BRNO UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Department of Intelligent Systems

Ing. Jan Fiedor

Practical Methods of Automated Verification of Concurrent Programs

Praktické metody automatizované verifikace paralelních programů

Extended Abstract of a Ph.D. Thesis

Supervisor: prof. Ing. Tomáš Vojnar, Ph.D.

Key Words

Dynamic analysis, concurrency, noise injection, testing, transactional memory, contracts.

Klíčová slova

Dynamická analýza, souběžnost, vkládání šumu, testování, transakční paměť, kontrakty.

The original of the thesis is available in the library of Faculty of Information Technology, Brno University of Technology, Czech Republic.

Contents

1 Introduction **5**
1.1 Detecting Errors in Multi-threaded Programs 5
1.2 Goals of the Thesis 8

2 Concurrency Errors **9**
2.1 Safety Errors 10
2.1.1 Data Races 10
2.1.2 Atomicity Violation 10
2.1.3 Order Violations 11
2.1.4 Deadlocks 11
2.1.5 Missed Signals 11
2.2 Liveness and Mixed Errors 12
2.2.1 Livelocks and Non-progress Behaviour 12
2.2.2 Blocked Threads 13

3 Combining Dynamic Analysis and Model Checking **14**
3.1 Experiments 15

4 The ANaConDA Framework **16**
4.1 Experiments 17

5 Improved Noise Injection **18**
5.1 Evaluating Improved Noise Injection 19
5.2 Evaluating New Heuristics 20

6 Transactional Memory Programs **23**
6.1 Experimental Evaluation 24

7 Contracts for Concurrency **26**
7.1 Experiments 27

8 Conclusion **29**

Bibliography **31**

Curriculum Vitae **37**

Abstract **38**

1 Introduction

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 [Kre06, CHPW00, Res03] 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 verification methods, such as, e.g., model checking [BK08, CGP99], 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 [AB01a].

Static analyses, such as [HP04], 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 [KMB10], but they often cannot discover concurrency-related errors because they do not model threads and their interactions [HP04]. Of course, there also exist static analyses which do consider concurrent threads, such as, e.g., [AB01b, NPSG09]. These analyses are able to detect concurrency-related errors, but they often produce many false alarms due to the abstractions they work with.

Testing [EFG+03b, HAP+12, MQB07, YNPP12] 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 [BH06, EQT07] 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 [HAP+12, MQB07, YNPP12] or noise injection [EFG+03b]). 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 [PGB+05].

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 [EFG+03b] 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 [VHBP00] 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) [EFG+03b] or the Microsoft Driver Verifier where the technique is called delay fuzzing [htt13]. 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 [HAP+12, MQB07, MQB+08, WTH+12, YNPP12] uses a deterministic control over the scheduling of threads. This approach can be seen as execution-based model checking which systematically tests as many thread interleaving scenarios as possible. Before the 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 [MQB07], totally ordered list of synchronization events [WTH+12], or simply in the form of a set of explored decisions [HAP+12, YNPP12]. 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.

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 [EQT07, PS03, SBN+97, LVK08a], atomicity violations [LTQZ06], or deadlocks [BH06, JPSN09, AS06]. These techniques may find more bugs in fewer executions than classical testing. Some of the techniques, e.g., [EQT07], 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 [Sen08], atomicity violations [PS08], and deadlocks [JPSN09]. 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 thesis 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. Most of the proposed techniques are implemented using the ANaConDA framework that allows one to easily analyse multi-threaded C/C++ programs, a common class of programs which is surprisingly often not supported by the implementations of various existing techniques. Also some of the newly invented techniques are able to detect some of the less-studied kinds of concurrency errors such as order violations. Moreover, the tools implementing the techniques are simple to use in practice, often give good results with just the default configuration, and can handle any C/C++ program even without its source code available.

The following sections summarise the contributions of the thesis, each section covering one or more of the goals mentioned above.

2 Concurrency Errors

Many works devoted to detection of concurrency errors have been published in recent years and many of them presented definitions of concurrency errors that the proposed algorithms are able to handle. These definitions are usually expressed in different terms suitable for a description of the particular considered algorithms, and they surprisingly often differ from each other in the meaning they assign to particular errors. To help understanding the errors and developing techniques for detecting them, this section strives to provide a uniform taxonomy of concurrency errors common in current programs.

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, however, often, the definitions have some shared basic *skeleton* which is *parameterised* by different underlying notions. 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.

2.1 Safety Errors

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

2.1.1 Data Races

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

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

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

2.1.2 Atomicity Violation

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

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

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

2.1.3 Order Violations

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

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

2.1.4 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* [CES71]. A Coffman deadlock happens in a state in which four conditions are met: (1) Processes have an exclusive access to the resources granted to them, (2) processes hold some resources and are waiting for additional resources, (3) resources cannot be forcibly removed from the tasks holding them until the resources are used to completion (no preemption on the resources), and (4) a circular chain of tasks exists in which each task holds one or more resources that are being requested by the next task in the chain. Such a definition perfectly fits deadlocks caused by blocking lock operations but does not cover deadlocks caused by message passing (e.g., a thread t_1 can wait for a message that could only be sent by a thread t_2, but t_2 is waiting for a message that could only be sent by t_1).

2.1.5 Missed Signals

Missed signals are another less studied class of concurrency errors. The notion of missed signals assumes that it is known which signal is *intended* to be

delivered to which thread or threads. A missed signal error can be defined as follows.

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

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

2.2 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* [Tan07].

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).

2.2.1 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 [Tan07].

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 [BBM07, Sta08, LŞW06, Tai94, HSH05]. Other papers [MR97, MP92] 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 [And91] 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.

2.2.2 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* [FNU03]. Another reason may be that a thread is waiting for a livelocked or deadlocked thread.

3 Combining Dynamic Analysis and Model Checking

Many existing approaches for detecting concurrency errors are 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 section, 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 [HKV09], we use the infrastructure offered by the *Contest* tool [EFG+03a] 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 [VHBP00] 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 searched state space, is suitably modified. Bounded

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

Table 1: 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

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.

3.1 Experiments

To demonstrate capabilities of DA-BMC, 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 1. 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 1 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 1 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).

4 The ANaConDA Framework

Many of the existing techniques are implemented 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, there are only a handful of such tools for C/C++.

To address the lack of tools for C/C++, this section presents the ANaConDA framework which is a framework for adaptable native-code concurrency-focused dynamic analysis built on top of PIN [LCM+05]. 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 section.

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.

Still, monitoring C/C++ programs on the binary level can be quite difficult. One of the problems to be dealt with is monitoring of function execution. This is because the monitoring code has to cope with that the control can be passed among several functions by jumps. Hence, the control can return from a different function than the one that was called. Another problem is that the monitoring code must properly trigger notifications for various special types of instructions such as atomic instructions, which access several memory locations at once but in an atomic way, or conditional and repeatable instructions, which might be executed more than once or not at all. Further, some pieces of information about the execution of instructions or functions (such as the memory locations accessed by them), which are crucial for various analyses, may be lost once the instruction or function finishes its execution, and it is necessary to explicitly preserve this information for later use. Finally, in order to support various multithreading libraries, the analysers must be abstracted from the concrete library used. The ANaConDA framework solves all of the above problems for the user which greatly simplifies the task of implementing various analyses for C/C++ programs. Moreover, the framework also provides debugging information, various kinds of backraces, and other information which may be used to precisely localise the detected errors.

4.1 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). 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 are considerably smaller (about 40k lines of code) and 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 severe as they may cause a crash (segmentation fault) of the program which uses these libraries. In both cases, one thread may reset a pointer to a callback function in between of the times when another thread checks the validity of this pointer and calls the function referenced by it.

Finally, we also successfully tested the framework on several Windows toy programs (100–500 lines of code). We have also used the framework for all of the experiments described in the upcoming sections.

5 Improved Noise Injection

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 is to use *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 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. Currently, various modifications of the *random* heuristic are used. The *random-all* heuristic put a noise before all possible locations with a given probability (called *frequency*). The *sharedVar-all* and *sharedVar-one* heuristics then restrict the possible locations just to accesses to all shared variables and one chosen shared variable, respectively. 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 most commonly used heuristics are *yield*, *sleep*, and *busyWait*, where *yield* injects one or more calls of `yield()`, *sleep* injects one call of `sleep()`, and *busyWait* just loops for some time. They are parametrised by the *strength of noise*. In case of the *sleep* and *busyWait* heuristics, the strength gives the time to wait or loop. In case of the *yield* heuristic, the strength says how many times the `yield()` routine should be called.

One of the main contributions of the thesis is a proposal of 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 technique as a concrete noise injection heuristic called *read/write* noise using the ANaConDA framework. This heuristic allows one to use different noise settings for read and write accesses separately. 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. We tested the heuristic on a set of smaller C/C++ projects and the obtained results show that using heuristics combining several noise seeding and noise placement heuristics can lead to a further increase of chances to spot a concurrency errors.

Apart from the technique mentioned above, we also present two additional noise injection heuristics—in particular, a new noise placement heuristic called *pattern* noise and a new noise seeding heuristic called *inverseNoise*. The *pattern* noise placement heuristic injects a noise before accesses to variables which were already accessed before within the same method or function. The *inverseNoise* noise seeding heuristic stops all but one thread and allows this one thread to get as far as possible in its execution. 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 [FV12b, LVK12].

5.1 Evaluating Improved Noise Injection

To evaluate how the improved noise injection approach can further increase the chances to detect errors in multi-threaded programs, we compared the *read/write* noise placement heuristic, which uses different noise seeding heuristics for read and write accesses, with the *random-all* heuristic, which uses a single noise seeding heuristic for all accesses. We implemented both heuristics in the ANaConDA framework [FV13] and then used them in conjunction with a C++ implementation of the AtomRace dynamic detector [LVK08b] to detect data races in 14 C programs implementing a simple ticket algorithm.

Results obtained for some selected noise injection configurations and test cases are shown in Tables 2 and 3. 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 possibilities how to combine them. One can use the same noise seeding heuristics, but parametrize them with different values of strength. As the results in Table 2 show, such configurations (Configuration no. 3) achieved better results than the configurations using the *random-all* heuristic (Configurations no. 1 and 2). Another possibility is to use different noise seeding heuristics for different memory accesses, e.g., the *sleep* heuristic

Table 2: 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**

Table 3: 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

for one type of memory accesses and the *yield* heuristic for the second one, and leave the values of strength the same. As the results in Table 2 show, such configurations (Configuration no. 4) achieved even better results than the ones combining different values of strength (Configuration no. 3).

Table 3 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.

5.2 Evaluating New Heuristics

To evaluate the new noise injection heuristics, we compared them with the best existing heuristics. We implemented both the new and existing heuristics in the ANaConDA framework [FV13] and used the AtomRace dynamic detec-

tor [LVK08b] to detect data races in 4 of the C programs mentioned in the previous section. For each execution of a test, the framework collects information about test duration and about the fact whether an error has manifested. 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. 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).

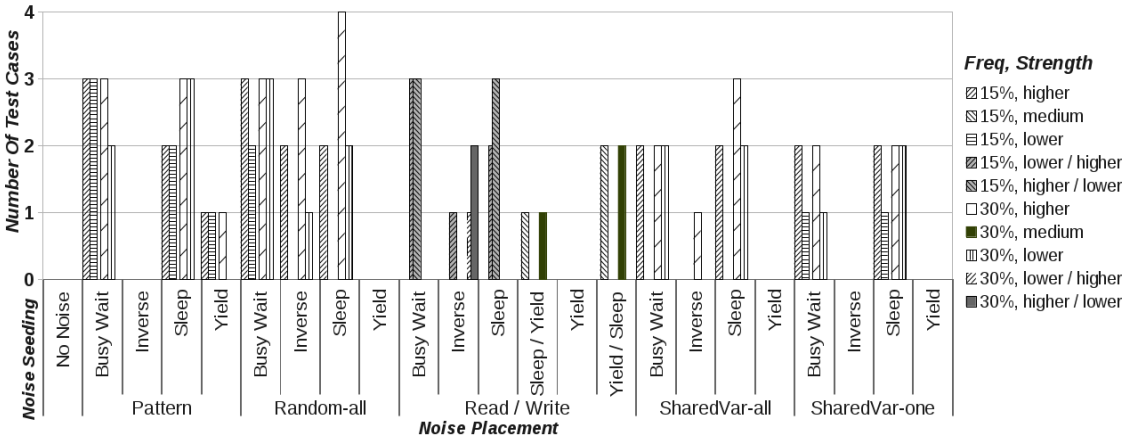

Figure 1: A comparison of configurations across all of the considered C test cases

Results. 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 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 heuristics.

6 Transactional Memory Programs

Transactional memory (TM) [GK10, HLR10] 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 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 [CMCKO08].

The lightest monitoring approach *statistics collector* (*sc*) tracks the number of started and committed transactions by maintaining two counters for each thread and each type of transaction. The *event logger* (*el*) approach tracks the same information by registering TM operations (events) in an event log and then processing them post mortem. Its variants *el-a* and *el-arw* additionally track aborts and also transactional reads and writes. The *el-ts*, *el-a-ts*, and *el-arw-ts* variants also collect time stamps for each logged event. While the lightweight implementation of these approaches tracks the TM operations by embedding the monitoring code into the monitored program itself and thus modifying its source code, the heavyweight implementation uses the ANaConDA framework which tracks these operations without changing the monitored program.

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.

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 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 [FV12a]. 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.

6.1 Experimental Evaluation

To evaluate the influence of the considered monitoring approaches on the behaviour of the monitored programs, we compare the impact of the lightweight and heavyweight implementations. 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.

Table 4 shows the average global number of aborts for each of the tested programs for the lightweight and heavyweight implementations of the monitoring approaches. 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.

Table 4: 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

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 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.

7 Contracts for Concurrency

In this section, 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 Section 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 section, designed to ease the process of writing concurrent programs, do not entirely avoid this problem and suffer from atomicity violations as well [DPL13].

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 [Mey92] 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 [SDFL15] 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) [SDFL15].

In this section, 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 [SBN+97] 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.

7.1 Experiments

We implemented the method based on the happens-before relation using the ANaConDA framework [FV13]. The method supports contracts with both parameters (data flow information) and targets/spoilers (contextual information). The ANaConDA framework is used to monitor method calls and synchronization events in running C/C++ programs. ANaConDA also provides us with heuristic *noise injection*. As discussed in the previous sections, 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 [AHB03, AHB04, VPG04] and were also used to evaluate a static validation method proposed in [DFF+16] (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

Table 5: Validation results for dynamic analysis.

Benchmark	T/S pairs	Contract Violations	False Positives	Potential AV	Real AV	SLOC	Time (s)
Coord03 [AHB03]	8	380	0	0	380	116	1.01
Coord04 [AHB04]	4	24	0	0	24	53	0.52
Local [AHB03]	4	2	0	0	2	27	0.52
NASA [AHB03]	1	100	0	0	100	96	0.60
Account [VPG04]	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

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 [JPPS11], 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 post mortem, one would need to store a trace with more than 17 million method calls (about 1.6 GB of data) while our method, which works on-the-fly, needed about 10 MB of data only.

Table 5 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.[2] The column *False Positives*, which was included for compatibility with the results of static contract validation as presented in [DFF$^+$16], 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

[2]Compared with the static approach [DFF$^+$16], 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.

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.[3] 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 in seconds.

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.

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

[3]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.

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 noise injection 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 Section 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, 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

[AB01a] 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 Computer Society.

[AB01b] Cyrille Artho and Armin Biere. Applying Static Analysis to Large-Scale, Multi-Threaded Java Programs. In *Proc. of ASWEC'01*. IEEE Computer Society, 2001.

[AHB03] Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. *Software Testing, Verification and Reliability*, 13(4):207–227, December 2003.

[AHB04] 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.

[And91] Gregory R. Andrews. *Concurrent programming: principles and practice*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1991.

[AS06] Rahul Agarwal and Scott D. Stoller. Run-time Detection of Potential Deadlocks for Programs with Locks, Semaphores, and Condition Variables. In *Proc. of PADTAD'06*, pages 51–60, New York, NY, USA, 2006. ACM.

[BBM07] Johann Blieberger, Bernd Burgstaller, and Robert Mittermayr. Static Detection of Livelocks in Ada Multitasking Programs. In *Proc. of Ada-Europe'07*, pages 69–83, Berlin, Heidelberg, 2007. Springer-Verlag.

[BH06] Saddek Bensalem and Klaus Havelund. Dynamic Deadlock Analysis of Multi-threaded Programs. In *Proc. of HVC'05*, volume 3875 of LNCS, pages 208–223, Berlin, Heidelberg, 2006. Springer-Verlag.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[CES71] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Comput. Surv.*, 3:67–78, June 1971.

[CGP99] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[CHPW00] 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.

[CMCKO08] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *Proc. of IISWC'08*, 2008.

[DFF+16] 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`.

[DPL13] Ricardo J. Dias, Vasco Pessanha, and JoĂŁo M. LourenĂ§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.

[EFG+03a] Orit Edelstein, Eitan Farchi, Evgeny Goldin, Yarde 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.

[EFG+03b] 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.

[EQT07] 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.

[FNU03] 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.

[FV12a] 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.

[FV12b] 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.

[FV13] 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.

[GK10] Rachid Guerraoui and Michal Kapalka. *Principles of Transactional Memory*. Morgan and Claypool Publishers, 2010.

[HAP+12] Shin Hong, Jaemin Ahn, Sangmin Park, Moonzoo Kim, and Mary Jean Harrold. Testing Concurrent Programs to Achieve High Synchronization Coverage. In *Proc. of ISSTA'12*, pages 210–220, New York, NY, USA, 2012. ACM.

[HKV09] Vendula Hrubá, Bohuslav Křena, and Tomáš Vojnar. Self-healing assurance using bounded model checking. In *Proc. of EUROCAST'09*. LNCS 5717, Springer, 2009.

[HLR10] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory, 2nd Edition*. Morgan and Claypool Publishers, 2010.

[HP04] David Hovemeyer and William Pugh. Finding Concurrency Bugs in Java. In *Proc. of PODC'04*, July 2004.

[HSH05] Alex Ho, Steven Smith, and Steven Hand. On deadlock, livelock, and forward progress. Technical report, University of Cambridge, 2005.

[htt13] Power Framework Delay Fuzzing. Online at: http://msdn.microsoft.com/en-us/library/hh454184(v=vs.85).aspx, April 2013.

[JPPS11] 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.

[JPSN09] 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.

[KMB10] Devin Kester, Martin Mwebesa, and Jeremy S. Bradbury. How Good is Static Analysis at Finding Concurrency Bugs? In *Proc. of SCAM'10*, pages 115–124. IEEE Computer Society, 2010.

[Kre06] Brian Krebs. A time to patch II: Mozilla, 2006. Last visited March 2016.

[LCM+05] 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*. ACM, 2005.

[LŞW06] 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.

[LTQZ06] 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.

[LVK08a] 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.

[LVK08b] 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.

[LVK12] 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.

[Mey92] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.

[MP92] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[MQB07] M. Musuvathi, S. Qadeer, and T. Ball. CHESS: A Systematic Testing Tool for Concurrent Software. Technical Report MSR-TR-2007-149, Microsoft Research, 2007.

[MQB+08] 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.

[MR97] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Trans. Comput. Syst.*, 15(3):217–252, 1997.

[NPSG09] Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. Effective Static Deadlock Detection. In *Proc. of ICSE'09*, pages 386–396. IEEE Computer Society, 2009.

[PGB+05] Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.

[PS03] 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.

[PS08] 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.

[Res03] 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.

[SBN+97] 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.

[SDFL15] Diogo G. Sousa, Ricardo J. Dias, Carla Ferreira, and JoĂŁo M. LourenĂ§o. Preventing atomicity violations with contracts. *eprint arXiv:1505.02951*, May 2015.

[Sen08] Koushik Sen. Race Directed Random Testing of Concurrent Programs. In *Proc. of PLDI'08*, pages 11–21, New York, NY, USA, 2008. ACM.

[Sta08] William Stallings. *Operating Systems: Internals and Design Principles*. Prentice Hall, 6 edition, April 2008.

[Tai94] 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.

[Tan07] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2007.

[VHBP00] Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model Checking Programs. In *Proc. of ASE'00*, page 3, Washington, DC, USA, 2000. IEEE Computer Society.

[VPG04] 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.

[WTH+12] 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.

[YNPP12] 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.

Curriculum Vitae

Personal Data

Name:	Jan Fiedor
Born:	April 2, 1985, Bohumín, Czech Republic
E-mail:	`fiedorjan@centrum.cz`
Telephone:	+420 721 344 247

Education

2009	–	now	PhD. (ongoing)—Faculty of Information Technology, Brno University of Technology. Research theme: *Practical Methods of Automated Verification of Concurrent Programs.*
2007	–	2009	Master's degree (Ing.)—Faculty of Information Technology, Brno University of Technology. Graduated with honors. Master thesis: *Design and Implementation of a Tool for Formal Verification of Systems Specified in RT-Logic Language.*
2004	–	2007	Bachelor's degree (Bc.)—Faculty of Information Technology, Brno University of Technology. Graduated with honors. Bachelor thesis: *Instant Messaging System.*
1996	–	2004	Secondary education—Gymnázium Orlová, Orlová.

Experience

09/2013	–	12/2013	IT department of Faculty of Science and Technology, Universidade Nova de Lisboa, Lisbon, Portugal. Short Time Scientific Mission within the European COST Action project IC1001 (Euro-TM) focused on developing methods for analysing multi-threaded programs using Transactional Memory.
2009	–	now	Part-time development engineer at *FIT BUT*, Brno, CZ. Development of C/C++ tools for dynamic analyses and testing of multi-threaded C/C++ programs.
2006	–	now	Lecturer at *ApS Brno*, Brno, CZ. Courses focusing on C/C++ programming and administration of MS Windows (MCP 70-270/290/291, MCT 70-620/640, MCSA).

Language skills

Czech, English.

Abstract

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.