. The communication is synchronized using locks. The used configuration simulates 4 elevators.

Moreover, to demonstrate that the testing environment also plays an important role in the testing process, two prominent test cases in which the probability of spotting the error is extremely low (namely, Rover and Crawler) were executed on two different hardware configurations (the results are then referred to as *Crawler2* and *Rover2*). The Airlines, Animator, Crawler, and Rover test cases were executed on Intel i5-2500 machines with 2GB memory running Linux with the 2.6.32 kernel and 64bit Sun (Oracle) JVM version 1.6. The Crawler2, Elevator, FtpServer, and Rover2 test cases were executed on Intel i7-3770K machines with 4GB memory running Linux with the 3.2.0 kernel and 64bit OpenJDK JVM version 1.6.

Experimental Results

In this section, results comparing efficiency of the considered noise configurations from the most important point of view, namely their efficiency in error detection, are presented. Then, a short discussion of the results these heuristics achieved in terms of coverage is provided. Next, results achieved by the newly proposed heuristics are highlighted. And finally, the influence of the testing environment is discussed.

In a vast majority of the test cases, the error does not manifest during the 20 minutes long testing of non-instrumented code. Instrumentation of the test cases usually increases the probability to spot an error a bit because the instrumented code is executed in locations suitable for noise injection. In particular, the *nonoise* configuration was able to detect an error within the given time slot in two test cases, namely, the Airlines (the error manifested in 8 % of runs) and FTPServer (the error manifested in 66 % of runs).

The success of noise-based testing in detection of concurrency errors is summarized in Figure 5.2. The figure shows configurations grouped by the noise placement and noise seeding heuristics on the x-axis (the noise frequency and strength are represented by the different hatch of the bars). The y-axis shows the number of test cases (out of 8) in which the particular configuration was able to detect concurrency errors within the given time. In most cases, there were only a few configurations which were able to detect errors in the given time (ranging from 2 in the Elevator test case to 9 in the Crawler test case). In the Airlines and FTPServer test cases where the probability of spotting an error is much higher

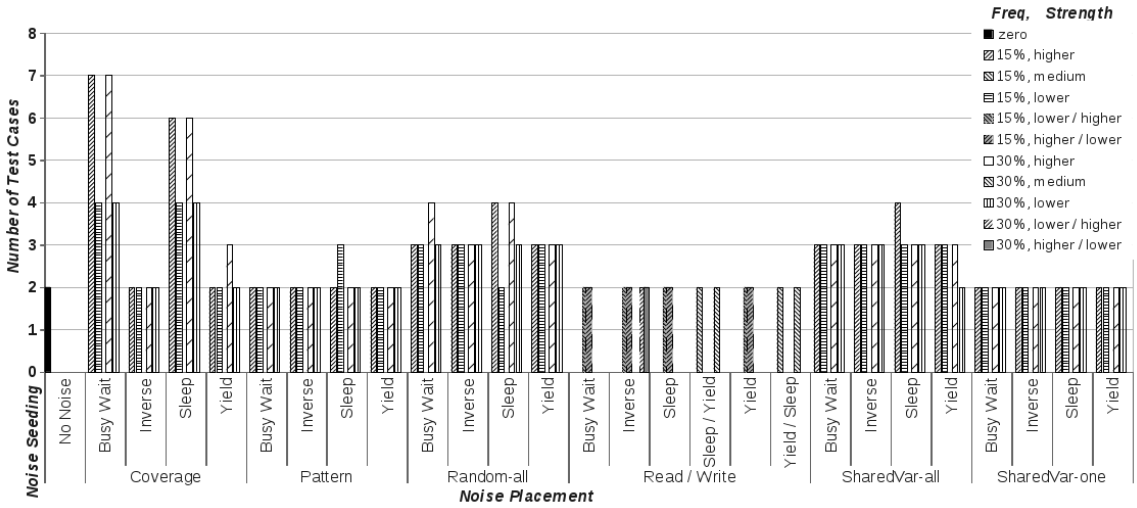


Figure 5.2: A comparison of noise configurations across all of the Java test cases

than in the other test cases, all of the noise configurations were able to detect the errors.

The figure shows that there is no silver bullet among the considered heuristics. Indeed, none of the columns reached value 8 which would mean that the heuristics worked for all the considered test cases. Moreover, one can clearly see that some of the configurations were mostly successful (mainly the configurations combining the *coverage* heuristic with the *sleep* and *busyWait* heuristics) and some were successful only in the easy Airlines and FTPServer test cases (for instance, the *pattern* noise placement heuristic combined with most of the noise placement heuristics).

A further analysis of the results also shows that in most of the cases choosing the right combination of noise placement and noise seeding heuristics was far more important then tweaking the noise frequency and noise strength parameters. Many configurations provided similar results when any value of strength and frequency was used.

Overall, the results focused on the detection of concurrency errors show that noise-based testing is able to dramatically increase the probability of finding concurrency errors. It is enough to use any combination of noise injection heuristics in order to detect errors that do manifest during normal test executions even through only rarely (as can be seen from the Airlines and FTPServer test cases). Moreover, in the case of truly rarely manifesting concurrency errors which are hard to spot even during the noise-based testing, a careful choice of the combination of noise placement and noise seeding heuristics and their parameters is necessary.

As for the coverage obtained under considered coverage metrics, the results clearly show a positive impact of noise-based testing in comparison with the *nonoise* configuration. In some cases, a high achieved coverage correlated with a success in error detection (for instance, in the Elevator test case), sometimes this correlation could be identified only between the error detection ability and one of the coverage metrics (e.g., Avio* metric in the Airlines test case and HBPair* in the Rover2 test case), and sometimes there was no correlation between the error detection ability and any of the considered metrics (for instance, in the Animator and FTPServer test cases). Therefore, one cannot claim that

there is in general a correlation between the ability to detect errors and to achieve a high Avio* or HBPair* coverage. However, a further analysis of the results indicates that it might be the case that if the error depends on a behavior reflected by the coverage metrics, the configurations which achieve a high concurrency coverage are able to detect the error (for instance, Avio* and the atomicity violation in the Airlines test case and HBPair* and the deadlock error in the Rover test case).

Next, we discuss efficiency of the newly proposed heuristics (namely, the *pattern* and *inverseNoise*) and the *read/write* heuristic. The newly proposed heuristics did not help much in detection of concurrency errors which was a bit surprising because the preliminary results obtained on the Rover test case (when the noise with frequency of 5 % only was injected and coverage data were not collected) the combination of the newly proposed heuristics achieved the highest error detection improvement[3]. Nevertheless, the heuristics achieved good results in obtaining a high coverage in some cases (e.g., in the Airlines test case). On the contrary, the *read/write* heuristic achieved very good results in improving the ability to detect concurrency errors in the Airlines and FTPServer test cases. Errors in these test cases were found by all the considered noise configurations, but noise configurations with the *read/write* heuristic increased the percentage of the detected erroneous runs the most.

Finally, the influence of the testing environment described in Section 5.4.3 (in particular, the different hardware used) on programs under test was analyzed on the Crawler/Crawler2 and Rover/Rover2 test cases. In the Crawler/Crawler2 test cases, the results clearly show the influence of the environment. The error was detected by 9 noise configurations in the Crawler test case. In the Crawler2 test case, the number of successful configurations increased to 39 including all the 9 configurations which worked for the Crawler test case. Additionally, in the Crawler2 test case which was executed on a machine with more available cores as described above, the obtained results show higher numbers of achieved coverage and a higher error detection ratio (i.e. the number of executions in which a suitable configuration was able to detect the error). Conversely, in the Rover/Rover2 test cases, the influence of the environment was minimal. The same configurations were able to detect the error and the achieved coverage reached almost the same levels.

5.4.4 Specifics of Noise Implementation

Before the comparison of the results obtained for C and Java test cases is provided, differences in implementing the noise injection techniques for C/C++ and Java programs are discussed here. There are various ways to insert noise injection code into a program. The code might be inserted directly to the source code of the program, to its intermediate code (e.g., Java bytecode), or to the binary code. In general, inserting the code to the source code of the program have several disadvantages. It requires to have the source code of the program (and all of the libraries it uses), which might not always be available. It is also less precise as the compiler might, e.g., move the code elsewhere because of some optimizations. Therefore, the ANaConDA framework [55] used for the C/C++ programs and the IBM ConTest framework [42] used for the Java programs insert the noise injection code on the binary and bytecode levels, respectively. In this section, a short summary of the experiences with implementing the noise injection techniques on the binary level of C/C++ code and the Java bytecode level is presented.

[3] A hypothesis to be tested in the future is that the positive impact of the new heuristics on error detection is reduced by the further noise associated with collecting coverage data.

Inserting some code to the bytecode of a program is not a big problem as the bytecode instructions are quite simple and JVM uses minimum optimizations complicating this task. However, inserting code to binaries of a C/C++ program is not such easy task. On the binary level there are used highly optimized instructions such as conditional and repeat instructions [54]. While the conditional instructions might not be executed when the control reaches them, the repeat instructions may be executed more than once as though they were placed in a loop. Moreover, the `rep`-prefixed instructions, designed for manipulating continuous sequences of memory locations (e.g., within string operations), are both conditional and repeat instructions since they may be executed a fixed number of times, until some condition is met, or sometimes not executed at all. When the binary code contains such instruction, one has to be sure that the noise is injected only when the instruction was really executed or every time the instruction was executed in a loop.

Distinguishing local and shared variables represents another problem. In Java, local data are stored on the current thread stack and possibly shared data are stored on the heap. Since there exist different instructions for accessing stack and heap, it is easy to distinguish accesses to the heap and apply noise only to them. On the binary level, local variables are on the stack too but the stack is just a reserved part of memory which might be accessed in the same way as the memory containing globally accessible variables. If noise injection before accesses to local variables is not desirable, one has to determine before each access whether the accessed variable is stored in memory containing the stack or not.

Finally, in some cases (e.g., in the implementation of the access pattern detector for the *pattern* noise placement heuristic), tracking of method or function entry and exit events is necessary. Again, such events were fairly easily identified in Java bytecode but fairly difficult to detect on the binary level of C/C++ programs where returning from functions is often heavily optimized by the compiler, e.g., using jumps between functions with the effect of the control effectively returning from another function than the one that was called [54], etc.

To sum up, the implementation of the actual noise generators is of equal difficulty in C and Java. On the other hand, instrumentation and execution monitoring is much harder on the binary level as described above. Overcoming the obstacles of the binary level optimizations has a negative impact on the overhead of the ANaConDA framework for noise-based testing and dynamic analysis.

5.4.5 Comparison of Results Obtained for C and Java Test Cases

In this section, discovered commonalities and dissimilarities when analyzing the obtained results of experiments with C and Java programs are briefly described. Note that this comparison may be partially influenced by the used infrastructures for noise injection in C and Java which differ as highlighted in the previous section and the test cases which are also not directly comparable (the comparison studies simple C programs created by students which implement a solution for the same problem and Java programs of various size implementing different problems). Nevertheless, the findings presented here might still be of interest for the users of noise-based testing.

The Java experiments indicate that the success of noise-based testing depends mainly on carefully choosing the noise placement and noise seeding heuristics (tweaking the frequency and strength parameters did not improve the results much). On the contrary, the results for the C programs show that strength and frequency of noise are also very important in the considered test cases. Further analysis of the results achieved for the Java Airlines test

case indicate that they share a few characteristics with the C programs. In particular, the same heuristics (including the new ones) provided good results, tweaking of frequency and strength did considerably affect the results, and stronger noise provided often good results. The Airlines test case is of similar size, contains a similar data-depending error, and the error manifestation ratio without any noise heuristics is also comparable.

In the considered C test cases, the results clearly show that a majority of noise configurations provided similar results across the four considered test cases. A configuration which provided good results in one test case was successful also in the other test cases and vice versa the configurations which provided poor improvement achieved poor results in all considered test cases. This is most probably caused by the similarity of the test cases. Indeed, the very similar results were also achieved for the Crawler/Crawler2 and Rover/Rover2 test cases in Java. The configurations which provided good results for the Crawler (Rover) test case were among the good ones even under the slightly different conditions represented by the Crawler2 (Rover2) test cases.

5.4.6 Hints for Noise-based Testing

The results presented above indicate that there is no single optimal noise configuration. The same noise setting may provide significantly different results for different test cases, testing goals, as well as testing environment. Moreover, using a wrong noise injection technique can in some cases even degrade the quality of the testing process. Therefore, if no information concerning the tested program is available, a good option is to start with the random setting which selects noise heuristics and their parameters at random before each execution of the tested program. This setting often does not achieve the overall best results as mentioned above but it provides reasonably good results with a minimal effort. Further, if one has at least a suspicion that the program under test may contain a data-dependent error (such as a data race or an atomicity violation), based on the experience, using some of the heuristics focused on shared variables (or restricting the random choice of noise heuristics to those focusing on shared variables) might be a good idea.

If one has to set up the noise seeding and placement heuristics manually (i.e., there is no support for the random choice of noise heuristics in repeated test), based on the results, one can recommend using the *yield*, *synchYield*, *wait*, and *busyWait* heuristics, which often provided good results in the experiments described above. The *yield* and *synchYield* heuristics have a smaller impact on the performance while still providing good improvement in some cases. The *wait* and *busyWait* heuristics cause a considerable performance degradation, but they can help to test even rarely executed synchronization scenarios. Further, the results indicate that using a low noise frequency (in particular, below 5 %) or using a high noise frequency (in particular, over 50 %) does not bring a higher probability of spotting an error or obtaining a higher coverage. On the contrary, a high noise frequency used with a demanding heuristic (e.g., *busyWait*) has a negative impact on the efficiency of the test.

All the considered advanced noise seeding heuristics (i.e., *timeoutTampering*, *haltOneThread*) including the newly proposed heuristics (i.e., *coverage*, *read/write*, and *pattern-based*) provide in some cases a considerable improvement of the testing process. Therefore, it is worth to enable them and test whether they positively affect results of the considered test case. If they do, the results indicate that the same heuristics might be providing good results even if the test is executed in a different environment. This is because the efficiency of these heuristics depends on appearance of certain code patterns in the program under test. Therefore, a simple static analysis of the program might help with the decision making

(e.g., an analysis which detects appearance of `wait()` and `sleep()` could indicate that the *timeoutTampering* heuristic might provide good results). Next, the results also indicate that heuristics which put noise at carefully selected locations only provide better results than heuristics which simply put noise randomly or at too many locations.

To sum up, above a number of hints that may be useful when applying noise-based testing is provided. But, it is important to repeat that choosing a suitable noise configuration is a difficult task, and the hints need not work in all cases. Hence, the final advice is to—if possible—experiment with more different noise settings. There also exists various automated approaches for solving this problem, i.e, for finding suitable noise settings, based on using search techniques [52], multi-objective genetic algorithms [40], or data mining [12].

5.5 Conclusion

This chapter discussed noise-based testing that helps to examine different thread interleavings during testing and dynamic analysis of concurrent programs and hence increases chances of finding concurrency-related errors.

We have proposed an improvement to the basic noise injection approach to be used to increase chances of spotting an error when testing or dynamically analysing a multithreaded C/C++ program. We have experimentally validated the proposed improvement on a set of C/C++ programs, and we have also discussed the effect of various noise settings when dealing with such types of programs.

Then we presented an overview of multiple results in the area of noise-based testing. Beside discussing various existing heuristics, we further created two new heuristics for the noise placement and noise seeding problems that play a crucial role in noise-based testing. Results of some previously performed comparisons of noise injection techniques were summarized and used as a basis for performing a new, more thorough comparison of the most promising noise injection heuristics as well as the new noise heuristics proposed in this chapter. The heuristics were compared according to their ability to find concurrency errors, to increase concurrency coverage, and to cause an acceptable performance degradation.

The presented experimental results show that noise injection can indeed very significantly improve the testing process, but there is no silver bullet among the many noise injection techniques. Their performance depends on the test case, test goal, as well as test environment. Hence, for a new test case, experimenting with the various noise injection heuristics may be needed—or, as often done in industrial practice (e.g., within the industrial use of ConTest in IBM mentioned in Section 1.1), one can apply a randomly selected mix of the heuristics. Alternatively, one can use techniques such as genetic programming or data mining to automatically learn from so-far performed test runs what the best noise setting for the given context might be.

Several promising directions for future work were envisaged: (1) One could try to introduce some more sophisticated noise heuristics, e.g., tailored for a specific detector or type of concurrency errors. In fact, going in this direction, a noise placement heuristic similar to the pattern noise is used in Chapter 7 by the method for dynamic contract validation to increase the probability to detect a contract violation. (2) Since using different noise injection configurations for the read and write accesses proved to be useful in our experiments, it may be interesting to look more into the fine-grained use of noise and find more rules how to use it, perhaps supported by some preliminary analysis of the program at test or allowing the noise settings to be automatically suitably adjusted during a test execution based on the so-far obtained results. (3) One could think of new heuristics and approaches

combining the simplicity of noise injection with the recent developments in the field of systematic testing. For instance, one could use noise-based testing to roughly explore the behavior of the tested program and use systematic testing to test only particular areas of the program behavior. (4) Noise injection is a lightweight testing approach that has a moderate impact on the performance of the test. Nevertheless, there is a simple possibility to further improve its performance by using partial instrumentation of the code. In this case, only selected parts of the code would be instrumented, and therefore affected by the noise. All parts of the code which would be known to be safe or to contain no concurrency related behavior could be omitted during the instrumentation. (5) Finally, there is still a lot of space for new combinations of static and dynamic analyses, further improving efficiency of the testing processes.

Chapter 6

Transactional Memory Programs

This chapter proposes several approaches for monitoring Transactional memory (TM) programs and studies their impact on the behaviour of the monitored programs. The considered approaches range from specialised lightweight monitoring to generic heavyweight monitoring. The implemented monitoring tools are publicly available and the implementation techniques used for lightweight monitoring may be used as an inspiration for developing other specialised lightweight monitors.

6.1 Introduction

Transactional memory (TM) [67, 71] is an increasingly popular technique for synchronising threads in multi-threaded programs, which is both easy to use and provides good performance. When using TM, the threads are synchronised by defining transactions that may be executed optimistically in parallel and will succeed if they do not interfere with each other. Even though using TM may be easier, there are still various opportunities to make mistakes that lead to performance degradation and errors, which rises a clear demand for tools for analysing and debugging TM programs. Because performance analyses usually require the program to be executed to be able to analyse its performance, dynamic analysis is often used here as it would be able to address both correctness and performance-related issues of TM programs.

In order to be able to implement various dynamic analyses of the behaviour of TM programs, one first needs to monitor their execution. However, the monitoring code may influence the monitored program's behaviour and hamper the results of some analyses. That is why, in this chapter, we propose several different ways of monitoring C/C++ TM programs and then experimentally study their influence on the behaviour of the monitored programs. Our monitoring approaches range from lightweight to heavyweight monitoring. The monitored programs are taken from the well-known STAMP benchmark [26].

As our primary metric for evaluating the influence of the different monitoring approaches, we use the number of transactions that aborted during the execution of the monitored TM programs as this metric gives a good insight into their contention level, i.e., into the number of conflicting concurrent transactions. The more conflicts and aborts the more work for the TM system.

In this chapter, we also present an experimental evaluation of the influence of different kinds of lightweight and heavyweight monitoring approaches for TM programs that we propose, both in terms of global numbers of aborts as well as numbers of aborts for different

types of transactions. Moreover, we also show that the obtained results can be significantly influenced by the environment in which the monitoring is performed.

The results presented in this chapter can be used in several ways. First, they can show researchers or developers interested in monitoring TM programs how the behaviour of these programs can be influenced by different monitoring techniques as well as the environment. Second, the proposed and implemented monitoring techniques are available to the scientific community and can be used in other settings, which is especially easy for the case of heavyweight monitoring since we implemented a quite generic TM monitoring platform on top of the ANaConDA framework [53]. The lightweight monitoring approaches are rather specialised; however, the described implementation techniques can be useful if there is a need for implementing yet another lightweight monitor.

Related work. To the best of our knowledge, there are only a couple of works dealing with monitoring of TM programs, namely the works [27,105]. These works aim at providing the users with a variety of interesting data about the execution of a TM program by tracing its operations. However, only the authors of [27] discuss how their monitoring influences the monitored programs, and this discussion is rather brief and addresses only the global number of aborts. We provide a much more detailed study of the influence of monitoring on the monitored programs, using more and/or different monitoring approaches and considering other metrics besides the global numbers of aborts.

6.2 Monitoring Transactional Memory Programs

In this section, we briefly recall general principles and properties of both lightweight and heavyweight monitoring techniques, and we propose several ways to use these approaches in monitoring TM programs. The influence of these techniques on the monitored programs is then experimentally studied in the next section.

6.2.1 Lightweight and Heavyweight Monitoring

Lightweight monitoring [105] strives to minimize the impact of the monitoring activity on the behaviour of the monitored TM program. To achieve this goal, only a limited amount of information is collected, mainly the kind of information that can be obtained fast enough and with minimal intrusion. This makes lightweight monitoring particularly suitable for analysing a program for performance issues. To achieve the highest performance, the monitoring code is usually embedded into the monitored program itself by modifying its source or intermediate code, or even its binary. In all these cases, the monitored program is modified and differs from the original one.

Besides the limited amount of information provided, another disadvantage of the lightweight approach is its lack of automation and/or versatility. The program must be modified again and again for each change in the information to be collected, no matter how small that change is. Sometimes, the required information can be acquired by modifying only some of the libraries used by the program (such as the TM run-time libraries in our case), but then the monitoring will be restricted to those programs that use this specific library. Moreover, embedding monitoring code into a library may be problematic if it is being shared with other programs running on the system, requiring one to manage and maintain multiple versions of the same library.

Heavyweight monitoring [111] trades performance for versatility. It frequently uses a specific run-time environment, such as some kind of a low-level virtual machine, to execute the code of the given program and to monitor its execution. Executing the program in such an environment slows down its execution considerably but enables the acquisition of nearly any information required about the execution of the program. Moreover, environments supporting dynamic instrumentation are able to insert (or remove) the monitoring code during the execution of the program, leaving its original code untouched. Finally, by having full control of the code being executed, these environments are able to monitor even self-modifying or self-generating code.

6.2.2 Lightweight Monitoring of TM Programs

In order to study the impact of monitoring on the behaviour of monitored TM programs, we proposed and implemented several lightweight monitoring approaches. These approaches differ in how much information they are collecting and how they are collecting this information. TM libraries usually provide information about the global numbers of started, committed, and aborted transactions. We take the possibility of obtaining this information as a starting point, and our monitoring approaches allow one to obtain various refinements of this information.

Our lightest monitoring approach (denoted as the *statistics collector* or *sc* in the experiments) allows one to obtain not only the global numbers of started, committed, and aborted transactions, but also all of these numbers separately for each thread and each type of transaction. In order to be as lightweight as possible, this information is obtained in such a way that the monitoring code maintains two counters for each thread and each type of transaction: the first one tracking the number of started transactions and the second one recording the number of committed transactions. These counters are stored in a two-dimensional array so that each combination of a thread and a type of transaction has its own exclusive set of counters. As each thread is accessing a different part of the array, no additional synchronization is introduced. Further, to achieve the best performance, the array is static with a defined maximum number of supported threads and types of transactions, and no boundary checks are done during the monitoring—the monitoring code just accesses a counter and increments it. The numbers of aborts are then computed from the numbers of started and successfully committed transactions.

Our next monitoring approach (denoted as the *event logger* or *el* in the experiments) is based on registering TM operations (events) in an event log (list) during a program execution, followed by a *post mortem* processing of these events. An event is generated (and stored in the event log) only when a transaction starts or successfully commits, and the number of aborts is computed later. In order to minimize the probe effect, each thread has its own event log which resides in the main memory, and hence no additional synchronization between the threads or interaction with the file system is needed[1].

Finally, we have implemented several variants of the event logger. The *el-a* variant differs from the basic event logger in that it is explicitly tracking the aborts and does not compute them from the number of started and successfully committed transactions. The *el-arw* variant does additionally track transactional reads and writes, which significantly increases the number of events collected. Further, we extend all the three above mentioned event logger approaches by collecting and associating a time stamp for each logged event

[1] Eliminating the interaction with the file system is very important as writing to a file introduces a significant intrusion to the execution of a program.

(leading to variants denoted as *el-ts*, *el-a-ts*, and *el-arw-ts* in the experiments). The time stamp is retrieved from the Intel TSC (Time Stamp Counter) register, and storing the time stamp doubles the data size of each event.

The implementation of all of our monitoring approaches is available[2] and can be used either directly or serve as an inspiration for implementing other specialized monitors. The current implementation is restricted to the TL2 library and requires a modification of the source code of the program to be monitored. Since the TL2 library provides a set of macros representing the TM operations and these macros are used by the testing programs, our implementation inserts the monitoring code into the programs by modifying these macros. Thus, the source code of the programs is modified at compile time when the modified macros are being expanded by the compiler. Still, we need to recompile the programs with a different set of macros every time we need to change the way the monitoring is done or the type of information to be acquired.

6.2.3 Heavyweight Monitoring of TM Programs

For versatile heavyweight monitoring of TM programs, we have proposed and implemented an extension of the ANaConDA framework [53]. The ANaConDA framework is based on PIN [111], a dynamic binary instrumentation tool from Intel. ANaConDA enables monitoring of multi-threaded C/C++ programs and allows one to obtain information about common synchronisation operations, such as memory accesses or lock acquisitions and releases. In order to support (heavyweight) monitoring of TM programs, we extended the ANaConDA framework to include a support for monitoring TM operations as described below.

The C/C++ programming languages usually include a support for TM by making use of a software library. In this setting, monitoring the TM operations implies intercepting the calls of the functions in this library. As there are many libraries implementing TM for C/C++, our extension is not restricted to a specific library and may be easily instantiated for any TM library. This allows one to analyse a broad variety of TM programs, not only a subset of programs using a specific library. Regardless of the concrete implementation/library used, TM is supported by five basic operations: three operations for managing transactions (*txStart*, *txCommit*, and *txAbort*); and two operations for managing the transactional accesses to the main memory (*txRead* and *txWrite*).

To be able to monitor the five basic TM operations of a concrete TM library with ANaConDA, the user has to identify which library functions implement these operations and which of their parameters reference memory locations. After that, the extended ANaConDA framework is able to monitor any TM program that uses that particular TM library. Currently, we instantiated the extended ANaConDA framework with a support for monitoring programs that use the TL2-x86[3] or the TinySTM[4] libraries.

We implemented all of the approaches described in the previous sections as plug-ins for the extended ANaConDA framework. The framework monitors the execution of a TM program and sends notifications of the relevant TM events to the plug-in. The plug-in then processes the events in the same way as the lightweight monitoring approaches. Unlike in the case of lightweight monitoring, the heavyweight monitoring does not require customized versions of the monitored program specifically tailored for a particular monitoring strategy.

[2] http://github.com/fiedorjan/lightweight-stm-monitoring
[3] http://stamp.stanford.edu/releases.shtml#tl2-x86
[4] http://tmware.org/tinystm

Table 6.1: Average number of aborts in original runs and runs with lightweight monitoring.

		genome	intruder	kmeans		ssca2	vacation		yada
	variant			high	low		high	low	
Lightweight	orig	**2.6** $\cdot 10^4$	**4.3** $\cdot 10^7$	**5.6** $\cdot 10^6$	**5.2** $\cdot 10^6$	**2.6** $\cdot 10^2$	**4.9** $\cdot 10^5$	**2.6** $\cdot 10^4$	**2.7** $\cdot 10^6$
	sc	**2.8** $\cdot 10^4$	**4.3** $\cdot 10^7$	**5.4** $\cdot 10^6$	**5.1** $\cdot 10^6$	**3.5** $\cdot 10^2$	**4.9** $\cdot 10^5$	**2.7** $\cdot 10^4$	**2.6** $\cdot 10^6$
	el	**2.3** $\cdot 10^4$	**3.8** $\cdot 10^7$	**4.3** $\cdot 10^6$	**4.0** $\cdot 10^6$	**2.7** $\cdot 10^2$	**4.6** $\cdot 10^5$	**2.5** $\cdot 10^4$	**2.6** $\cdot 10^6$
	el-ts	**2.2** $\cdot 10^4$	**3.5** $\cdot 10^7$	**3.7** $\cdot 10^6$	**3.4** $\cdot 10^6$	**2.0** $\cdot 10^2$	**4.4** $\cdot 10^5$	**2.4** $\cdot 10^4$	**2.3** $\cdot 10^6$
	el-a	**2.3** $\cdot 10^4$	**3.7** $\cdot 10^7$	**4.0** $\cdot 10^6$	**3.7** $\cdot 10^6$	**2.0** $\cdot 10^2$	**4.4** $\cdot 10^5$	**2.4** $\cdot 10^4$	**2.5** $\cdot 10^6$
	el-a-ts	**2.1** $\cdot 10^4$	**3.4** $\cdot 10^7$	**2.9** $\cdot 10^6$	**2.7** $\cdot 10^6$	**2.2** $\cdot 10^2$	**3.9** $\cdot 10^5$	**2.1** $\cdot 10^4$	**2.1** $\cdot 10^6$
	el-arw	**2.1** $\cdot 10^4$	**1.1** $\cdot 10^7$	**3.2** $\cdot 10^6$	**3.4** $\cdot 10^6$	**1.9** $\cdot 10^2$	**0.5** $\cdot 10^5$	**0.8** $\cdot 10^4$	**1.8** $\cdot 10^6$
	el-arw-ts	**2.5** $\cdot 10^4$	**0.8** $\cdot 10^7$	**2.3** $\cdot 10^6$	**2.7** $\cdot 10^6$	**2.5** $\cdot 10^2$	**0.5** $\cdot 10^5$	**0.8** $\cdot 10^4$	**1.5** $\cdot 10^6$

Based on the type of information requested by each plug-in, the framework instruments the original code of the monitored program upon loading it into the main memory with the code which collects the required information.

6.3 Experimental Evaluation of the Impact of Monitoring

We will now present a set of experiments that evaluate the influence of the monitoring approaches described in the previous section on the behaviour of a set of benchmark TM programs from several different points of view. For our experiments, we used 6 out of 8 programs from the STAMP benchmark suite [26], namely `genome`, `intruder`, `kmeans`, `scca2`, `vacation`, and `yada`. These programs utilise transactional memory to solve a wide variety of problems. In case of the `kmeans` and `vacation` programs, we also distinguish the `high` and `low` variants that use respectively the high and low contention configurations available in the benchmark. The remaining two benchmarks, `bayes` and `labyrinth`, were excluded due to technical problems unrelated with the work described in this paper.

For the experiments, we used two different environments. The first environment, which we will refer to as *x5355-64GB*, consists of a single machine with 4-core Intel Xeon X5355 2.66 GHz CPU and 64 GB of memory, running Linux with the 3.2.0 kernel. The second environment, which we will refer to as *x3450-8GB*, is a cluster containing three identical nodes with 4-core Intel Xeon X3450 2.66 GHz CPUs and 8 GB of memory, running Linux with the 2.6.26 kernel. As all of the CPUs which we used support Hyper-threading, up to 8 threads may run seemingly simultaneously on any of these machines. To achieve maximal concurrency, all of the benchmarks were configured to use 8 threads. For lightweight monitoring, programs were compiled with `-g` and `-O3` flags.

6.3.1 Comparison of Lightweight Monitoring Approaches

First, we evaluate the impact of the different variants of lightweight monitoring that we proposed on the behaviour of the monitored programs. As a metric, we use the global number of transactions aborted during the program run. The presented experiments were performed in the *x5355-64GB* environment.

Table 6.1 shows the average global number of aborts (out of 100 runs) for each of the tested programs when executed with the different variants of lightweight monitoring described in Section 6.2.2. The variant *orig* represents a run without any monitoring, i.e.,

Table 6.2: Average aborts in original runs and runs with lightweight monitoring without outliers.

		genome	intruder	kmeans		ssca2	vacation		yada
	variant			high	low		high	low	
Lightweight	orig	**2.6** $\cdot 10^4$	**4.3** $\cdot 10^7$	**5.6** $\cdot 10^6$	**5.0** $\cdot 10^6$	**2.6** $\cdot 10^2$	**4.9** $\cdot 10^5$	**2.5** $\cdot 10^4$	**2.6** $\cdot 10^6$
	sc	**2.7** $\cdot 10^4$	**4.4** $\cdot 10^7$	**5.4** $\cdot 10^6$	**5.0** $\cdot 10^6$	**2.5** $\cdot 10^2$	**4.9** $\cdot 10^5$	**2.6** $\cdot 10^4$	**2.6** $\cdot 10^6$
	el	**2.2** $\cdot 10^4$	**3.8** $\cdot 10^7$	**4.2** $\cdot 10^6$	**3.9** $\cdot 10^6$	**1.7** $\cdot 10^2$	**4.6** $\cdot 10^5$	**2.5** $\cdot 10^4$	**2.6** $\cdot 10^6$
	el-ts	**2.1** $\cdot 10^4$	**3.5** $\cdot 10^7$	**3.7** $\cdot 10^6$	**3.3** $\cdot 10^6$	**1.6** $\cdot 10^2$	**4.3** $\cdot 10^5$	**2.4** $\cdot 10^4$	**2.3** $\cdot 10^6$
	el-a	**2.3** $\cdot 10^4$	**3.7** $\cdot 10^7$	**3.9** $\cdot 10^6$	**3.6** $\cdot 10^6$	**1.9** $\cdot 10^2$	**4.4** $\cdot 10^5$	**2.4** $\cdot 10^4$	**2.5** $\cdot 10^6$
	el-a-ts	**2.1** $\cdot 10^4$	**3.4** $\cdot 10^7$	**2.9** $\cdot 10^6$	**2.6** $\cdot 10^6$	**1.6** $\cdot 10^2$	**3.9** $\cdot 10^5$	**2.1** $\cdot 10^4$	**2.1** $\cdot 10^6$
	el-arw	**2.1** $\cdot 10^4$	**1.1** $\cdot 10^7$	**3.2** $\cdot 10^6$	**3.2** $\cdot 10^6$	**1.8** $\cdot 10^2$	**0.5** $\cdot 10^5$	**0.8** $\cdot 10^4$	**1.8** $\cdot 10^6$
	el-arw-ts	**2.4** $\cdot 10^4$	**0.9** $\cdot 10^7$	**2.3** $\cdot 10^6$	**2.6** $\cdot 10^6$	**1.7** $\cdot 10^2$	**0.5** $\cdot 10^5$	**0.8** $\cdot 10^4$	**1.5** $\cdot 10^6$

the execution of the original program with no modifications. The parameters of each of the programs were set to the values recommended for the so-called standard runs of the programs in the STAMP benchmark suite[5].

When performing the most lightweight monitoring (*sc*), the global number of aborts does not change much and stays almost always within a range of 5 % from the original runs. The only exception is the `ssca2` benchmark which gets near 35 % more aborts than in the original runs. This is caused by the so-called outliers, i.e., rare runs that achieve a number of aborts much higher than usual, which distorts the results. This effect is more noticeable in the cases where the global number of aborts is relatively low and even one of such outlying runs may change the average values considerably. For example, the results for the `ssca2` benchmark using the *sc* monitoring approach contained two runs with 4300 and 3800 global numbers of aborts. When we look at the global number of aborts and remove the 10 runs identified as outliers, we get close to the original global number of aborts even for the `ssca2` benchmark. These results can be seen in Table 6.2. In particular, we take as outliers the runs which achieved a significantly different global number of aborts than the rest of the runs based on their Euclidian distance from the 10 runs with the closest global number of aborts.

When we try to obtain the same information as above using the event logger approach (*el*), we see that the global number of aborts drops much more than when using the *sc* approach—changing up to 25 % of the original value. This is because logging the events in a list is more intrusive than just incrementing a counter. This demonstrates that it is indeed quite important how the monitored information is acquired and registered as even slightly different methods that obtain the same information may have considerably different impact on the behaviour of the monitored TM programs.

When we start collecting more information (events) than just the number of started and committed transactions, we get an even lower global number of aborts. When logging the number of aborts as well (using the *el-a* approach), the drop in the number of aborts is not that significant yet (up to 30 % of the original value) as the number of events of this type is not that high. However, when we start tracking the read and write operations as well (using the *el-arw* approach), the global number of aborts often suffers large drops (the

[5]These parameters are recommended by the STAMP authors when running the benchmarks natively, i.e., directly on a concrete operating system, not in a simulator or another tool negatively affecting its performance.

change is up to 90 % of the original value). This is related to the fact that the number of reads and writes is usually much higher than the number of starts and commits.

If we also start collecting the time stamps (using the *el-ts*, *el-a-ts*, and *el-arw-ts* approaches), the global number of aborts does also drop when compared with the variants not collecting the time stamps. However, in general, despite collecting time stamps is usually more intrusive than tracking the aborts, it is less intrusive than tracking the reads and writes.

6.3.2 Comparison of Lightweight and Heavyweight Monitoring

In this section, we compare the impact of the lightweight and heavyweight implementations of the considered monitoring approaches. Since heavyweight monitoring greatly slows down the tested programs, for these experiments the parameters of the benchmarking programs were set to the values recommended by the STAMP authors for the so-called simulation runs, which are suitable when executing a program in a simulator or another tool that negatively affects its performance. Since the simulation runs generate much less aborts than the standard ones, meaning that the results might be negatively influenced by the outliers, we remove 10 (out of 100) runs marked as the outliers during the evaluation. Due to the higher time cost of these tests, the experiments were performed in the *x3450-8GB* environment.

Table 6.3 shows the average global number of aborts for each of the tested programs for the lightweight and heavyweight implementations of the monitoring approaches described in Section 6.2.2. The heavyweight implementations come in two different versions. The first version, called *PIN*, does the monitoring by executing the lightweight monitoring implementation, i.e., the modified versions of the programs, in the PIN framework without doing any instrumentation of the program. The purpose of this version is to show how the use of PIN's low-level virtual machine changes the behaviour of the monitored program even without the influence of the instrumentation needed to capture the monitored events. The second version, denoted as *ANaConDA*, is the true heavyweight implementation where the counter incrementation and event collection is done through the callbacks provided by the extended ANaConDA framework.

First of all, let us note that compared with the results of the standard runs (Table 6.2), the results of the simulation runs exhibit the same tendencies when monitored using the lightweight approaches (and hence we can consider their use instead of the standard runs meaningful). The main difference is that the simulation runs are more prone to problems with outliers as their execution time is quite short and even a very short disruption during the execution may change significantly the overall results. For example, the results obtained for the `yada` benchmark using the *sc* monitoring approach contain several runs with significantly greater global number of aborts even after the 10 outliers have been removed (in fact, in this batch of runs there were 14 runs with a very high global number of aborts).

When we start monitoring the programs using the heavyweight versions of the monitoring approaches, we can see a massive drop in the global number of aborts (more than 95 %). This drop is mainly caused by PIN's low-level virtual machine as just running the original (non-modified) version (*orig*) of a program in PIN leads to an extreme drop in the global number of aborts (more than 95 %). The additional disruption introduced by the monitoring code does not influence much the behaviour. In fact, rather than having the effect of decreasing the global number of aborts, like in the case of the lightweight monitoring, inserting the monitoring code actually helps to increase the number of aborts a little

Table 6.3: A comparison of average number of aborts for lightweight and heavyweight monitoring.

		genome	intruder	kmeans		ssca2	vacation		yada
	variant			high	low		high	low	
Lightweight	orig	67.6	22850.0	3804.7	1626.1	6.5	23.4	4.9	9362.3
	sc	73.3	22013.1	4115.7	1721.5	7.2	23.3	5.3	11659.3
	el	63.1	17663.5	2722.9	1245.9	12.2	25.2	5.3	9354.7
	el-ts	61.3	16797.2	2402.7	1236.4	13.0	22.6	4.7	8118.7
	el-a	65.8	16504.1	2204.3	1091.0	16.6	22.6	4.0	8096.3
	el-a-ts	64.3	16112.9	1696.8	942.8	15.6	19.7	3.8	6846.7
	el-arw	72.7	8238.9	2891.2	1877.0	18.0	19.9	3.7	5804.0
	el-arw-ts	107.1	9499.4	3463.6	2121.3	22.0	22.6	4.7	4458.0
PIN	orig	3.7	85.8	0.2	0.1	0.0	2.1	0.2	595.1
	sc	3.4	81.1	0.4	0.1	0.0	2.0	0.3	584.4
	el	8.6	92.2	7.2	6.7	0.5	2.4	0.5	589.3
	el-ts	9.4	106.9	9.0	7.8	0.7	2.5	0.3	571.2
	el-a	7.0	101.6	14.9	12.2	0.5	2.1	0.2	580.2
	el-a-ts	7.4	95.7	17.5	14.6	0.6	2.4	0.3	576.6
	el-arw	13.2	476.8	36.6	28.6	0.9	10.1	1.6	715.2
	el-arw-ts	24.1	1567.1	213.2	139.3	1.0	14.6	2.8	902.4
ANaConDA	orig	10.8	71.4	0.3	0.1	0.0	1.9	0.2	595.6
	sc	9.3	109.8	0.2	0.1	0.0	3.4	0.6	729.6
	el	13.7	109.7	8.6	7.8	0.6	4.0	0.5	704.3
	el-ts	11.3	119.2	9.8	8.6	0.8	4.0	0.4	687.4
	el-a	12.3	126.0	20.8	16.7	0.9	3.6	0.7	702.4
	el-a-ts	11.0	133.8	24.5	18.0	0.9	4.0	0.5	682.3
	el-arw	20.8	1653.4	178.5	126.9	1.3	17.4	2.8	1100.1
	el-arw-ts	34.4	3132.9	480.8	305.8	1.5	19.1	3.7	1260.8

in the heavyweight monitoring. This effect increases as we collect more information while monitoring, which is a completely opposite tendency compared to the lightweight monitoring. Also, the monitoring code inserted by ANaConDA has a greater effect on increasing the global number of aborts than using the lightweight monitoring code executed in PIN.

Another effect that the heavyweight monitoring has on the considered programs is that it suppresses the outliers. Table 6.3 contains the results evaluated from the runs not marked as outliers, but the results are nearly identical even when considering all of the runs.

6.3.3 Impact of the Monitoring on Different Types of Transactions

The global number of aborts is an important performance metric and hence also a good basic metric of how the behaviour of the monitored programs is influenced by the monitoring layer. However, one may want to get a more detailed information about the behaviour of a program and also about the way how it is influenced by monitoring. To go one step further in this direction, we now consider monitoring numbers of aborts of different types of transactions and the influence of monitoring on these numbers. Since TM libraries do not give us statistics for different types of transactions, we use the information obtained using the *sc* monitoring approach as a baseline behaviour of a program in this case. As the global number of aborts when using the *sc* monitoring approach is very similar to the

Table 6.4: Average number of aborts for different types of transactions.

		intruder			kmeans-high		
	variant	Tx1	Tx2	Tx3	Tx4	Tx5	Tx6
Lightweight	sc	**13.9** $\cdot 10^6$	**91.0** $\cdot 10^5$	**20.5** $\cdot 10^6$	**51.7** $\cdot 10^5$	**24.9** $\cdot 10^4$	**51.0** $\cdot 10^0$
	el	**9.5** $\cdot 10^6$	**85.2** $\cdot 10^5$	**19.9** $\cdot 10^6$	**40.9** $\cdot 10^5$	**22.1** $\cdot 10^4$	**44.0** $\cdot 10^0$
	el-ts	**8.1** $\cdot 10^6$	**83.5** $\cdot 10^5$	**18.9** $\cdot 10^6$	**35.1** $\cdot 10^5$	**21.8** $\cdot 10^4$	**36.0** $\cdot 10^0$
	el-a	**9.5** $\cdot 10^6$	**86.0** $\cdot 10^5$	**19.0** $\cdot 10^6$	**37.8** $\cdot 10^5$	**21.9** $\cdot 10^4$	**37.0** $\cdot 10^0$
	el-a-ts	**8.7** $\cdot 10^6$	**83.0** $\cdot 10^5$	**17.0** $\cdot 10^6$	**26.8** $\cdot 10^5$	**22.2** $\cdot 10^4$	**33.0** $\cdot 10^0$
	el-arw	**5.1** $\cdot 10^6$	**23.6** $\cdot 10^5$	**3.3** $\cdot 10^6$	**31.3** $\cdot 10^5$	**8.3** $\cdot 10^4$	**12.0** $\cdot 10^0$
	el-arw-ts	**5.1** $\cdot 10^6$	**22.3** $\cdot 10^5$	**1.1** $\cdot 10^6$	**22.6** $\cdot 10^5$	**7.7** $\cdot 10^4$	**11.0** $\cdot 10^0$

original global number of aborts, we may safely assume that this behaviour is very close to the original one. The presented experiments were again performed in the *x5355-64GB* environment.

Table 6.4 shows the average number of aborts for each type of transactions present in the `intruder` and `kmeans` benchmarks (in the latter case, for the variant with high contention). As can be seen, the various kinds of monitoring influence each type of transactions differently. When looking at transactions of Type *Tx2* and *Tx3* for the `intruder` benchmark or at transactions of Type *Tx5* for the `kmeans` benchmark, one can see that utilizing the event logger with or without direct tracking of aborts (*el* and *el-a*, respectively) does not influence the average number of aborts much. The drop in the number of aborts is around 10 % here. Also, the collection of time stamps (the *el-ts* and *el-a-ts* approaches) changes these numbers minimally. However, when we start tracking the reads and writes (the *el-arw* approach), the number of aborts drops considerably (by around 65–85 %).

On the other hand, some types of transactions, like transactions of Type *Tx1* for the `intruder` benchmark and transactions of Type *Tx4* for the `kmeans` benchmark are more affected by the event logger (*el*) approach and exhibit a significant decrease in the number of aborts (by around 20–30 %). The number of aborts does not drop much when we add the direct tracking of aborts (*el-a*), but it lowers again (by around 10–20 %) when we include the collection of time stamps (the *el-ts* and *el-a-ts* approaches). When we start tracking the reads and writes in these types of transactions, the number of aborts drops again (by around 10–30 %), but this drop is not that significant as in the case of the previously described transaction types.

One may think that the abrupt drop in the number of aborts that we saw in the transactions of Type *Tx2*, *Tx3*, or *Tx5* when we started tracking the reads and writes is connected to the number of memory accesses in these types of transactions since the influence of the monitoring should be different for transactions with a high and low number of memory accesses, respectively. However, our analysis of the data showed no clear dependency between the number of accesses and the drops in the number of aborts. For example, transactions of Type *Tx2* perform on average 110 accesses to the TM, while transactions of Type *Tx3* just 3 and transactions of Type *Tx5* only 2. Still, the tendencies they exhibit for the various monitoring approaches are the same. The exact cause of this behaviour remains an interesting direction for future work.

Table 6.5: Average aborts in runs with lightweight monitoring in the *x3450-8GB* environment.

		genome	intruder	kmeans		ssca2	vacation		yada
	variant			high	low		high	low	
Lightweight	orig	**3.0** $\cdot 10^4$	**3.0** $\cdot 10^7$	**5.7** $\cdot 10^6$	**4.1** $\cdot 10^6$	**6.3** $\cdot 10^2$	**3.6** $\cdot 10^5$	**3.1** $\cdot 10^4$	**5.0** $\cdot 10^6$
	sc	**3.1** $\cdot 10^4$	**3.0** $\cdot 10^7$	**6.0** $\cdot 10^6$	**4.4** $\cdot 10^6$	**11.7** $\cdot 10^2$	**3.6** $\cdot 10^5$	**3.2** $\cdot 10^4$	**5.0** $\cdot 10^6$
	el	**2.7** $\cdot 10^4$	**2.9** $\cdot 10^7$	**4.9** $\cdot 10^6$	**3.7** $\cdot 10^6$	**3.4** $\cdot 10^2$	**3.4** $\cdot 10^5$	**3.0** $\cdot 10^4$	**4.6** $\cdot 10^6$
	el-ts	**2.6** $\cdot 10^4$	**2.9** $\cdot 10^7$	**4.5** $\cdot 10^6$	**3.3** $\cdot 10^6$	**1.9** $\cdot 10^2$	**3.3** $\cdot 10^5$	**2.8** $\cdot 10^4$	**4.4** $\cdot 10^6$
	el-a	**2.8** $\cdot 10^4$	**2.8** $\cdot 10^7$	**4.2** $\cdot 10^6$	**3.1** $\cdot 10^6$	**5.2** $\cdot 10^2$	**3.3** $\cdot 10^5$	**2.7** $\cdot 10^4$	**4.3** $\cdot 10^6$
	el-a-ts	**2.6** $\cdot 10^4$	**2.5** $\cdot 10^7$	**3.1** $\cdot 10^6$	**2.3** $\cdot 10^6$	**2.3** $\cdot 10^2$	**3.0** $\cdot 10^5$	**2.5** $\cdot 10^4$	**3.6** $\cdot 10^6$
	el-arw	**2.4** $\cdot 10^4$	**0.8** $\cdot 10^7$	**3.4** $\cdot 10^6$	**3.7** $\cdot 10^6$	**5.1** $\cdot 10^2$	*timeout*	**3.5** $\cdot 10^4$	**2.9** $\cdot 10^6$
	el-arw-ts	**2.8** $\cdot 10^4$	**0.7** $\cdot 10^7$	**2.5** $\cdot 10^6$	**2.2** $\cdot 10^6$	**2.4** $\cdot 10^2$	*timeout*	*timeout*	*timeout*

6.3.4 Influence of the Environment

In the previous sections, we discussed that even a slight disturbance of the monitored TM program's execution by the monitoring code could impact its behaviour. However, changes in the monitoring code are not the only factor that may influence the behaviour of the monitored program. Other factors include changes of the environment in which the monitoring is done. That is why we now compare both of our execution environments used for acquiring the experimental results.

In particular, Table 6.5 shows results of the same experiments with lightweight monitoring as Table 6.1 but this time from the *x3450-8GB* environment instead of *x5355-64GB*.[6] We can see that the tendencies for the various monitoring approaches are similar to the ones presented before. However, the average global number of aborts changed for some of the benchmarks. For example, the `intruder` benchmark achieved around 30 % less aborts on this machine regardless of the monitoring approach used. On the other hand, the `yada` benchmark got twice as many aborts with any monitoring approach used.

Moreover, interestingly, some of the benchmarks seem to behave the same way as on the previously used machine when looking at the global number of aborts only. However, when looking at aborts for different types of transactions, one finds out that the program is in fact behaving differently. When looking at the `kmeans` benchmark, the average global number of aborts for the original run (*orig*) is nearly the same, but this is not true when we compare the number of aborts per transactions type.

In particular, Table 6.6 contains the average number of aborts for each type of transactions present in the `intruder` and `kmeans` (high contention variant) benchmarks. When we look at the *sc* monitoring approach and compare transactions of Type *Tx4* and *Tx5* with the results presented in Table 6.4, we see that the number of aborts for transactions of Type *Tx4* increases by about 20 % while the number of aborts for transactions of Type *Tx5* drops by more than 85 %. Moreover, the tendencies exhibited by transactions of type *Tx5* change: now, the number of aborts starts actually increasing when more intrusive monitoring approaches are used. Also, the time stamp collection greatly increases the number of aborts here.

We see a similar change in the behaviour in the `intruder` benchmark for transactions of

[6] The missing values for some of the benchmarks for the *el-arw* and *el-arw-ts* monitoring approaches in Table 6.5 are caused by all of the runs timing out due to the extensive swapping as the main memory was rapidly filled out with the collected events.

Table 6.6: Average aborts for different types of transactions in the *x3450-8GB* environment.

		intruder			kmeans-high		
	variant	Tx1	Tx2	Tx3	Tx4	Tx5	Tx6
Lightweight	sc	**3.2** $\cdot 10^6$	**88.9** $\cdot 10^5$	**17.5** $\cdot 10^6$	**59.8** $\cdot 10^5$	**3.6** $\cdot 10^4$	**6.0** $\cdot 10^0$
	el	**3.8** $\cdot 10^6$	**84.1** $\cdot 10^5$	**16.5** $\cdot 10^6$	**48.7** $\cdot 10^5$	**6.3** $\cdot 10^4$	**7.0** $\cdot 10^0$
	el-ts	**4.2** $\cdot 10^6$	**85.9** $\cdot 10^5$	**16.5** $\cdot 10^6$	**44.0** $\cdot 10^5$	**7.6** $\cdot 10^4$	**8.0** $\cdot 10^0$
	el-a	**3.9** $\cdot 10^6$	**85.9** $\cdot 10^5$	**15.4** $\cdot 10^6$	**41.0** $\cdot 10^5$	**6.4** $\cdot 10^4$	**7.0** $\cdot 10^0$
	el-a-ts	**4.0** $\cdot 10^6$	**83.9** $\cdot 10^5$	**13.1** $\cdot 10^6$	**29.9** $\cdot 10^5$	**7.7** $\cdot 10^4$	**8.0** $\cdot 10^0$
	el-arw	**3.7** $\cdot 10^6$	**15.3** $\cdot 10^5$	**2.3** $\cdot 10^6$	**33.4** $\cdot 10^5$	**6.9** $\cdot 10^4$	**7.0** $\cdot 10^0$
	el-arw-ts	**4.4** $\cdot 10^6$	**14.6** $\cdot 10^5$	**1.1** $\cdot 10^6$	**23.5** $\cdot 10^5$	**10.1** $\cdot 10^4$	**14.0** $\cdot 10^0$

Type *Tx1*. While the other two types of transactions exhibit similar tendencies and number of aborts, the number of aborts in transactions of Type *Tx1* drops by more than 75 % when using the *sc* monitoring approach. Using the more intrusive monitoring approaches then increases the number of aborts.

6.4 Analysis of the Impact of Heavyweight Monitoring

It is hard to explain all the above presented changes in the behaviour of the monitored TM programs since, for that, one would typically need some additional information about their original behaviour. However, gathering such information is usually impossible without monitoring and hence without again changing the behaviour.

Nevertheless, the situation is a bit different for the specific case when one wants to analyse differences between what happens within lightweight and heavyweight monitoring. In this case, the environment used for heavyweight monitoring has more influence on the behaviour than the actual collection of information about the monitored program. Hence, one may come with a hypothesis why the behaviour changes in a certain way in heavyweight monitoring and then try to support the hypothesis by analysing differences of suitable data collected about the behaviour of the monitored program during lightweight and heavyweight monitoring processes. We follow this path below.

Our hypothesis why the behaviour of the monitored TM programs changes so significantly during heavyweight monitoring is as follows. The run-time environment used in heavyweight monitoring has to execute not only the code of the monitored program but also the monitoring code that collects desired information about the execution of the program as well as other essential code for managing the running threads, for determining when and where to execute the monitoring code, etc. As a result, there is more code to be executed inside each transaction block, but there is even more code to be executed outside of the transactions. This, of course, influences the timing of the transactions as their execution is moved further apart in the program's execution, and even though their execution is longer, their chances to overlap and possibly abort are decreased. This phenomenon is illustrated in Figure 6.1 (where an abort of a transaction within the normal execution is highlighted in red hatching).

To support the above hypothesis, we computed how much time is spent inside and outside the transactional blocks (using recorded timestamps of starts, aborts, and commits of transactions). The results are shown in Table 6.7. One can clearly see that the relative

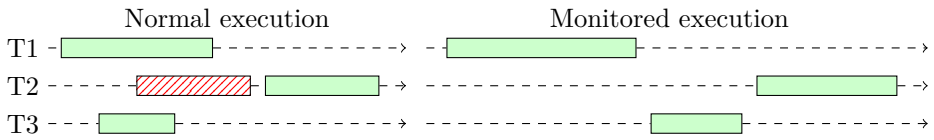


Figure 6.1: Differences between normal and monitored execution.

Table 6.7: Average percentage of time spent in transactions.

		genome	intruder	kmeans		ssca2	vacation		yada
	variant			high	low		high	low	
Light	el-a-ts	45.4%	71.6%	33.1%	26.9%	50.8%	96.2%	95.4%	89.0%
	el-arw-ts	60.3%	95.3%	78.6%	75.0%	63.8%	99.0%	98.9%	97.2%
Heavy	el-a-ts	13.9%	15.6%	8.1%	6.3%	3.4%	29.7%	27.8%	56.3%
	el-arw-ts	24.9%	29.9%	22.7%	23.4%	5.0%	65.1%	61.7%	74.1%

time spent inside transactions is much lower when using heavyweight monitoring than when using lightweight monitoring. This confirms our hypothesis and explains why we get significantly less aborts during heavyweight monitoring. Moreover, the table also shows that when we start registering transactional reads and writes, we spend more time in transactions, and, correspondingly, we also get more aborts (cf. Table 6.3).

6.5 Conclusion

We have presented several approaches of lightweight and heavyweight monitoring of TM programs. The proposed monitoring techniques are publicly available and can be used directly or serve as an inspiration for implementing other specialized monitors. We have also presented an experimental evaluation of the influence of these monitoring approaches on the number of aborts, both at the global level and for each type of transactions present in the monitored programs. Further, we have shown that not only the monitoring process influences the number of aborts, but also the environment in which the monitoring is performed has a great impact on the overall behaviour.

From our experiments, we concluded that when using lightweight monitoring strategies, the more information we monitor the less aborts we usually get, both globally and per transaction type as well. However, one has to be careful of the role of outliers and of the fact that the number of aborts does not decrease in the same way across different types of transactions. Moreover, sometimes, the number of aborts can even increase when we increase the amount of monitoring. Such a behaviour is easily observed when the environment used causes a massive initial drop in the number of aborts. This is, in particular, visible when using environments for heavyweight monitoring.

In the future, it would be interesting to find analytical explanations for the various phenomena observed during the experiments reported in this paper. Such explanations could then perhaps be used as a basis for finding means for neutralizing the influence of the monitoring approaches on the monitored runs. Furthermore, one can use the developed monitoring layer as a basis for developing various dynamic analyses allowing one to detect errors in the monitored programs.

Chapter 7

Contracts for Concurrency

This chapter describes how to detect errors in multi-threaded programs using the so-called contracts for concurrency. Contracts for concurrency define how functions should be used in a concurrent setting so that they do not interfere with each other. Besides extending the basic notion of contracts for concurrency from the literature with data and contextual information, the chapter proposes two novel methods for their dynamic validation. These methods were able to unveil previously unknown errors in several programs from the industry.

7.1 Introduction

The divide-and-conquer strategy is frequently applied to the development of large software products where the whole application is divided into interacting software modules, collaboratively developed by multiple teams. Objects in object-oriented programming languages are an example of such software modules. Accessing the services provided by a software module requires one to follow a protocol that includes: (i) the syntax of the service, i.e., the name of the service and the type of its input and output parameters; (ii) the semantics of the service, i.e., the expected behavior of the service for a given set of input parameters; and (iii) the service access restrictions, e.g., the domain of the valid values for each parameter, dependency relations between services, atomicity requirements for execution in a concurrent setting, etc.

Violating the protocol of a service may cause all sorts of misbehaviors—from subtle, perhaps admissible but wrong results to fault-stop fails, such as exceptions and segmentation faults. Compilers take good care of Aspect (i) of the protocol, i.e., syntax validation. Aspect (ii), service semantics, although not verified by compilers, is usually at least documented. Aspect (iii), service access restrictions, is usually not verified by compilers nor documented, which results in a deep dependency on programmers' clairvoyance on the usage of the services—in particular, when concurrency issues are involved.

In this chapter, we aim at reducing the above problem by addressing Aspect (iii), i.e., service access restrictions, for the context of *concurrent (multi-threaded)* programs. In particular, we address restrictions of using services provided by software modules in a concurrent setting with the aim of avoiding atomicity violations and similar concurrency-related errors. *Atomicity violations* (see Chapter 2) are a class of errors which result from an incorrect definition of the scope of an atomic region. Such errors are usually hard to localise and diagnose, which becomes even harder when using (third-party) software libraries where it is

unknown to the programmer how to form the atomic regions correctly when accessing the library. Even new synchronisation techniques, such as transactional memories discussed in the previous chapter, designed to ease the process of writing concurrent programs, do not entirely avoid this problem and suffer from atomicity violations as well [39].

One way to address the problem of proper atomicity is to associate a *contract* with each program module/library and then check whether the contract is indeed respected. In fact, the notion of contract is, in general, not restricted to concurrent programs. In the general case, a contract [116] regulates the use of methods of an object by specifying a set of pre-conditions the program must meet before calling the object methods. For the particular case of concurrent programs, Sousa et al. proposed in [142] the concept of the so-called *contracts for concurrency*. A contract for concurrency is a particular case of a software protocol that allows one to enumerate sequences of public methods of a module that are required to be executed atomically. Contracts may be written by the software module/library developer or inferred automatically from the program (based on its typical usage patterns) [142].

In this chapter, assuming that the appropriate contracts for concurrency have been obtained, we propose two methods for dynamically verifying that such contracts are respected at program run time. In particular, the first method belongs among the so-called *lockset-based* dynamic analyses whose classic example is the Eraser algorithm for data race detection [137] and whose common feature is that they track sets of locks that are held by various threads and used for various synchronization purposes. The tracked lock sets are used to extrapolate the synchronization behaviour seen in the witnessed test runs, allowing one to warn about possible errors even when they do not directly appear in the witnessed test runs. We have implemented our approach in a prototype tool, and we present experimental results obtained with our implementation.

While the lockset-based method works well in many cases, it may produce both false positives and negatives. Some of these problems are caused by the method itself as lockset-based methods are imprecise in general. However, many of the problems are caused by the limitations of the (basic) contracts which does not allow one to precisely describe which situations are errors and which not. To address this problem, we extended the notion of contracts for concurrency by allowing them to reflect both the *data flow* between the methods (in that a sequence of method calls only needs to be atomic if they manipulate the same data) and the *contextual information* (in that a sequence of method calls needs not be atomic wrt all other sequences of methods but only some of them). Then, we propose a method for dynamic validation of contracts based on the *happens-before relation* which utilises *vector clocks* in a way optimized for contract validation. This method does not suffer from false alarms and supports the extended contracts. We implemented this method using the ANaConDA framework and obtained promising experimental results, including discovery of previously unknown errors in large real-world programs.

Plan of the chapter. The rest of the chapter is organised as follows. In Section 7.2, we discuss related work. In Section 7.3, we present the notion of contracts for concurrency. In Section 7.4, we present a simple method for dynamic validation of contracts based on locksets and discuss its advantages and disadvantages. In Section 7.5, we provide experimental results showing that even the simple method can detect many contract violations. In Section 7.6, we extend the basic contracts to consider the data flow and/or the contextual information of method calls. In Section 7.7, we describe a more advanced method for dynamic validation of contracts based on the happens-before relation which supports the extended contracts. We also provide results of experiments with this method showing

that it is able to detect a broader variety of concurrency-related errors, not only atomicity violations. Section 7.8 concludes the chapter.

7.2 Related Work

Design by contract was introduced by Meyer [117] as a way to write robust code, using contracts between programs and objects, checked at runtime. In this context, a contract consists of a pre- and post-condition of a method such that when the call of a method satisfies its pre-condition, the post-condition is guaranteed to be satisfied upon return from the method.

Cheon et al. [28] proposed a way of using contracts to specify protocols for accessing objects in a sequential setting. The contracts use regular expressions describing sequences of calls that can be executed for a given object. Hurlin [79] extended [28] with operators allowing one to specify which methods may be executed concurrently. The work, however, does not show how to validate such contracts, it only proposes a technique for automatically generating programs from contracts that are to be proven correct (e.g., by theorem proving) to show that the contracts adhere to the protocols they specify.

The basic contracts for concurrency that we are building on here first appeared in [141], where the authors propose a static approach which can formally prove that no contract violation is possible. For that, however, they assume that properly handled contracts must appear in code blocks declared as atomic (with the atomicity assured by the run-time support). If a different way of guarding the contracts is used, a false alarm is issued. Moreover, the approach scales to relatively small programs only. For more complex programs, one has to restrict the analysis to program fragments, e.g., individual methods, in order to achieve a reasonable performance. This leads to a loss of precision as contracts may span across several methods and thus be missed by the analysis.

Another problem with the static approach is related to the fact that contracts for concurrency are required to operate atomically only when all the involved method calls operate on the same object. This is a natural requirement since the atomic execution is critical only when working with data elements that are mutually related, which is assumed to be reflected in that they are stored within one object. However, static validation does not have precise information on which objects the methods are called on. Hence, calls of methods on different objects are mixed together, leading to possible false alarms. Classic alias and escape analyses can be used to infer this information from the source code of the program, but these analyses provide only approximate information and may still lead to false alarms.

Both of our approaches for dynamic contract validation avoid the above false alarms since they have precise run-time information about the objects that particular methods are executed on. Moreover, they also scale quite well. On the other hand, despite the use of locksets and happens-before relation to extrapolate the behaviour of the witnessed test runs, these approaches can miss some contract violations that do not happen in the witnessed test runs nor they can be deduced from the locking patterns or other synchronisation used in these traces. In order to minimize the number of possibly missed contract violations, one can combine our approaches with noise injection techniques [52] that maximize the number of thread interleavings witnessed in a set of test runs.

ICFinder [103] uses static analysis to automatically infer which pairs of calls to a module are incorrect. This is achieved by identifying and applying two common incorrect composition patterns: one capturing stale value errors and the other one trying to infer correlations

between method calls by analyzing the CFG of the client's program. These patterns are extremely broad and yield many false positives. The authors address this issue by filtering the results from the static analysis with a dynamic analysis that only considers violations defined in [149]. This analysis assumes that the notion of atomic set was correctly inferred by ICFinder.

In [22, 135], *typestates* are used to specify protocols for accessing objects. A typestate can describe both the legal sequences of method calls and the data these methods may work with. In [22], the protocol must be defined by the user and then validated using three static analyses. If these analyses cannot establish correctness of the program, dynamic analysis is used to find protocol violations. In [135], a dynamic analysis is used to automatically infer protocols from program runs and then static analysis is used to check the protocols. All the protocols, however, do not consider concurrency-related issues. Beckman et al. [16] showed how to use *typestates* in concurrent scenarios. Their approach, however, requires the user not only to define the protocols to be checked, but also to annotate the code with additional information needed by the static checker to check if the protocols are respected. Typestate specifications are also much more complex compared with the specifications based on contracts we propose in this chapter.

The work [112] deals with JavaMOP specifications of desired program properties that are validated dynamically at runtime. Using the approach, one can specify that some sequence of methods must be atomic, but the specific way of ensuring the atomicity (e.g., the fact that some lock must be held) has to be encoded by the user in the specification. On the other hand, when our contracts are used for checking atomicity, the user just specifies the sequence of method calls and does not have to care about the way the atomicity should be ensured.

Leaving the specific area of contracts and focusing in general on finding errors in concurrent programs, most works have concentrated on detecting data races and deadlocks (as we have indeed discussed in previous chapters too). These errors are, however, of a different nature than those captured by contracts, and hence methods and tools developed for detecting them—including well-known ones, such as, Eraser [137], RaceTrack [164], GoldiLocks [45], FastTrack [60], or GoodLock [72]—cannot be used for contract violation detection.

Significantly less works targeted detection of various kinds of atomicity violation [58, 109, 152], including different forms of high-level data races [10, 38, 51] or stale value errors [11, 24, 38]. Detectors based on access patterns to shared variables [106, 149], type systems [25], semantic invariants [35], and dynamic analysis [58, 63, 161] have been proposed for detecting this kind of errors. Despite atomicity violation is closer to contract violation, contract violation is still more general. This is, atomicity violations can be detected as contract violations (possibly with a need to view accesses to variables as method calls) but not vice versa. An example of an error that can be captured via contract validation but not atomicity validation is that of order violation. Such an error happens in the Link Manager, described in Section 7.7.3, where a shared queue is used before it is initialised. As the queue (variable) is accessed only once in each of the threads and both accesses are guarded by the same lock, it is neither an atomicity violation nor a data race, and yet we were able to detect it.

7.3 Contracts for Concurrency

A *contract for concurrency* [141] (or simply *contract* herein) is a protocol for accessing public services of a module, i.e., the methods of its public API, expressing which of the

methods are correlated and should be executed in the same atomic context (wrt its API usage) if applied on the same computational object. Therefore, a program that conforms to a contract is guaranteed to be safe from atomicity violations.

Formally, let $\Sigma_{\mathbb{M}}$ be a set of all public method names (the API) of a software module (or library). A *contract* is a set $\mathbb{R}$ of *clauses* where each clause $\varrho \in \mathbb{R}$ is a regular expression over $\Sigma_{\mathbb{M}}$. A contract violation occurs if any of the sequences represented by the contract clauses is interleaved with an execution of methods from $\Sigma_{\mathbb{M}}$ over the same object.

Example. Consider the `java.util.ArrayList` implementation of a resizable array of the Java standard library, and, for simplicity, take the following subset of the available methods: `add(obj)`, `contains(obj)`, `indexOf(obj)`, `get(idx)`, `set(idx, obj)`, `remove(idx)`, and `size()`. The below clauses belong to the contract for the `ArrayList` library:

(ϱ_1) `contains indexOf`
(ϱ_2) `indexOf ( set | remove | get )`
(ϱ_3) `size ( remove | set | get )`
(ϱ_4) `add ( get | indexOf )`

Clause ϱ_1 states that the execution of `contains()` followed by `indexOf()` should be atomic. Otherwise, the program may confirm the existence of an object in the array but fail to obtain its index as a concurrent thread can, e.g., remove the object. Clause ϱ_2 represents a similar scenario where the index of an object is obtained and then the index is used to modify the object. Without atomicity, a concurrent change of the array may shift the position of the object and cause malfunction. Clause ϱ_3 deals with programs that verify whether a given index is in a valid range (e.g., `index < size()`) and then access the array. To ensure `size()` is still valid when accessing the array, the calls must execute atomically. Clause ϱ_4 represents a scenario where an object is added to the array and then the program tries to obtain information about it by querying the array. Without atomicity, the object may no longer exist or its position in the array may have shifted.

Another relevant clause in the contract of `ArrayList` is:

(ϱ_5) `contains indexOf ( set | remove )`

However, the contract's semantic already enforces this clause since it results from the composition of clauses ϱ_1 and ϱ_2.

7.4 Lockset-based Dynamic Validation of Contracts

In this section, we propose a dynamic approach to check whether a contract is violated or not. Our dynamic validation looks for contract violations based on concrete program executions. Possible violations not witnessed during the execution of the program may be missed, but all of the methods encountered during the execution are taken into account, and so contract violations caused by method calls from all over the program are detected. Since all of the threads are running and all objects are known when the program is executing, we know precisely whether all of the methods called in a sequence use the same object, and we do not report any false alarms due to mixing calls on different objects as is common in static analysis.

Since we look for contract violations based on concrete executions, we can avoid some false alarms, but on the other hand, we can miss some errors. In order to minimise this possibility, we employ one of the dynamic analysis techniques—namely, the *lock sets* [137]—to

extrapolate the actually witnessed behaviour and hence detect possible contract violations even when they were not actually witnessed. Moreover, we utilise noise injection techniques [52] (see Chapter 5) to enforce synchronisation scenarios, which normally appear only rarely, leading to behaviours (and possibly contract violations) that would not be covered by extrapolation of the common synchronisation scenarios only.

7.4.1 Detection of Contracts

In order to validate a contract, we first need to detect the sequences it contains in the execution of a program. To do that, we encode each contract, i.e., all of its sequences, as a single finite state automaton. As each clause of the contract represents a regular expression, we use standard methods for transforming (star-free) regular expressions[1] into finite automata to perform the conversion and then merge all these automata into a single one. The transitions of the automaton represent method calls and the accepting states represent situations where a contract sequence was detected.

Each thread manages a list of finite state automata instances which represent the currently encountered incomplete contract sequences. Whenever a method $m \in \Sigma_{\mathbb{M}}$ is encountered, we try to advance each of these instances using the method m. If we cannot advance the instance, the contract sequence is invalid and we discard it. If we successfully advanced the instance to the next state, call it q, we check if q is an accepting state. If yes, a contract sequence is detected; if not, we leave the instance in q and go on. Moreover, we check if we can advance any of the finite state automata from their starting state using the method m. If yes, then the beginning of another contract sequence was detected and we create a new instance of the automaton which will monitor the execution of this contract sequence to check if it can be accepted or not.

7.4.2 Checking the Atomicity Condition

When a contract sequence is detected, the next step is to check if the atomicity condition is met, i.e., if the program ensures that all methods of this contract sequence are executed atomically. The static approach does this by checking if all of the methods of the detected contract sequence are enclosed in code blocks declared as atomic, which can be done by analysing the source code of the program.

We propose a lockset-based method, inspired by [137], to perform these checks which is more suited for dynamic analysis. This method checks if at least one lock is held during the execution of a contract sequence by monitoring the lock acquisitions and releases during the execution of a contract sequence. If this condition is not satisfied, i.e., no locks are held throughout the execution, then the contract is being violated.

The method works online, i.e., it performs the contract validation during the execution of a program, and is based on the analysis state $\sigma = (A, H, R)$ where:

- $A : \mathbb{T} \to 2^{\mathbb{L}}$ records the set of locks acquired by a thread.
- $H : \mathbb{T} \times \mathbb{R} \to 2^{\mathbb{L}}$ records the set of locks held by a thread when a contract sequence starts.
- $R : \mathbb{T} \times \mathbb{R} \to 2^{\mathbb{L}}$ records the set of locks released by a thread during the execution of a contract sequence.

[1] Star-free regular expressions are used in the static contract validation approach [142]. We can, however, easily generalize our approach to general regular expressions.

In the initial analysis state, all sets of locks are empty, reflecting that at the beginning of the execution, no locks are held by any thread, i.e., $\sigma_0 = (\emptyset, \emptyset, \emptyset)$. Fig. 7.1 shows rules according to which the analysis state is updated for each operation of the target program.

The rule [Contract sequence start] records that a thread t is starting an execution of a contract sequence ϱ by remembering the locks which are currently held by the thread. It also clears the set of locks released by the thread as no locks could have been released yet.

The rule [Contract sequence end] records that a contract sequence ϱ was detected in a thread t and checks the atomicity condition by comparing the set of locks held when the contract sequence started its execution with the set of locks released during its execution. If at least one lock was held all the time the contract sequence was executed, the contract is valid, and no error is issued. If all locks held at the beginning of the execution of the contract sequence were released before its execution finished, a contract violation is reported.

[Contract sequence start]

$$\frac{H' := H_t[\varrho := A_t] \quad R' := R_t[\varrho := \emptyset]}{(A, H, R) \Rightarrow^{seq\_start(t,\varrho)} (A, H', R')}$$

[Contract sequence end]

$$\frac{\texttt{if } H_t(\varrho) \setminus R_t(\varrho) = \emptyset \texttt{ then ERROR}}{(A, H, R) \Rightarrow^{seq\_end(t,\varrho)} (A, H, R)}$$

[Lock acquired]

$$\frac{A' := A[t := A_t \cup \{m\}]}{(A, H, R) \Rightarrow^{acq(t,m)} (A', H, R)}$$

[Lock released]

$$\frac{\forall \varrho \in \mathbb{R} : R' := R_t[\varrho := R_t(\varrho) \cup \{m\}]}{(A, H, R) \Rightarrow^{rel(t,m)} (A, H, R')}$$

Figure 7.1: Analysis rules.

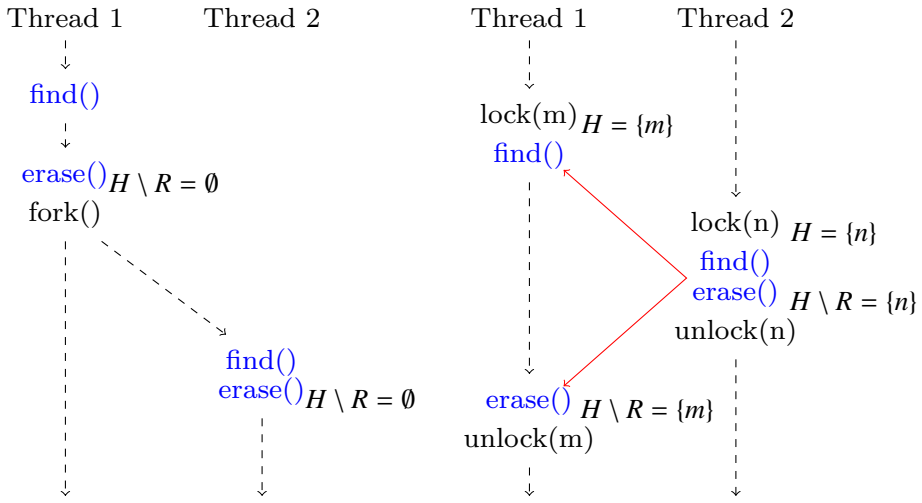
The rule [Lock acquired] records that a thread t acquired a lock m, and it updates the set of locks currently acquired by this thread. Finally, the rule [Lock released] records that a thread t released a lock m, and it updates the set of locks released by the thread for each contract sequence currently executed by this thread.

7.4.3 Discussion of the Proposed Approach

The above method may produce both false positives (i.e., false alarms) as well as false negatives. False positives may be caused by the fact that not guarding an execution of a contract sequence with a single lock throughout its entire duration does not mean that it will not be executed atomically. Take the situation shown in Fig. 7.2(a) as an example. Not a single one of the contract sequences is guarded by a lock, yet there is no contract violation as the synchronisation ensures that the contract sequence in `Thread 1` is always executed before the contract sequence in `Thread 2`. Therefore there is no interference between these two contract sequences, and hence no contract violation. Yet the method reports both of the contract sequences being violated.

False negatives may happen since holding a lock when executing a contract sequence does not always ensure that no other thread interferes with it. Take the situation in Fig. 7.2(b) as an example. The executions of the contract sequence in both `Thread 1` and `Thread 2` are guarded by a lock. However, these locks are different and thus the execution of the contract sequence in `Thread 1` may be interleaved with the execution of the contract sequence in `Thread 2`, violating the contract sequence in `Thread 1`. Yet the method does not report any error.

To solve the above problems, we need to take into account thread interleavings. When guarding the same contract sequence with two different locks in two different threads, we should issue an error only when these two threads may interleave each other. Conversely,



(a) An atomically executed contract reported as errorneous.

(b) A not reported contract violation.

Figure 7.2: Examples of situations where the contract validation fails.

when a contract sequence is not guarded by a lock, we should report an error only when this thread may be interleaved by another thread executing the same contract sequence. When using a static approach such as [141], this information is hard to obtain as one would need to infer it from the source code of the program where the scheduling of threads is unknown. On the other hand, the dynamic approach actually sees the concrete thread interleavings, and so it is easier to get the needed information. Unfortunately, the lockset method does not work with it in any way. Moreover, incorporating this information into the lockset method would be counterproductive as it would kill the extrapolation which increases chances to detect errors. A way to go here seems to be a use of dynamic analysis based on the *happens-before relation* as used, e.g., in the GoldiLock data race detector [45]. This is the direction that we follow in Section 7.7.

7.5 Experiments

This section presents an experimental comparison of the proposed dynamic validation of contracts with the static approach of [142]. To compare the approaches, we implemented the method described in Section 7.4 as a plug-in for the IBM Concurrency Testing Tool (ConTest) [43]. The ConTest infrastructure provides a fully automatic Java byte-code instrumentation and a listeners architecture that facilitated the implementation of the proposed method as well as execution and dynamic analysis of the benchmarks.

The comparison of the static and dynamic approaches is done using a subset of the small benchmark programs which were previously used to evaluate the static approach [142], namely, the Account, Allocate Vector, Arithmetic DB, Jigsaw, Store, and Vector fail test cases. All these benchmark programs had to be slightly modified in order to allow us to execute them and use ConTest to analyse their runs. Namely, we did the following modifications by hand: (1) test arguments were provided if missing; (2) infinite loops (which are not a problem for the static approach, but cannot be present during a dynamic analysis) were transformed to finite loops with a small number of iterations to avoid infinite

Table 7.1: An experimental comparison of static and dynamic contract validation.

Program			*Static analysis*			*Dynamic analysis*		
Benchmark	LOC	Contract Clauses	Duration (sec.)	CFG Nodes	Detected Violations	Duration (sec.)	Detected Violations	Failed Assertions
Account	68	2	0,041	158	2	0,011	2	0,96
Allocate Vector	167	1	0,120	882	1	0,099	1	0,00
Arithmetic DB	325	2	0,272	2256	2	0,010	2	0,06
Jigsaw	147	1	0,044	125	1	0,009	1	0,44
Store	769	1	0,090	559	1	0,303	1	1,00
VectorFail	100	2	0,048	244	2	0,009	2	0,09

executions; (3) exceptions generation and handling (commented out due to limits of the static approach) were uncommented; (4) the `Atomic` annotations preferred by the static approach were turned back to `synchronized` blocks; and (5) all assertions and correctness checks already present in the test cases were extended to send notifications to our ConTest plug-in. The dynamic analysis tests were executed on a Linux machine with an i5-4200M CPU (i.e., comparable with the machine used to evaluate the static approach in [142]), running Linux 3.16, and OpenJDK 1.6 JVM.

Table 7.1 summarises results of the comparison between our dynamic approach and the static approach of [142]. The table is divided into three sections. In the leftmost part, basic characteristics of the benchmark programs are provided. In particular, the test case name, the number of effective lines of the original Java code (without our modifications, which added only a few extra lines of code), and the number of contract clauses for the benchmark program as manually identified by the authors of the static approach.

The middle part of Table 7.1 characterizes results of the static analysis obtained in [142]. Namely, the average analysis duration in seconds is provided with the number of control flow graph (CFG) nodes generated and processed. Finally, the number of detected violations of contract clauses for each benchmark program is shown.

The rightmost part of the table shows the average results (from 1000 test executions) obtained with our dynamic approach. In particular, the average execution time of the instrumented test in seconds is provided, followed by the average number of detected violations of contract clauses. Finally, the average ratio of failed assertions (usually implemented as conditions checking memory consistency) provided by the authors of the tests is reported. The standard deviations of the execution times as well as failed assertions were quite low. The standard deviation of the number of violated contract was zero (i.e., the algorithm always detected all the violations).

Concerning both the considered static as well as dynamic approach, there are two interesting aspects we would like to emphasize: (i) the ability of both approaches to detect contract violations; and (ii) the very low execution time taken by both approaches (of course, a further evaluation on larger test cases remains to be done).

In both approaches, all violations were *always* correctly reported. Such a good result of the dynamic approach depends on the quality of the test that executes the problematic part of the code and on the ability of the lockset approach to extrapolate other behaviours from the witnessed execution, and therefore to detect possible violations even from executions where the problem did not occur. This can, in particular, be demonstrated on the Allocate Vector, Arithmetic DB, and VectorFail benchmark programs where the assertion-based detection reported the problem in less than 10 % of executions while the dynamic approach

```
void replace(int a, int b) {
    if (array.contains(a)) {
        int idx=array.indexOf(a);
        array.set(idx,b);          } }
```

Figure 7.3: Example of atomicity violation with data dependencies.

always detected a possible violation.

Let us now get back to the time consumed by the analyses. In both cases, the analysis itself took less than one second for the considered simple test programs. However, there was a significant difference in the overhead of the underlying infrastructures. The initialisation of the static approach within the Soot analysis environment [142] took nearly 40 seconds for each benchmark. The dynamic approach was much faster. The bytecode instrumentation took about 0.5 seconds. The slowdown of the test execution was within 5 % because only the *method entry* and *lock operation* events were instrumented (i.e., most of the code was executed with no instrumentation and hence no overhead).

7.6 Extending Contracts for Concurrency

As it turns out, the basic definition of contracts for concurrency, introduced in Section 7.3, is sometimes quite restrictive and can classify valid concurrent programs as unsafe. Hence, in this section, we propose two extensions that improve the expressiveness of contracts: one extends them with parameters, making it possible to consider the data flow between method calls; and the other adds contextual information that restricts the situations in which atomicity shall be enforced.

7.6.1 Extending Contracts with Parameters

Figure 7.3 illustrates a situation where basic contracts may be too restrictive. It shows a procedure that replaces item `a` in an array by item `b`. The procedure contains two atomicity violations: (i) item `a` needs not exist anymore when `indexOf` is called; and (ii) the index obtained may be outdated when `set` is executed. A basic contract of Section 7.3 could cover this situation by a clause (ϱ_6) `contains indexOf set`. However, the given sequence needs to be executed atomically only if `contains` and `indexOf` have the same argument, and the result of `indexOf` is used as the first argument of `set`.

To express in a contract how the flow of data influences the dependencies between methods, we extend the contract specification by considering *method call parameters* and *return values*, expressed as *meta-variables*. Then, if a contract should be enforced only if the same object appears as an argument or as the return value of multiple calls in the given call sequence, we may express that by using the same meta-variable at the position of all the concerned parameters and/or return values.

Clause ϱ_6 may then be refined as follows—in particular, note the repeated use of meta-variables `X`/`Y`, requiring the same objects o_1/o_2 to appear at the positions of `X`/`Y`, resp.: (ϱ_6') `contains(X) Y = indexOf(X) set(Y,_)`. Here, the underscore is a free meta-variable that imposes no restrictions.

Example. With the above extension, it is possible to refine the contract for the standard

library `java.util.ArrayList` as follows:

(ϱ'_1) `contains(X) indexOf(X)`
(ϱ'_2) `X = indexOf(_) ( remove(X) | set(X, _) | get(X) )`
(ϱ'_3) `X = size() ( remove(X) | set(X, _) | get(X) )`
(ϱ'_4) `add(X) ( get(X) | indexOf(X) )`

This contract captures in detail the dependencies between method calls, expressing the relations that are problematic, excluding those that do not constitute atomicity violations.

7.6.2 Extending Contracts with Spoilers

Interleaving a sequence of calls listed in a contract clause with some methods of the given API may lead to an atomicity violation, while this is not the case for other methods. This is, however, not reflected in the basic contracts. For example, the clause `contains indexOf` states that this sequence of calls must always be executed atomically (wrt methods of the given module), regardless of which methods the other threads are executing. Interleaving a thread executing this sequence with another one is thus a contract violation regardless of whether the other thread executes `remove` or `get`, not distinguishing that the former is harmful while the latter not.

To cope with the above, we propose to augment contracts with *contextual information*, allowing one to express in which context the contract clauses shall be enforced. For that, each clause of the basic contract (now called a *target*) will be coupled with a set of *spoilers* that restrict its application. A spoiler represents a set of sequences of methods that may violate its target. Client programs must then ensure that each target is executed atomically wrt its spoilers, whenever executed on the same object. For the target clause `contains indexOf`, a possible spoiler is `remove`, and the extended clause would be: `contains indexOf ⇜ remove`.

Formally, as before, let $\mathbb{R}$ be the set of *target* clauses where each target $\varrho \in \mathbb{R}$ is a regular expression over $\Sigma_{\mathbb{M}}$. Let $\mathbb{S}$ be the set of *spoilers* where each spoiler $\sigma \in \mathbb{S}$ is a regular expression over $\Sigma_{\mathbb{M}}$. We also define the alphabets $\Sigma_{\mathbb{R}} \subseteq \Sigma_{\mathbb{M}}$ and $\Sigma_{\mathbb{S}} \subseteq \Sigma_{\mathbb{M}}$ for the methods used in the targets or spoilers, respectively.

A *contract* is then a relation $\mathbb{C} \subseteq \mathbb{R} \times \mathbb{S}$ which defines for each target the spoilers that may cause atomicity violations. Note that one target may be violated by more than one spoiler and also one spoiler may violate more than one target. A contract is violated if any sequence represented by a target $\varrho \in \mathbb{R}$ executed on the same object o is fully interleaved with an execution of the sequence representing its spoiler $\sigma \in \mathbb{C}(\varrho)$ on the object o. A target sequence r is fully interleaved by a spoiler sequence s if the execution of r starts before the execution of s and the execution of s ends before that of r.[2]

Example. The basic contract for `java.util.ArrayList` with spoilers extending it with contextual information is below:

(ϱ''_1) `contains indexOf ⇜ remove`
(ϱ''_2) `indexOf (remove | set | get) ⇜ remove | add | set`
(ϱ''_3) `size (remove | set | get) ⇜ remove`
(ϱ''_4) `add indexOf ⇜ remove | set`

[2]Partial interleavings of targets and spoilers are not considered to cause an error. If they do, this can be handled by adding a new contract clause (target) whose spoiler is the appropriate fraction of the original spoiler.

This contract explicitly captures which interferences are harmful and which interleavings shall be forbidden. All other interleavings, not captured by spoilers, are considered safe.

Finally, the extension of contracts with spoilers can be combined with the extension with parameters, allowing one to define fine-grained atomicity requirements for the methods of a module. This can be illustrated by the below clause:

`contains(X) indexOf(X)` ⇜⇝ `remove(_)`.

This clause requires sequences of `contains` and `indexOf` to be executed atomically but only when executed over the same object, when dealing with the same item `X`, and only wrt concurrent execution of `remove`. This captures the fact that any concurrent removal may lead to an atomicity violation, by either removing object `X` or by altering its position in the array. Note that `add` is not a spoiler since it does not interfere with the position of `X` as elements are added to the end of the array.

7.7 Happens-before-based Dynamic Contract Validation

We now propose a *dynamic contract validation* method for contracts with contextual information (i.e., using both targets and spoilers) as defined in Section 7.6.2. Though not discussed here in detail, the method can be easily extended to support parameters by considering separate instances of target/spoiler pairs for different values of parameters (as done in our implementation). The method was published in [37] together with a complementary method for static validation of contracts. While the static approach can formally prove that no contract violation is possible, it does not scale well, supports contracts with parameters only, and produces many false alarms. As the method for static validation of contracts is not part of the contribution of this thesis, it is not discussed in this chapter in more detail.

Below, we first formalize a notion of multi-threaded program traces used as the input of our analysis. Then we define the happens-before relation that captures the ordering of events in program traces. Next, we describe our method for detecting contract violations. Finally, we provide results of experiments with a prototype implementation of the approach.

7.7.1 Preliminaries

For the below, we fix a set of threads $\mathbb{T}$, a set of targets $\mathbb{R}$, a set of spoilers $\mathbb{S}$, a set of contracts $\mathbb{C} \subseteq \mathbb{R} \times \mathbb{S}$, and a set of locks $\mathbb{L}$. We consider program traces in the form of sequences of events of the following types: a thread entering/exiting a method, a thread acquiring/releasing a lock, and a thread forking/joining another thread. Since each of the events can appear multiple times in a trace, we assume the events to be indexed by their position in the trace. However, we do not take the indices into account when looking for matches of the regular expressions of targets/spoilers in a trace. We denote the set of all events that can be generated by a thread $t \in \mathbb{T}$ as $\mathbb{E}_t$, and let $\mathbb{E} = \cup_{t \in \mathbb{T}} \mathbb{E}_t$. Then, a *trace* is a sequence $\tau = e_1 \dots e_n \in \mathbb{E}^+$. We let $e_i \in \tau$ denote that the event e_i is present in the trace τ. By $start(t)/end(t)$, we denote the first/last event generated by a thread t.

Given a trace $\tau = e_1 \dots e_n \in \mathbb{E}^+$, we call its sub-sequence $r = e_{i_1} e_{i_2} \dots e_{i_k}$, $1 < k \leq n$, an *instance* of a target $\varrho \in \mathbb{R}$ iff (1) r consists of well-paired method enter/exit events executed by a thread $t \in \mathbb{T}$, (2) when restricted to the enter events only, r matches the regular expression of ϱ (if ϱ contains stars, the longest possible matches are considered only), and (3) apart from the events e_{i_1}, ..., e_{i_k} there is no event from the alphabet of ϱ executed by t between the indices i_1 and i_k in τ. Intuitively, an instance of a target can

interleave with events that are not its part, but only if they are outside of its alphabet. For instance, for a target $\varrho = abc$ and a trace $\tau = aabdc$, there is an instance of ϱ between indices 2 and 5 but not between 1 and 5. We denote by $e_i \in r$ that the event e_i is present in the target instance r. We let $start(r) = e_{i_1}$ and $end(r) = e_{i_k}$ denote the first/last event of r, respectively. We let $[\varrho]^\tau$ be the set of all instances of a target $\varrho \in \mathbb{R}$ in a trace τ and $[\mathbb{R}]^\tau = \cup_{\varrho \in \mathbb{R}}[\varrho]^\tau$ be the set of all instances of all targets from $\mathbb{R}$ in τ.

Likewise, we define the notion of an instance s of a spoiler $\sigma \in \mathbb{S}$ in a trace τ, its beginning/end events $start(s)/end(s)$, respectively, the set $[\sigma]^\tau$ of all instances of σ in τ, and the set $[\mathbb{S}]^\tau = \cup_{\sigma \in \mathbb{S}}[\sigma]^\tau$ of all instances of all spoilers from $\mathbb{S}$ in τ.

A *happens-before relation* $\prec_{hb}$ over a trace $\tau = e_1 \dots e_n \in \mathbb{E}^+$ is the smallest transitively-closed relation on the set $\{e_1, ..., e_n\}$ of events in τ such that $e_j \prec_{hb} e_k$ holds whenever $j < k$ and one of the following holds: (i) Both events e_j and e_k are performed by the same thread (program order). (ii) Both events e_j and e_k acquire or release the same lock. (iii) One of the events e_j and e_k is a fork/join of a thread u in a thread t and the other is executed by u (fork-join synchronization). If two indices in a trace are not related by a happens-before relation, then the corresponding events are considered to be *concurrent*.

A contract $(\varrho, \sigma) \in \mathbb{C}$ is *violated* in a trace τ iff there is a target instance $r \in [\varrho]^\tau$ and a spoiler instance $s \in [\sigma]^\tau$ s.t. $start(s) \not\prec_{hb} start(r) \wedge end(r) \not\prec_{hb} end(s)$. Intuitively, the contract (ϱ, σ) is violated in τ if there are instances r/s of ϱ/σ, resp., where r may start before s and end after s, i.e., the target instance can be fully interleaved with the spoiler instance.

7.7.2 On-the-Fly Dynamic Contract Validation

If the entire trace is available, dynamic contract validation is easy. For all possibly conflicting instances of targets and spoilers, one simply checks whether a target is fully interleaved with a spoiler or not, i.e., $\forall(\varrho, \sigma) \in \mathbb{C}, \forall r \in [\varrho]^\tau, \forall s \in [\sigma]^\tau$ checks if $start(s) \not\prec_{hb} start(r) \wedge end(r) \not\prec_{hb} end(s)$ is satisfied. If it is, an error is reported.

However, this approach is not very practical. It scales poorly with the size of the trace, which can be huge. In some cases, e.g., for reactive programs, the trace can even be infinite. To address this problem, we propose an on-the-fly dynamic contract validation algorithm which does not require the whole trace to be available and yet guarantees that if a contract is violated in the trace, this will be detected.

Trace Windows

A crucial concept for our on-the-fly dynamic contract validation is the concept of a *trace window*, providing a gradually moving, partial view of the trace. Formally, a trace window υ is a subsequence of the trace τ. While, in the extreme case, the trace window may actually contain the entire trace, the goal is to keep it as small as possible. Later, we show that there is a maximum number of events that we need to keep in the window in order not to miss any error and that this number grows only with the number of targets and spoilers, not with the size of the trace.

We denote by $[\varrho]^\upsilon$ the set of all instances of a target $\varrho \in \mathbb{R}$ in a window υ and by $[\mathbb{R}]^\upsilon = \cup_{\varrho \in \mathbb{R}}[\varrho]^\upsilon$ the set of all instances of all targets from $\mathbb{R}$ in υ. In the same manner, we define the set $[\sigma]^\upsilon$ of all instances of spoiler $\sigma \in \mathbb{S}$ in υ and the set $[\mathbb{S}]^\upsilon = \cup_{\sigma \in \mathbb{S}}[\sigma]^\upsilon$ of all instances of all spoilers from $\mathbb{S}$ in υ.

We move events into the trace window υ as soon as they occur. However, in order for the window not to grow indefinitely, we also have to remove some events from it. We define the $\upsilon \rightarrow e$ operation which removes e from υ. We also generalize this operation for

instances of targets/spoilers. The $\upsilon \to r$ operation removes all events from $r \in [\mathbb{R}]^\upsilon$ from υ provided they do not belong to another currently tracked instance of a target or spoiler, i.e., $\forall e_i \in r : \upsilon \to e_i \iff (\forall x \in [\mathbb{R}]^\upsilon \cup [\mathbb{S}]^\upsilon, x \neq r : e_i \notin x) \wedge (\forall x \in [\mathbb{R}]^\tau \cup [\mathbb{S}]^\tau, start(x) \in \upsilon \wedge end(x) \notin \upsilon : e_i \notin x)$. Likewise, we define the $\upsilon \to s$ operation that removes all events from $s \in [\mathbb{S}]^\tau$ from υ. As we show below, one can discard events corresponding to some of the older spoiler and target instances when newer ones appear in the window. The conditions allowing us to discard such instances are safe in that at least one instance of a violation of each target by each spoiler is always reported. However, if there are multiple occurrences of the conflict, just one is guaranteed to be preserved.

Discarding Spoilers

First, we aim at reducing the number of spoiler instances in a trace window. We say that discarding a spoiler instance s (i.e., removing this particular instance from the current trace window and not considering it in further contract violation detection) is *safe* iff whenever a contract violation can be detected using s, it can be detected without s too. The below lemma shows that, under some natural assumptions, reflected in our analysis, an instance s_1 of a spoiler σ can be safely discarded from the window provided the window contains a newer instance of the spoiler σ, i.e., an instance s_2 that started later than s_1.

In particular, we assume that events appear in the window υ as soon as they appear in the trace τ. Moreover, we assume that as soon as an instance r of a target ϱ appears in the window υ, i.e., $r \in [\varrho]^\upsilon$ becomes true, r is checked for contract violation against all instances s of all spoilers $\sigma \in \mathbb{C}(\varrho)$ conflicting with the given target ϱ that appear in the window υ, i.e., $s \in [\sigma]^\upsilon$. Then the following holds.

Lemma 1. *Let $s_1, s_2 \in [\sigma]^\upsilon$ be instances of a spoiler $\sigma \in \mathbb{S}$ present in a window υ of a trace τ. If s_1 started before s_2, i.e., $start(s_1) \prec_{hb} start(s_2)$, it is safe to discard s_1 from υ.*

Proof of Lemma 1. By contradiction. Assume there is a trace τ with a window υ and two spoiler instances $s_1, s_2 \in [\sigma]^\upsilon$ of a spoiler $\sigma \in \mathbb{S}$ where $start(s_1) \prec_{hb} start(s_2)$, and it is not safe to discard s_1 from υ. Then, there must be a contract $(\varrho, \sigma) \in \mathbb{C}$ and an instance $r \in [\varrho]^\tau$ of the target ϱ in the trace τ s.t. s_1 violates r, but s_2 does not violate r. This means that the following conditions must hold: (1) $start(s_1) \not\prec_{hb} start(r) \wedge end(r) \not\prec_{hb} end(s_1)$ since s_1 violates r. (2) $start(s_2) \prec_{hb} start(r) \vee end(r) \prec_{hb} end(s_2)$ since s_2 does not violate r.

Consider first that $start(s_2) \prec_{hb} start(r)$ holds. Then, since s_1 and s_2 are such that $start(s_1) \prec_{hb} start(s_2)$, we get $start(s_1) \prec_{hb} start(r)$. However, this contradicts the first condition above, which requires that $start(s_1) \not\prec_{hb} start(r)$ holds.

Hence, it must be the case that $end(r) \prec_{hb} end(s_2)$ holds. This means that r must appear in υ before s_2 appears in there. However, then, a contract violation is detected before s_1 is removed from the window, and once a contract violation has already been detected, any further optimization is safe (in fact, the analysis can be stopped once a contract violation is detected).

□

Using Lemma 1 and the fact that spoiler instances in a single thread are ordered wrt $\prec_{hb}$, we can prove the below lemma that limits the number of spoiler instances to be preserved.

Lemma 2. *Let $T = \{\ t \in \mathbb{T} \mid start(t) = e_l \Rightarrow l \leq j\ \}$ be the set of threads that started before the end of a window $\upsilon = e_i \dots e_j$. For each thread $t \in \mathbb{T}$ and for each spoiler $\sigma \in \mathbb{S}$, we need to preserve just the last instance of σ in υ running within t.*

Proof of Lemma 2. Take any thread $t \in T$ and any spoiler $\sigma \in \mathbb{S}$. Since the definition of spoiler instances rules out overlapping of spoiler instances within a particular thread, if the set of instances of σ that appear in υ within t is not empty, we can order these instances into a sequence $s_1, \ldots, s_n$ such that $\mathit{start}(s_i) \prec_{hb} \mathit{start}(s_j)$ for any $1 \leq i < j \leq n$. Then, by Lemma 1, it suffices to preserve just the last spoiler instance s_n in the sequence.

□

Discarding Targets

We now aim at reducing the number of target instances, which turns out to be more challenging than for spoilers. We say that discarding a target instance r is *safe wrt a spoiler instance* s iff whenever a contract violation between r and s can be detected, then a conflict between s and some other target instance r' can be detected too. Note that, unlike in the case of spoilers, discarding a target instance is defined as safe wrt a given spoiler instance and not in general.

First, Lemma 3 shows that, given instances r_1 and r_2 of a target ϱ where r_1 ends before r_2 starts, r_1 can be safely discarded wrt any spoiler instance that (i) has not even started before the end of the window or that (ii) started even before r_1.

Lemma 3. *Let $\upsilon = e_i \ldots e_j$ be a window of a trace τ with two instances $r_1, r_2 \in [\varrho]^\upsilon$ of a target $\varrho \in \mathbb{R}$ such that $\mathit{end}(r_1) \prec_{hb} \mathit{start}(r_2)$. It is safe to discard r_1 wrt any instance $s \in [\sigma]^\tau$ of a spoiler $\sigma \in \mathbb{S}$ forming a contract with ϱ, i.e., $(\varrho, \sigma) \in \mathbb{C}$, whenever either (i) s starts behind the window υ, meaning that if $\mathit{start}(s) = e_l$, then $j < l$, or (ii) s starts before r_1 starts, i.e., $\mathit{start}(s) \prec_{hb} \mathit{start}(r_1)$.*

Proof of Lemma 3. By contradiction. Assume that there is a trace τ with a window $\upsilon = e_i \ldots e_j$, two target instances $r_1, r_2 \in [\varrho]^\upsilon$ of a target $\varrho \in \mathbb{R}$ such that $\mathit{end}(r_1) \prec_{hb} \mathit{start}(r_2)$, and it is not safe to discard a spoiler instance $s \in \mathbb{C}(\varrho)$ despite it is the case that either (i) s starts behind the window υ, i.e., if $\mathit{start}(s) = e_l$, then $j < l$, or (ii) s starts before r_1, i.e., $\mathit{start}(s) \prec_{hb} \mathit{start}(r_1)$.

Then, there must be some instance $s \in [\sigma]^\tau$ of a conflicting spoiler $\sigma \in \mathbb{C}(\varrho)$ that appears in the trace τ and that is violated by r_1 but not r_2. For the spoiler instance s, the following conditions must hold: (1) $\mathit{start}(s) \not\prec_{hb} \mathit{start}(r_1) \wedge \mathit{end}(r_1) \not\prec_{hb} \mathit{end}(s)$ because s violates r_1. (2) $\mathit{start}(s) \prec_{hb} \mathit{start}(r_2) \vee \mathit{end}(r_2) \prec_{hb} \mathit{end}(s)$ because s does not violate r_2.

Clearly, if it is the case that s starts before r_1, i.e., $\mathit{start}(s) \prec_{hb} \mathit{start}(r_1)$, we immediately have a contradiction with Condition (1), which requires $\mathit{start}(s) \not\prec_{hb} \mathit{start}(r_1)$. Hence, assume that s starts only after the end of the window.

Next, assume that $\mathit{end}(r_2) \prec_{hb} \mathit{end}(s)$ holds. Since it is moreover the case that $\mathit{end}(r_1) \prec_{hb} \mathit{start}(r_2)$ holds, and the program order guarantees that $\mathit{start}(r_2) \prec_{hb} \mathit{end}(r_2)$, $\mathit{end}(r_1) \prec_{hb} \mathit{end}(s)$ holds too. However, this contradicts with Condition (1), which requires that $\mathit{end}(r_1) \not\prec_{hb} \mathit{end}(s)$ holds.

Hence, it must be the case that $\mathit{start}(s) \prec_{hb} \mathit{start}(r_2)$ holds. However, this means that s must start in υ, which is a contradiction.

□

Next, we consider the case when an instance s of a spoiler σ is running at the end of the window υ, there are two instances r_1 and r_2 of the same target ϱ conflicting with σ, r_1

ends before r_2 starts, but s does not start before r_1 and r_2. Lemma 4 shows that, in this case, discarding r_1 is safe wrt s.

Lemma 4. *Assume a window υ of a trace τ with two target instances $r_1, r_2 \in [\varrho]^\upsilon$ of a target $\varrho \in \mathbb{R}$ s.t. $end(r_1) \prec_{hb} start(r_2)$. Let $s \in [\sigma]^\tau$ be an instance of a spoiler $\sigma \in \mathbb{S}$ that forms a contract with ϱ, i.e., $(\varrho, \sigma) \in \mathbb{C}$, it is running at the end of υ, i.e., $start(s) \in \upsilon$ but $end(s) \notin \upsilon$, and it has not started before the given target instances, i.e., $start(s) \not\prec_{hb} start(r_2)$. Then discarding r_1 is safe wrt s.*

Proof of Lemma 4. By contradiction. Assume that there is a trace τ with a window υ, two target instances $r_1, r_2 \in [\varrho]^\upsilon$, and an instance $s \in [\sigma]^\tau$ of a spoiler $\sigma \in \mathbb{C}(\varrho)$ such that $start(s) \in \upsilon$, $end(s) \notin \upsilon$, and $start(s) \not\prec_{hb} start(r_2)$. Further, assume that $end(r_1) \prec_{hb} start(r_2)$, and yet discarding r_1 from υ is not safe wrt s.

It is not safe to remove r_1 from υ wrt s iff r_1 can be violated by s while r_2 cannot. Then, the following conditions must hold: (1) $start(s) \not\prec_{hb} start(r_1) \wedge end(r_1) \not\prec_{hb} end(s)$ because s violates r_1. (2) $start(s) \prec_{hb} start(r_2) \vee end(r_2) \prec_{hb} end(s)$ because s does not violate r_2.

First, assume that $end(r_2) \prec_{hb} end(s)$ holds. Since $end(r_1) \prec_{hb} start(r_2)$ holds, and the program order guarantees that $start(r_2) \prec_{hb} end(r_2)$, we get $end(r_1) \prec_{hb} end(s)$. However, this contradicts with Condition (1) above, which requires that $end(r_1) \not\prec_{hb} end(s)$ holds.

Hence, it must be the case that $start(s) \prec_{hb} start(r_2)$ holds. However, this contradicts with the assumption of the lemma that $start(s) \not\prec_{hb} start(r_2)$.

□

Since we check each spoiler instance against all target instances that are currently in the trace window as soon as the spoiler instance gets into the window, we can prove the below upper bound on the number of target instances to be preserved. Intuitively, by Lemma 3, one instance is kept wrt all not yet started and—on the other hand—old but still running spoiler instances. Further, by Lemma 4, one instance per thread in which a newer spoiler instance is running is to be preserved.

Lemma 5. *Let $T_1 = \{\ t \in \mathbb{T} \mid start(t) = e_l \Rightarrow l \leq j\ \}$ be the threads that started before the end of a window $\upsilon = e_i \dots e_j$, and let $T_2 = \{\ t \in T_1 \mid end(t) = e_l \Rightarrow l > j\ \}$ be the threads running at the end of υ. For each thread in T_1 and each target $\varrho \in \mathbb{R}$, we need to preserve at most $|T_2| + 1$ instances of ϱ.*

Proof of Lemma 5. Take any thread $t \in T_1$ and any target $\varrho \in \mathbb{R}$. Due to immediate checks of conflicts between any target instance in the window and any spoiler instance that appears in the trace window, from the point of view of preserving target instances, we care about their possible conflicts with only those spoiler instances that have not yet terminated or that have not even started yet.

The definition of target instances implies that we do not have to consider overlapping target instances within particular threads. Therefore, if the set of instances of ϱ that appear in υ within the thread t is not empty, we can order these instances into a sequence $r_1, \dots, r_n$ such that $end(r_i) \prec_{hb} start(r_j)$ for any $1 \leq i < j \leq n$.

Further, the definition of spoiler instances implies that we do not have to consider overlapping instances of spoilers running within a single thread. Hence, there can be at most $|T_2|$ running instances of spoilers from $\mathbb{S}$ at the end of υ: one instance in each thread of T_2. For each of them, Lemma 4 may be applicable to a subsequence of the target instances $r_1, \dots, r_m$, $m \leq n$. These subsequences may differ just in the value of m. Lemma 4 allows us

to preserve just the last instance r_m. However, since the value of m can be different for each of the subsequences, we may end up preserving $|T_2|$ target instances, one for each spoiler instance running at the end of υ.

Next, if Lemma 4 is not applicable wrt some running spoiler instance s to a suffix $r_{m+1}, \ldots, r_n$, $m \geq 0$, of the sequence of target instances running within t in υ because $start(s) \prec_{hb} r_{m+1}$, Lemma 3 allows us to preserve just r_n. The same r_n is to be preserved for each running spoiler instance. Moreover, due to Lemma 3, it is also enough to preserve r_n with respect to all spoiler instances that have not yet started. Hence, we get the upper bound of $|T_2| + 1$ target instances.

□

Vector Clocks and Further Optimizations

Next, as a further optimization, we will first introduce an application of *vector clocks* for efficiently tracking information about the happens-before relation between the spoiler/target instances that are (or were) in the current trace window. Essentially, instead of remembering the entire sequence of events forming a target/spoiler instance, we will remember the vector clocks of their start and end only. Keeping just these two vector clocks is sufficient as we need to know the happens-before relation only between the starts and ends of conflicting target/spoiler instances. Next, from Lemma 5, we know that we need to track—in the worst-case—for each thread and for each target, one instance of the target for each thread in which some potentially conflicting spoiler instance is running (a consequence of Lemma 4) plus one further instance for all other running or not yet started spoiler instances (a consequence of Lemma 3). We will propose an optimisation which will allow us to preserve, for each thread t and each target ϱ, the vector clocks of both the beginning and end just for the last instance of ϱ in t only. For the other instances required to be tracked by Lemma 4, we will remember the vector clock of their end only.

In general, a *vector clock* $VC : \mathbb{T} \to \mathbb{N}$ contains a clock value for each thread $t \in \mathbb{T}$ recorded at a certain point. In particular, we maintain, for each $t \in \mathbb{T}$, a vector clock $\mathbb{C}_t$ whose entries $\mathbb{C}_t(u)$ record, for each $u \in \mathbb{T}$, the clock value of the last operation of u that happens before the current operation of t. The t-component of this vector clock then represents the clock of the thread t. It is incremented at each lock release or fork operation. Next, we maintain a vector clock $\mathbb{L}_l$ for each lock $l \in \mathbb{L}$. These vector clocks are updated on synchronisation operations that impose a happens-before order of operations from different threads in a way described in [60].

Further, we assign to each event $e \in \tau$ executed by a thread $t \in \mathbb{T}$ a vector clock VC_e. This vector clock is set to the value of $\mathbb{C}_t$ when e is encountered in the execution of the program. It can then be determined whether an event e_t executed in the thread t happens before an event e_u executed in a thread u, i.e., $e_t \prec_{hb} e_u$, by checking whether $VC_{e_t}(t) \leq VC_{e_u}(t)$.

To allow for checking the conditions determining if a contract was violated or not, it now suffices to record the vector clocks of the start and end of the spoiler and target instances that are to be kept in the window wrt Lemmas 1, 3, and 4.

Moreover, for the target instances r to be remembered according to Lemma 4, i.e., those for which there is some running spoiler instance s that can collide with r, we can reduce the amount of stored information even further as follows. Instead of storing the vector clocks of the beginning and end of each target instance r of the above kind that appears in some thread t, we proceed as follows: (1) We remember in which threads u there are running spoiler instances s satisfying the first condition of contract violation

wrt r, i.e., $start(s) \not\prec_{hb} start(r)$. (2) We remember the time when r ends its execution, i.e., $VC_{end(r)}(t)$, which is needed to check the second condition of contract violation, i.e., $end(r) \not\prec_{hb} end(s)$, once s ends. Both of these pieces of information can be remembered by maintaning a mapping $PV_t^{\varrho,\sigma} : \mathbb{T} \to \mathbb{N}$ for the thread $t \in \mathbb{T}$, the target $\varrho \in \mathbb{R}$ whose instance r is, and the spoiler $\sigma \in \mathbb{S}$ whose instance s is. Namely, for each thread u containing a spoiler instance s satisfying the first condition of contract violation, we may set $PV_t^{\varrho,\sigma}(u)$ to $VC_{end(r)}(t)$, while setting the other entries of $PV_t^{\varrho,\sigma}$ to 0.[3]

Using the above, when a spoiler instance s finishes its execution in a thread t, it suffices to check $PV_u^{\varrho,\sigma}(t)$ for each thread u other than t (as we do not consider conflicts within a single thread).[4] If the value is not 0, we know that the first condition of contract violation between s and the target instance r that ran in the thread u that we remebered through $PV_u^{\varrho,\sigma}$ only was satisfied. Then, by checking $PV_u^{\varrho,\sigma}(t) \leq VC_{end(s)}(u)$, we can determine if a violation occurred or not.

Method Description

We now summarise our optimized on-the-fly contract violation detection. Most of it is done by Algorithm 2 at method exit events. Algorithm 2 handles both conflicts between the latest, so far fully remembered spoiler and target instances (lines 3, 11) as well as between newly finished spoiler instances and older target instances partially remembered via $PV_t^{\varrho,\sigma}$ (lines 12–13). Algorithm 2 also discards older target/spoiler instances r'/s' (lines 7, 9) and maintains the $PV_t^{\varrho,\sigma}$ mapping (line 6). The latter is done by recording the above described data about an older target instance r' that can still collide with some running spoiler instance s according to Lemma 4, which is tested on lines 4–6, before r' is removed from the window.

Apart from the above, at an entry to a method, we perform recognition of target/spoiler instances. That is done using finite automata for recognising sequences of events matching the regular expressions representing the corresponding targets/spoilers, respectively. New runs through the automata may be initiated at each event, and, at the same time, an attempt to extend all so-far unfinished runs is done (if such a run cannot be extended via the current event and the event belongs to the alphabet of the concerned automaton, the run is discarded). When an exit from a method is encountered, a check is performed to see whether some of the runs has reached an accepting state (this will then be recognised via the $end(r)/end(s)$ predicates on lines 1/8 of Algorithm 2).

7.7.3 Implementation and Experiments

We implemented the above approach extended to distinguish values of one parameter by tracking different target/spoiler instances for its different values. We used the ANaConDA framework [55] to monitor method calls and synchronization events in running C/C++ programs. ANaConDA also provides us with heuristic *noise injection*. As discussed in the

[3] By setting $PV_v^{\varrho,\sigma}(u)$ to $VC_{end(r)}(t)$, we remember both that the first condition of contract violation has been satisfied between r and s and the time when r ended. The time is remembered multiple times for possibly different threads u, but we tolerate this for the sake of obtaining uniform data structures. Since the space needed to store $PV_t^{\varrho,\sigma}$ corresponds to that of a vector clock, and we have a single $PV_t^{\varrho,\sigma}$ instead of two vector clocks for each target instance that needs to be remembered according to Lemma 4, we save up to $2 \cdot |T_2| - 1$ vector clocks where T_2 is the set of currently running threads.

[4] The meaning of the threads is swapped here wrt the previous paragraph in order to have the explanation in line with the code in Fig. 2.

Algorithm 2: Contract violation detection at method exit.

Data: window υ, event $e \in \mathbb{E}$ generated by thread $t \in \mathbb{T}$

1 **if** $\exists \varrho \in \mathbb{R}, r \in [\varrho]_t^\upsilon : e = end(r)$ **then** // Target ended
2 **for** $\sigma \in \mathbb{C}(\varrho), u \in \mathbb{T} : u \neq t$ **do**
3 **if** $\exists s \in [\sigma]_u^\upsilon : start(s) \not\prec_{hb} start(r) \wedge end(r) \not\prec_{hb} end(s)$ **then** r is violated by s ;
4 **if** $\exists s \in [\sigma]_u^\tau : start(s) \in \upsilon \wedge end(s) \notin \upsilon$ **then**
5 **if** $start(s) \prec_{hb} start(r)$ **then**
6 **if** $\exists r' \in [\varrho]_t^\upsilon : r' \neq r \wedge start(s) \not\prec_{hb} start(r')$ **then** $PV_t^{\varrho,\sigma}(u) = VC_{end(r')}(t)$;
7 **if** $\exists r' \in [\varrho]_t^\upsilon : r' \neq r$ **then** $\upsilon \rightarrow r'$;
8 **if** $\sigma \in \mathbb{S}, s \in [\sigma]_t^\upsilon : end(s) = e$ **then** // Spoiler ended
9 **if** $\exists s' \in [\sigma]_t^\upsilon : s' \neq s$ **then** $\upsilon \rightarrow s'$;
10 **for** $\varrho \in \mathbb{C}(\sigma), u \in \mathbb{T} : u \neq t$ **do**
11 **if** $\exists r \in [\varrho]_u^\upsilon : start(s) \not\prec_{hb} start(r) \wedge end(r) \not\prec_{hb} end(s)$ **then** r is violated by s ;
12 **if** $PV_u^{\varrho,\sigma}(t) \neq 0 \wedge PV_u^{\varrho,\sigma}(t) \leq VC_{end(s)}(u)$ **then**
13 an instance of ϱ is violated by s;

Table 7.2: Validation results for dynamic analysis.

Benchmark	T/S pairs	Contract Violations	False Positives	Potential AV	Real AV	SLOC	Time (s)
Coord03 [10]	8	380	0	0	380	116	1.01
Coord04 [11]	4	24	0	0	24	53	0.52
Local [10]	4	2	0	0	2	27	0.52
NASA [10]	1	100	0	0	100	96	0.60
Account [152]	1	176	0	0	176	54	0.53
Link Manager	2	1	0	0	1	1.5K	1.14
Chromium-1	2	2	0	0	2	7.5M	49.12

previous text, this can increase the number of witnessed interleavings and hence chances to see an interleaving from which our analysis can deduce that a contract violation is possible. We thus use two orthogonal methods to find rare concurrency-related bugs: noise injection and extrapolation based on the happens-before relation. Moreover, we use a specific kind of noise tailored for the given purpose. In particular, we inject noise before the last method of each target instance which prolongs its execution and increases chances to encounter a spoiler instance capable of interleaving the target instance and causing a contract violation.

We tested our implementation on a set of small benchmarks with known atomicity violations as well as two real-world programs, Link Manager and Chromium-1. The small programs were taken from [10, 11, 152] and were also used to evaluate the static validation method [37] (we used a C++ version as close as possible to the Java version).

Link Manager is a component of a cloud-connected thermostat used for managing parallel task processing (we were not allowed to identify the company developing it). A *manager* thread is issuing tasks to *executor* threads, which send results of the assigned tasks back to

the *manager* through a shared queue. Our tool was used in the early stages of development of this program, and it uncovered an order violation error that happened when an *executor* sent the result of its task before the *manager* initialised the queue used to transfer the data. This caused the *manager* to wait forever for the task to be finished. One of the contracts we checked required that the queue cannot be used before it is initialised, i.e., no `send` or `receive` can occur between the start of the *manager* and the initialisation of the queue. The error occurred very rarely, so normal tests were unable to detect it. Our tool, however, was able to detect the error, and it was then promptly fixed.

Chromium-1 is a program from the RADBench benchmark [80], an older version of the Chrome browser (version 6.0.472.35) containing a known atomicity violation leading to an assertion failure. As this error can be described using a contract, we tried our tool to find the error. The experiment was successful, showing that our tool can handle even large programs. Interestingly, to find the error without the on-the-fly approach, one would need to store a trace with more than 17 million method calls (about 1.6 GB of data) while the on-the-fly method needed about 10 MB of data only.

Table 7.2 provides results of experiments with our dynamic approach. The *T/S Pairs* column gives the number of target/spoiler pairs considered. The column *Contract Violations* gives the number of instances of such pairs found violated.[5] The column *False Positives*, which was included for compatibility with the results of static contract validation as presented in [37], contains zeros only as, unlike the static approach, the dynamic one considers solely executable sequences of method calls (indeed, they were seen to execute). The column *Potential AV* contains numbers of detected contract violations that need not stay real if the values of more than one parameter per contract are taken into account (which is not yet supported in our tool). The column contains zeros only showing that we sufficed with tracking a sole parameter in all our experiments.[6] The column *Real AV* gives numbers of contract violations guaranteed to be real as they used at most one parameter, and our tool was thus able to distinguish the needed instances. Finally, the columns *SLOC* and *Time* give the numbers of lines of the considered programs and the analysis time.

The results show that our approach can be used to find real errors in real-world programs. Moreover, it can be used to detect not only atomicity violations, but also order violations which are hard to be found using exiting techniques.

7.8 Conclusion

We proposed two approaches for dynamic validation of contracts in concurrent code, one based on locksets and the other on happens-before relation. When compared with previously proposed static approaches, our approaches can suppress some of the false alarms produced by the static approaches, and they are also more scalable. On the other hand, since we build on observing concrete runs, our approaches can miss some errors that would not be missed by the static approaches.

To detect as many contract violations as possible, the first approach employs a lockset-based extrapolation of the synchronization behaviour observed in performed test runs,

[5] Compared with the static approach [37], we look for contract violations in the *execution* of a program, not its source code. As the code containing a contract violation may be executed repeatedly, we can detect (and report) the same contract violation many times. The static approach reports it only once.

[6] We tried an experiment in which we tracked no parameter values at all. Then, for Chromium-1, our tool reported 14 potential violations instead of the 2 real ones, showing that distinguishing target/spoiler instances is important.

which allows the method to warn about possible contract violations even when they were not seen in a concrete execution. Unfortunately, this extrapolation can suffer from both false positives and negatives due to the fact that it does not utilise any information about thread interleavings. In order to solve this problem, we have further introduced a second approach based on the happens-before relation, which does reflect thread interleavings, and we extended the basic notion of contracts for concurrency with arguments and spoilers, each of the extensions allowing one to describe contracts more precisely.

We have evaluated both approaches on a set of simple as well as real-world programs. While the lockset-based approach can detect atomicity violations only, the happens-before-based approach can also detect errors such as order violations. Moreover, it produces no false alarms. In addition, both of these approaches may use noise injection to increase the number of observed thread interleavings and hence to increase the chances to see interleavings containing a contract violation or at least symptoms that such a violation is possible.

There are many possibilities for future work. For instance, while it is conceptually easy to support contracts with both arguments and spoilers in the dynamic approach, this can be rather costly in practice due to many target and spoiler instances to be tracked. Suitable optimisations are thus likely needed. Further, it seems promising to combine the dynamic approach with the existing static approaches—e.g., by letting some static approach to drive the dynamic one to likely problematic code. More involved ways of automatically deriving contract candidates are also an interesting issue for further work.

Chapter 8

Conclusion

The main goal of the thesis is to advance the research in the area of detecting errors in multi-threaded programs. Despite the fact that a plenty of detection techniques were invented over the years, many companies still have a hard time finding tools which can detect errors they are encountering in the programs they are developing. There are several main reasons why they cannot use the techniques already available, the most common ones being that the tools implementing the techniques do not support the programs they have, the programs contain types of errors the techniques are unable to detect, or the techniques just cannot handle larger software. While it is impossible to create a technique which would be universal and effective at the same time, developing practical techniques usable in the industry is always welcome, especially nowadays where multi-threaded programs can be found even in the smallest devices.

While the most common types of concurrency errors already have a lot of techniques for their detection, detection methods for other kinds of errors are clearly lacking in this area. It is not that these types of errors are so rare that it is not worth to deal with them. Actually, many software developers encounter these errors more often than they would like. It is mostly that these types of errors are more complex, and thus it is much harder to develop techniques to detect them.

This thesis contributes to the research in the area of detecting concurrency errors in several ways. It starts with improving existing techniques by combining dynamic analysis and (bounded) model checking, exploiting their strengths and suppressing their weaknesses. The idea is to reduce the state space to be searched by a (bounded) model checker using information provided by a dynamic analysis. While the resulting technique is much more efficient and precise than either of the two utilised techniques, it still cannot handle large real-world programs where even the reduced state space is too big.

Another contribution is a proposal of improved noise injection which allows one to combine several noise placement and noise seeding heuristics in a single run of a program. This improvement further increases the chances of noise injection to uncover rare executions containing errors and thus makes any technique whose detection capabilities rely on seeing such executions more efficient. Besides improving the noise injection in general, the thesis also introduces several new noise injection heuristics which can be used separately or combined together.

The third contribution is the development of the ANaConDA framework which allows one to analyse C/C++ programs on the binary level. While many of the existing detection techniques are applicable on C/C++ programs, there are only few implementations of these techniques for this class of programs. The goal of this framework is to ease the

development of tools for analysing C/C++ programs. The framework also supports various noise injection techniques that can be used to increase the efficiency of any technique implemented using this framework.

Beside allowing one to analyse C/C++ programs, the framework was later extended to support transactional memory, a new kind of synchronisation which starts to be used more and more. The next contribution of the thesis is the discussion of various caveats of monitoring programs using transactional memory (which is required for their analysis) and a usage of noise injection techniques for fixing the behaviour of such programs (as the behaviour is often changed because of the monitoring).

The last contribution is the invention of a brand new technique for detecting several kinds of concurrency errors, namely the well-known atomicity violations and also the less studied order violations and missed signals. All of these errors may be described using the so called contracts for concurrency and then checked using the new technique. The technique also works very well with the noise injection techniques, it fact, a slightly modified version of one of the previously invented noise injection techniques is used here to greatly increase the efficiency of the detection technique.

The results were published in proceedings of Eurocast 2011 and 2015, RV 2011 and 2012, PADTAD 2012, and MEMICS 2014, and in the special issue of the STVR journal focusing on concurrency. A part of the content of Chapter 7 is currently under review at the ICST 2017 conference. Moreover, they have been implemented in the ANaConDA framework. A paper describing this framework also won the best tool paper award at the RV 2012 conference. The framework has been tested in collaboration with industrial partners (we were not allowed to mention them explicitly) and various errors were found in their programs.

As for future work, we discussed it already in the different chapters. Let us highlight just some general directions here. As there is no silver bullet heuristic when using noise injection, it is always important to pick up (and combine) the right heuristics and choose the right parameters for them. So developing new noise injection heuristics, e.g., tailored for specific concurrency errors, detection techniques, or classes of programs, and inventing (automatic) approaches for determining the best parameters for them, e.g., by utilising search techniques, generic algorithms, or data mining, is more than helpful. With the ANaConDA framework, it is easier than ever to implement the existing dynamic analyses for C/C++ programs or to quickly develop new analyses and test how they work in practice. There are also many ways how to extend the framework. To mention a few, one can add a support for analysing multi-process programs (in addition to multi-threaded programs), for handling additional kinds of synchronisation such as RCU (read-copy-update), or for extracting a more detailed information about the code being analysed.

Bibliography

[1] Power Framework Delay Fuzzing. Online at: http://msdn.microsoft.com/en-us/library/hh454184(v=vs.85).aspx, April 2013.

[2] 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.

[3] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. In *SIGMOD'93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 207–216, New York, NY, USA, 1993. ACM.

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

[5] Gregory R. Andrews. *Concurrent programming: principles and practice.* Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1991.

[6] Tadashi Araragi and Seung Mo Cho. Checking liveness properties of concurrent systems by reinforcement learning. pages 84–94, 2007.

[7] 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.

[8] Cyrille Artho and Armin Biere. Applying Static Analysis to Large-Scale, Multi-Threaded Java Programs. In *Proc. of ASWEC'01*, pages 68–76, Washington, DC, USA, 2001. IEEE.

[9] Cyrille Artho and Armin Biere. Applying static analysis to large-scale, multi-threaded java programs. In *Proceedings of the 13th Australian Conference on Software Engineering*, ASWEC '01, pages 68–. IEEE Computer Society, 2001.

[10] Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. *Software Testing, Verification and Reliability*, 13(4):207–227, December 2003.

[11] Cyrille Artho, Klaus Havelund, and Armin Biere. Using block-local atomicity to detect stale-value concurrency errors. *Automated Technology for Verification and Analysis*, pages 150–164, 2004.

[12] Renata Avros, Vendula Dudka, Bohuslav Křena, Zdeněk Letko, Hana Pluháčková, Shmuel Ur, Tomáš Vojnar, and Zeev Volkovich. Boosted decision trees for behaviour mining of concurrent programs. In *Proceedings of MEMICS'14*, pages 15–27. NOVPRESS s.r.o., 2014.

[13] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

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

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

[16] Nels E. Beckman, Kevin Bierhoff, and Jonathan Aldrich. Verifying correct usage of atomic blocks and typestate. *SIGPLAN Not.*, 43(10):227–244, October 2008.

[17] Yosi Ben-Asher, Yaniv Eytani, Eitan Farchi, and Shmuel Ur. Noise makers need to know where to be silent - producing schedules that find bugs. In *ISOLA'06*, pages 458–465, Washington, DC, USA, 2006. IEEE Computer Society.

[18] Yosi Ben-Asher, Eitan Farchi, and Yaniv Eytani. Heuristics for Finding Concurrent Bugs. In *Proc. of IPDPS'03*, pages 288–296, Washington, DC, USA, 2003. IEEE.

[19] Saddek Bensalem and Klaus Havelund. Dynamic Deadlock Analysis of Multi-threaded Programs. In *Proc. of PADTAD'05*, volume 3875 of LNCS, pages 208–223, Berlin, Heidelberg, 2005. Springer-Verlag.

[20] Johann Blieberger, Bernd Burgstaller, and Robert Mittermayr. Static detection of livelocks in ada multitasking programs. In *Ada-Europe'07: Proceedings of the 12th international conference on Reliable software technologies*, pages 69–83, Berlin, Heidelberg, 2007. Springer-Verlag.

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

[22] Eric Bodden and Laurie Hendren. The clara framework for hybrid typestate analysis. *International Journal on Software Tools for Technology Transfer*, 14(3):307–326, 2012.

[23] Jeremy S. Bradbury and Kevin Jalbert. Defining a catalog of programming anti-patterns for concurrent java. In *Proc. of the 3rd International Workshop on Software Patterns and Quality (SPAQu'09)*, pages 6–11, Oct. 2009.

[24] M. Burrows and K.R.M. Leino. Finding stale-value errors in concurrent programs. *Concurrency and Computation: Practice and Experience*, 16(12):1161–1172, 2004.

[25] Luís Caires and João C. Seco. The type discipline of behavioral separation. *SIGPLAN Not.*, 48(1):275–286, January 2013.

[26] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *Proc. of IISWC'08*, 2008.

[27] Marcio Castro, Kiril Georgiev, Vania Marangozova-Martin, Jean-Francois Mehaut, Luiz Gustavo Fernandes, and Miguel Santana. Analysis and Tracing of Applications Based on Software Transactional Memory on Multicore Architectures. In *Proc. of PDP'11*. IEEE CS, 2011.

[28] Yoonsik Cheon and Ashaveena Perumandla. Specifying and checking method call sequences of java programs. *Software Quality Control*, 15(1):7–25, March 2007.

[29] 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.

[30] 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.

[31] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[32] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Comput. Surv.*, 3:67–78, June 1971.

[33] Byron Cook, Alexey Gotsman, Andreas Podelski, Andrey Rybalchenko, and Moshe Y. Vardi. Proving That Programs Eventually Do Something Good. In *Proc. of POPL'07*. ACM, 2007.

[34] Crispin Cowan, Heather Hinton, Calton Pu, and Jonathan Walpole. The cracker patch choice: An analysis of post hoc security techniques. In *Proceedings of the 23rd National Information Systems Security Conference*. USENIX Association, 2000.

[35] R. Demeyer and W. Vanhoof. A framework for verifying the application-level race-freeness of concurrent programs. In *22nd Workshop on Logic-based Programming Environments (WLPE 2012)*, page 10, 2012.

[36] Jyotirmoy Deshmukh, E. Allen Emerson, and Sriram Sankaranarayanan. Symbolic deadlock analysis in concurrent libraries and their clients. In *ASE'09: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 480–491, Washington, DC, USA, 2009. IEEE.

[37] Ricardo F. Dias, Carla Ferreira, Jan Fiedor, João M. Lourenço, Aleš Smrčka, Diogo G. Sousa, and Tomáš Vojnar. Verifying concurrent programs using contracts. Technical report, 2016. http://www.fit.vutbr.cz/~vojnar/Publications/tr-contracts-16.pdf.

[38] Ricardo J. Dias, Vasco Pessanha, and JoÃŁo M. LourenÃ§o. Precise detection of atomicity violations. In *Hardware and Software: Verification and Testing*, Lecture Notes in Computer Science. Springer Berlin / Heidelberg, November 2012. HVC 2012 Best Paper Award.

[39] Ricardo J. Dias, Vasco Pessanha, and JoÃŁo M. LourenÃ§o. Precise detection of atomicity violations. In Armin Biere, Amir Nahir, and Tanja Vos, editors, *Hardware and Software: Verification and Testing*, volume 7857 of *Lecture Notes in Computer Science*, pages 8–23. Springer Berlin / Heidelberg, November 2013. HVC 2012 Best Paper Award.

[40] Vendula Dudka, Bohuslav Křena, Zdeněk Letko, Hana Pluháčková, and Tomáš Vojnar. Multi-objective genetic optimization for noise-based testing of concurrent software. In *SSBSE'14*, LNCS 8636, pages 107–122. Springer Verlag, 2014.

[41] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded Java Program Test Generation. *IBM Systems Journal*, 41:111–125, January 2002.

[42] 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.

[43] 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.

[44] 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.

[45] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: A Race and Transaction-aware Java Runtime. In *Proc. of PLDI'07*, pages 245–255, New York, NY, USA, 2007. ACM.

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

[47] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19:624–633, November 1976.

[48] Yaniv Eytani and Timo Latvala. Explaining Intermittent Concurrent Bugs by Minimizing Scheduling Noise. In *Proc. of HVC'06*, volume 4383 of LNCS, pages 183–197, Berlin, Heidelberg, 2007. Springer-Verlag.

[49] David Faragó and Peter H. Schmitt. Improving non-progress cycle checks. In *Proceedings of the 16th International SPIN Workshop on Model Checking Software*, pages 50–67, Berlin, Heidelberg, 2009. Springer-Verlag.

[50] 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, Washington, DC, USA, 2003. IEEE.

[51] Eitan Farchi, Itai Segall, João M. Lourenço, and Diogo Sousa. Using program closures to make an application programming interface (api) implementation thread safe. In *Proceedings of the 2012 Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, PADTAD 2012, pages 18–24, New York, NY, USA, 2012. ACM.

[52] Jan Fiedor, Vendula Hrubá, Bohuslav Křena, Zdeněk Letko, Shmuel Ur, and Tomáš Vojnar. Advances in noise-based testing. *STVR*, 24(7):1–38, 2014.

[53] Jan Fiedor and Tomáš Vojnar. ANaConDA: A Framework for Analysing Multi-threaded C/C++ Programs on the Binary Level. In *Proc. of RV'12*. LNCS 7687, Springer, 2012.

[54] Jan Fiedor and Tomáš Vojnar. Noise-based Testing and Analysis of Multi-threaded C/C++ Programs on the Binary Level. In *Proc. of PADTAD'12*, pages 36–46, New York, NY, USA, 2012. ACM.

[55] Jan Fiedor and Tomáš Vojnar. ANaConDA: A Framework for Analysing Multi-threaded C/C++ Programs on the Binary Level. In *Proc. of RV'13*, volume 7687 of LNCS, pages 35–41. Springer-Verlag, 2013.

[56] 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.

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

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

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

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

[61] Cormac Flanagan and Stephen N. Freund. The roadrunner dynamic analysis framework for concurrent programs. In *PASTE'10*, pages 1–8, New York, NY, USA, 2010. ACM.

[62] Cormac Flanagan, Stephen N. Freund, Marina Lifshin, and Shaz Qadeer. Types for atomicity: Static checking and inference for java. *ACM Trans. Program. Lang. Syst.*, 30(4):1–53, 2008.

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

[64] Dimitra Giannakopoulou, Corina S. Pasareanu, Michael Lowry, and Rich Washington. Lifecycle Verification of the NASA Ames K9 Rover Executive. In *Proc. of ICAPS'05*, page 11. AAAI Press, 2005.

[65] Patrice Godefroid. Software model checking: The verisoft approach. *Form. Methods Syst. Des.*, 26(2):77–101, 2005.

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

[67] Rachid Guerraoui and Michal Kapalka. *Principles of Transactional Memory.* Morgan and Claypool Publishers, 2010.

[68] Philip Jia Guo. A scalable mixed-level approach to dynamic analysis of C and C++ programs. Master's thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 5, 2006.

[69] H. H. Hallal, E. Alikacem, W. P. Tunney, S. Boroday, and A. Petrenko. Antipattern-based detection of deficiencies in java multithreaded software. In *QSIC'04: Proceedings of the Quality Software, Fourth International Conference*, pages 258–267, Washington, DC, USA, 2004. IEEE.

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

[71] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory, 2nd Edition.* Morgan and Claypool Publishers, 2010.

[72] Klaus Havelund. Using runtime analysis to guide model checking of java programs. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 245–264, London, UK, 2000. Springer-Verlag.

[73] Alex Ho, Steven Smith, and Steven Hand. On deadlock, livelock, and forward progress. Technical report, University of Cambridge, 2005.

[74] Gerard Holzmann. *Spin model checker, the: primer and reference manual.* Addison-Wesley Professional, 2003.

[75] Shin Hong, Jaemin Ahn, Sangmin Park, Moonzoo Kim, and Mary Jean Harrold. Testing Concurrent Programs to Achieve High Synchronization Coverage. In *Proc. of ISSTA 2012*, pages 210–220, New York, NY, USA, 2012. ACM.

[76] David Hovemeyer and William Pugh. Finding Concurrency Bugs in Java. In *Proc. of PODC'04*, July 2004.

[77] Vendula Hrubá, Bohuslav Křena, Zdeněk Letko, Shmuel Ur, and Tomáš Vojnar. Testing of Concurrent Programs Using Genetic Algorithms. In *Proc. of SSBSE'12*, volume 7515 of LNCS, pages 152–167, Berlin, Heidelberg, 2012. Springer-Verlag.

[78] Vendula Hrubá, Bohuslav Křena, and Tomáš Vojnar. Self-healing assurance using bounded model checking. In *Proc. of EUROCAST'09*. LNCS 5717, Springer, 2009.

[79] Clément Hurlin. Specifying and checking protocols of multithreaded classes. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 587–592, New York, NY, USA, 2009. ACM.

[80] Nicholas Jalbert, Cristiano Pereira, Gilles Pokam, and Koushik Sen. Radbench: A concurrency bug benchmark suite. In *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism*, HotPar'11, Berkeley, CA, USA, 2011. USENIX Association.

[81] Alexandra Jimborean, Luis Mastrangelo, Vincent Loechner, and Philippe Clauss. Vmad: An advanced dynamic program analysis and instrumentation framework. In *CC'12*, volume 7210 of *LNCS*, pages 220–239. Springer, 2012.

[82] Pallavi Joshi, Mayur Naik, Chang-Seo Park, and Koushik Sen. Calfuzzer: An extensible active testing framework for concurrent programs. In *CAV'09*, pages 675–681, Berlin, Heidelberg, 2009. Springer-Verlag.

[83] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. A Randomized Dynamic Program Analysis Technique for Detecting Real Deadlocks. In *Proc. of PLDI'09*, pages 110–120, New York, NY, USA, 2009. ACM.

[84] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *PLDI'09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 110–120, New York, NY, USA, 2009. ACM.

[85] 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.

[86] Devin Kester, Martin Mwebesa, and Jeremy S. Bradbury. How good is static analysis at finding concurrency bugs? In *SCAM*, pages 115–124. IEEE Computer Society, 2010.

[87] KyungHee Kim, Tuba Yavuz-Kahveci, and Beverly A. Sanders. Precise data race detection in a relaxed memory model using heuristic-based model checking. In *ASE*, pages 495–499. IEEE, 2009.

[88] Brian Krebs. A time to patch II: Mozilla, 2006. Last visited March 2016.

[89] Bohuslav Křena, Zdeněk Letko, Yarden Nir-Buchbinder, Rachel Tzoref-Brill, Shmuel Ur, and Tomáš Vojnar. A concurrency testing tool and its plug-ins for dynamic analysis and runtime healing. In *Runtime Verification*, Lecture Notes in Computer Science, Volume 5779/2009, pages 101–114. Springer Verlag, 2009.

[90] Bohuslav Křena, Zdeněk Letko, Rachel Tzoref, Shmuel Ur, and Tomáš Vojnar. Healing Data Races On-the-fly. In *Proc. of PADTAD'07*, pages 54–64, New York, NY, USA, 2007. ACM.

[91] Bohuslav Křena, Zdeněk Letko, Rachel Tzoref, Shmuel Ur, and Tomáš Vojnar. Healing data races on-the-fly. In *PADTAD'07*, pages 54–64, New York, NY, USA, 2007. ACM.

[92] Bohuslav Křena, Zdeněk Letko, and Tomáš Vojnar. Coverage Metrics for Saturation-based and Search-based Testing of Concurrent Software. In *Proc. of RV'11*, volume 7186 of LNCS, pages 177–192, Berlin, Heidelberg, 2012. Springer-Verlag.

[93] Bohuslav Křena, Zdeněk Letko, Tomáš Vojnar, and Shmuel Ur. A Platform for Search-based Testing of Concurrent Software. In *PADTAD'10*, pages 48–58, New York, NY, USA, 2010. ACM.

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

[95] M.A. Laurenzano, M.M. Tikir, L. Carrington, and A. Snavely. Pebil: Efficient static binary instrumentation for linux. In *ISPASS'10*, pages 175–183, 2010.

[96] Zdeněk Letko. *Analysis and Testing of Concurrent Programs*. PhD thesis, 2012.

[97] Zdeněk Letko, Tomáš Vojnar, and Bohuslav Křena. Atomrace: data race and atomicity violation detector and healer. In *PADTAD'08*, pages 1–10, New York, NY, USA, 2008. ACM.

[98] Zdeněk Letko, Tomáš Vojnar, and Bohuslav Křena. AtomRace: Data Race and Atomicity Violation Detector and Healer. In *Proc. of PADTAD'08*, pages 1–10, New York, NY, USA, 2008. ACM.

[99] Zdeněk Letko, Tomáš Vojnar, and Bohuslav Křena. Influence of Noise Injection Heuristics on Concurrency Coverage. In *Proc. of MEMICS'11*, volume 7119 of LNCS, pages 123–131, Berlin, Heidelberg, 2012. Springer-Verlag.

[100] Stefan Leue, Alin Ştefănescu, and Wei Wei. A livelock freedom analysis for infinite state asynchronous reactive systems. In Christel Baier and Holger Hermanns, editors, *CONCUR 2006 – Concurrency Theory*, volume 4137 of *Lecture Notes in Computer Science*, pages 79–94. Springer Berlin / Heidelberg, 2006.

[101] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

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

[103] Peng Liu, Julian Dolby, and Charles Zhang. Finding incorrect compositions of atomicity. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 158–168. ACM, 2013.

[104] 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, Washington, DC, USA, 2003. IEEE.

[105] João M. Lourenço, Ricardo J. Dias, João Luís, Miguel Rebelo, and Vasco Pessanha. Understanding the Behavior of Transactional Memory Applications. In *Proc. of PADTAD'09*. ACM, 2009.

[106] J. Lourenço, D. Sousa, B. Teixeira, and R. Dias. Detecting concurrency anomalies in transactional memory programs. *Computer Science and Information Systems/ComSIS*, 8(2):533–548, 2011.

[107] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou. Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. *SIGOPS Oper. Syst. Rev.*, 41(6):103–116, 2007.

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

[109] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *Proc. of ASPLOS'06*, pages 37–48, New York, NY, USA, 2006. ACM.

[110] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. Avio: detecting atomicity violations via access interleaving invariants. In *ASPLOS-XII*, pages 37–48, New York, NY, USA, 2006. ACM.

[111] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proc. of PLDI'05*, pages 190–200, New York, NY, USA, 2005. ACM.

[112] Qingzhou Luo, Yi Zhang, Choonghwan Lee, Dongyun Jin, Patrick O'Neil Meredith, Traian Florin Şerbănuţă, and Grigore Roşu. *RV-Monitor: Efficient Parametric Runtime Verification with Simultaneous Properties*, pages 285–300. Springer International Publishing, Cham, 2014.

[113] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.

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

[115] 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.

[116] Bertrand Meyer. Applying „design by contract“. *Computer*, 25(10):40–51, October 1992.

[117] Bertrand Meyer. Applying „design by contract“. *Computer*, 25(10):40–51, October 1992.

[118] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Trans. Comput. Syst.*, 15(3):217–252, 1997.

[119] M. Musuvathi, S. Qadeer, and T. Ball. CHESS: A Systematic Testing Tool for Concurrent Software. Technical Report MSR-TR-2007-149, Microsoft Research, 2007.

[120] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI'08*, pages 267–280, Berkeley, CA, USA, 2008. USENIX Association.

[121] Mayur Naik. Chord: A static and dynamic program analysis platform for java bytecode. URL: http://code.google.com/p/jchord.

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

[123] Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. Effective static deadlock detection. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 386–396. IEEE Computer Society, 2009.

[124] Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. Effective static deadlock detection. In *ICSE'09: Proceedings of the 31st International Conference on Software Engineering*, pages 386–396, Washington, DC, USA, 2009. IEEE.

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

[126] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI'07*, pages 89–100, New York, NY, USA, 2007. ACM.

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

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

[129] 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.

[130] Chang-Seo Park and Koushik Sen. Randomized Active Atomicity Violation Detection in Concurrent Programs. In *Proc. of SIGSOFT'08/FSE-16*, pages 135–145, New York, NY, USA, 2008. ACM.

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

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

[133] Eli Pozniansky and Assaf Schuster. Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs. In *Proc. of PPoPP'03*, pages 179–190, New York, NY, USA, 2003. ACM.

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

[135] Michael Pradel, Ciera Jaspan, Jonathan Aldrich, and Thomas R. Gross. Statically checking api protocol conformance with mined multi-object specifications. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 925–935, Piscataway, NJ, USA, 2012. IEEE Press.

[136] Eric Rescorla. Security holes... who cares? In *Proceedings of the 12th Conference on USENIX Security Symposium*, pages 75–90, Berkeley, CA, USA, 2003. USENIX Association.

[137] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multi-threaded Programs. In *Proc. of SOSP'97*, pages 27–37, New York, NY, USA, 1997. ACM.

[138] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multi-threaded programs. In *SOSP'97*, pages 27–37, New York, NY, USA, 1997. ACM.

[139] Viktor Schuppan. *Liveness checking as safety checking to find shortest counterexamples to linear time properties*. PhD thesis, ETH ZĂLrich, MĂLnchen, 2006.

[140] Koushik Sen. Race Directed Random Testing of Concurrent Programs. In *Proc. of PLDI'08*, pages 11–21, New York, NY, USA, 2008. ACM.

[141] D. G. Sousa, R. J. Dias, C. Ferreira, and J. M. Lourenço. Preventing atomicity violations with contracts. *arXiv preprint arXiv:1505.02951*, May 2015.

[142] Diogo G. Sousa, Ricardo J. Dias, Carla Ferreira, and JoĂŁo M. LourenĂ§o. Preventing atomicity violations with contracts. *eprint arXiv:1505.02951*, May 2015.

[143] William Stallings. *Operating Systems: Internals and Design Principles*. Prentice Hall, 6 edition, April 2008.

[144] Scott D. Stoller. Testing Concurrent Java Programs using Randomized Scheduling. In *Proc. of RV'02*, volume 70(4) of *Electronic Notes in Theoretical Computer Science*. Elsevier, July 2002.

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

[146] Kuo-Chung Tai. Definitions and detection of deadlock, livelock, and starvation in concurrent programs. In *ICPP'94: Proceedings of the 1994 International Conference on Parallel Processing*, pages 69–72, Washington, DC, USA, 1994. IEEE.

[147] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2007.

[148] Ehud Trainin, Yarden Nir-Buchbinder, Rachel Tzoref-Brill, Aviad Zlotnick, Shmuel Ur, and Eitan Farchi. Forcing Small Models of Conditions on Program Interleaving for Detection of Concurrent Bugs. In *Proc. of PADTAD'09*, pages 1–6, New York, NY, USA, 2009. ACM.

[149] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *ACM SIGPLAN Notices*, volume 41, pages 334–345. ACM, 2006.

[150] 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.

[151] Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model checking programs. In *ASE'00: Proceedings of the 15th IEEE international conference on Automated software engineering*, page 3, Washington, DC, USA, 2000. IEEE.

[152] C. Von Praun and T.R. Gross. Static detection of atomicity violations in object-oriented programs. *Journal of Object Technology*, 3(6):103–122, 2004.

[153] Christoph von Praun and Thomas R. Gross. Object Race Detection. In *Proc. of OOPSLA'01*, pages 70–82, New York, NY, USA, 2001. ACM.

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

[155] Liqiang Wang and Scott D. Stoller. Static analysis of atomicity for programs with non-blocking synchronization. In *PPoPP'05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 61–71, New York, NY, USA, 2005. ACM.

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

[157] Amy Williams, William Thies, and Michael D. Ernst. Static deadlock detection for java libraries. In *ECOOP 2005—Object-Oriented Programming, 19th European Conference*, pages 602–629, Glasgow, Scotland, July 27–29, 2005.

[158] Jingyue Wu, Yang Tang, Gang Hu, Heming Cui, and Junfeng Yang. Sound and Precise Analysis of Parallel Programs through Schedule Specialization. In *Proc. of PLDI'12*, pages 205–216, New York, NY, USA, 2012. ACM.

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

[160] Y. Yang, A. Gringauze, D. Wu, and H. Rohde. Detecting data race and atomicity violation via typestate-guided static analysis. Technical Report MSR-TR-2008-108, Microsoft Research, 2008.

[161] Jaeheon Yi, Caitlin Sadowski, and Cormac Flanagan. SideTrack: Generalizing Dynamic Atomicity Analysis. In *Proceedings of the 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, PADTAD '09, pages 8:1–8:10, New York, NY, USA, 2009. ACM.

[162] Jie Yu and Satish Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. *SIGARCH Comput. Archit. News*, 37(3):325–336, 2009.

[163] Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam. Maple: A Coverage-driven Testing Tool for Multithreaded Programs. In *Proc. of OOPSLA'12*, pages 485–502, New York, NY, USA, 2012. ACM.

[164] 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.

[165] Wei Zhang, Chong Sun, and Shan Lu. Conmem: detecting severe concurrency bugs through an effect-oriented approach. In *ASPLOS'10: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, pages 179–192, New York, NY, USA, 2010. ACM.