BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

ADVANCED STATIC ANALYSIS OF ATOMICITY IN CONCURRENT PROGRAMS THROUGH FACEBOOK INFER

POKROČILÁ STATICKÁ ANALÝZA ATOMIČNOSTI V PARALELNÍCH PROGRAMECH V PROSTŘEDÍ FACEBOOK INFER

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. DOMINIK HARMIM**
AUTOR PRÁCE

SUPERVISOR **prof. Ing. TOMÁŠ VOJNAR, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2021

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2020/2021

Master's Thesis Specification

Student: **Harmim Dominik, Bc.**
Programme: Information Technology and Artificial Intelligence
Specialization: Software Verification and Testing
Title: **Advanced Static Analysis of Atomicity in Concurrent Programs through Facebook Infer**
Category: Software analysis and testing

Assignment:

1. Study limitations of the atomicity analyser Atomer developed in your bachelor thesis as well as the latest developments concerning the Facebook Infer framework.
2. Propose ways of significantly improving precision and/or scalability of the analysis even if for the price of the user providing more input and/or combining it with dynamic analysis.
3. Implement a new version of Atomer including the proposed improvements and supporting analysis of programs written in more programming languages than just C supported by the first version of Atomer.
4. Evaluate the new version of Atomer on suitable benchmarks, including at least real-life code in which some atomicity problems were previously detected.
5. Describe and discuss the achieved results and their further possible improvements.

Recommended literature:

1. Rival, X., Yi, K.: Introduction to Static Analysis: An Abstract Interpretation Perspective. MIT Press, 2020.
2. Blackshear, S., Gorogiannis, N., O'Hearn, P. W., Sergey, I.: RacerD: Compositional Static Race Detection. In: Proc. of OOPSLA'18, PACMPL 2(OOPSLA):144:1-144:28, 2018.
3. Gorogiannis, N., O'Hearn, P.W., Sergey, I.: A True Positives Theorem for a Static Race Detector. In: Proc. of POPL'19, PACMPL 3(POPL):57:1-57:29, 2019.
4. Dias, R.J., Ferreira, C., Fiedor, J., Lourenço, J.M., Smrčka, A., Sousa, D.G., Vojnar, T.: Verifying Concurrent Programs Using Contracts, In: Proc. of ICST'17, IEEE, 2017.
5. Harmim, D.: Static Analysis Using Facebook Infer to Find Atomicity Violations. Bachelor thesis, Brno University of Technology, 2019.
6. Marcin, V.: Static Analysis Using Facebook Infer Focused on Deadlock Detection. Bachelor thesis, Brno University of Technology, 2019.

Requirements for the semestral defence:

- Item 1 and at least some development falling under items 2 and 3 of the assignment.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Vojnar Tomáš, prof. Ing., Ph.D.**
Head of Department: Hanáček Petr, doc. Dr. Ing.
Beginning of work: November 1, 2020
Submission deadline: May 19, 2021
Approval date: November 11, 2020

Abstract

Atomer is a *static analyser* based on the idea that if some *sequences of functions* of a *multi-threaded program* are executed *under locks* in some runs, likely, they are *always intended to execute atomically*. Atomer thus strives to look for such sequences and then detects for which of them the atomicity may be broken in some other program runs. The author of this master's thesis proposed and implemented the first version of Atomer as a plugin of the *Facebook Infer framework* in his bachelor's thesis. In the master's thesis, a new and *significantly improved* version of Atomer is proposed. The improvements aim at both increasing *scalability* as well as *precision*. Moreover, support for several initially not supported programming features has been added (including, e.g., the possibility of analysing *C++ and Java programs* or support for *re-entrant locks* or *lock guards*). Through a number of experiments (including experiments with *real-life code* and *real-life bugs*), it is shown that the new version of Atomer is indeed much *more general, scalable, and precise*.

Abstrakt

Nástroj *Atomer* je *statický analyzátor* založený na myšlence, že pokud jsou některé *sekvence funkcí vícevláknového programu* prováděny v některých bězích *pod zámky*, je pravděpodobně zamýšleno, že mají být *vždy provedeny atomicky*. Analyzátor Atomer se tudíž snaží takové sekvence hledat a poté zjišťovat, pro které z nich může být v některých jiných bězích programu porušena atomicita. Autor této diplomové práce ve své bakalářské práci navrhl a implementoval první verzi nástroje Atomer jako zásuvný modul *aplikačního rámce Facebook Infer*. V této diplomové práci je navržena nová a *výrazně vylepšená* verze analyzátoru Atomer. Cílem vylepšení je zvýšení jak *škálovatelnosti*, tak *přesnosti*. Kromě toho byla přidána podpora pro několik původně nepodporovaných programovacích vlastností (včetně např. možnosti analyzovat *programy napsané v jazycích C++ a Java* nebo podpory pro *reentrantní zámky* nebo *stráže zámků*, tzv. „lock guards“). Prostřednictvím řady experimentů (včetně experimentů s *reálnými programy* a *reálnými chybami*) se ukázalo, že nová verze nástroje Atomer je skutečně *mnohem obecnější, přesnější a lépe škáluje*

Keywords

Facebook Infer, static analysis, abstract interpretation, contracts for concurrency, atomicity violation, concurrent programs, programs analysis, atomicity, incremental analysis, modular analysis, compositional analysis, interprocedural analysis, scalability, Atomer, function calls sequence, multi-threaded programs

Klíčová slova

Facebook Infer, statická analýza, abstraktní interpretace, kontrakty pro paralelismus, porušení atomicity, paralelní programy, analýza programů, atomicita, inkrementální analýza, modulární analýza, kompoziční analýza, interprocedurální analýza, škálovatelnost, Atomer, sekvence volání funkcí, vícevláknové programy

Reference

HARMIM, Dominik. *Advanced Static Analysis of Atomicity in Concurrent Programs through Facebook Infer*. Brno, 2021. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor prof. Ing. Tomáš Vojnar, Ph.D.

Rozšířený abstrakt

[[Tady nějak zkombinovat a zkrátit úvod + závěr a přeložit do češtiny.]]

Advanced Static Analysis of Atomicity in Concurrent Programs through Facebook Infer

Declaration

Hereby I declare that this master's thesis was prepared as an original author's work under the supervision of professor Tomáš Vojnar. All the relevant information sources used during this thesis's preparation are appropriately cited and included in the reference list.

.....
Dominik Harmim
18th May 2021

Acknowledgements

I thank my colleagues from VeriFIT for their assistance. I would particularly like to thank my supervisor Tomáš Vojnar. I also wish to acknowledge Nikos Gorogiannis from the Infer team at Facebook for valuable discussions about the analyser's development. Lastly, I acknowledge the financial support received from the H2020 ECSEL projects AQUAS, Arrowhead Tools, and VALU3S.

Contents

Chapter 1

Introduction

Bugs have been present in computer programs ever since the inception of the programming discipline. Unfortunately, they are often hidden in unexpected places, and they can lead to unexpected behaviour, which may cause significant damage. Nowadays, developers have many possibilities of catching bugs in the early development process. *Dynamic analysers* or tools for *automated testing* are often used, and they are satisfactory in many cases. Nevertheless, they can still leave too many bugs undetected because they can analyse only *particular program flows* dependent on the input data. An alternative solution is *static analysis* (despite it, of course, suffers from some problems too — such as the possibility of reporting many *false alarms*, i.e., *spurious errors*). Quite some tools for static analysis were implemented, e.g., Coverity or CodeSonar. However, they are often proprietary and difficult to openly evaluate and extend.

Recently, Facebook introduced *Facebook Infer*: an *open-source* tool for creating *highly scalable*, *compositional*, *incremental*, and *interprocedural* static analysers. Facebook Infer has grown considerably, but it is still under active development by many teams across the globe. It is employed every day not only in Facebook itself but also in other companies, such as Spotify, Uber, Mozilla, or Amazon. Currently, Facebook Infer provides several analysers that check for various types of bugs, such as buffer overflows, data races and some forms of deadlocks and starvation, null-dereferencing, or memory leaks. However, most importantly, Facebook Infer is a *framework* for building new analysers quickly and easily. Unfortunately, the current version of Facebook Infer still lacks better support for *concurrency bugs*. While it provides a reasonably advanced data race analyser, it is limited to Java and C++ programs only and fails for C programs, which use a *lower-level lock manipulation*. Moreover, the only available checker of *atomicity of call sequences* is the first version of *Atomer* [15] proposed in the bachelor's thesis of the author.

At the same time, in *concurrent programs*, there are often *atomicity requirements* for the execution of specific sequences of instructions. Violating these requirements may cause many kinds of problems, such as unexpected behaviour, exceptions, segmentation faults, or other failures. *Atomicity violations* are usually not verified by compilers, unlike syntactic or some sorts of semantic rules. Moreover, atomicity requirements, in most cases, are not even documented at all. Therefore, in the end, programmers themselves must abide by these requirements and usually lack any tool support. Furthermore, in general, it is difficult to avoid errors in *atomicity-dependent programs*, especially in large projects, and even more

laborious and time-consuming is finding and fixing them. The papers [11, 13, 33, 37] discuss the importance of *atomicity-related bugs*, and they also show some bugs in *real-world programs*. Unfortunately, tool support for automatically discovering such kinds of errors is currently minimal.

As already mentioned, within the author's bachelor's thesis [15], *Atomer*[1] was proposed — a *static analyser* for finding some forms of *atomicity violations* implemented as a Facebook Infer's module. In particular, the stress is put on the *atomic execution of sequences of function calls*, which is often required, e.g., when using specific library calls. For example, assume the function `replace` from Listing 1.1 that replaces item `a` in an array by item `b`. It contains an atomicity violation — the index obtained may be outdated when `set` is executed (because, e.g., a *concurrent thread* can modify the array), i.e., `index_of` and `set` should be executed atomically. The analysis is based on the assumption that sequences of function calls executed *atomically once* should probably be executed *always atomically*. Hence, the checker naturally works with sequences. In fact, the idea of checking the atomicity of certain sequences of function calls is inspired by the works of *contracts for concurrency* [11, 33]. In the terminology of [11, 33], the atomicity of specific sequences of calls is the most straightforward (yet very useful in practice) kind of contracts for concurrency. However, while the idea of using sequences in the given context is indeed natural and rather exact, it quite severely limits the *scalability* of the analysis (indeed, even with a few functions, there can appear numerous different orders in which they can be called). Moreover, the implementation of the first version of Atomer targets mainly *C programs* using *PThread locks*. Consequently, there was no support for other languages and their locking mechanisms in the first version of Atomer.

```
1 void replace(int *array, int a, int b)
2 {
3     int index = index_of(array, a);
4     if (index >= 0) set(array, index, b);
5 }
```

Listing 1.1: An example of an *atomicity violation*

Within this thesis, Atomer has been *significantly improved and extended*. In particular, to improve scalability, working with *sequences* of function calls was *approximated* by working with *sets* of function calls. Furthermore, several new features were implemented: support for *C++ and Java*, including various advanced kinds of *locks* these languages offer (such as *re-entrant locks* or *lock guards*); or a more precise way of *distinguishing between different lock instances*. Moreover, the analysis has been *parameterised* by function names to concentrate on during the analysis and limits of the number of functions in *critical sections*. These parameters aim to reduce the number of false alarms. Their proposal is based on the author's analysis of false alarms produced by the first Atomer's version. Lastly, new experiments were performed to test capabilities of a new version of Atomer.

The development of the original Atomer started under the H2020 ECSEL projects AQUAS and Arrowhead Tools. The development of its new version is supported by the H2020 ECSEL project VALU3S. It has been discussed with the developers of Facebook Infer too. Parts of the thesis concerning the preliminaries and the basic version of Atomer are partially

[1]The implementation of **Atomer** is available at GitHub as an *open-source* repository (in a branch `atomicity-sets`): https://github.com/harmim/infer.

taken from the thesis [15]. Moreover, some preliminary results were also published in the Excel@FIT'21 paper [16] written by the author.

The rest of the thesis is organised as follows. Chapter 2 describes all the topics related to and essential to this thesis (including *static analysis*, *abstract interpretation*, *Facebook Infer*, and *contracts for concurrency*). The original version of *Atomer*, its limitations, and related work are described in Chapter 3. Subsequently, Chapter 4 presents all the proposed extensions and improvements. The implementation of these extensions is then covered in Chapter 5. The experimental evaluation of the new Atomer's features and other experiments performed within this thesis are discussed in Chapter 6 together with the future work. Finally, the thesis is concluded in Chapter 7. Besides, there are included the following appendices. The content of the attached memory media is listed in Appendix A. In the end, Appendix B serves as an installation and user manual.

Chapter 2

Preliminaries

This chapter explains the theoretical background that the thesis builds on. It also explains and describes the existing tools used in the thesis. Also, note that the author already partially published the contents of the sections in this chapter in his bachelor's thesis [15] and paper [16].

Multi-Threaded Programs [13, 26] *Multiple threads* of control are commonly used in the software development process because they help reduce *latency*, increase *throughput*, and better utilise *multiple processor computers*. Threads or *processes* are independent *sequences of instructions* that may be performed simultaneously. A process represents a single running program. It has its own address space and a unique identifier. One process can consist of multiple threads, i.e., a thread is a so-called *lightweight process*. All threads of a single process share the same address space (i.e., code and data). However, reasoning about the behaviour and correctness of a multi-threaded system is complex due to the need to consider all possible interleavings of the executions of different threads. An integral part of parallel programming is the *synchronisation* of individual threads. Usually, synchronisation mechanisms ensure the *mutual exclusion* of shared resources or synchronise actions that threads perform. Operating systems provide basic synchronisation primitives that can be often used in programming languages in a higher-level manner. Such fundamental mechanisms are *semaphores* (binary semaphores are called *mutexes*), *barriers*, *read-copy-update* techniques, or *monitors*. In this work (and also in practice), for simplicity, it will be used a notion of *locks* for semaphores, mutexes, monitors, etc. as a mechanism that mutually excludes access to a *critical section*.

The use of locks and access to shared data in parallel programs involves the risk of errors not known in sequential programming. Much previous work on checking *thread interference* has focused on *data races* (or, in general, on *race conditions*). A data race occurs when two threads simultaneously access the same data variable, and, at least, one of the access is a write. In practice, data races are commonly avoided by guarding each data structure with a lock. Unfortunately, the absence of data races is insufficient to guarantee the absence of errors due to unexpected interference between threads. [13, 37]

Atomicity Violations [13] Another possible source of errors due to unexpected interference between threads are *atomicity violations*. A method (or, in general, a code block)

is *atomic* iff, for every (arbitrarily interleaved) program execution, there is an equivalent execution with the same overall behaviour where the atomic method is executed serially. In other words, the method's execution is not interleaved with instructions of other threads. Also, atomicity provides a powerful, indeed maximal, guarantee of non-interference among threads. In short, atomicity is a generally applicable and fundamental *correctness property* of multi-threaded code. Nevertheless, commonly used *testing approaches* are deficient in verifying atomicity. While testing may reveal a concrete interleaving in which an atomicity violation causes erroneous behaviour, the exponentially large amount of possible interleavings does fundamentally impossible to get suitable *test coverage*. Research results [11, 13, 33, 37] have shown that defects related to atomicity are common, even in well-tested libraries.

It has to be described which blocks of code should be executed atomically to detect atomicity violations. Algorithms for the detection of this error often deal only with the atomicity of operations on variables. Examples of a complex description of *atomicity requirements* are *contracts* that require atomicity of the execution of certain methods/functions (defined in Section 2.4). [26]

Further, Section 2.1 outlines the fundamental notions and approaches in *program analysis*. In Section 2.2, there is an explanation of *static analysis* by *abstract interpretation*, which is used in *Facebook Infer*, i.e., the key framework used in this thesis. Facebook Infer, its principles, and features are covered in Section 2.3. The concept of *contracts for concurrency* is discussed and defined in Section 2.4.

2.1 Concepts in Program Analysis

This section provides a fundamental intuition about the main principles of *program analysis*. It discusses several standard techniques for reasoning about programs. The section is based on a few first chapters from the book [31] and the overview paper [19].

2.1.1 What to Analyse

The first question is *what programs to analyse*. An obvious characterisation of programs to analyse is the programming language in which the programs are written. Moreover, certain specific families of input programs may be distinguished. However, besides the language and the family of programs to consider, the way input programs are processed can also differ and affects how the analysis works. The most straightforward way to handle input programs is working directly with a *source code* just like a compiler would. Nevertheless, different representations of programs are used in program analysis likewise. The two below types of techniques are customarily distinguished.

Program-Level Analysis The first possibility is to run the analysis on a *source code* of input programs (e.g., programs written in conventional programming languages like C or Java; or hardware described in VHDL, Verilog, etc.) or on *executable binaries*. This technique typically involves some *front-end* comparable to a compiler's one that constructs the *syntax trees* of input programs.

Model-Level Analysis An alternative option is to consider a different input language that aims at modelling the semantics of programs. Then, the analysis takes as an input a description that *models* the program to analyse. Such models either need to be created manually, or some specialised tools are used. These models may hide implementation difficulties or inaccuracies. Examples of the models are automata of any kind, UML diagrams, Petri nets, Markov chains, or specialised modelling languages like, e.g., Promela.

2.1.2 Static vs Dynamic Approaches

Another critical question in program analysis is *when* the analysis is made. In particular, whether it operates *during* or *before* the program's execution.

One solution is to analyse programs at *run-time*, i.e., *during* the program's execution. This approach is called *dynamic analysis*. It takes place while the program computes, often, over a number of executions.

A second approach is to analyse the program *before* execution, which is called *static analysis*. It is done independently from any execution.

Static and dynamic approaches are significantly distinct in many ways. They come with different benefits and weaknesses. While dynamic approaches are often simpler to design and implement, they often have problems with performance at run-time. They do not force developers to fix bugs before program execution. Moreover, some properties cannot be checked (or, at least, it is very challenging) dynamically. For instance, dynamically detecting whether an execution does not *terminate* would require an infinite program run. Particular static and dynamic techniques are further discussed in Section 2.1.5.

2.1.3 Automation and Scalability

Automation is a further relevant aspect in program analysis. It would be ideal if program analysis methods were *fully automated* (i.e., no human help is needed). Unfortunately, this is not always possible due to the consequences of *Rice's theorem* [30]. Thus, sometimes, it is needed to give up on automation and let program analyses ask some *user input* (i.e., some human help is required). In that case, the user is asked to give some information to the analysis, e.g., *invariants*[1]. That is, the analysis is partially manual since users need to compute parts of the results themselves. However, having to provide this information may be unwieldy because input programs can be huge or complex.

Scalability is another essential characteristic of program analysis algorithms. Even if a program analysis is fully automatic, it is not guaranteed that it will generate a result within a reasonable time, depending on the complexity of the algorithms. A program analysis tool may not *scale* to extensive programs due to time costs or, e.g., memory constraints.

2.1.4 Soundness and Completeness

In order to preserve automation and/or scalability, the conditions about program analysis may be relaxed. Namely, the analysis can be proposed to return *inaccurate* results (altern-

[1] An **invariant** is a *logical property* that can be proved to be inductive for a given program.

atively, it can return a non-conclusive "do not know" answer). For this purpose, two dual properties (forms of *approximations* or *inaccuracies*) are used. To express these notions, let $\mathcal{L}$ be a *Turing-complete language*, φ be a *non-trivial semantic property* of interest of programs of $\mathcal{L}$, and T be an analysis tool that decides whether φ holds in a given program. Ideally, if T were absolutely precise, it would be such that:

$$\text{for every program } P \in \mathcal{L} : T(P) = true \iff P \models \varphi$$

The above, of course, can be decomposed into a pair of implications:

$$\begin{cases} \forall P \in \mathcal{L} : T(P) = true \implies P \models \varphi \\ \forall P \in \mathcal{L} : T(P) = true \impliedby P \models \varphi \end{cases}$$

Soundness A *sound* program analysis satisfies the first implication.

Definition 2.1.1 (**Soundness** [31])**.** The program analyser T is **sound** w.r.t. property φ whenever, for any program $P \in \mathcal{L}$, $T(P) = true \implies P \models \varphi$.

When a sound analysis terminates and claims that the analysed program has property φ, it guarantees that the program indeed satisfies φ, i.e., no errors are missed. In other words, a sound analysis will refuse all programs that do not satisfy φ. In terms of errors in *binary classification*, there are not *false-negative* errors[2] when using sound analysis. On the other hand, there is a chance of *false-positive* errors[3].

Completeness A *complete* program analysis satisfies the second implication.

Definition 2.1.2 (**Completeness** [31])**.** The program analyser T is **complete** w.r.t. property φ whenever, for any program $P \in \mathcal{L}$, $P \models \varphi \implies T(P) = true$.

A complete program analysis will accept every program that satisfies property φ. In other words, when a complete analysis refuses an analysed program, it is guaranteed that the program indeed fails to satisfy φ, i.e., there are no false-positive errors. However, there can be false-negative errors.

Due to the *computability* barrier [30], it is not possible to design a general analysis to determine which programs satisfy any non-trivial property for a Turing-complete language that is sound, complete, fully automated, and scalable at the same time. Some of these have to be sacrificed, or the analysis has to be proposed to operate only on a specific set of input programs.

2.1.5 Program Analysis Techniques

This section describes several standard *program analysis techniques*. Although this thesis focuses on *static analysis* (or maybe rather *bug finding* based on static analysis), it is essential to see the differences between other analysis techniques. Hence, a brief overview is presented in the below sections. Finally, Table 2.1 compares the techniques (how they are usually used) based on the earlier criteria.

[2] A **false-negative** error is a real error that is undetected by an analysis tool.

[3] A **false-positive** error (also called a **false alarm**) is a *spurious error*, i.e., it is detected by an analysis tool, but the error does not exist in the real program.

Testing and Dynamic Analysis The *testing* approach checks a finite set of finite program executions. In the development process, several levels of testing are used at various stages of the software/hardware life-cycle, e.g., *unit testing* or *integration testing*. In general, it is challenging to achieve suitable *test coverage* because of infinite program paths when using, e.g., *random testing*. However, several more advanced techniques that improve the coverage have been introduced. These techniques are often combined with other verification approaches. For instance, *concolic testing* combines testing with *symbolic execution*, *dynamic analysis* that observes behaviour in a testing run (such behaviour can be *extrapolated* to behaviour not seen in the given testing run), or *search-based techniques* that can generate test data or parameters. Moreover, to test programs with non-deterministic semantics (i.e., concurrent programs), techniques like *noise injection* are applied. In particular, many advanced dynamic analysers have been proposed to detect data races or deadlocks, including, e.g., Eraser or FastTrack.

Testing has the following features. In general, it is simple to automate. It is unsound in almost all cases (besides programs that have a finite number of finite paths). Since failed testing runs provide incorrect concrete executions, the testing is complete.

Other analysis techniques mentioned below use a static approach. The significant difference between static and dynamic approaches is the following. It is well-known that testing may expose errors, but it cannot prove their absence. It was also famously stated by Edsger W. Dijkstra: "*Program testing can be used to show the presence of bugs, but never to show their absence!*". However, static approaches may be able to prove their absence — with some *approximation* — they can check *all possible executions* of a program and provide guarantees about its properties. Another static approaches' benefit is that the analysis can be performed during the development process, so the program does not have to be executable yet, and it already can be analysed. The biggest drawback of static approaches, in general, is that they can produce a lot of false alarms (though this is not the case, e.g., for *theorem proving*), which is often resolved by accepting unsoundness. Another crucial issue of static approaches (this is, however, also an issue of dynamic analyses) is ensuring sufficient scalability — in fact, typically, the more precise the analysis, the less scalable it becomes.

Deductive Verification The *semi-automated* approach *deductive verification* (or *theorem proving*) uses *inference systems* for inferring theorems about the analysed system from the facts known about the system and from general theorems of different logical theories. The approach falls under *machine-assisted* techniques, which means that users may be required to provide extra information to the analysis (usually *loop invariants*, procedure *pre-conditions/post-conditions*, assertions, or some other invariants). This can be demanding and require some level of expertise. However, a substantial part of the verification can usually still be carried out in a fully automated fashion. This approach is very general, but there is a problem with generating diagnostic information for incorrect systems. There exist a number of *interactive theorem proving* tools like Coq, Isabelle/HOL, PVS, ACL2, etc. The user usually guides the inference process in these tools.

These techniques also involve *automated decision procedures* (or *satisfiability solvers*) for various logical theories. Such solvers are often used as back-end components for higher-level verification methods, such as *symbolic execution* or *predication abstraction* in model checking. Commonly used solvers are *SAT-solvers* (e.g., CaDiCaL and Glucose) and *SMT-*

solvers (e.g., Z3 and CVC4). Various tools allow the user to provide some *logical annotations* in a code and then automatically attempt to prove specific properties using decision procedures. Examples of such tools are VCC and Dafny.

Theorem proving techniques have the following properties. They are not fully automatic, i.e., *high user expertise* is often needed. They are sound w.r.t. the model of the program used in the proof, and they are usually complete up to the capabilities of the proof assistant.

Model Checking Another technique called (*finite-state*) *model checking* aims at finite systems. It automatically verifies whether a system or its model satisfies a particular property based on an *algorithmic exploration of the system's state space*. Unfortunately, the biggest issue here is the *state space explosion* problem [35]. However, in practice, model checking tools use effective data structures (such as *binary decision diagrams* or *hierarchical storage of states*) to describe program behaviours and avoid enumerating all executions thanks to approaches that reduce the search space. In addition, other techniques are used to cope with this problem. For instance, various abstractions are used (e.g., *predicate abstraction*) or *bounded model checking*[4]. Properties are usually defined using *temporal logics*, such as LTL, CTL, or PCTL. Finite model checking has the following characteristics. It is automatic (up to the need of modelling the system or its environment). It is sound and usually complete, w.r.t. the model. Other advantages are that it is pretty general and provides diagnostic data for incorrect systems.

Model checking is typically done at the *model level*, i.e., a model of the program needs to be built, either manually or automatically. In practice, model checking tools usually implement a front-end for that purpose. The problem is that the model generally cannot precisely capture the input program's behaviours since programs are usually infinite systems. Thus, the checking of the synthesised model may be either incomplete or unsound, w.r.t. the input program. Some model checking techniques can automatically *refine* the model when they fail to prove a property due to a *spurious counterexample*, although a *termination* must be ensured. In practice, model checking tools are often sound and incomplete w.r.t. the input program.

Model checking has found many successful applications, including hardware verification, verification of *concurrent* and *distributed* systems, *probabilistic* systems, *biological* systems, etc. Examples of hardware model checkers are RuleBase, Incisive Verifier, or NuSMV. Model checkers for concurrent and distributed systems include Spin or DIVINE. Tools, such as CPAchecker or BLAST, are model checkers that use predicate abstraction. PRISM and Storm are state-of-the-art tools for model checking probabilistic systems. Finally, Uppaal is a model checker that verifies temporal logic formulas on *timed automata*.

Static Analysis The last technique (and the most important one for the thesis) is *static analysis*. It relies on other techniques to compute *conservative* (sound but incomplete) descriptions of program behaviours using finite resources rather than building a finite model. The fundamental idea is to finitely *over-approximate* the set of all program behaviours using a particular set of properties.

[4] **Bounded model checking** explores models up to a *fixed depth* only. This technique is often referred to as *bug finding* because it sacrifices completeness and often also soundness. Examples of tools that implement such technique are CMBC, LLBMC, or JBMC.

According to [27], static analysis of programs is reasoning about the behaviour of computer programs without really executing them. It has been used since the 1960s in *optimising compilers* for generating efficient code. More recently, it has proven valuable also for automatic error detection, verification of the correctness of programs, and it is used in other tools that can help programmers. Intuitively, a static program analyser is a program that reasons about the behaviour of other programs by looking for some *syntactic patterns* in the code and/or by assigning the program statements some *abstract semantics* and then deriving a characterisation of the behaviour in terms of the abstract semantics. Nowadays, static analysis is one of the leading concepts in *formal verification*. It can even automatically answer considerably complicated questions about a given program, such as [27]:

- Does the program *terminate* on every input?
- Do two pointers refer to *disjoint data structures* in memory?
- Are *data races* possible? Can the program *deadlock*?
- Does there exist an input that leads to a *null-pointer dereference*, *division-by-zero*, or *arithmetic overflow*?
- Are arrays always accessed *within their bounds*?
- **Are certain operations executed *atomically*?**

In general, conservative static analysis has the following characteristics. It is automatic, it can often handle large systems, and it does not ordinarily need a model of the environment. It produces sound results, and it is generally incomplete, i.e., it can produce many false alarms. However, it is possible to drop soundness to minimise the number of false alarms and preserve automation, which, in fact, is done by many tools. Such techniques are often instead called *bug finding* or *bug hunting*, which is based not only on static analysis but also, e.g., on model checking. Since the primary motivation of these approaches is to discover bugs (and not to prove their absence), they are neither sound nor complete, and they aim at swiftly catching bugs.

The above explanation of static analysis is quite general. Thus, even model checking or deductive verification may be considered as static analysis. However, when setting these approaches aside, the most traditional techniques in the static analysis include:

- *Syntactic checks* looking for various *bug patterns* (*anti-patterns*) — implemented, e.g., in Lint, Cppcheck, or in analyses in VisualStudio, Clang, GCC.
- *Data-flow analysis* — it is often combined with bug pattern searching, and it is usually unsound. Many successful tools of this kind have been implemented. They are often proprietary and hard to openly evaluate or extend. Examples of some well-known tools are Coverity, CodeSonar, Klocwork, FindBugs/SpotBugs, SonarQube.
- *Constraint-based analysis*, *type-based analysis* (type and effect systems).
- *Symbolic execution* — implemented, e.g., in KLEE, Pex, Symbiotic.

- *Abstract interpretation* (see Section 2.2) — examples of sound tools are Polyspace, AbsInt/Astrée, or Sparrow. State-of-the-art frameworks that allow creating sound analysers include Facebook Infer (see Section 2.3), Facebook SPARTA, or Frama-C. These tools also offer already implemented sound as well as unsound checkers.

Table 2.1: A summary of *program analysis techniques* [31]

Technique	**Automatic**	**Sound**	**Complete**	**Object**	**Approach**
Testing	Yes	No	Yes	Program	Dynamic
Deductive verification	No	Yes	Yes/No	Model/Program	Static
Model checking	Yes	Yes	Yes/No	Model/Program	Static
Conservative static analysis	Yes	Yes	No	Program	Static
Bug finding	Yes	No	No	Program	Static

2.2 Abstract Interpretation

This section explains and defines the basics of static analysis technique *abstract interpretation*. The description is based on [6, 7, 8, 9, 10, 19, 21, 27, 28, 31]. In these works, there can be found a more detailed and formal explanation.

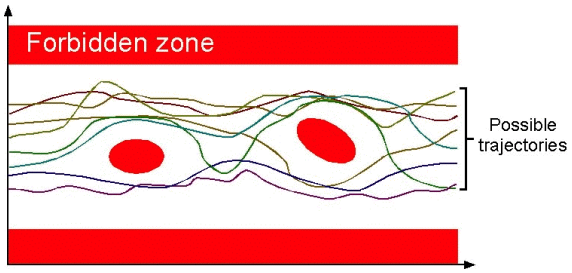
Abstract interpretation was introduced and formalised by French computer scientist Patrick Cousot and his wife Radhia Cousot at POPL'77 [9]. It is a generic *framework* for static analyses. It allows one to create particular analyses by providing specific *components* (discussed in Section 2.2.1) to the framework. The obtained analysis is guaranteed to be *sound* if specific properties of the components are met.

In general, in *set theory*, which is independent of the application setting, abstract interpretation is considered a theory for *approximating* sets and set operations. A more restricted formulation of abstract interpretation is to interpret it as a theory of approximation of the behaviour of the *formal semantics* of programs. *Fixpoints* may characterise those behaviours (see Section 2.2.3), which is why the primary part of the theory provides efficient techniques for *fixpoint approximation* [28]. Therefore, for standard semantics, abstract interpretation is used to derive the approximate abstract semantics over an *abstract domain* (explained in Section 2.2.1). The abstract semantics obtained from program analysis can then be used for verification, optimisation, code generation, etc. [8]

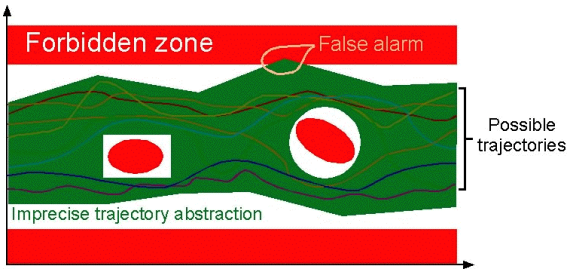
To be sound, it is essential that an approximation performed using abstract interpretation should be an *over-approximation*. This means that the considered abstract semantics should be a *superset* of the concrete semantics, i.e., the abstract semantics should cover all possible cases. Whence, if the abstract semantics satisfies a given property, then the concrete semantics satisfies it too. Moreover, the consequence of the over-approximation of the possible executions is that inexisting executions are considered, leading to *false alarms*. Thus, due to a lack of accuracy, abstract interpretation is usually *incomplete*. [7]

Patrick Cousot intuitively and informally illustrates abstract interpretation in [7] as follows. Figure 2.1a shows the concrete semantics of a program by a set of curves, which represents

the set of all possible executions of the program in all possible execution environments. Each curve shows the evolution of the vector $x(t)$ of input, state, and output values of the program as a function of the time t. *Forbidden zones* on this figure represent a set of erroneous states of the program execution. Proving that the intersection of the concrete semantics of the program with the forbidden zones is empty may be *undecidable* because the program concrete semantics is, in general, *not computable*. As demonstrated in Figure 2.1b, abstract interpretation deals with the abstract semantics, i.e., the superset of the concrete program semantics. The abstract semantics includes all possible executions. That implies that if the abstract semantics is safe — i.e., it does not intersect the forbidden zones — the concrete semantics is safe as well. However, the over-approximation of the possible program executions causes that inexisting program executions are considered, which leads to false alarms, as demonstrates the figure. It is the case when the abstract semantics intersects the forbidden zone, whereas the concrete semantics does not intersect it.



(a) The *concrete semantics* of programs with *forbidden zones*



(b) The *abstract semantics* of programs with *imprecise abstraction*

Figure 2.1: An *abstract interpretation* demonstration [7] (horizontal axes: the time t; vertical axes: the vector $x(t)$ of input, state, and output values of the considered program)

2.2.1 Components of Abstract Interpretation

Before the formal definition of abstract interpretation is given, below are intuitively described basic components of the framework in accordance with [6, 8, 21, 28]:

- **Abstract Domain Q**:
  - An abstraction of the possible concrete program states (or their parts) in the form of *abstract properties*[5].
  - In other words, it is a set of *abstract states* (or *abstract contexts*), where an abstract state represents a set of program states reachable at a particular program location.
  - For instance, several practical domains have been defined: numerical intervals, polyhedra, octagons, congruences, or various heap domains.
- **Abstract Transformers τ**:
  - There is a *transform function* τ for each program operation (instruction) representing the impact of the operation executed on an abstract state.

[5] **Abstract properties** approximate *concrete properties* behaviours. [8]

- **Join Operator** $\sqcup$:
  - Joins abstract states from individual program branches into a single one.
- **Widening Operator** $\triangledown$:
  - Enforces *termination* of the abstract interpretation.
  - It is used to over-approximate the *least fixed points* of program semantics (it is performed on a sequence of abstract states at a certain location).
  - Usually, the later in the analysis this operator is applied, the more accurate the result is (but the analysis takes more time).
- **Narrowing Operator** $\triangle$:
  - The approximation obtained by the widening operator can be *refined* using the narrowing operator, i.e., it may be used to refine the result of widening.
  - It is used when a fixpoint is approximated by the widening operator.

Note that neither the widening operator nor narrowing operator are required. However, at least the widening operator is frequently used. The narrowing operator can be sometimes dropped.

2.2.2 Formal Definition of Abstract Interpretation

The definitions below consider notions from *lattice theory*. More information about lattices, functions on lattices, *partial orders*, and set theory can be found, e.g., in [27, 28, 31].

Definition 2.2.1 (**Abstract Interpretation**)**.** According to [9, 21], **abstract interpretation** I of a program P with the instruction set $Instr$ is a tuple

$$I = (Q, \sqcup, \sqsubseteq, \top, \bot, \tau)$$

where

- Q is the *abstract domain* (the set of *abstract states*),
- $\sqcup : Q \times Q \to Q$ is the *join operator* for accumulation of abstract states,
- $\sqsubseteq \subseteq Q \times Q$ is an *ordering* defined as $x \sqsubseteq y \iff x \sqcup y = y$ where
  - $\langle Q, \sqsubseteq \rangle$ is a *complete* $\sqcup$*-semilattice*,
- $\top \in Q$ is the *supremum* of $\langle Q, \sqsubseteq \rangle$,
- $\bot \in Q$ is the *infimum* of $\langle Q, \sqsubseteq \rangle$ — thus, $\langle Q, \sqsubseteq \rangle$ is, in fact, a *complete lattice*,
- $\tau : Instr \times Q \to Q$ defines the *abstract transformers* for specific instructions,
  - τ must be *monotone*[6] on Q for each instruction from $Instr$.

[6] Assume two *partially-ordered sets* $\langle E, \preceq_E \rangle$ and $\langle F, \preceq_F \rangle$, and a function $f : E \to F$. f is **monotone** iff $\forall x, y \in E : x \preceq_E y \implies f(x) \preceq_F f(y)$. [31]

Using a so-called *Galois connection* [9, 10, 21, 27, 28, 31], one can ensure the *soundness* of the abstract interpretation, i.e., the correspondence between the *concrete semantics* and its *abstract semantics* can be formalised by a Galois connection (also a so-called pair of *adjoined functions*).

Definition 2.2.2 (Galois Connection). If $\mathcal{P} = \langle P, \preceq \rangle$ and $\mathcal{Q} = \langle Q, \sqsubseteq \rangle$ are *partially-ordered sets*, then a quadruple $\pi = (\mathcal{P}, \alpha, \gamma, \mathcal{Q})$ is a **Galois connection**, written

$$\langle P, \preceq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle Q, \sqsubseteq \rangle$$

iff $\alpha : P \to Q$ and $\gamma : Q \to P$ are *monotone* functions such that

$$\forall p \in P : \forall q \in Q : \alpha(p) \sqsubseteq q \iff p \preceq \gamma(q)$$

where P is the concrete domain and Q is the abstract domain. Furthermore, α is an *abstraction function*, and γ is a *concretisation function*. Consequently, $\alpha(p)$ is the abstraction of p, i.e., the most precise approximation of $p \in P$ in Q, and $\gamma(q)$ is the concretisation of q, i.e., the most imprecise element of P which $q \in Q$ can soundly approximate. A Galois connection is illustrated in Figure 2.2.

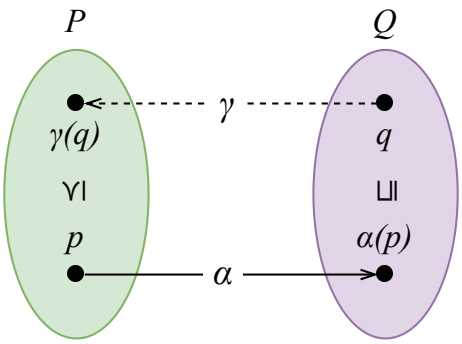


Figure 2.2: An illustration of a *Galois connection*

If the abstraction and concretisation functions of the abstract interpretation form a Galois connection, then applying the abstraction function and concretising the result back yield a less (or equally) precise result. However, it is a safe (*conservative*) approximation. That is, formally written, $\forall p \in P : p \preceq \gamma(\alpha(p))$. Finally, the abstract interpretation may only over-approximate the concrete semantics (i.e., it is sound) if for each *concrete transformer* (instruction) $i : P \to P$ from the instruction set $Instr$ and the appropriate *abstract transformer* $\tau : Instr \times Q \to Q$ respects a Galois connection. This is $\forall p \in P : \alpha(i(p)) \sqsubseteq \tau(i, \alpha(p))$.

2.2.3 Fixpoint Approximation [9, 10, 21, 27, 28, 31]

Most program properties can be represented as *fixpoints*[7]. This reduces the program analysis to the *fixpoint approximation*. The complete analysis of a program using abstract interpretation can be then viewed as finding the *least/greatest* fixpoint of the equation

[7] Consider a function $f : L \to L$ on a set L. A **fixed point** (or **fixpoint**) of f is an element $l \in L$ iff $f(l) = l$. The set of all fixpoints of f is denoted as $Fix(f) = \{l \in L \mid f(l) = l\}$. [28]

$\overline{Q} = \overline{\tau}(\overline{Q})$, where $\overline{Q}$ is a vector of abstract states and $\overline{\tau}$ is an extension of τ to the whole program. This fixpoint equation is then solved iteratively. *Knaster-Tarski theorem* [34] guarantees these fixpoints' existence.

The computation of the most precise abstract fixpoint is not generally guaranteed to *terminate*, in particular, when a given program contains a loop (or recursion) and uses an infinite domain (or even finite but very large). In order to enforce or accelerate the convergence, the fixpoint is often over-approximated using the *widening* operator ∇. The approximation may be later refined using the *narrowing* operator Δ. These two operators are defined below. In practice, the analysis is usually done by iterating the abstract transformers over the *control-flow graph*[8] (CFG). The join operator $\sqcup$ is applied at program locations where different branches come across. Moreover, the widening operator ∇ is applied at loop junctions (and afterwards, the narrowing operator Δ may be used).

Definition 2.2.3 (**Widening**)**.** Assume the abstract interpretation from Definition 2.2.1. The **widening operator** over the abstract domain Q is a binary operator $\nabla : Q \times Q \to Q$ such that $\forall x, y \in Q : x \sqcup y \sqsubseteq x \nabla y$, and for all *increasing chains* $x_0 \sqsubseteq x_1 \sqsubseteq \ldots$, the increasing chain defined by $y_0 = x_0, \ \ldots \ , \ y_{i+1} = y_i \nabla x_{i+1}, \ \ldots$ is not *strictly increasing*. Note that the chain eventually stabilises since the result of ∇ is an *upper bound*.

Definition 2.2.4 (**Narrowing**)**.** Assume the abstract interpretation from Definition 2.2.1. The **narrowing operator** over the abstract domain Q is a binary operator Δ: $Q \times Q \to Q$ such that $\forall x, y \in Q : y \sqsubseteq x \implies y \sqsubseteq x \Delta y \sqsubseteq x$, and for all *decreasing chains* $x_0 \sqsupseteq x_1 \sqsupseteq \ldots$, the decreasing chain defined by $y_0 = x_0, \ \ldots \ , \ y_{i+1} = y_i \Delta x_{i+1}, \ \ldots$ is not *strictly decreasing*. Note that the chain eventually stabilises since the result of Δ is a *lower bound*.

2.3 Facebook Infer — Static Analysis Framework

This section describes the principles and features of *Facebook Infer*. The description is based on information provided at the Facebook Infer's website[9] and in [2].

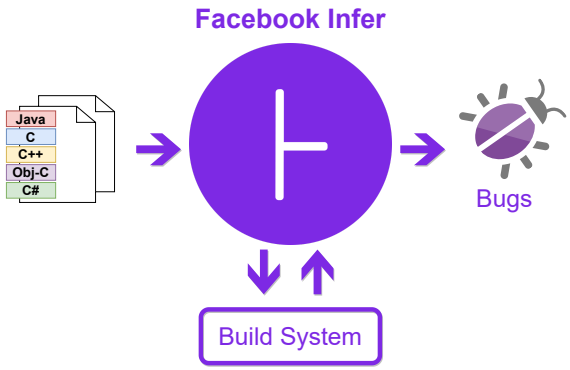


Figure 2.3: Static analysis in *Facebook Infer*

[8] A **control-flow graph** (**CFG**) is a *directed graph* in which the nodes represent *basic blocks* of the program, and the edges represent control-flow paths among them [1]. A basic block is a *maximal sequence of statements* such that all transfers to it are to the first statement in the sequence, and all statements in the sequence are executed sequentially [28].

[9] The **Facebook Infer's** website: `https://fbinfer.com`.

Facebook Infer is an *open-source*[10] *static analysis framework*, which can discover various kinds of software bugs and which stress the *scalability* of the analysis. The Facebook Infer's basic usage is illustrated in Figure 2.3. A more detailed explanation of its architecture is given in Section 2.3.2. Facebook Infer is implemented in *OCaml*[11] — a *functional* programming language, also supporting *imperative* and *object-oriented* paradigms. Further details about OCaml can be found in the book [25]. Infer has initially been a rather specialised tool focused on *sound verification* of the absence of *memory safety violations*, which was first published in the well-known paper [5]. Once Facebook has purchased it, its scope significantly widened and abandoned the focus on sound analysis only.

Facebook Infer can analyse programs written in the following languages: C, C++, Java, Objective-C (and support for C# has been recently implemented [36]). Moreover, it is possible to extend Facebook Infer's *frontend* for supporting other languages. Currently, Facebook Infer contains many analyses focusing on various kinds of bugs, e.g., *Inferbo* [38] (buffer overruns); *RacerD* [3, 4, 14] (data races); and other analyses that check for buffer overflows, some forms of deadlocks and starvation, null-dereferencing, memory leaks, resource leaks, etc. Since Facebook Infer is a relatively popular and open-source framework, many experimental analysers arise pretty often. For instance, the promising experimental deadlock checker L2D2 [23] has been implemented at FIT BUT not long ago.

2.3.1 Abstract Interpretation in Facebook Infer

Facebook Infer is a general framework for static analysis of programs, and it is based on *abstract interpretation*. Despite the original approach taken from [5], Facebook Infer aims to find bugs rather than perform *formal verification*. It is still possible to propose *sound* and *complete* analyses in the framework. However, the majority of the checkers already implemented in Facebook Infer are both unsound and incomplete.

It can be used to develop new sorts of *compositional* and *incremental* analysers quickly (both *intraprocedural* and *interprocedural* [28]) based on the concept of function *summaries*. In general, a summary χ represents a set of *pre-condition/post-condition* pairs for a function. In particular, it records under which pre-condition a function can be performed leading to a given post-condition. Formally, it can be described using *Hoare triples* [17], because it can be viewed as a triple $\{P\}\ S\ \{R\}$, where P is a pre-condition, S is a program (or a single statement/command)[12], and R is a description of the result of the execution of S (i.e., a post-condition). In theory, P and R are considered formulae in a suitable logic. The triple may be interpreted as follows. Suppose S is executed from a state satisfying P, and the execution of S terminates. In that case, the program state after S terminates satisfies R. However, in practice, a summary is a custom data structure that may be used for storing any information resulting from the analysis of particular functions. Usually, a summary consists of the relevant parts of *abstract states* for a particular analysis.

Facebook Infer generally does not compute the summaries during the analysis along the CFG as it is done in classical analyses based on the concepts from [29, 32]. Instead, Facebook Infer performs the analysis of a program *function-by-function along the call-tree*, starting from its leaves (demonstrated later in Example 2.3.1). Therefore, a function is

[10] The **Facebook Infer's open-source repository** at GitHub: https://github.com/facebook/infer.
[11] The **OCaml's** website: https://ocaml.org.
[12] In Facebook Infer's summaries considered as Hoare triples, S is usually one function.

analysed, and a summary is computed without knowledge of the call context. Then, the summary of the function is used at all its call sites. It is needed to deduce under which pre-conditions a function can produce post-conditions appropriate for the given analysis. Since the summaries do not differ for different contexts, each function is analysed precisely once, and the analysis becomes more scalable, but it can lead to a loss of accuracy. However, of course, it is more troublesome to propose such a *bottom-up* analysis.

In order to create a new intraprocedural analyser in Facebook Infer, it is required to define the following (the listed items are described in more detail in Section 2.2):

1. The *abstract domain* Q, i.e., the type of *abstract states*.
2. The *ordering operator* $\sqsubseteq$, i.e., an ordering of abstract states.
3. The *join* operator $\sqcup$, i.e., the way of joining two abstract states.
4. The *widening* operator $\triangledown$, i.e., the way how to enforce termination of the computation.
5. The *transfer functions* τ, i.e., transformers that take an abstract state as an input and produce an abstract state as an output.

Furthermore, to create an interprocedural analyser, it is required to define additionally:

1. The type of function summaries χ.
2. The logic for using summaries in transfer functions and the logic for transforming an intraprocedural abstract state to a summary.

An important Facebook Infer's feature, which improves its scalability, is the *incrementality* of the analysis. It allows one to analyse separate code changes only, instead of analysing the whole codebase. It is more suitable for extensive and variable projects where ordinary analysis is not feasible. The incrementality is based on *re-using summaries* of functions for which there is no change in them neither in the functions transitively invoked from them, as shown in Example 2.3.1 later on.

2.3.2 Architecture of the Infer.AI Framework

The architecture of the abstract interpretation framework of Facebook Infer (**Infer.AI**) may be split into three major parts, as demonstrated in Figure 2.4: the *frontend*, an *analysis scheduler* (and the *results database*), and a set of *analyser plugins*.

The frontend compiles input programs into the *Smallfoot Intermediate Language* (SIL) and represents them as a CFG. There is a separate CFG representation for each analysed function. Nodes of this CFG are formed as SIL instructions. The SIL language consists of the following underlying instructions:

- `LOAD` — reading into a temporary variable;
- `STORE` — writing to a program variable, field of a structure, or array;
- `CALL` — a function call;

- `PRUNE e` — the evaluation of an expression `e` in a condition, cycle, etc.

The frontend allows one to propose *language-independent* analyses (to a certain extent) because it supports input programs to be written in multiple languages.

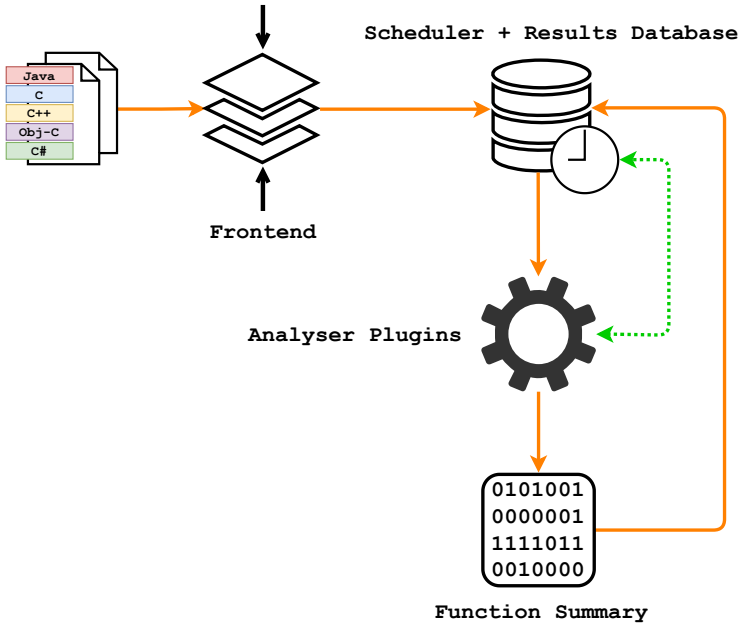


Figure 2.4: The *architecture* of the Facebook Infer's *abstract interpretation framework* [15]

The next part of the architecture is the scheduler, which defines the order of the analysis of single functions according to the appropriate *call graph*[13]. The scheduler also checks if it is possible to simultaneously analyse some functions, allowing Facebook Infer to run the analysis in parallel.

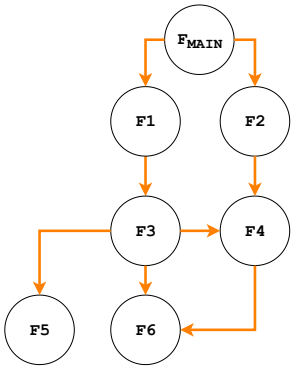


Figure 2.5: A *call graph* for an illustration of Facebook Infer's *analysis process* [15]

Example 2.3.1. For demonstrating the order of the analysis in Facebook Infer and its incrementality, assume the call graph given in Figure 2.5. At first, leaf functions `F5` and `F6` are analysed. Further, the analysis goes on towards the root of the call graph — $\texttt{F}_{\texttt{MAIN}}$, while considering the dependencies denoted by the edges. This order ensures that a summary is available once a nested function call is abstractly interpreted within the analysis. When there is a subsequent code change, only directly changed functions and all the functions up the call path are re-analysed. For instance, if there is a change of source code of function `F4`, Facebook Infer triggers re-analysis of functions `F4`, `F2`, and $\texttt{F}_{\texttt{MAIN}}$ only.

The last part of the architecture consists of analyser plugins. Each plugin performs some analysis by interpreting SIL instructions. The result of the analysis of each function (func-

[13] A **call graph** is a *directed graph* describing call dependencies among functions.

tion summary) is stored in the results database. The interpretation of SIL instructions (*commands*) is made using the *abstract interpreter* (also called the *control interpreter*) and *transfer functions* (also called the *command interpreter*). The transfer functions take a previously generated abstract state of an analysed function as an input, and by applying the interpreting command, produce a new abstract state. The abstract interpreter interprets the command in an abstract domain according to the CFG. This workflow is shown in a simplified form in Figure 2.6.

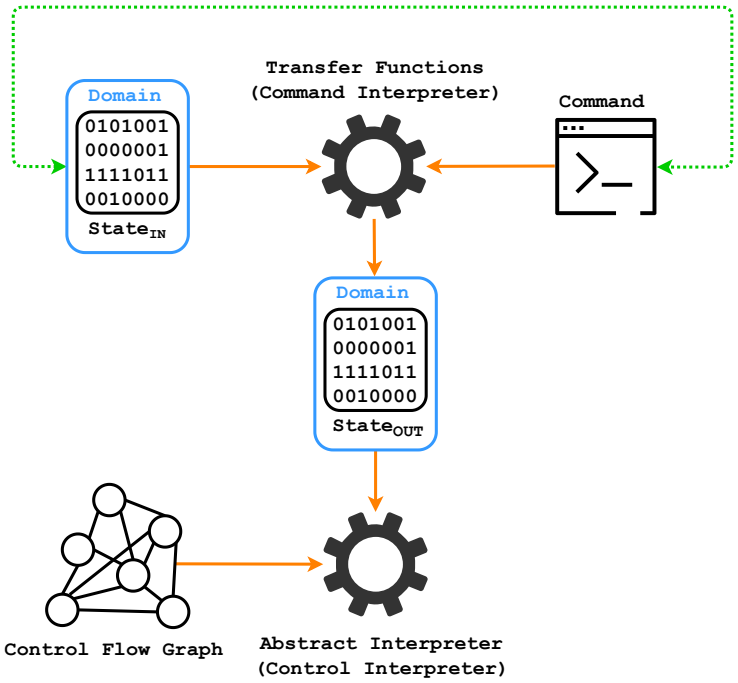


Figure 2.6: The Facebook Infer's *abstract interpretation process* [15]

2.4 Contracts for Concurrency

This section introduces the concept of *contracts for concurrency* [11, 33]. Examples and listings in the section are pieces of programs written in the C language.

Respecting the *protocol* of a software module — that defines which *sequences of functions* are legal to invoke — is one of the requirements for the correct behaviour of the module. For example, a module that deals with a file system typically requires that a programmer using this module should call a function `open` at first, followed by an optional number of functions `read` and `write`, and at last, call a function `close`. A program utilising such a module that does not follow this protocol is erroneous. The methodology of *design by contract* [24] requires programs to meet such well-defined behaviours. [33]

In *concurrent programs*, contracts for concurrency allow one — in the simplest case — to specify sequences of functions that are needed to be *executed atomically* in order to avoid *atomicity violations*. In general, contracts for concurrency specify sets of sequences of calls called *spoilers* and sets of sequences of calls called *targets*. It is then required that no target overlaps fully with any spoiler. A developer may manually specify such contracts

or automatically generate them by a program (analyser). These contracts can be used to verify the correctness of programs as well as they can serve as helpful documentation.

Section 2.4.1 defines the notion of *basic contracts for concurrency*. Further, Section 2.4.2 defines contracts extended to consider the *data flow* between functions, where a sequence of function calls must be atomic only if they handle the same data. Finally, Section 2.4.3 presents those mentioned above more general contracts for concurrency with spoilers and targets, which essentially extend the basic contracts with some *contextual information*.

2.4.1 Basic Contracts for Concurrency

In [11, 33], a *basic contract for concurrency* is formally defined as follows. Let $\Sigma_{\mathbb{M}}$ be a set of all function names of a software module $\mathbb{M}$. A contract is a set $\mathbb{R}$ of *clauses*, where each clause $\varrho \in \mathbb{R}$ is a *star-free regular expression*[14] over $\Sigma_{\mathbb{M}}$. A *contract violation* occurs if any of the sequences expressed by the contract clauses are interleaved with the execution of functions from $\Sigma_{\mathbb{M}}$. In other words, each sequence specified by any clause ϱ must be executed atomically. Otherwise, there is a violation of the contract. The number of sequences defined by a contract is finite since the contract is a union of *star-free languages*.

Example 2.4.1. Consider the following example from [11, 33]. Assume a module $\mathbb{M}$ implementing a resizable array of integers with the following interface functions $\Sigma_{\mathbb{M}} = \{$`add, contains, index_of, get, set, remove, size`$\}$ defined as:

```
void add(int *array, int element)

bool contains(int *array, int element)

int index_of(int *array, int element)

int get(int *array, int index)

void set(int *array, int index, int element)

void remove(int *array, int index)

int size(int *array)
```

The module's contract contains clauses $\mathbb{R} = \{\varrho_1, \varrho_2, \varrho_3, \varrho_4\}$ such that:

(ϱ_1) `contains index_of`

The execution of `contains` followed by the execution of `index_of` should be atomic. Otherwise, the program may fail to get the index because after checking the presence of an element in an array, it can be removed by some concurrent thread.

(ϱ_2) `index_of ( get | set | remove )`

The execution of `index_of` followed by the execution of `get`, `set`, or `remove` should be atomic. Otherwise, the received index may be outdated when applied to address an array element because a concurrent modification of the array may shift the element's position.

[14] **Star-free regular expressions** are regular expressions that use only the *concatenation operator* and the *alternative operator* (|), without the *Kleene star operator* (*).

(ϱ_3) `size ( get | set | remove )`

The execution of `size` followed by the execution of `get`, `set`, or `remove` should be atomic. Otherwise, an array may be empty when accessing it because of a concurrent change of the array. This can be an issue since a given index is not in a valid range anymore (e.g., testing `index < size`).

(ϱ_4) `add index_of`

The execution of `add` followed by the execution of `index_of` should be atomic. Otherwise, the added element needs no longer exist in an array.

2.4.2 Contracts for Concurrency with Parameters

The above definition of basic contracts for concurrency is quite limited in some circumstances and can consider valid programs as erroneous (i.e., *false alarms* may be reported). Hence, in this section, there is introduced an extension of basic contracts — *contracts for concurrency with parameters* (defined in [11, 33]) — which takes into consideration the *data flow* within function calls.

Example 2.4.2. Consider the following example from [11, 33], given Listing 2.1. There is the function `replace` that replaces item `a` in an array by item `b`. The implementation of this function comprises two atomicity violations:

1. when `index_of` is invoked, item `a` does not need to be in the array anymore; and
2. the acquired index can be obsolete when `set` is executed.

A basic contract could cover this scenario by the following clause:

(ϱ_5) `contains index_of set`

It can be obtained from the composition of clauses ϱ_1 and ϱ_2. Nevertheless, this definition is too restrictive because the functions are required to be executed atomically only if `contains` and `index_of` have the same arguments `array` and `element`; `index_of` and `set` have the same argument `array`; and the returned value of `index_of` is used as the argument `index` of the function `set`.

```
1 void replace(int *array, int a, int b)
2 {
3     if (contains(array, a))
4     {
5         int index = index_of(array, a);
6         set(array, index, b);
7     }
8 }
```

Listing 2.1: An example of an atomicity violation with *data dependencies* [11, 33]

To respect function call *parameters* and *return values* of functions in contracts, the basic contracts are extended by dependencies among functions in [11, 33] as follows. Function

call parameters and return values are expressed as *meta-variables*. Further, if a contract should be required to be observed exclusively if the same object emerges as an argument or as the return value of multiple calls in a given call sequence, it may be denoted by using the same meta-variable at the position of all these occurrences of parameters and return values.

Clause ϱ_5 can be extended as follows (repeated application of meta-variables `X/Y/Z` requires the same objects $o_1/o_2/o_3$ to be used at the positions of `X/Y/Z`, respectively; and the underscore indicates a *free meta-variable* that does not restrict the contract clause):

$$(\varrho_5')\ \texttt{contains(X, Y) Z = index\_of(X, Y) set(X, Z, \_)}$$

Example 2.4.3. With the extension described above, it is possible to extend the contract from Example 2.4.1 to capture the dependencies between function calls as follows:

(ϱ_1') `contains(X, Y) index_of(X, Y)`

(ϱ_2') `Y = index_of(X, _) ( get(X, Y) | set(X, Y, _) | remove(X, Y) )`

(ϱ_3') `size(X) ( get(X, _) | set(X, _, _) | remove(X, _) )`

(ϱ_4') `add(X, Y) index_of(X, Y)`

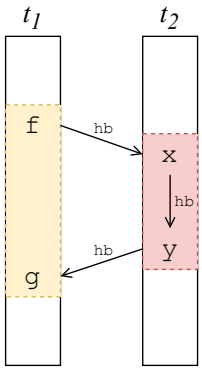
2.4.3 Contracts for Concurrency with Spoilers

Interleaving a sequence of function calls from a contract clause (considering the basic contracts for concurrency or contracts with parameters) with some function calls of a given module can cause atomicity violations. At the same time, this is not the case for some other function calls from the module. However, this is not possible to describe in the contracts introduced so far. For instance, clause (ϱ_1) `contains index_of` requires that this sequence of calls must always be performed atomically, i.e., it does not matter which functions are executed by other threads. Thus, interleaving the execution of this sequence with, e.g., `remove` or `get` is a contract violation. However, in effect, only the execution of `remove` may be problematic, while the execution of `get` may not.

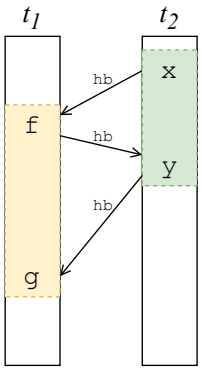
The paper [11] proposes a solution to the above issue — an extension of the basic contracts for concurrency with *contextual information* — allowing one to describe in which context the contract clauses should be enforced. Each clause of the basic contract is now called *target*. For each target, there is a set of so-called *spoilers* that restrict its application. That is, a spoiler is a sequence of function calls that can violate its target. In the end, it has to be ensured that each target is executed atomically w.r.t. its spoilers. Assuming the earlier example, let `contains index_of` be the target clause. Then, a possible spoiler is `remove`. The syntax for this description is as follows: `contains index_of` $\rightsquigarrow$ `remove`.

Formally, let $\Sigma_{\mathbb{M}}$ be a set of all function names of a software module $\mathbb{M}$. Further, let $\mathbb{R}$ be a set of *target clauses*, where each clause $\varrho \in \mathbb{R}$ is a star-free regular expression over $\Sigma_{\mathbb{M}}$. Let $\mathbb{S}$ be a set of *spoiler clauses*, where each clause $\sigma \in \mathbb{S}$ is a star-free regular expression over $\Sigma_{\mathbb{M}}$. Moreover, let $\Sigma_{\mathbb{R}} \subseteq \Sigma_{\mathbb{M}}$ and $\Sigma_{\mathbb{S}} \subseteq \Sigma_{\mathbb{M}}$ be the alphabets of function names used in the target or spoiler clauses, respectively. Then, a contract is a relation $\mathbb{C} \subseteq \mathbb{R} \times \mathbb{S}$, which states for each target the spoilers that can cause atomicity violations. One spoiler may violate more than one target, and, on the contrary, one target may be violated by more

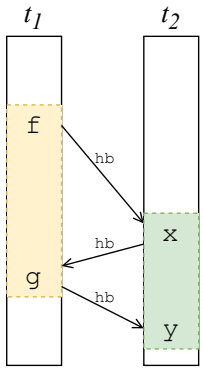
than one spoiler. A contract is violated iff any executed sequence expressed by a target $\varrho \in \mathbb{R}$ is *completely interleaved* with the execution of the sequence representing its spoiler, i.e., $\sigma \in \mathbb{C}(\varrho)$. A target sequence r is completely interleaved by a spoiler sequence s iff the execution of r *starts before* the execution of s, and the execution of s *ends before* the execution of r. *Partial interleavings* of targets and spoilers are here not taken into account to cause an error. However, if needed, this can be resolved by adding a new target clause with an appropriate spoiler. Whether a sequence is executed before another one is, is defined using the so-called *"happened before" relation* ($\xrightarrow{\texttt{hb}}$) [20]. Complete interleaving is illustrated in Figure 2.7a, and partial interleavings are shown in Figures 2.7b and 2.7c.



(a) *Complete interleaving*: the contract is violated



(b) *Partial interleaving*: the contract is not violated (s *started before* r)



(c) *Partial interleaving*: the contract is not violated (s *ended after* r)

Figure 2.7: A *contract violation* demonstration with *target and spoiler interleavings*. In each sub-figure, a thread t_1 executes a target sequence $r =$ `f g` and a thread t_2 executes a spoiler sequence $s =$ `x y`. Consequently, there is the following contract: `f g` $\rightsquigarrow$ `x y` [26]

Example 2.4.4. With the extension of spoilers, the contract from Example 2.4.1 can be refined with contextual information to refuse *unsafe interleavings* as follows (all other interleavings, not described by spoilers, are deemed safe):

(ϱ_1'') `contains index_of` $\rightsquigarrow$ `remove | set`

(ϱ_2'') `index_of ( get | set | remove )` $\rightsquigarrow$ `remove | set`

(ϱ_3'') `size ( get | set | remove )` $\rightsquigarrow$ `remove`

(ϱ_4'') `add index_of` $\rightsquigarrow$ `remove | set`

Lastly, it is possible to combine the extension of contracts for concurrency with spoilers and with parameters. The following clause can demonstrate it:

```
Y = index_of(X, _) get(X, Y) ⇝ remove(X, _)
```

The clause requires sequences of `index_of` and `get` to be performed atomically, but only when working with the same element `X` and index `Y`, and only w.r.t. the concurrent execution of `remove`. Note that the argument `index` of `remove` is not restricted because any concurrent removal may produce an atomicity violation — by either removing an element on index `Y` or by shifting its position.

Chapter 3

Atomer — Atomicity Violations Detector

This chapter describes the principles and limitations of the basic version of the *Atomer static analyser* proposed as a module of *Facebook Infer* (introduced in Section 2.3) for finding some forms of *atomicity violations*. Atomer was proposed and in detail described in the bachelor's thesis [15] of the author of this thesis. Therefore, naturally, the description in Section 3.2 is based on the mentioned thesis. Already existing solutions in this area (besides Atomer) are discussed in Section 3.1. In particular, it deals with other existing approaches and tools for finding atomicity violations, their advantages, disadvantages, features, availability, and so on. In Section 3.3, there are discussed limitations and shortcomings of Atomer. Some of the thoughts mentioned in this section are taken into consideration already in [15].

3.1 Related Work

Atomer is slightly inspired by ideas from [11, 33]. In these papers, there is described a proposal and implementation of a *static* approach for finding *atomicity violations of sequences of function calls*, which is based on *grammars* and *parsing trees*. Note that in the paper [11], there is also described and implemented a *dynamic* approach for the validation. The authors of these papers implemented a stand-alone prototype static analyser called *Gluon*[1] for analysing programs written in Java. To the best author's knowledge, Gluon is the only static analyser that tries to go in a similar direction as Atomer does. Gluon led to some promising experimental results, but the *scalability* of Gluon was still limited.[2] Moreover, Gluon is no more actively developed, and it is not easy to use. Despite all author's efforts, it was not put into operation. Above that, the authors themselves note that the code of Gluon is very ad hoc, and many things are hard-coded in it. These facts, in fact, inspired the decision that eventually led to the implementation of the first version of Atomer, namely, to get inspired by the ideas of [11, 33] but reimplement them in *Facebook Infer*, redesigning it in accordance with the principles of Facebook Infer, which should make

[1] **Gluon** is a tool that implements a *static* approach for finding *atomicity violations of sequences of function calls* in Java programs. It is available at `https://github.com/trxsys/gluon`.

[2] Some of the experiments performed by Gluon are also similarly performed by Atomer, which is discussed in Sections 6.3, 6.4.

the resulting tool more scalable. In the end, however, due to adapting the analysis to the context of Facebook Infer, the implementation of Atomer's analysis is significantly different from [11, 33], as is presented in Chapter 4 of [15]. Furthermore, unlike Gluon, a new version of Atomer is capable of analysing a much wider range of programs because it also supports other languages than Java, and it supports more advanced *locking mechanisms*. On the other hand, Gluon implements *extended contracts for concurrency* (see Sections 2.4.2, 2.4.3) that consider *data flow* within functions and *contextual information*, while Atomer implements only the idea of *basic contracts for concurrency* (see Section 2.4.1). These extended contracts should improve the precision of the analysis. Nonetheless, it is the author's future work to implement the extended contracts in Atomer as well.

In Facebook Infer, there is already implemented analysis called *Lock Consistency Violation*[3]. It is a part of *RacerD* [3, 4, 14]. This analysis finds atomicity violations in C++ and Objective-C programs for *reads/writes* on single variables required to be executed atomically. Atomer is different; it finds atomicity violations for *sequences of functions* required to be executed atomically. Moreover, Atomer tries to automatically determine which sequences should indeed be executed atomically.

3.2 Design of Atomer and Its Principles

Atomer concentrates on checking the *atomicity of the execution of certain sequences of function calls*, which is often required for *concurrent programs'* correct behaviour. In principle, Atomer is based on the assumption that sequences of function calls executed *atomically once* should probably be executed *always atomically*.

The proposal of Atomer is based on the concept of *contracts for concurrency* described in Section 2.4. In particular, the proposal considers the *basic contracts* described in Section 2.4.1. Neither the contracts extended by *parameters* explained in Section 2.4.2 nor the contracts extended by *spoilers* and *targets* described in Section 2.4.3 are so far taken into account.

In general, basic contracts for concurrency allow one to define sequences of functions required to be executed atomically, as explained in more detail in Section 2.4. Atomer is able to automatically derive candidates for such contracts and then verify whether the contracts are fulfilled. In other words, Atomer can both automatically derive those sequences that are sometimes executed atomically as well as subsequently check whether they are indeed always executed atomically. Both of these steps are done statically. The proposed analysis is thus divided into two parts (*phases of the analysis* that are in-depth described in the sections below and illustrated in Figure 3.1):

Phase 1: Detection of (likely) *atomic sequences*.

Phase 2: Detection of *atomicity violations* (violations of the atomic sequences).

This section provides a high-level view of the *abstract interpretation* underlying Atomer. The concrete types of the *abstract states* (i.e., elements of the *abstract domain* Q) and

[3] **Lock Consistency Violation** is an *atomicity violations* analysis implemented in *Facebook Infer*. It is described at `https://fbinfer.com/docs/all-issue-types#lock_consistency_violation`.

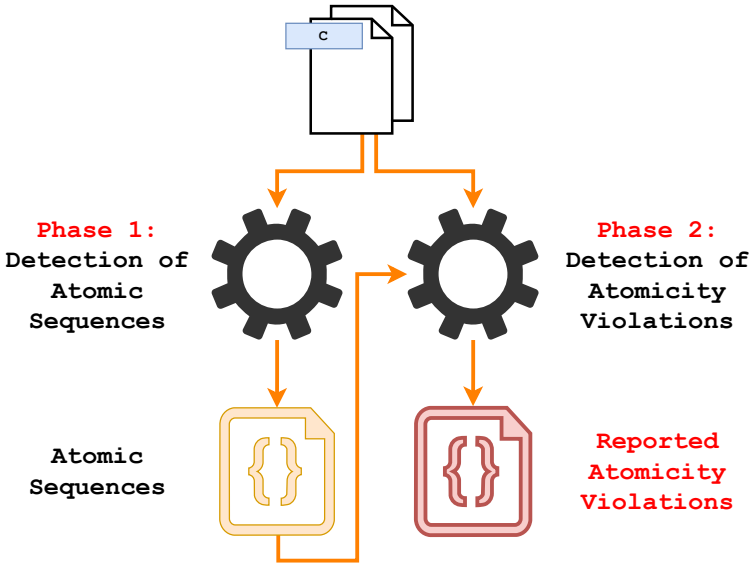


Figure 3.1: *Phases* of the analysis of the Atomer's first version and the analysis *high-level process* illustration [15]

the *summaries* χ, along with the implementation of all necessary *abstract interpretation operators* are stated in Chapter 4 of [15]. However, actually, the abstract states $s \in Q$ of both phases of the analysis are proposed as *sets*. So, in fact, the *ordering operator* $\sqsubseteq$ is implemented using testing for a *subset* (i.e., $s \sqsubseteq s' \iff s \subseteq s'$, where $s, s' \in Q$), the *join operator* $\sqcup$ is implemented as the *set union* (i.e., $s \sqcup s' \iff s \cup s'$), and the *widening operator* $\triangledown$ is implemented using the join operator (i.e., $s \triangledown s' \iff s \sqcup s'$) since the domains are *finite*. Furthermore, it is essential to note that the proposed analysis is neither *sound* nor *complete*. Its goal is to effectively find bugs rather than formally verify the absence of atomicity violations.

Function summaries are in the below sections reduced to the output parts only (*post-conditions* R). The input parts of summaries (*pre-conditions* P) are in case of the proposed analysis always empty because, so far, it is not necessary to have any pre-conditions for analysed functions. Thus, in this case, *Hoare triples* — $true\ S\ \{R\}$ — are used, where S is an analysed function, i.e., $P = true$.

Listings in the below sections are pieces of programs written in the C language (assuming `lock`/`unlock` functions for *mutual exclusion* to *critical sections*).

3.2.1 Phase 1 — Detection of Atomic Sequences

Before detecting *atomicity violations* may begin, it is required to have *contracts for concurrency* introduced earlier. Phase 1 of Atomer is able to produce such contracts, i.e., it detects *sequences of functions* that should be *executed atomically*. Intuitively, the detection is based on looking for sequences of functions executed atomically, in particular, under some *lock*, on some path through a program. The assumption is that if it is *once needed to execute a sequence atomically*, it should probably be *always executed atomically*.

For a description of the analysis, it is first needed to introduce a notion of a *reduced sequence* of function calls. Such a sequence denotes a sequence in which the *first appearance* of each function is recorded only. It is needed to ensure the *finiteness* of the sequences derived by the analysis, and hence the analysis's *termination*. The detection of sequences of calls to be executed atomically is based on analysing all paths through the CFG of a function and generating all pairs $(A, B) \in \Sigma^* \times \Sigma^*$ (where Σ is the set of functions of a given program) of reduced sequences of function calls for each path such that: A is a reduced sequence of function calls that appear between the beginning of the function being analysed and the first lock; between an unlock and a subsequent lock; or between an unlock and the end of the function being analysed. B is a reduced sequence of function calls that follow the calls from A, and that appear between a lock and unlock (or between a lock and the end of the function being analysed). Thus, the abstract states of the analysis are elements of the set $2^{2^{\Sigma^* \times \Sigma^*}}$ because there is a set of the (A, B) pairs for each program path.

It would be more precise to generate longer sequences of type $A_1 \cdot B_1 \cdot A_2 \cdot B_2 \cdot \ldots$ instead of the sets of the pairs (A, B). Nevertheless, it would be more challenging to ensure the above longer sequences' finiteness and the sets of these sequences' finiteness. Moreover, there would be a significantly larger *state space explosion problem* [35]. So, the proposed representation of the sets of pairs of sequences has been chosen to compromise accuracy and efficiency. However, the experiments described in Chapter 5 of [15] show that it needs even more pronounced abstraction for appropriate *scalability*.

Formally, the *initial abstract state* of a function is defined as $s_{init} = \{\{(\varepsilon, \varepsilon)\}\}$, where ε indicates an empty sequence. To formalise the analysis of a function, let $\mathtt{f}$ be a called leaf function. Further, let $s_{\mathtt{g}}$ be the abstract state of a function $\mathtt{g}$ being analysed before the function $\mathtt{f}$ is called. After the call of $\mathtt{f}$, the abstract state will be changed as follows:

$$\begin{aligned} s_{\mathtt{g}} = \{p' \in 2^{\Sigma^* \times \Sigma^*} \mid \exists\, p \in s_{\mathtt{g}} : p' = \{(A', B') \in \Sigma^* \times \Sigma^* \mid \exists\, (A, B) \in p : \\ [\neg actual(p, (A, B)) \wedge (A', B') = (A, B)] \vee [actual(p, (A, B)) \\ \wedge [(lock \wedge (A', B') = (A, B \cdot \mathtt{f})) \vee (\neg lock \wedge (A', B') = (A \cdot \mathtt{f}, B))]]\}\} \end{aligned}$$

where *actual* is a Boolean function that determines whether a given (A, B) pair is the most recent pair of sequences of the current program state for a given program path. Furthermore, *lock* is a predicate indicating whether the current program state is inside an *atomic block*. Further, let $s_{\mathtt{g}}$ be the abstract state of a function $\mathtt{g}$ being analysed before an unlock is called. After the unlock is called, a new (A, B) pair is created and labelled as an actual using the function *setActual* as follows:

$$\begin{aligned} s_{\mathtt{g}} = \{p' \in 2^{\Sigma^* \times \Sigma^*} \mid \exists\, p \in s_{\mathtt{g}} : p' = \{(A, B) \in \Sigma^* \times \Sigma^* \mid \\ [(A, B) = (\varepsilon, \varepsilon) \wedge setActual(p, (A, B))] \vee (A, B) \in p\}\} \end{aligned}$$

Example 3.2.1. For an explanation of the computation of the sets of the pairs (A, B), assume that a state of the analysis of a program S is the following sequence of function calls: $\mathtt{f} \cdot \mathtt{g}$; and a state of the analysis of a program S' is the following sequence of function calls: $\mathtt{f} \cdot \mathtt{g}\ [\mathtt{m} \cdot \mathtt{n}$. The square brackets are used to indicate an *atomic sequence* (the closing square bracket is missing in the case of the program S', which means that the program state is currently inside an atomic block). The computed abstract state for the program S is $s_S = \{\{(\mathtt{f} \cdot \mathtt{g},\ \varepsilon)\}\}$, and for the program S', it is $s_{S'} = \{\{(\mathtt{f} \cdot \mathtt{g},\ \mathtt{m} \cdot \mathtt{n})\}\}$. Now, if the next instruction is a call of a function $\mathtt{x}$, in the case of the program S, the call will be added to the first A sequence, and in the case of the program S', the call will be added to the

first B sequence as follows: $s_S = \{\{(\texttt{f} \cdot \texttt{g} \cdot \texttt{x},\ \varepsilon)\}\}$, $s_{S'} = \{\{(\texttt{f} \cdot \texttt{g},\ \texttt{m} \cdot \texttt{n} \cdot \texttt{x})\}\}$. Subsequently, if the next step in the program S is a lock call, the next function calls will be added to the first B sequence of the set s_S. And if the next step in the program S' is an unlock call, it will be created a new element of the first set of the set $s_{S'}$, and the next function calls will be added to the A sequence of this element. Finally, if a function `y` is called, the resulting sets will look like follows: $s_S = \{\{(\texttt{f} \cdot \texttt{g} \cdot \texttt{x},\ \texttt{y})\}\}$, $s_{S'} = \{\{(\texttt{f} \cdot \texttt{g},\ \texttt{m} \cdot \texttt{n} \cdot \texttt{x}), (\texttt{y},\ \varepsilon)\}\}$. Note that the final sequences of function calls look like follows: $\texttt{f} \cdot \texttt{g} \cdot \texttt{x}\ [\texttt{y}$ and $\texttt{f} \cdot \texttt{g}\ [\texttt{m} \cdot \texttt{n} \cdot \texttt{x}]\ \texttt{y}$ for the programs S and S', respectively.

A *summary* $\chi_{\texttt{f}} \in 2^{\Sigma^*} \times \Sigma^*$ of a function `f` is then a pair $\chi_{\texttt{f}} = (\mathcal{B}, C)$, where:

- $\mathcal{B}$ is a set constructed such that it contains all the B sequences that appear on program paths through `f`, i.e., those computed within the (A, B) pairs at the exit of `f`. Formally, $\mathcal{B} = \{B' \in \Sigma^* \mid \exists p \in s_{\texttt{f}} : \exists (A, B) \in p : B \neq \varepsilon \wedge B' = B\}$, where $s_{\texttt{f}}$ is the abstract state at the end of the abstract interpretation of `f`. In other words, this component of the summary is a set of sequences of atomic function calls appearing in `f`.
- C is a *concatenation* of all the A and B sequences with removed duplicates of function calls. In particular, assume that the following set of (A, B) pairs is computed at the exit of `f`: $\{(A_1, B_1), (A_2, B_2), \ldots, (A_n, B_n)\}$, then the result is the sequence $A_1 \cdot B_1 \cdot A_2 \cdot B_2 \cdot \ldots \cdot A_n \cdot B_n$ with removed duplicates. Formally,

$$C = reduce(\dot{\bigcup_{c \in C'}} c)$$

  where $C' = \{c \in \Sigma^* \mid \exists p \in s_{\texttt{f}} : \exists (A, B) \in p : c = A \cdot B\}$, $\dot{\bigcup}$ concatenates all sequences of a set, and *reduce* is a function that removes duplicates of function calls. Intuitively, in this component of the summary, the analysis gathers occurrences of all called functions within the analysed function obtained by a concatenation of all the A and B sequences. C is recorded to facilitate the derivation of atomic call sequences that show up higher in the *call hierarchy*. Indeed, while locks/unlocks can appear in such a *higher-level* function, parts of the call sequences can appear lower in the call hierarchy.

Example 3.2.2. For instance, the analysis of the function `f` from Listing 3.1 produces the following sequences:

$$\overbrace{\texttt{x} \cdot \cancel{\texttt{x}} \cdot \texttt{y}}^{A_1}\ \overbrace{[\texttt{a} \cdot \texttt{b} \cdot \cancel{\texttt{b}}]}^{B_1}\ \overbrace{\texttt{y} \cdot \cancel{\texttt{y}}}^{A_2}\ \overbrace{[\texttt{a} \cdot \texttt{c}]}^{B_2}\ \overbrace{\cancel{\texttt{y}}}^{A_3}\ \overbrace{\cancel{[\texttt{a} \cdot \texttt{a} \cdot \texttt{c}]}}^{B_3}$$

The functions `a`, `b`, `c`, `x`, `y` are not deeper analysed because it is assumed that these functions are leaf nodes of the call graph. The strikethrough of the functions `b`, `x`, `y` denotes removing already recorded function calls in the A and B sequences to get the reduced form. The strikethrough of the entire sequence $\texttt{y}\ [\texttt{a} \cdot \texttt{a} \cdot \texttt{c}]$ means discarding sequence already seen before. For the above, the abstract state at the end of the abstract interpretation of the function `f` is $s_{\texttt{f}} = \{\{(\texttt{x} \cdot \texttt{y},\ \texttt{a} \cdot \texttt{b}), (\texttt{y},\ \texttt{a} \cdot \texttt{c}), (\varepsilon,\ \varepsilon)\}\}$. The derived summary $\chi_{\texttt{f}}$ for the function `f` is $\chi_{\texttt{f}} = (\mathcal{B}, C)$, where $\mathcal{B} = \{\texttt{a} \cdot \texttt{b},\ \texttt{a} \cdot \texttt{c}\}$, i.e., B_1 and B_2; and $C = \texttt{x} \cdot \texttt{y} \cdot \texttt{a} \cdot \texttt{b} \cdot \texttt{c}$, i.e., the concatenation of A_1, B_1, A_2, B_2 from which duplicate function calls were removed.

```
1  void f()
2  {
3      x(); x(); y();
4      lock(&L); // a . b
5      a(); b(); b();
6      unlock(&L);
7      y(); y();
8      lock(&L); // a . c
9      a(); c();
10     unlock(&L);
11     y();
12     lock(&L); // a . c
13     a(); a(); c();
14     unlock(&L);
15 }
```

Listing 3.1: A code snippet used for an illustration of the derivation of *sequences of functions called atomically*

Further, it is demonstrated how the results of the analysis of *nested functions* are used during the detection of atomic sequences. The result of the analysis of a nested function is used as follows. When calling an already analysed function, one plugs the sequence from the second component of its summary (i.e., the C sequence) into the most recent A or B sequence of all the program paths (where a program path corresponds to a single element of an abstract state, i.e., a set of the (A, B) pairs). In particular, assume that (A, B) is the most recent pair of sequences of the program state of a path being analysed. Subsequently, it is called a function $\mathtt{f}$ with a non-empty summary (i.e., $C \neq \varepsilon$). If the current program state of the analysed function is inside an atomic block, the analysis in this step will transform the pair (A, B) to a new (A', B') pair as follows: $(A', B') = (A, B \cdot \mathtt{f} \cdot C)$. Otherwise, $(A', B') = (A \cdot \mathtt{f} \cdot C, B)$. In such cases where a summary is empty, i.e., there are no function calls in a called function, or it is a leaf node of the call graph, just the function name is appended to the most recent A or B sequences of all the program paths. To formalise this process, let $\mathtt{f}$ be a called function that was already analysed, and the second component of its summary is C. Further, let $s_{\mathtt{g}}$ be the abstract state of a function $\mathtt{g}$ being analysed before the function $\mathtt{f}$ is called. After the call of $\mathtt{f}$, the abstract state will be changed as follows:

$$s_{\mathtt{g}} = \{p' \in 2^{\Sigma^* \times \Sigma^*} \mid \exists\, p \in s_{\mathtt{g}} : p' = \{(A', B') \in \Sigma^* \times \Sigma^* \mid \exists\, (A, B) \in p : [\neg actual(p, (A, B)) \wedge (A', B') = (A, B)] \vee [actual(p, (A, B)) \wedge [(lock \wedge (A', B') = (A, B \cdot \mathtt{f} \cdot C)) \vee (\neg lock \wedge (A', B') = (A \cdot \mathtt{f} \cdot C, B))]]\}\}$$

Example 3.2.3. This example shows how the function $\mathtt{g}$ from Listing 3.2 would be analysed using the result of the analysis of the function $\mathtt{f}$ from Listing 3.1. The second component of $\chi_{\mathtt{f}}$ is $C = \mathtt{x} \cdot \mathtt{y} \cdot \mathtt{a} \cdot \mathtt{b} \cdot \mathtt{c}$. The analysis of the function $\mathtt{g}$ produces the following sequence:

$$\mathtt{x} \cdot \mathtt{f} \cdot \mathtt{x} \cdot \mathtt{y} \cdot \mathtt{a} \cdot \mathtt{b} \cdot \mathtt{c} \cdot \mathtt{z}\ [\mathtt{f} \cdot \mathtt{x} \cdot \mathtt{y} \cdot \mathtt{a} \cdot \mathtt{b} \cdot \mathtt{c}]$$

For the above, the abstract state at the end of the abstract interpretation of the function $\mathtt{g}$ is $s_{\mathtt{g}} = \{\{(\mathtt{x} \cdot \mathtt{f} \cdot \mathtt{y} \cdot \mathtt{a} \cdot \mathtt{b} \cdot \mathtt{c} \cdot \mathtt{z},\ \mathtt{f} \cdot \mathtt{x} \cdot \mathtt{y} \cdot \mathtt{a} \cdot \mathtt{b} \cdot \mathtt{c}\cdot), (\varepsilon,\ \varepsilon)\}\}$. The derived summary $\chi_{\mathtt{g}}$ for the function $\mathtt{g}$ is $\chi_{\mathtt{g}} = (\{\mathtt{f} \cdot \mathtt{x} \cdot \mathtt{y} \cdot \mathtt{a} \cdot \mathtt{b} \cdot \mathtt{c}\cdot\},\ \mathtt{x} \cdot \mathtt{f} \cdot \mathtt{y} \cdot \mathtt{a} \cdot \mathtt{b} \cdot \mathtt{c} \cdot \mathtt{z})$.

```
1  void g()
2  {
3      x(); f(); z();
4      lock(&L); // f . x . y . a . b . c
5      f();
6      unlock(&L);
7  }
```

Listing 3.2: A code snippet used to illustrate the derivation of sequences of functions called atomically with a *nested function call* (function `f` is defined in Listing 3.1)

Cases Where Lock/Unlock Calls Are Not Paired in a Function

For treating cases where *lock/unlock calls are not paired* in a function — as demonstrated in Listing 3.3 — the following solution is implemented in the basic version of Atomer.

Everything is unlocked at the end of a function, i.e., one *virtually appends* an unlock to the end of the function if it is necessary. Then, for the function `m` from Listing 3.3, the atomic section is virtually closed. Hence, there is detected an atomic sequence `a`. In particular, the summary is as follows: $\chi_{\texttt{m}} = (\{\texttt{a}\},\ \texttt{a})$.

Moreover, *all unlock calls not preceded by a lock are ignored.* Thus, in the function `n` from Listing 3.3, there are not detected any atomic sequences: $\chi_{\texttt{n}} = (\emptyset,\ \texttt{a})$.

```
1  void m()
2  {
3      lock(&L); // a
4      a();
5  }
6  void n()
7  {
8      a();
9      unlock(&L);
10 }
```

Listing 3.3: A code snippet used to illustrate treating cases where *lock/unlock calls are not paired* in a function

Summary of Phase 1 The derived sequences of calls assumed to execute atomically — the $\mathcal{B}$ sequences — from the summaries of all analysed functions are stored into a file used during Phase 2, which is described later on.

3.2.2 Phase 2 — Detection of Atomicity Violations

In the second phase of the analysis, i.e., when *detecting violations* of the atomic sequences obtained from Phase 1, the analysis looks for *pairs of functions* that *should be called atomically* (or just for single functions if there is only one function call in an atomic sequence) and that are not executed atomically (i.e., under a lock) on some path through the CFG. The pairs of function calls to be checked for atomicity are obtained as follows. For each function `f`

with a Phase 1 summary $\chi_{\mathtt{f}} = (\mathcal{B}, C)$ in a given program S, where $\mathcal{B} = \{B_1, B_2, \ \ldots \ , B_n\}$, the analysis considers *every pair* $(\mathtt{x}, \mathtt{y}) \in \Sigma \times \Sigma$ of functions that appear as a *substring* in some of the B_i sequences, i.e., $B_i = w \cdot \mathtt{x} \cdot \mathtt{y} \cdot w'$ for some sequences w, w'. Note that $\mathtt{x}$ could be ε (an empty sequence) if some B_i consists of a single function. All these "atomic pairs" are put into the set $\Omega \in 2^{\Sigma \times \Sigma}$. More formally,

$$\Omega = \{(\mathtt{x}, \mathtt{y}) \in \Sigma \times \Sigma \mid \exists\, (\mathcal{B}, C) \in X_S : \exists\, B \in \mathcal{B} : [|B| = 1 \wedge (\mathtt{x}, \mathtt{y}) = (\varepsilon, B)] \\ \vee [|B| > 1 \wedge \exists\, w, w' \in \Sigma^* : B = w \cdot \mathtt{x} \cdot \mathtt{y} \cdot w' \wedge (\mathtt{x}, \mathtt{y}) \neq (\varepsilon, \varepsilon)]\}$$

where $X_S \in 2^{2^{\Sigma^*} \times \Sigma^*}$ is a set of all Phase 1 summaries of the program S.

Example 3.2.4. For instance, assume that in Phase 1, there was analysed a function `f`. It produced the summary $\chi_{\mathtt{f}} = (\mathcal{B}, C)$, where $\mathcal{B} = \{\mathtt{a} \cdot \mathtt{b} \cdot \mathtt{c},\ \mathtt{a} \cdot \mathtt{c} \cdot \mathtt{d}\}$, i.e., a set of sequences of functions that should be called atomically. The analysis will then look for the following pairs of functions that are not called atomically: $\Omega = \{\mathtt{a} \cdot \mathtt{b},\ \mathtt{b} \cdot \mathtt{c},\ \mathtt{a} \cdot \mathtt{c},\ \mathtt{c} \cdot \mathtt{d}\}$.

An element of this phase's abstract state is a triple $(\mathtt{x}, \mathtt{y}, \delta) \in \Sigma \times \Sigma \times 2^{\Sigma \times \Sigma \times \mathbb{N}}$ where $(\mathtt{x}, \mathtt{y})$ is a pair of the most recent calls of functions performed on the program path being explored, and δ is a set of so far detected *pairs that violate atomicity* on particular lines of code. Thus, the abstract states are elements of the set $2^{\Sigma \times \Sigma \times 2^{\Sigma \times \Sigma \times \mathbb{N}}}$. Whenever a function `f` is called on some path that led to an abstract state $(\mathtt{x}, \mathtt{y}, \delta)$, a new pair $(\mathtt{x}', \mathtt{y}')$ of the most recent function calls is created from the previous pair $(\mathtt{x}, \mathtt{y})$ such that $(\mathtt{x}', \mathtt{y}') = (\mathtt{y}, \mathtt{f})$. Further, when the current program state is not inside an atomic block, the analysis checks whether the new pair (or just the last call) violates atomicity (i.e., $(\mathtt{x}', \mathtt{y}') \in \Omega \vee (\varepsilon, \mathtt{y}') \in \Omega$). When it does, it is added to the set δ of pairs that violate atomicity.

Formally, the initial abstract state (in this phase) of a function is defined as $s_{init} = \{(\varepsilon, \varepsilon, \emptyset)\}$. To formalise the analysis of a function, let `f` be a called leaf function on a line c. Further, let $s_{\mathtt{g}}$ be the abstract state of a function `g` being analysed before the function `f` is called. After the call of `f`, the abstract state will be changed as follows:

$$s_{\mathtt{g}} = \{(\mathtt{x}', \mathtt{y}', \delta') \in \Sigma \times \Sigma \times 2^{\Sigma \times \Sigma \times \mathbb{N}} \mid \exists\, (\mathtt{x}, \mathtt{y}, \delta) \in s_{\mathtt{g}} : (\mathtt{x}', \mathtt{y}') = (\mathtt{y}, \mathtt{f}) \\ \wedge [(\neg \mathit{lock} \wedge \delta' = \{(\mathtt{x}'', \mathtt{y}'', c') \in \Sigma \times \Sigma \times \mathbb{N} \mid (\mathtt{x}'', \mathtt{y}'', c') \in \delta \\ \vee [((\mathtt{x}'', \mathtt{y}'') = (\mathtt{x}', \mathtt{y}') \vee (\mathtt{x}'', \mathtt{y}'') = (\varepsilon, \mathtt{y}')) \wedge (\mathtt{x}'', \mathtt{y}'') \in \Omega \wedge c' = c]\}) \vee (\mathit{lock} \wedge \delta' = \delta)]\}$$

A *summary* of the second phase of the analysis — $\chi_{\mathtt{f}} \in 2^{\Sigma \times \Sigma \times \mathbb{N}}$ — of a function `f` is then $\chi_{\mathtt{f}} = \Delta$, where Δ is a set of pairs that violate atomicity within the function `f`. Δ is constructed such that it contains a union of all the δ sets that appear on program paths through `f`. Formally, $\Delta = \bigcup_{\delta' \in \Delta'} \delta'$, where $\Delta' = \{\delta' \in 2^{\Sigma \times \Sigma \times \mathbb{N}} \mid \exists\, p \in s_{\mathtt{f}} : \exists\, (\mathtt{x}, \mathtt{y}, \delta) \in p : \delta \neq \emptyset \wedge \delta' = \delta\}$ and $s_{\mathtt{f}}$ is the abstract state at the end of the abstract interpretation of `f`.

The analysis of functions with *nested function calls* and cases where *lock/unlock calls are not paired* within functions are handled analogically as in Phase 1. For a detailed explanation, see [15].

Example 3.2.5. To demonstrate the detection of an atomicity violation, assume the functions `f` and `g` from Listing 3.4. The set of atomic sequences of the function `f` with the first phase's summary $\chi_{\mathtt{f}} = (\mathcal{B}, C)$ is $\mathcal{B} = \{\mathtt{a} \cdot \mathtt{b} \cdot \mathtt{c}\}$, thus, $\Omega = \{(\mathtt{a}, \mathtt{b}), (\mathtt{b}, \mathtt{c})\}$. In the function `g`, an atomicity violation is detected because the pair of functions `b`, `c` is not called atomically on line 12, i.e., $(\mathtt{b}, \mathtt{c}) \in \Omega$. Consequently, the derived summary $\chi_{\mathtt{g}}$ for the function `g` for the second phase of the analysis is $\chi_{\mathtt{g}} = \{(\mathtt{b}, \mathtt{c}, 12)\}$.

```
1  void f()
2  {
3      x();
4      lock(&L); // a . b . c
5      a(); b(); c();
6      unlock(&L);
7      y();
8  }
9  void g()
10 {
11     x();
12     b(); c(); // ATOMICITY_VIOLATION: (b, c)
13     y();
14 }
```

Listing 3.4: An example of an *atomicity violation*

Summary of Phase 2 The sets of pairs that violate atomicity — the Δ sets — from the summaries of all analysed functions are finally reported to the user.

3.3 Atomer's Limitations

The basic version of Atomer has been proposed as it is detailed in Section 3.2. The first version of the analyser has also been implemented in [15], and it works as expected. Moreover, it can be used in practice to analyse various kinds of programs, and it may find *real-world atomicity related bugs*. Nevertheless, there are still several *limitations* and cases where the original version of Atomer would not work correctly, i.e., cases not addressed during the original proposal. Some of these cases are briefly discussed already in [15] and further described in [16].

So far, Atomer does not distinguish *different lock instances* used simultaneously in a program. Only calls of locks/unlocks are identified, and the parameters of these calls — *lock objects* — are not considered at all. Therefore, if there are several lock objects used, the analysis does not work correctly. Although this may happen in *real-life programs*, insomuch as one could have, e.g., another (smaller) atomic section inside a current atomic section (this does not have to be evident at first because the *inner atomic section* could be, e.g., inside a nested function). For example,

... `lock(L1)`; ... `lock(L2)`; ... `unlock(L2)`; ... `unlock(L1)`; ...

Another possibility is an *alternating sequence of locks*, e.g., two locks are locked at first, and then, they are unlocked in the same order, i.e.,

... `lock(L1)`; ... `lock(L2)`; ... `unlock(L1)`; ... `unlock(L2)`; ...

Another limitation of Atomer's basic version is that it supports only the analysis of programs written in the *C language* that use *PThread* locks to *synchronise concurrent threads*. Of course, in practice, many other *types of locks* for synchronisation of concurrent threads or even synchronisation of *concurrent processes* are used. Although the first version of

Atomer can analyse C programs with other types of locks, these locks are not recognised as locks. Thus, the analysis would not work as expected. It would definitely be helpful also to analyse other languages than just C. As described in Section 2.3, Facebook Infer is capable of analysing programs written in C, C++, Objective-C, Java, and C#. The analysis algorithm could then be the same for all these languages because Facebook Infer's *intermediate language* is analysed instead of directly analysing the input languages. Again, the first version of Atomer should be able to analyse the above languages, but it has not been tested within [15]. However, most importantly, other languages may use *very different lock types*, which would not be recognised. Examples of some advanced locking mechanisms (not supported by the basic version of Atomer) are *lock guards*, *re-entrant locks*, or *try-locks*.

Regarding *scalability*, the basic version of Atomer can have problems with more *extensive* and *complex* programs, as mentioned in [15] (problems with *memory* as well as problems with the *analysis time*). The problem is working with the sets of (A, B) pairs of *sequences* in the abstract states of Phase 1 and working with *sequences* of calls in the summaries of this phase. It may be necessary to store many of these sequences, and they can be very long (due to all different paths through the CFG of an analysed program). It may lead to the *state space explosion problem* [35].

One of the main reasons that Atomer's first version reports *false alarms* is that in *critical sections*, in real-world programs, there are sometimes called *generic functions* that do not influence atomicity violations (such as functions for printing to the standard output, functions for recasting variables to different types, functions related to iterators, and whatever other "safe" functions for particular program types). Often, to find some atomicity violations, it is sufficient to focus only on certain "critical" functions. In practice, another issue is that in an analysed program, there can be "large" critical sections or critical sections in which appear function calls with a *deep hierarchy of nested function calls*. All the above cases may cause massive and "imprecise" atomic sequences that are the source of false alarms. However, regardless of the above issues, Atomer can still report quite some false alarms. It is due to the assumption that *sequences called atomically once* should *always be called atomically*, but this does not always have to hold. None of the above reasons that can generate false alarms is resolved in the first version of Atomer.

The next source of false alarms is something that the author of this work calls *local atomicity violations*. Imagine a function `f` that contains non-atomic calls of functions `a`, `b`, and these functions should always be invoked atomically. Obviously, this is an atomicity violation. However, suppose that `f` is called exclusively from atomic sections of other functions higher in the call hierarchy. In this case, in effect, that is not a real atomicity violation (it can be considered as a local atomicity violation within a single function, but globally, it is not). As a consequence, a false alarm would be reported by the basic version of Atomer. In real-life programs, this situation may be fairly common due to *complicated call graphs*.

Atomer considers only the *basic contracts for concurrency*, defined in Section 2.4.1. It is pretty limited in some circumstances, and therefore, Atomer can report *false alarms*. The basic contracts do not take into consideration the *data flow* within function calls. However, a better idea is to work with the assumption that a sequence of function calls must be atomic only if it *handles the same data*. Assume that functions `f`, `g` are manipulating with the same container `C` as follows: `f(C); g(C);`. These are called atomically. Somewhere else — where `f`, `g` are not called atomically — it does not necessarily cause an atomicity violation because they can be invoked with different arguments, which could be valid.

This behaviour corresponds to the *extended contracts with parameters* (see Section 2.4.2). Another (more complex) limitation is that basic contracts do not consider any *contextual information*. It would be more precise to consider as atomicity violations such sequences that could be violated only by particular ("dangerous") function calls, not by any calls. For example, suppose that there is the following sequence of functions called atomically: `f(); g();`. While somewhere else, these functions are not called atomically, it does not necessarily cause that it is an atomicity violation because, in this particular context, none of the "dangerous" functions can be executed by any concurrent thread. The *extended contracts with spoilers* formally describe these scenarios in Section 2.4.3.

A remarkable problem (though it is not directly a problem of Atomer) is identifying whether a reported atomicity violation is a *real bug* or whether it is just a false alarm. It could be really challenging, especially in *extensive real-life* programs.

Solutions for some of the above issues and limitations are proposed in Chapter 4 and further implemented in a new version of Atomer, detailed in Chapter 5.

Chapter 4

Proposal of Enhancements for Atomer

In this chapter, the author proposes solutions for some of the Atomer's limitations stated in Section 3.3. The solutions enhance the analysis's *precision* and *scalability*. In order to formally define these enhancements, the notions and symbols introduced in Section 3.2 are used. Some of the enhancements were briefly discussed already in [16].

Section 4.1 proposes an optimisation of Atomer's scalability. The following Sections 4.2, 4.3, 4.4 cover precision improvements, i.e., extensions of Atomer with additional features that improve its ability to cope with cases that were not supported in the first version of Atomer, and that can be seen in *real-life code*. Furthermore, Chapter 5 provides an overview of the implementation of all the below improvements in a new version of Atomer.

In the following sections, to give an intuition, there are used listings with pieces of C programs (assuming `lock/unlock` functions for *mutual exclusion* to *critical sections*). In addition, there are used C++ and Java programs to illustrate the *locking mechanisms* in these languages.

4.1 Approximation of Sequences by Sets

Regarding *scalability*, the basic version of Atomer can have problems with more *extensive* and *complex* programs, which can manifest both in its *time* and *memory* consumption. The problems arise primarily due to working with the sets of (A, B) pairs of **sequences** of function calls in *abstract states* (during Phase 1). It may be necessary to store many of these sequences, and they could be very long (due to all different paths through the CFG of an analysed program). The author's idea is to *approximate* these sets by working with sets of (A, B) pairs of **sets** of function calls. Apart from representing the abstract states of the first phase of the analysis, elements of these pairs do also appear in the first phase's *summaries*, and they are then used during Phase 2 as well. Thus, it is needed to make a certain approximation in the summaries and their subsequent usage too. The approximated phases of the analysis and their collaboration are illustrated in Figure 4.1 (one can compare that with the illustration of the first version of Atomer in Figure 3.1).

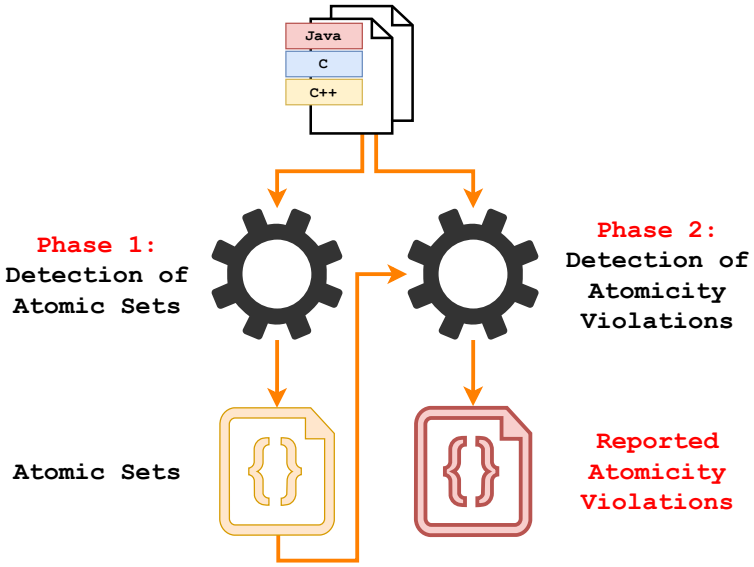


Figure 4.1: An illustration of the *phases* of the Atomer's analysis and the *high-level analysis process* with an *approximation* of working with *sequences* by working with *sets* (moreover, note that a new version of Atomer accepts programs also written in C++ and Java languages, which is described in Section 5.3)

In particular, the proposed solution is more scalable because the ordering of function calls that appear in the pairs is not relevant anymore. Therefore, less memory is required because different sequences of function calls can map the same set. The analysis is also faster since there are stored fewer sets of function calls to work with. On the other hand, the analysis is less accurate because the new approach causes some loss of information. In practice, this loss of information could eventually lead to *false alarms*. However, the number of such false alarms is typically not that high as this thesis's experimental evidence shows. Moreover, later, there are discussed some techniques that may rid of these false alarms.

4.1.1 Approximation with Sets in Phase 1

The *detection of sequences of calls to be executed atomically* now generates all (A, B) pairs of **sets** of function calls for each path instead of pairs of **sequences**, i.e., $(A, B) \in 2^{\Sigma} \times 2^{\Sigma}$. Here, A, B are not *reduced sequences* (the notion of a reduced sequence is not needed anymore) but sets. The purpose of the pairs is preserved. Hence, the abstract states are elements of the set $2^{2^{\Sigma} \times 2^{\Sigma}}$. In all the implemented algorithms and definitions, it is sufficient to work with:

- *sets* 2^{Σ} of functions, instead of *sequences* Σ^* of functions;
- the *empty sets* $\emptyset$, instead of the *empty sequences* ε; and
- *unions* $\cup$ of sets, instead of the *concatenation* $\cdot$ of sequences.

The above implies that the *initial abstract state* of a function is changed to $s_{init} = \{\{(\emptyset, \emptyset)\}\}$. During the analysis of a function `g` with an abstract state $s_{\mathtt{g}}$, when a leaf function `f` is called,

the abstract state's transformation is changed as follows:

$$s_{\mathtt{g}} = \{p' \in 2^{2^{\Sigma} \times 2^{\Sigma}} \mid \exists\, p \in s_{\mathtt{g}} : p' = \{(A', B') \in 2^{\Sigma} \times 2^{\Sigma} \mid \exists\, (A, B) \in p : [\neg \mathit{actual}(p, (A, B)) \wedge (A', B') = (A, B)] \vee [\mathit{actual}(p, (A, B)) \wedge [(\mathit{lock} \wedge (A', B') = (A, B \cup \{\mathtt{f}\})) \vee (\neg \mathit{lock} \wedge (A', B') = (A \cup \{\mathtt{f}\}, B))]]\}\}$$

Further, when an unlock is called, a new (A, B) pair is created as follows:

$$s_{\mathtt{g}} = \{p' \in 2^{2^{\Sigma} \times 2^{\Sigma}} \mid \exists\, p \in s_{\mathtt{g}} : p' = \{(A, B) \in 2^{\Sigma} \times 2^{\Sigma} \mid [(A, B) = (\emptyset, \emptyset) \wedge \mathit{setActual}(p, (A, B))] \vee (A, B) \in p\}\}$$

Other algorithms (e.g., calling an already analysed *nested* function) are modified analogically.

Another approximation was made in the summaries. The first component of the summary has to be changed to a set of sets of function calls because it is constructed from the B items from the abstract states, which are now sets. The second component of the summary can be changed to a set of function calls because even before, it was a reduced sequence of all the (A, B) pairs. Therefore, the ordering of function calls was significantly approximated even so. Moreover, it is used to analyse functions higher in the *call hierarchy* where it is appended to A or B, which are now sets. Thus, it would make no sense to store it in summaries as a sequence. Formally, the form of summaries χ changes from $2^{\Sigma^*} \times \Sigma^*$ to $2^{2^{\Sigma}} \times 2^{\Sigma}$. In particular, a summary $\chi_{\mathtt{f}} \in 2^{2^{\Sigma}} \times 2^{\Sigma}$ of a function `f` is redefined as $\chi_{\mathtt{f}} = (\mathcal{B}, C)$, where:

- $\mathcal{B} = \{B' \in 2^{\Sigma} \mid \exists\, p \in s_{\mathtt{f}} : \exists\, (A, B) \in p : B \neq \emptyset \wedge B' = B\}$, where $s_{\mathtt{f}}$ is the abstract state at the end of the abstract interpretation of `f`.
- $C = \bigcup_{c \in C'} c$, where $C' = \{c \in 2^{\Sigma} \mid \exists\, p \in s_{\mathtt{f}} : \exists\, (A, B) \in p : c = A \cup B\}$.

Example 4.1.1. For demonstrating the approximation of the analysis to sets, assume functions `f` and `g` from Listing 4.1. Further, assume that `a`, `b`, `x`, `y` are leaf nodes of the call graph. Before the approximation, when the analysis was working with sequences of function calls, Phase 1 of the analysis produced the following abstract states and summaries while analysing the functions:

- $s_{\mathtt{f}} = \{\{(\mathtt{x} \cdot \mathtt{y},\ \mathtt{a} \cdot \mathtt{b}), (\mathtt{y} \cdot \mathtt{x},\ \mathtt{b} \cdot \mathtt{a}), (\varepsilon,\ \varepsilon)\}\}$, $\chi_{\mathtt{f}} = (\{\mathtt{a} \cdot \mathtt{b},\ \mathtt{b} \cdot \mathtt{a}\},\ \mathtt{x} \cdot \mathtt{y} \cdot \mathtt{a} \cdot \mathtt{b})$;
- $s_{\mathtt{g}} = \{\{(\mathtt{y} \cdot \mathtt{x},\ \mathtt{b} \cdot \mathtt{a}), (\varepsilon,\ \varepsilon)\}\}$, $\chi_{\mathtt{g}} = (\{\mathtt{b} \cdot \mathtt{a}\},\ \mathtt{y} \cdot \mathtt{x} \cdot \mathtt{b} \cdot \mathtt{a})$.

Whereas, after the approximation, the produced abstract states and summaries are as follows: $s_{\mathtt{f}} = s_{\mathtt{g}} = \{\{(\{\mathtt{x}, \mathtt{y}\}, \{\mathtt{a}, \mathtt{b}\}), (\emptyset, \emptyset)\}\}$, $\chi_{\mathtt{f}} = \chi_{\mathtt{g}} = (\{\{\mathtt{a}, \mathtt{b}\}\}, \{\mathtt{a}, \mathtt{b}, \mathtt{x}, \mathtt{y}\})$. They are the same for both functions because there are the same locked/unlocked function calls; only the order of calls differs.

4.1.2 Approximation with Sets in Phase 2

The *detection of atomicity violations* in Phase 2 then works almost the same way as before the approximation. However, there is one difference. Before, the analysis implemented in

```
1  void f()
2  {
3      x(); y();
4      lock(&L); // a . b -> {a, b}
5      a(); b();
6      unlock(&L);
7      y(); x();
8      lock(&L); // b . a -> {a, b}
9      b(); a();
10     unlock(&L);
11 }
12 void g()
13 {
14     y(); x();
15     lock(&L); // b . a -> {a, b}
16     b(); a();
17     unlock(&L);
18 }
```

Listing 4.1: A code snippet used to illustrate the proposed *approximation* of the first phase of the analysis in a new version of Atomer using *sets of function calls*

the second phase looked for violations of atomic sequences obtained from Phase 1. Now, **atomic sets** are obtained from Phase 1; hence, the detection of atomicity violations needs to work with sets too. Again, the analysis looks for *pairs of functions that should be called atomically*, while this is not the case on some path through the CFG. This algorithm is identical to the algorithm before the approximation.

Nevertheless, it is needed to propose a new algorithm that derives the pairs of function calls (from the atomic sets) to be checked for atomicity (i.e., the set $\Omega \in 2^{\Sigma \times \Sigma}$). Intuitively, the second phase of the analysis now looks for non-atomic execution of any pair of functions `f`, `g` such that $\{\mathtt{f}, \mathtt{g}\}$ is a *subset* of some set of functions that were found to be executed atomically. In order to obtain the pairs, all possible pairs of functions are taken from atomic sets from Phase 1, i.e., all *2-element variations*. Formally, let S be an analysed program, and let $X_S \in 2^{2^{2^\Sigma} \times 2^\Sigma}$ be a set of all summaries of the program S. Then, all the atomic pairs (the first item of a pair may be empty if an atomic set consists of a single function) are obtained as follows:

$$\Omega = \{(\mathtt{x}, \mathtt{y}) \in \Sigma \times \Sigma \mid \exists\, (\mathcal{B}, C) \in X_S : \exists\, B \in \mathcal{B} : [|B| = 1 \wedge (\mathtt{x}, \mathtt{y}) \in \{\varepsilon\} \times B] \\ \vee\, [|B| > 1 \wedge (\mathtt{x}, \mathtt{y}) \in B \times B \wedge \mathtt{x} \neq \mathtt{y}]\}$$

Example 4.1.2. For example, assume that Phase 1 analysed a function `f`, which produced the summary $\chi_{\mathtt{f}} = (\mathcal{B}, C)$. Assume that before the approximation, a set of sequences of functions that should be called atomically was as follows: $\mathcal{B} = \{\mathtt{a} \cdot \mathtt{b} \cdot \mathtt{c}\}$. Then, the analysis looked for the following pairs of functions that are not called atomically: $\Omega = \{(\mathtt{a}, \mathtt{b}), (\mathtt{b}, \mathtt{c})\}$. Since the result of the first component of the summary was changed to the set $\mathcal{B}$ of sets of functions that should be called atomically as follows: $\mathcal{B} = \{\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}\}$, the analysis now looks for the following pairs of functions that are not called atomically (all 2-element variations): $\Omega = \{(\mathtt{a}, \mathtt{b}), (\mathtt{a}, \mathtt{c}), (\mathtt{b}, \mathtt{a}), (\mathtt{b}, \mathtt{c}), (\mathtt{c}, \mathtt{a}), (\mathtt{c}, \mathtt{b})\}$.

4.2 Advanced Manipulation with Locks

The original version of Atomer does not distinguish *different lock instances* in a program. Only calls of locks/unlocks are identified, and the parameters of these calls — *lock objects* — are not considered. Thus, if there are several lock objects used, the analysis does not work correctly.

In order to consider lock objects, it was proposed to distinguish between them using Facebook Infer's built-in mechanism called *access paths* [22], explained below. The analyser does not perform a classical *alias analysis*, i.e., it does not perform a precise analysis for saying when arbitrary pairs of accesses to lock objects may alias (such an analysis is considered too expensive).

Access Paths The *syntactic access paths* [22] represent *heap locations* via the paths used to access them, i.e., they have the form of an expression consisting of a base variable followed by a sequence of fields. More formally, let Var be a set of all variables that can occur in a given program. Let $Field$ be a set of all possible field names that can be used in the program (e.g., structure fields). An access path π from the set Π of all access paths is then defined as follows:

$$\pi \in \Pi ::= Var \times Field^*$$

Access paths are already implemented in Facebook Infer. For instance, the principle of using access paths is used in an existing analyser in Facebook Infer — RacerD [3, 4, 14] — for data race detection. In general, no sufficiently precise *alias analysis* works *compositionally* and *at scale*. That is the motivation for using access paths in Facebook Infer.

Given a pair of accesses to lock objects, to determine whether these locks are equal, it is needed to answer the following question: "Can the accesses touch the same address?". Remarkably, according to the authors of [3], access paths alone *almost* convey enough semantic information to answer the above question on their own. If two access paths are syntactically equal, it is almost (but not quite) true that they must refer to the same address. Syntactically identical paths can refer to different addresses if (i) they refer to different instances of the same object, or (ii) a prefix of the path is reassigned along one execution trace but not the other. These conditions cannot hold if an access path is *stable*, i.e., if none of its proper prefixes appears in assignments during a given execution trace, then it touches the same memory as all other stable accesses to the syntactic path. Therefore, the access paths' syntactic equality is a reasonably efficient way to say (in an *under-approximate fashion*) that heap access touches the same address. Also, by using access paths, RacerD detected many errors in real-world programs, proving that the use of access paths can reveal real errors. This is why it was decided to use this principle to represent locks in Atomer.

During the analysis performed by Atomer (in both phases), each atomic section is identified by an access path of the lock that guards the section; see Sections 4.2.1, 4.2.2. Because *syntactically identical access paths* are used as the means for distinguishing atomic sections, some atomicity violations could be missed (or some false alarms could be reported) due to distinct access paths that refer to the same memory. However, the analysis's precision is still significantly improved this way while preserving its *scalability*, and the stress is anyway put on finding likely violations, not on being *sound*.

Another limitation of Atomer in its basic version is that it does not count with *re-entrant locks* when a process can lock the same object multiple times without blocking itself, and then it should unlock the lock object the same number of times. This approach is, in fact, widespread, e.g., in Java, where so-called *synchronised blocks* are used, as demonstrates Listing 4.2. These blocks are re-entrant by default. To consider re-entrant locks in the analysis, the number of locks of individual lock objects is tracked in the abstract states of both phases of the analysis. A lock is unlocked as soon as this number decreases to 0. Also, an input parameter $t \in \mathbb{N}$ was proposed to limit the *upper bound* to which the analysis tracks precisely the number of times a given lock is locked. When this bound is reached, the *widening operator* is used to abstract the number to any value bigger than the bound. This is to ensure *termination* of the analysis. The idea of this upper bound limit comes from the approach used in RacerD.

```
1 public synchronized void f() {
2     x();
3     synchronized (this) {
4         a();
5         synchronized (this) { b(); c(); }
6         d();
7     }
8     y();
9 }
```

Listing 4.2: An example of *re-entrant locks* in Java using the `synchronized` keyword, which is implemented as a *monitor*. In the example, there are three locks used simultaneously over the same object. The entire method `f` is synchronised, which implicitly uses `this` as a *lock object*. Furthermore, the two *synchronised blocks* explicitly use the lock object `this`

4.2.1 Advanced Manipulation with Locks in Phase 1

Recall that the *detection of sets of calls to be executed atomically* is based on generating the pairs $(A, B) \in 2^{\Sigma} \times 2^{\Sigma}$. Now, these pairs are to be extended to store the access paths and the number of locks of lock objects that guard calls executed atomically, i.e., the B sets. Therefore, the (A, B) pairs are extended to tuples $(A, B, \pi, l) \in 2^{\Sigma} \times 2^{\Sigma} \times \Pi \times \mathbb{N}^{\top}$ where π is an access path that identifies a lock object that locks the atomic section that contains the calls from B, and l is the number of locks of the lock identified by π. For the clarity of the below description, let π be just a base variable, i.e., $\pi \in \Pi ::= Var \cup \{\varepsilon\}$. Note that π could also be ε, which is a special case when there is no lock associated with the (A, B) pair so far, i.e., B is empty and a lock was not called yet. $\mathbb{N}^{\top}$ denotes $\mathbb{N} \cup \{\top\}$, where $\top$ represents a number larger than t. Thus, the *abstract states* are elements of the set $2^{2^{\Sigma} \times 2^{\Sigma} \times \Pi \times \mathbb{N}^{\top}}$.

The analysis works as follows. When a function is called, it is appended to the A set of the element without an associated lock, i.e., the element where $\pi = \varepsilon$. Besides, the function is appended to the B sets of all the elements that have associated some lock that is currently locked, i.e., $\pi \neq \varepsilon$ and $l > 0$. When a lock is called, its identifier π_i is associated with the element without any lock associated with it, and the counter l of this element is set to 1. Then, l is incremented in all the elements where $\pi = \pi_i$ and $l > 0$. Moreover, it is created a new empty element without a lock. Finally, when an unlock with the identifier π_i is called,

l is decremented in all the elements where $\pi = \pi_i$ and $l > 0$. To formalise this process, let $inc_\top : \mathbb{N}^\top \to \mathbb{N}^\top$ and $dec_\top : \mathbb{N}^\top \to \mathbb{N}^\top$ be functions for incrementing and decrementing the number of locks of some lock objects w.r.t. the upper bound $\top$, respectively. These functions are defined as follows:

$$inc_\top(l) = \begin{cases} l+1 & \text{for } l \neq \top \text{ and } l+1 < t \\ \top & \text{otherwise} \end{cases} \qquad dec_\top(l) = \begin{cases} l-1 & \text{for } l \neq \top \text{ and } l-1 \geq 0 \\ 0 & \text{for } l = 0 \\ t-1 & \text{otherwise, i.e., } l = \top \end{cases}$$

The *initial abstract state* of a function is changed to $s_{init} = \{\{(\emptyset, \emptyset, \varepsilon, 0)\}\}$. During the analysis of a function `g` with an abstract state $s_{\mathtt{g}}$, when a leaf function `f` is called, the abstract state's transformation is changed as follows:

$$\begin{aligned} s_{\mathtt{g}} = \{p' \in 2^{2^\Sigma \times 2^\Sigma \times \Pi \times \mathbb{N}^\top} \mid \exists p \in s_{\mathtt{g}} : p' = \{(A', B', \pi', l') \in 2^\Sigma \times 2^\Sigma \times \Pi \times \mathbb{N}^\top \mid \\ \exists (A, B, \pi, l) \in p : [\pi = \varepsilon \wedge l = 0 \wedge (A', B', \pi', l') = (A \cup \{\mathtt{f}\}, B, \pi, l)] \vee [\pi \neq \varepsilon \\ \wedge ([l > 0 \wedge (A', B', \pi', l') = (A, B \cup \{\mathtt{f}\}, \pi, l)] \vee [l = 0 \wedge (A', B', \pi', l') = (A, B, \pi, l)])]\}\} \end{aligned}$$

Further, when a lock identified by an access path π_i is called, the abstract state changes as follows:

$$\begin{aligned} s_{\mathtt{g}} = \{p' \in 2^{2^\Sigma \times 2^\Sigma \times \Pi \times \mathbb{N}^\top} \mid \exists p \in s_{\mathtt{g}} : p' = \{(A, B, \pi, l) \in 2^\Sigma \times 2^\Sigma \times \Pi \times \mathbb{N}^\top \mid \\ (A, B, \pi, l) = (\emptyset, \emptyset, \varepsilon, 0) \vee [(A, B, \varepsilon, 0) \in p \wedge \pi = \pi_i \wedge l = inc_\top(0)] \vee [(A, B, \pi, l') \in p \\ \wedge \pi \neq \varepsilon \wedge [((l' = 0 \vee \pi \neq \pi_i) \wedge l = l') \vee (l' > 0 \wedge \pi = \pi_i \wedge l = inc_\top(l'))]]\}\} \end{aligned}$$

Furthermore, when an unlock identified by the access path π_i is called, the abstract state changes as follows:

$$\begin{aligned} s_{\mathtt{g}} = \{p' \in 2^{2^\Sigma \times 2^\Sigma \times \Pi \times \mathbb{N}^\top} \mid \exists p \in s_{\mathtt{g}} : p' = \{(A, B, \pi, l) \in 2^\Sigma \times 2^\Sigma \times \Pi \times \mathbb{N}^\top \mid \\ (A, B, \pi, l) = (\emptyset, \emptyset, \varepsilon, 0) \vee [(A, B, \pi, l') \in p \wedge \pi \neq \varepsilon \\ \wedge [((l' = 0 \vee \pi \neq \pi_i) \wedge l = l') \vee (l' > 0 \wedge \pi = \pi_i \wedge l = dec_\top(l'))]]\}\} \end{aligned}$$

Other algorithms (e.g., calling an already analysed *nested* function) are changed analogically.

A *summary* $\chi_{\mathtt{f}} \in 2^{2^\Sigma} \times 2^\Sigma$ of a function `f` is the same as earlier. Only access paths and lock counters from abstract states are ignored. This is, $\chi_{\mathtt{f}} = (\mathcal{B}, C)$, where:

- $\mathcal{B} = \{B' \in 2^\Sigma \mid \exists p \in s_{\mathtt{f}} : \exists (A, B, \pi, l) \in p : B \neq \emptyset \wedge B' = B\}$, where $s_{\mathtt{f}}$ is the abstract state at the end of the abstract interpretation of `f`.
- $C = \bigcup_{c \in C'} c$, where $C' = \{c \in 2^\Sigma \mid \exists p \in s_{\mathtt{f}} : \exists (A, B, \pi, l) \in p : c = A \cup B\}$.

Example 4.2.1. Consider the function `f` from Listing 4.3. There are two lock objects `L1` and `L2`, which are used simultaneously. Moreover, `L2` is locked several times without unlocking in between. Further, assume that `a`, `b`, `c` are leaf nodes of the call graph. After the extension described above, the produced summary is as follows: $\chi_{\mathtt{f}} = (\{\{\mathtt{b}\}, \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}\}, \{\mathtt{a}, \mathtt{b}, \mathtt{c}\})$. Without the extension, the summary would be $\chi'_{\mathtt{f}} = (\{\{\mathtt{a}\}\}, \{\mathtt{a}, \mathtt{b}, \mathtt{c}\})$. The reason is that only the first locks/unlocks were detected. Other locks inside atomic sections and other unlocks outside atomic sections were ignored. Moreover, the abstract state after the execution of line 7 is as follows: $s_{\mathtt{f}_7} = \{\{(\emptyset, \{\mathtt{a}, \mathtt{b}\}, \mathtt{L1}, 1), (\{\mathtt{a}\}, \{\mathtt{b}\}, \mathtt{L2}, 2), (\{\mathtt{b}\}, \emptyset, \varepsilon, 0)\}\}$.

```
1  void f()
2  {
3      lock(&L1); // {a, b, c}
4      a();
5      lock(&L2); lock(&L2); // {b}
6      lock(&L2); unlock(&L2);
7      b();
8      unlock(&L2); unlock(&L2);
9      c();
10     unlock(&L1);
11 }
```

Listing 4.3: A code snippet used to illustrate the *advanced manipulation with locks* during the first phase of the analysis

4.2.2 Advanced Manipulation with Locks in Phase 2

The pairs Ω of functions that should be called atomically are computed the same way as earlier during the *detection of atomicity violations* in Phase 2. However, dealing with access paths and re-entrant locks must, of course, be reflected in the second phase of the analysis as well. For that, while looking for *atomicity violations of pairs of function calls*, from now, the analysis stores (in addition to pairs of the most recent function calls $(\mathtt{x}, \mathtt{y})$ and the set δ of pairs that have so far been identified as violating atomicity) all the most recent pairs of function calls locked under individual locks. Hence, the *abstract state* element gets the form $(\mathtt{x}, \mathtt{y}, \delta, \lambda) \in \Sigma \times \Sigma \times 2^{\Sigma\times\Sigma\times\mathbb{N}} \times 2^{\Sigma\times\Sigma\times\Pi\times\mathbb{N}^{\top}}$, where $(\mathtt{x}, \mathtt{y}), \delta$ are as before, and λ is the set of the most recent function calls with their lock access paths and the number of locks of lock objects of these locks. Thus, the abstract states are elements of the set $2^{\Sigma\times\Sigma\times 2^{\Sigma\times\Sigma\times\mathbb{N}}\times 2^{\Sigma\times\Sigma\times\Pi\times\mathbb{N}^{\top}}}$.

The analysis works as follows. When a function `f` is called on some path that led to an abstract state $(\mathtt{x}, \mathtt{y}, \delta, \lambda)$, a new pair $(\mathtt{x}', \mathtt{y}')$ of the most recent function calls is created from the previous pair $(\mathtt{x}, \mathtt{y})$ such that $(\mathtt{x}', \mathtt{y}') = (\mathtt{y}, \mathtt{f})$. This pair is also stored in the locked pairs λ if there are any locks currently locked. Further, it is checked whether the new pair (or just the last call) violates the atomicity, and at the same time, the pair is not locked by any of the stored locks (i.e., $((\mathtt{x}', \mathtt{y}') \in \Omega \wedge \nexists\, (\mathtt{x}_\pi, \mathtt{y}_\pi, \pi, l) \in \lambda : (\mathtt{x}_\pi, \mathtt{y}_\pi) = (\mathtt{x}', \mathtt{y}')) \vee ((\varepsilon, \mathtt{y}') \in \Omega \wedge \nexists\, (\mathtt{x}_\pi, \mathtt{y}_\pi, \pi, l) \in \lambda : (\mathtt{x}_\pi, \mathtt{y}_\pi) = (\varepsilon, \mathtt{y}'))$). When the condition holds, the pair is added to the set δ of pairs that violate atomicity. When a lock with an identifier π_i is called, it is created a new empty element of λ with this identifier, and the lock counter l of this element is set to 1. Furthermore, the lock counter l of an element from λ with the access path π_i is incremented/decremented when a lock/unlock with the identifier π_i is called, respectively.

More formally, the *initial abstract state* of a function is defined as $s_{init} = \{\{(\varepsilon, \varepsilon, \emptyset, \emptyset)\}\}$. To formalise the analysis of a function, let `f` be a called leaf function on a line c. Further, let $s_{\mathtt{g}}$ be the abstract state of a function `g` being analysed before the function `f` is called.

After the call of `f`, the abstract state will be changed as follows:

$$s_{\mathtt{g}} = \{(\mathtt{x}', \mathtt{y}', \delta', \lambda') \in \Sigma \times \Sigma \times 2^{\Sigma\times\Sigma\times\mathbb{N}} \times 2^{\Sigma\times\Sigma\times\Pi\times\mathbb{N}^{\top}} \mid \exists\,(\mathtt{x}, \mathtt{y}, \delta, \lambda) \in s_{\mathtt{g}} : (\mathtt{x}', \mathtt{y}') = (\mathtt{y}, \mathtt{f}) \\ \wedge \lambda' = \{(\mathtt{x}'_\pi, \mathtt{y}'_\pi, \pi', l') \in \Sigma \times \Sigma \times \Pi \times \mathbb{N}^{\top} \mid \exists\,(\mathtt{x}_\pi, \mathtt{y}_\pi, \pi, l) \in \lambda : (\mathtt{x}'_\pi, \mathtt{y}'_\pi, \pi', l') = \\ (\mathtt{y}_\pi, \mathtt{f}, \pi, l)\} \wedge \delta' = \{(\mathtt{x}'', \mathtt{y}'', c') \in \Sigma \times \Sigma \times \mathbb{N} \mid (\mathtt{x}'', \mathtt{y}'', c') \in \delta \vee [((\mathtt{x}'', \mathtt{y}'') = (\mathtt{x}', \mathtt{y}') \vee \\ (\mathtt{x}'', \mathtt{y}'') = (\varepsilon, \mathtt{y}')) \wedge (\mathtt{x}'', \mathtt{y}'') \in \Omega \wedge c' = c \wedge \nexists\,(\mathtt{x}_\pi, \mathtt{y}_\pi, \pi, l) \in \lambda' : (\mathtt{x}_\pi, \mathtt{y}_\pi) = (\mathtt{x}'', \mathtt{y}'')]\}\}$$

Further, when a lock identified by an access path π_i is called, the abstract state is changed as follows:

$$s_{\mathtt{g}} = \{(\mathtt{x}', \mathtt{y}', \delta', \lambda') \in \Sigma \times \Sigma \times 2^{\Sigma\times\Sigma\times\mathbb{N}} \times 2^{\Sigma\times\Sigma\times\Pi\times\mathbb{N}^{\top}} \mid \exists\,(\mathtt{x}, \mathtt{y}, \delta, \lambda) \in s_{\mathtt{g}} : (\mathtt{x}', \mathtt{y}', \delta') = \\ (\mathtt{x}, \mathtt{y}, \delta) \wedge \lambda' = \{(\mathtt{x}_\pi, \mathtt{y}_\pi, \pi, l) \in \Sigma \times \Sigma \times \Pi \times \mathbb{N}^{\top} \mid (\mathtt{x}_\pi, \mathtt{y}_\pi, \pi, l) = (\varepsilon, \varepsilon, \pi_i, inc_{\top}(0)) \\ \vee [(\mathtt{x}_\pi, \mathtt{y}_\pi, \pi, l') \in \lambda \wedge [(\pi \neq \pi_i \wedge l = l') \vee (\pi = \pi_i \wedge l = inc_{\top}(l'))]]\}\}$$

Furthermore, when an unlock identified by the access path π_i is called, the abstract state is changed as follows:

$$s_{\mathtt{g}} = \{(\mathtt{x}', \mathtt{y}', \delta', \lambda') \in \Sigma \times \Sigma \times 2^{\Sigma\times\Sigma\times\mathbb{N}} \times 2^{\Sigma\times\Sigma\times\Pi\times\mathbb{N}^{\top}} \mid \exists\,(\mathtt{x}, \mathtt{y}, \delta, \lambda) \in s_{\mathtt{g}} : \\ (\mathtt{x}', \mathtt{y}', \delta') = (\mathtt{x}, \mathtt{y}, \delta) \wedge \lambda' = \{(\mathtt{x}_\pi, \mathtt{y}_\pi, \pi, l) \in \Sigma \times \Sigma \times \Pi \times \mathbb{N}^{\top} \mid l > 0 \\ \wedge [(\mathtt{x}_\pi, \mathtt{y}_\pi, \pi, l') \in \lambda \wedge [(\pi \neq \pi_i \wedge l = l') \vee (\pi = \pi_i \wedge l = dec_{\top}(l'))]]\}\}$$

A *summary* of the second phase of the analysis — $\chi_{\mathtt{f}} \in 2^{\Sigma\times\Sigma\times\mathbb{N}}$ — of a function `f` is the same as earlier. Only the locked pairs λ with their access paths from abstract states are ignored. This is, $\chi_{\mathtt{f}} = \Delta = \bigcup_{\delta' \in \Delta'} \delta'$, where $\Delta' = \{\delta' \in 2^{\Sigma\times\Sigma\times\mathbb{N}} \mid \exists p \in s_{\mathtt{f}} : \exists\,(\mathtt{x}, \mathtt{y}, \delta, \lambda) \in p : \delta \neq \emptyset \wedge \delta' = \delta\}$ and $s_{\mathtt{f}}$ is the abstract state at the end of the abstract interpretation of `f`.

Example 4.2.2. Consider the function `g` from Listing 4.4. There are two lock objects `L1` and `L2`, which are used simultaneously. Further, assume that `a`, `b` are leaf nodes of the call graph. Then, assume that the result of the first phase of the analysis is that the pair (`a`, `b`) should be called atomically, i.e., $\Omega = \{(\mathtt{a}, \mathtt{b})\}$. Before the extension distinguishing of multiple lock instances, the analysis would report an atomicity violation for these functions (line 6). This is because the locks are not distinguished, and the unlock of `L1` (line 5) would unlock everything. On the other hand, after the extension, there are not reported any atomicity violations because the pair is still locked using `L2` (i.e., in Phase 2, it is produced the following summary: $\chi_{\mathtt{g}} = \emptyset$). The abstract state of `g` after the execution of line 6 looks as follows: $s_{\mathtt{g}_6} = \{(\mathtt{a}, \mathtt{b}, \emptyset, \{(\mathtt{a}, \mathtt{b}, \mathtt{L2}, 1)\})\}$.

4.2.3 Lock Guards

Finally, support for so-called *lock guard objects* has been proposed. Lock guards are objects associated with lock objects. One lock guard can be associated with multiple lock objects, and one lock object can be associated with multiple lock guards. When a lock guard is created, all lock objects associated with it are locked. When a lock guard is destroyed (usually, when a *scope of variables* is left), all lock objects associated with it are automatically unlocked. Exceptionally, under certain circumstances, lock guards can be locked/unlocked manually. Lock guards are widely used, especially in C++, but they are used, e.g., in Java as well. To cope with them, the analysis has been extended such that they are also identified

```
1 void g()
2 {
3     lock(&L1); // {}
4     lock(&L2); // {a, b}
5     unlock(&L1);
6     a(); b();
7     unlock(&L2);
8 }
```

Listing 4.4: A code snippet used to illustrate the *advanced manipulation with locks* during the second phase of the analysis

by *access paths*. The association between a lock guard and lock objects is modelled as a pair $(\pi_g, L) \in \Pi \times 2^{\Pi}$, where π_g is an access path that identifies the lock guard and L is a set of access paths that identify the lock objects associated with the guard identified by π_g. In the *abstract states* of both phases of the analysis, the associations between lock guards and lock objects are maintained as a set that is an element of the set $2^{\Pi \times 2^{\Pi}}$, i.e., there is a set of associations between lock guards and lock objects. Subsequent locks/unlocks of lock guards are then interpreted as a sequence of locks/unlocks of lock objects associated with these lock guards, respectively.

Example 4.2.3. Using Listing 4.5, it will be demonstrated how the analysis works with lock guards. There are used two lock objects `L1`, `L2`. `L2` is a *re-entrant lock* while `L1` is not. Besides, there are also two lock guards `G1` and `G2`. On line 9, at the same time, the lock guard `G1` is associated with both `L1` and `L2`, and the guard `G2` is associated only with `L2`. Conversely, the lock object `L1` is associated only with `G1`, and `L2` is associated with both `G1` and `G2`. These associations are stored in the analysis's abstract states as the (π_g, L) pairs in the following set: $\{(\texttt{G1}, \{\texttt{L1}, \texttt{L2}\}), (\texttt{G2}, \{\texttt{L2}\})\}$. For instance, when the scope of `G1` is left, the analysis abstractly interprets it as a sequence `L1.unlock(); L2.unlock();`.

```
1  #include <mutex>
2  std::mutex L1; std::recursive_mutex L2;
3  void f()
4  {
5      std::scoped_lock G1(L1, L2); // {a, b, c}
6      a();
7      {
8          std::lock_guard<std::recursive_mutex> G2(L2); // {b}
9          b();
10     }
11     c();
12 }
```

Listing 4.5: An example of *lock guards* in C++. The entire function `f` is locked with lock objects `L1`, `L2` using `std::scoped_lock`, which is a type of lock guard that can lock multiple lock objects at once (note that `std::scoped_lock` is available since C++ 17). However, there is also an inner atomic section locked with `L2` using `std::lock_guard`. `L2` is, in effect, locked twice — it is a *re-entrant lock* of a type `std::recursive_mutex`

4.3 Analysis's Parametrisation

This section proposes solutions to reduce the number of *false alarms* reported by the basic version of Atomer. In particular, it is done by *parameterising* the analysis by some inputs provided by the user, i.e., in this case, the analysis does not have to be fully automatic. Namely, the parametrisation aims to filter out functions and *critical sections* that cause many reportings where most of them are very likely false alarms.

4.3.1 Specification of Critical Functions

One of the main reasons why Atomer in its first version reports false alarms is that, in practice, critical sections often interleave calls of functions that need to be executed atomically with common functions that need not be executed atomically (such as functions for printing to the standard output, functions for recasting variables to different types, functions related to iterators, and various other "safe" functions). Often, to find real atomicity violations, it is sufficient to focus on specific "critical" functions only.

For example, calls of *constructor* and *destructor* methods of classes do not lead to atomicity violations. Therefore, these calls can usually be ignored. Unfortunately, in general, it is not easy to differentiate between functions that should be set aside and functions to focus on because this distinction is application-specific. Therefore, the author decided to rely on the user to provide this information to the analysis. (Indeed, providing information of this kind is not so exceptional, e.g., for developers of libraries. A similar approach has also been chosen, e.g., in the *ANaConDA dynamic analyser* for concurrency issues [12], where the user can use so-called *hierarchical filters* to specify functions that the analysis should not monitor.) For this reason, the following input parameters of the analysis are proposed:

- a list of functions that will not be analysed,
- a list of functions that will be analysed (and all other functions will not be),
- a list of functions whose calls will not be considered, and
- a list of functions whose calls will be considered (and all other function calls will not be).

In other words, it is possible to specify *black-lists* and *white-lists* of functions to analyse and function calls to consider. It is also possible to combine these parameters, and they can be enabled for individual phases of the analysis. In the implementation of this approach mentioned in the following chapter, these parameters' values are read from input text files that contain one function name per line. Moreover, the implementation allows the user to specify sets of functions using *regular expressions* (in that case, the line must start with the letter `R` followed by whitespace).

4.3.2 Limitation of a Size of Critical Sections

Another issue often causing false alarms is that some programs contain "large" critical sections or critical sections that include function calls with a *deep hierarchy of nested function*

calls. Both cases can cause massive and "imprecise" atomic sets that are the source of false alarms. Indeed, such "large" and/or "deep" critical sections are likely to contain a number of calls of functions that are not critical.

To resolve the "large" critical sections' problem, the author proposes to parametrise the analysis by a parameter $p \in \mathbb{N}$ that limits the maximum length of a critical section to be taken into account. During its first phase, the analysis then discards all (A, B) pairs where $|B| > p$, i.e., it removes pairs where the number of functions in the set B (functions called atomically) is greater than the limit p.

To get to the above proposal of dealing with deeply nested critical functions, recall that, during the first phase of the analysis, when calling an already analysed *nested function*, the C set (i.e., the set of all called functions within a function) from its summary is used. If there is a deep hierarchy of nested function calls, the top level of the hierarchy uses function calls from all lower-level functions, leading to "large" critical sections. To avoid this problem, the summary $\chi = (\mathcal{B}, C) \in 2^{2^\Sigma} \times 2^\Sigma$ in Phase 1 is redefined as $2^{2^\Sigma} \times 2^{\mathbb{N} \times 2^\Sigma}$, i.e., C is no longer a set of all functions called within an analysed function. It is a set of pairs where each pair represents functions called at a particular level in the hierarchy of a nested function (0 means the top-level). For instance, the summary $\chi_{\texttt{f}} = (\emptyset, \{(0, \{\texttt{a}, \texttt{b}\}), (1, \{\texttt{x}, \texttt{y}\})\})$ of a function `f` means that there were called functions `a`, `b` in `f` and that there were called functions `x`, `y` in functions one level lower in the call-tree (i.e., in functions directly invoked from `f`). During the analysis, the summaries are passed among functions in the call hierarchy. Furthermore, the analysis uses a parameter $r \in \mathbb{N}$ to limit the number of levels considered during analysing nested functions.

More particularly, to derive these extended summaries, it is also needed to store in the abstract states the information about in which levels appear which function calls. When analysing a function `f` and a leaf function `g` is called, `g` is added to the level 0 of functions called within `f`. Furthermore, when calling an already analysed function `h`, the functions called in a level i of `h` are appended into the appropriate A or B sets of `f` if $i < r$. Finally, the information about in which levels appear which function calls is passed from a lower-level function to a higher-level function. In particular, if some function is called in a level i of `h`, it is added to the level $i + 1$ of the function `f` (which is one level higher) if $i + 1 < r$.

Example 4.3.1. Assume functions `x`, `y`, `z` from Listing 4.6. Their summaries for the first phase of the analysis are as follows: $\chi_{\texttt{z}} = (\emptyset, \{(0, \{\texttt{z1}, \texttt{z2}\})\})$, $\chi_{\texttt{y}} = (\emptyset, \{(0, \{\texttt{y1}, \texttt{y2}, \texttt{z}\}), (1, \{\texttt{z1}, \texttt{z2}\})\})$, $\chi_{\texttt{x}} = (\emptyset, \{(0, \{\texttt{x1}, \texttt{x2}, \texttt{y}\}), (1, \{\texttt{y1}, \texttt{y2}, \texttt{z}\}), (2, \{\texttt{z1}, \texttt{z2}\})\})$. When the value of the parameter r is set to 1, the summary of the function `x` is reduced as follows: $\chi'_{\texttt{x}} = (\emptyset, \{(0, \{\texttt{x1}, \texttt{x2}, \texttt{y}\}), (1, \{\texttt{y1}, \texttt{y2}, \texttt{z}\})\})$.

```
1 void z() { z1(); z2(); }
2 void y() { y1(); y2(); z(); }
3 void x() { x1(); x2(); y(); }
```

Listing 4.6: A code snippet used to illustrate the *limitation of considered nested functions*

4.4 Local/Global Atomicity Violations

The author of the thesis uses two notions — *local* and *global atomicity violations* — to distinguish two classes of "atomicity violations". A local atomicity violation is when there is

an atomicity violation within the *scope of a function* being analysed. However, in effect, it does not have to be an error in the context of the whole analysed program because the entire function may *always be locked* when it is used. Thus, it is ensured that all instructions of the function are always executed atomically in the program. On the other hand, when there is at least a single case of the function not being called atomically, it entails that there exists a possibility of a real error, which is called a global atomicity violation.

In the first version of Atomer, local/global atomicity violations are not distinguished. All atomicity violations are considered global ones. This is, of course, a source of *false alarms*. In extensive real-world programs, local atomicity violations are quite common because calls of some smaller functions are often included in atomic sections. Unfortunately, it can be challenging to differentiate between local and global atomicity violations in such programs due to *complex call graphs*.

Example 4.4.1. Consider Listing 4.7, where the functions `x`, `y` are leaf nodes of the call graph. `main` is the *top-level function* that contains the calls of functions `f` and `g`. From the first phase of the analysis, it is derived that the pair $(\texttt{x}, \texttt{y})$ should be called atomically because these functions are called in `g`, which is locked in `main`. In particular, the first phase's summary of `main` is $\chi_{\texttt{main}} = (\mathcal{B}, C) = (\{\{\texttt{g}, \texttt{x}, \texttt{y}\}\}, \{(0, \{\texttt{f}, \texttt{g}\}), (1, \{\texttt{x}, \texttt{y}\})\})$, and $\exists B \in \mathcal{B} : \{\texttt{x}, \texttt{y}\} \subseteq B$. During Phase 2, there are reported atomicity violations on lines 1 and 2 due to non-atomic executions of the pair $(\texttt{x}, \texttt{y})$ within functions `f` and `g`, respectively. Consequently, the second phase's summaries of these functions are as follows: $\chi_{\texttt{f}} = \{(\texttt{x}, \texttt{y}, 1)\}$, $\chi_{\texttt{g}} = \{(\texttt{x}, \texttt{y}, 2)\}$. Nevertheless, while the violation in `f` is global, the violation in `g` is, as a matter of fact, local because `g` is called exclusively from `main` where it is locked. Therefore, a false alarm is reported.

```
1 void f() { x(); y(); } // GLOBAL_ATOMICITY_VIOLATION: (x, y)
2 void g() { x(); y(); } // LOCAL_ATOMICITY_VIOLATION: (x, y)
3 void main()
4 {
5     f();
6     lock(&L); // {g, x, y}
7     g();
8     unlock(&L);
9 }
```

Listing 4.7: An example of so-called *local* and *global atomicity violations*

To identify local atomicity violations during Phase 2, the author proposes the following approach in a new Atomer's version. First, two *severities* of the atomicity violation reporting are defined:

- `Warning` for local atomicity violations and
- `Error` for (real) global atomicity violations.

In particular, the sets δ of pairs that violate atomicity in the abstract states and the appropriate sets Δ from the summaries are extended from $2^{\Sigma \times \Sigma \times \mathbb{N}}$ to $2^{\Sigma \times \Sigma \times \mathbb{N} \times \mathcal{S}}$, where $\mathcal{S} = \texttt{Warning} \mid \texttt{Error}$.

When an atomicity violation within an analysed function is detected as earlier, it is labelled as global, i.e., the `Error` severity is assigned to it. In the course of the analysis, the information about the violations from function summaries (i.e., the pairs from Δ) are passed from lower-level functions to higher-level functions along to the call-tree. Once a nested function call is performed inside an atomic section, all its violations are labelled as local, i.e., the `Warning` severity is assigned to them. Finally, atomicity violations (both local as well as global) are reported only from *top-level functions*, i.e., functions not called from any other functions; thus, they cannot be called under a lock somewhere higher in the call hierarchy.

Example 4.4.2. Consider Listing 4.7 again. The first analysis's phase derived that the pair $(\texttt{x}, \texttt{y})$ should be called atomically, as explained in Example 4.4.1. After extending Atomer for distinguishing local and global atomicity violations, the second phase of the analysis works as follows. First, functions `f` and `g` are analysed. This phase produces the following summaries for these functions: $\chi_{\texttt{f}} = \{(\texttt{x}, \texttt{y}, 1, \texttt{Error})\}$, $\chi_{\texttt{g}} = \{(\texttt{x}, \texttt{y}, 2, \texttt{Error})\}$. These violations must not be reported yet. Further, `main` is analysed. It receives summaries from `f` and `g`. All violations from `g` are, however, labelled as local atomicity violations. As a result, the summary of `main` is $\chi_{\texttt{main}} = \{(\texttt{x}, \texttt{y}, 1, \texttt{Error}), (\texttt{x}, \texttt{y}, 2, \texttt{Warning})\}$. In the end, the reporting is made from `main` because it is the only top-level function. On line 1, there is reported a global atomicity violation. On the contrary, it is reported just a warning (local violation) on line 2 because the call of this function is locked in `main`.

Chapter 5

Implementation of a New Version of Atomer

This chapter discusses the implementation of a new version of *Atomer*. It summarises the implementation of the whole analyser (i.e., it includes the implementation of the Atomer's basic version from [15], also described in Chapter 3) with a particular focus on the new enhancements proposed in Chapter 4. Atomer is implemented as a *module of Facebook Infer* introduced in Section 2.3. The implemented algorithms are illustrated using convenient *pseudocode* and listings of simplified code written in *OCaml*, which is an implementation language of Atomer and Facebook Infer. Sections 5.1 and 5.2 describe the implementation of *both phases* of the analysis in more detail. Moreover, Section 5.3 outlines the implementation of support for *programming languages* and *locking mechanisms* newly added to Atomer.

The implementation of both the basic version and the new version of Atomer is *publicly available* at GitHub[1]. An installation and user manual are available in Appendix B. Together with some examples, these manuals are also available in the attached memory media of the thesis (see Appendix A) and in Atomer's Wiki[2]. As it was already said, Atomer is implemented in OCaml, which is a *functional* programming language. However, it also allows using the *imperative* and *object-oriented* paradigms exploited in the Atomer's implementation as well. Since the implementation of the first version of Atomer, Facebook Infer was many times updated (in its Git repository, there are several new commits every day). Due to maintaining Atomer up-to-date with Facebook Infer, and due to some refactoring of Atomer itself, the code of the first version of Atomer since its first implementation is considerably different.

Atomer itself is implemented in files in a directory `infer/src/atomicity`. In particular, the first phase of the analysis is implemented in `AtomicSets.ml[i]`, and its *abstract domain* Q is implemented in `AtomicSetsDomain.ml[i]`. The second phase is implemented in `AtomicityViolations.ml[i]`, and its abstract domain Q is implemented

[1] The implementation of a new version of **Atomer** at GitHub as an *open-source* repository (in a branch `atomicity-sets`): https://github.com/harmim/infer. The implementation of the first version is in a branch `atomicity`. At the following address, there are available releases of both versions, including source code and executable binaries (v1.0.0 is the basic version, and v2.0.0 corresponds to the new version): https://github.com/harmim/infer/releases.

[2] **Atomer's Wiki** provides an installation and user manual together with some examples. It is available at GitHub: https://github.com/harmim/infer/wiki.

in `AtomicityViolationsDomain.ml[i]`. The phases are implemented as separate analysers in Facebook Infer. The output of the first phase is the input to the second one (as earlier shown in Figure 4.1). These analysers are defined as modules of Facebook Infer in `infer/src/base/Checker.ml[i]`, and they are registered to the framework in `infer/src/backend/registerCheckers.ml`. The analysers are enabled only by specific command-line arguments; see the mentioned manuals.

Note that in the listings in the below sections, `Domain` refers to an abstract domain Q of a particular phase of the analysis (the `AtomicSetsDomain`/`AtomicityViolationsDomain` module for the first/second phase of the analysis, respectively). Then, `Domain.t` is a type of *abstract state* of the particular phase. Furthermore, `Domain.Summary.t` is a type of *summary* χ in the particular phase.

5.0.1 The Main Analysis Function

For both phases of the analysis, the analyser is implemented as an *abstract interpreter* using the `LowerHil` module, which transforms *Smallfoot Intermediate Language (SIL) instructions* (mentioned in Section 2.3) into *High-level Intermediate Language (HIL) instructions*. HIL instructions wrap SIL instructions and simplify their utilisation, i.e., it is an abstraction over SIL. For representing functions, a filtered view of the *forward CFG* that skips *exceptional control-flow paths* is used. This type of CFG corresponds to the `ProcCfg.Normal` module in Facebook Infer. The final analyser is created as a module `Analyser` using the `LowerHil.MakeAbstractInterpreter` function.

```
1  let analyse_procedure data : Domain.Summary.t option =
2    (* Should be the analysis of this function skipped? *)
3    if f_is_ignored (Pdesc.get_pname data.pdesc) then None
4    else
5      let pre : Domain.t =
6        Domain.initial (* initial abstract state (pre-condition) *)
7      in
8      (* Compute the final abstract state (post-condition). *)
9      match Analyser.compute_post data ~initial:pre with
10     | Some (post : Domain.t) ->
11       (* Convert the abstract state to a summary. *)
12       let summary : Domain.Summary.t = Domain.Summary.create post in
13       Some summary
14     | None -> Logging.die InternalError "Analysis failed."
```

Listing 5.1: The *main analysis function* for analysing individual functions in a given program

The *main analysis function* `analyse_procedure` is implemented the same way for both phases of the analysis (for Phase 1, it is implemented in the `AtomicSets` module, and for Phase 2, it is implemented in the `AtomicityViolations` module). A simplified implementation is given in Listing 5.1. Facebook Infer's backend invokes this function for each function in an analysed program. It computes a *summary* of the analysed function. At first, the function checks whether the analysis of an analysed function should be ignored (line 3). It has to do with the *specification of critical functions* proposed in Section 4.3. The implement-

ation of that is covered in Section 5.0.4. Further, `analyse_procedure` computes the final abstract state (*post-condition*) for the analysed function using the `Analyser.compute_post` function (line 9). As a *pre-condition*, the *initial abstract state* `Domain.initial` from an abstract domain is used. On line 12, the abstract state is appropriately converted to an interprocedural function summary using `Domain.Summary.create`. Finally, this summary is returned. The crucial parts are here the types `Domain.t`, `Domain.Summary.t`, and the function `Domain.Summary.create`. These are different for individual phases of the analysis. They are described in Sections 5.1, 5.2.

```
1  let exec_instr (s : Domain.t) data : HilInstr.t -> Domain.t = function
2    | Call (Direct callee, params) (* function call instruction *)
3      (* Should be this function call ignored? *)
4      when f_is_ignored callee ~actuals:(Some params) -> s
5    | Call (Direct callee, params, loc) -> ( (* function call instruction *)
6      (* Does the call relate to the locking? *)
7      match ConcurrencyModels.get_lock_effect callee params with
8      (* Process locks/unlocks/lock guards. *)
9      | Lock locks ->
10       Domain.apply_locks (get_paths locks) s
11     | Unlock locks ->
12       Domain.apply_unlocks (get_paths locks) s
13     | GuardConstruct {guard; locks} ->
14       Domain.apply_guard_construct (get_path guard) (get_paths locks) s
15     | GuardDestroy guard ->
16       Domain.apply_guard_destroy (get_path guard) s
17     | NoEffect -> ( (* the call of a classical function *)
18       let s : Domain.t =
19         (* Process the called function. *)
20         Domain.apply_call ~fName:(Pname.to_string callee) loc s
21       in
22       (* Read a summary of an already analysed nested function. *)
23       match data.analyze_dependency callee with
24       | Some (summary : Domain.Summary.t) ->
25         (* Apply the summary to the abstract state. *)
26         Domain.apply_summary summary loc s
27       | None -> s (* leaf node *) ) )
28   | _ -> s (* Do nothing for other instructions. *)
```

Listing 5.2: The implementation of the *abstract transformers* (also called *transfer functions*)

5.0.2 Abstract Transformers

The *abstract transformers* $\tau : Instr \times Q \to Q$ over an abstract domain Q (also called *transfer functions*) are implemented almost the same way for both phases of the analysis. In general, the abstract transformer takes an abstract state as its input and produces an abstract state as an output while interpreting an instruction of an analysed program. The implementation is illustrated in Listing 5.2. It is implemented using the `exec_instr` function (in the `AtomicSets`/`AtomicityViolations` module for Phase 1/Phase 2, respectively). It

considers only function calls, i.e., instruction `CALL`. The abstract state remains unchanged when other instructions are executed (line 28). On line 4, there is checked whether the called function should be ignored (see Section 4.3). Otherwise, line 7 identifies calls related to *locking*. When the called function is some lock, `Domain.apply_locks` is called to update the abstract state appropriately (line 10). When the call is a construction of a *lock guard*, `Domain.apply_guard_construct` is used (line 14), etc. Note that besides the cases used in the pattern matching shown in the listing, there are also processed `GuardRelease`, `GuardLock`, and `GuardUnlock` for advanced manipulation with lock guards in the actual implementation. The functions `get_path`/`get_paths` are used for acquiring *access paths* of *lock objects*. The functions `Domain.apply_locks` and `Domain.apply_unlocks` are implemented differently for both phases of the analysis. This implementation is described in Sections 5.1, 5.2. However, the functions `Domain.apply_guard_construct` and `Domain.apply_guard_destroy` are the same for both phases. Thus, they are explained below. If the called function is not related to locking (line 17), the function call is interpreted in the abstract domain using `Domain.apply_call`. Further, line 23 tries to read a summary of the called function. If it is not a leaf node of the call tree, its summary is used to update the abstract state using `Domain.apply_summary` on line 26. The functions `Domain.apply_call`, `Domain.apply_summary` are essential. Thus, they are introduced in Sections 5.1, 5.2. Moreover, note that on line 5, there is a parameter `loc` of the called function. It is the location of the call in the source program. It is used to identify the location of an atomicity violation. Therefore, it is needed only in the implementation of the abstract transformers for the second phase of the analysis.

When a *lock guard is constructed*, an abstract state is updated using the function `Domain.apply_guard_construct` illustrated in Algorithm 5.1. It first appends the association of the lock guard with the underlying locks to the *guards* field of each element (program path) of the abstract state. Then, all the underlying locks are immediately locked using `Domain.apply_locks`, which is defined later. Both the lock guard and the underlying locks are identified through access paths.

Algorithm 5.1: Updating an abstract state after the *construction* of a *lock guard*

Data: lock guard's access path $\pi_g \in \Pi$; access paths $L \in 2^{\Pi}$ of locks associated with π_g; abstract state s

```
1 def apply_guard_construct(π_g, L, s):
2 |   for p ∈ s do p.guards ← p.guards ∪ {(π_g, L)};
3 |   return apply_locks(L, s);
4 end
```

After the *destruction of a lock guard*, an abstract state is changed using the function `Domain.apply_guard_destroy` given in Algorithm 5.2. For each element of the abstract state, it obtains locks associated with the lock guard being destroyed. This association is then removed from the abstract state's field *guards*. Finally, the associated locks are unlocked using the function `Domain.apply_unlocks` extended to apply unlock calls for a particular program path only.

Algorithm 5.2: Updating an abstract state after the *destruction* of a *lock guard*

Data: lock guard's access path $\pi_g \in \Pi$; abstract state s

```
1 def apply_guard_destroy(π_g, s):
2     for p ∈ s do
3         L ← p.guards(π_g);
4         p.guards ← p.guards \ {(π_g, L)};
5         s ← apply_unlocks(L, s, p);
6     end
7     return s;
8 end
```

5.0.3 Abstract Interpretation Operators

The abstract domains of both phases of the analysis are quite different. However, the *abstract interpretation operators* of these domains are the same because for both phases, the abstract states $s \in Q$ are *sets* of elements that represent an abstraction of program paths. The implementation of the operators is shown in Listing 5.3. Note that `t` is a type of abstract state. In fact, it is an alias for `TSet.t`, a module representing a set of structures where the fields of these structures are defined differently in each phase of the analysis. Thus, each phase defines its own `TSet`. Particular abstract interpretation operators (implemented in Listing 5.3) are defined as follows:

- The *ordering operator* $\sqsubseteq$ (`leq`) is defined as follows. Let `lhs` be the left-hand side of the operator and `rhs` the right-hand side. Then, `lhs` $\sqsubseteq$ `rhs` iff `lhs` is a *subset* of `rhs`. In other words, $s \sqsubseteq s' \iff s \subseteq s'$, where $s, s' \in Q$.
- The *join operator* $\sqcup$ (`join`) is defined simply as the *union* of two abstract states, i.e., $s \sqcup s' \iff s \cup s'$.
- The *widening operator* $\triangledown$ (`widen`) is defined as joining the previous and the next abstract state since the domains are *finite*. In particular, $s \triangledown s' \iff s \sqcup s'$.

```
1 (* lhs <= rhs if lhs is a subset of rhs. *)
2 let leq ~(lhs : t) ~(rhs : t) : bool = TSet.subset lhs rhs
3 (* Union of abstract states. *)
4 let join (s1 : t) (s2 : t) : t = TSet.union s1 s2
5 (* Join the previous and the next abstract state. *)
6 let widen ~(prev : t) ~(next : t) ~(num_iters : int) : t = join prev next
```

Listing 5.3: The implementation of the *abstract interpretation operators*

5.0.4 Specification of Critical Functions

In Section 4.3, there are proposed *input parameters* for specifying *black-lists*/*white-lists* of functions to analyse or function calls to consider. The decision of whether a function is analysed or a function call is considered is made in `analyse_procedure` (Listing 5.1) and

`exec_instr` (Listing 5.2), respectively. In particular, the function `f_is_ignored` implements the decision making whether a particular function is considered/ignored according to the input parameters. These parameters may be combined. For example, one can allow only specific functions to be analysed and ignore several selected function calls considered during the analysis. Moreover, these parameters may be enabled for individual phases of the analysis or specified differently for each phase. `f_is_ignored` is implemented in the `AtomicityUtils` module.

The parameters' values are read from *input text files*. How to enable the parameters (i.e., how to specify the file names of the input files using command-line arguments) is described in the manual. The input file should contain one function name to consider/ignore per line. There may be empty lines, and line comments can be used with the character `#`. Moreover, it is possible to specify sets of function names using *regular expressions* (in that case, the line must start with the letter `R` followed by any number of whitespaces). The syntax of the regular expressions corresponds to the OCaml's regular expressions syntax[3]. An example of such an input file is in Listing 5.4. Processing of input files is implemented in the `functions_from_file` class in the `AtomicityUtils` module.

```
intValue # this is a comment
R [A-Z]+.*

foo
Logger.log
R \(String\)?\.format
#this is another comment
R   \(Int\|Float\)\.toString
```

Listing 5.4: An example of an *input text file* with a *black-list/white-list* of functions to analyse or function calls to consider

5.0.5 Representation of Locks in Abstract States

```
1  module Lock = struct
2    (* A lock's access path with the number of times it has been acquired. *)
3    type t = AccessPath.t * int (* (p, l) *)
4    (* The bottom value of 'l'. *)
5    let bot : int = 0
6    (* The top value of 'l'. *)
7    let top : int = Config.atomicity_lock_level_limit
8    (* Increases 'l'. *)
9    let lock ((p, l) : t) : t = if l = top then (p, l) else (p, l + 1)
10   (* Decreases 'l'. *)
11   let unlock ((p, l) : t) : t = if l = bot then (p, l) else (p, l - 1)
12 end
```

Listing 5.5: The implementation of a module that represents *locks in abstract states*

[3]The syntax of *OCaml's regular expressions*: https://ocaml.org/api/Str.html.

5.1 Phase 1 — Detection of Atomic Sets

5.2 Phase 2 — Detection of Atomicity Violations

5.3 Support for New Languages and Locks

[[Některé věci částečně převzít z projektové praxe a z Excelu.]]

Chapter 6

Experimental Evaluation of the New Version of Atomer

[[Některé věci převzít z Excelu a z projektové praxe.]] [18]

6.1 Testing on Hand-Crafted Examples

6.2 Scalability Benchmark

6.3 Evaluation on Validation Programs Derived from Gluon

6.4 Experiments with Real-Life Programs

6.5 Summary of the Evaluation and Future Work

Chapter 7

Conclusion

Bibliography

[1] Allen, F. E. Control Flow Analysis. In: *Proceedings of a Symposium on Compiler Optimization*. Urbana-Champaign, Illinois: ACM, New York, NY, USA, July 1970, p. 1–19. DOI: 10.1145/800028.808479. ISBN 9781450373869.

[2] Blackshear, S. *Getting the most out of static analyzers* [online]. San Jose Convention Center: The @Scale Conference, 2. September 2016 [cit. 2021-04-21]. Available at: https://atscaleconference.com/videos/getting-the-most-out-of-static-analyzers.

[3] Blackshear, S., Gorogiannis, N., O'Hearn, P. W. and Sergey, I. RacerD: Compositional Static Race Detection. *Proceedings of the ACM on Programming Languages*. New York, NY, USA: Association for Computing Machinery. October 2018, vol. 2, OOPSLA'18, p. 144:1–144:28. DOI: 10.1145/3276514. ISSN 2475-1421.

[4] Blackshear, S. and O'Hearn, P. W. Facebook Engineering. *Open-sourcing RacerD: Fast static race detection at scale* [online]. 19. October 2017 [cit. 2021-04-21]. Available at: https://code.fb.com/android/open-sourcing-racerd-fast-static-race-detection-at-scale.

[5] Calcagno, C., Distefano, D., O'Hearn, P. W. and Yang, H. Compositional Shape Analysis by Means of Bi-Abduction. In: *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Savannah, GA, USA: ACM, New York, NY, USA, January 2009, p. 289–300. POPL'09. DOI: 10.1145/1480881.1480917. ISBN 978-1-60558-379-2.

[6] Cousot, P. *Abstract Interpretation* [online]. Revised 5. August 2008 [cit. 2021-04-21]. Available at: https://www.di.ens.fr/~cousot/AI.

[7] Cousot, P. *Abstract Interpretation in a Nutshell* [online]. [cit. 2021-04-21]. Available at: https://www.di.ens.fr/~cousot/AI/IntroAbsInt.html.

[8] Cousot, P. Abstract Interpretation Based Formal Methods and Future Challenges, invited paper. In: Wilhelm, R., ed. *« Informatics — 10 Years Back, 10 Years Ahead »*. Berlin, Heidelberg: Springer Berlin Heidelberg, March 2001, vol. 2000, p. 138–156. Lecture Notes in Computer Science. DOI: 10.1007/3-540-44577-3_10. ISBN 978-3-540-44577-7.

[9] Cousot, P. and Cousot, R. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Los Angeles, California: ACM Press, New York,

NY, January 1977, p. 238–252. POPL'77. DOI: 10.1145/512950.512973. ISBN 9781450373500.

[10] Cousot, P. and Cousot, R. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation, invited paper. In: Bruynooghe, M. and Wirsing, M., ed. *Proceedings of the International Workshop Programming Language Implementation and Logic Programming*. Springer-Verlag, Berlin, Germany, January 1992, p. 269–295. PLILP'92. Leuven, Belgium, 13–17 August 1992, Lecture Notes in Computer Science 631. DOI: 10.1007/3-540-55844-6_101. ISBN 978-3-540-47297-1.

[11] Dias, R. J., Ferreira, C., Fiedor, J., Lourenço, J. M., Smrčka, A., Sousa, D. G. and Vojnar, T. Verifying Concurrent Programs Using Contracts. In: *10th IEEE International Conference on Software Testing, Verification and Validation*. Tokyo, Japan: IEEE Computer Society, Los Alamitos, CA, USA, March 2017, p. 196–206. ICST'17. DOI: 10.1109/ICST.2017.25. ISBN 9781509060313.

[12] Fiedor, J., Mužikovská, M., Smrčka, A., Vašíček, O. and Vojnar, T. Advances in the ANaConDA Framework for Dynamic Analysis and Testing of Concurrent C/C++ Programs. In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Amsterdam, Netherlands: Association for Computing Machinery, New York, NY, USA, July 2018, p. 356–359. ISSTA'18. DOI: 10.1145/3213846.3229505. ISBN 978-1-4503-5699-2.

[13] Flanagan, C. and Freund, S. N. Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs. In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Venice, Italy: Association for Computing Machinery, New York, NY, USA, January 2004, p. 256–267. POPL'04. DOI: 10.1145/964001.964023. ISBN 158113729X.

[14] Gorogiannis, N., O'Hearn, P. W. and Sergey, I. A True Positives Theorem for a Static Race Detector. *Proceedings of ACM Programming Languages*. New York, NY, USA: Association for Computing Machinery. January 2019, vol. 3, POPL'19, p. 57:1–57:29. DOI: 10.1145/3290370. ISSN 2475-1421.

[15] Harmim, D. *Static Analysis Using Facebook Infer to Find Atomicity Violations*. Brno, CZ, 2019. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Department of Intelligent Systems. Supervisor Vojnar, T. Available at: https://www.fit.vut.cz/study/thesis/21689.

[16] Harmim, D. Advanced Static Analysis of Atomicity in Concurrent Programs through Facebook Infer. In: *Proceedings of the Excel@FIT'21*. Brno, CZ: Brno University of Technology, Faculty of Information Technology, 2021. Available at: http://excel.fit.vutbr.cz/submissions/2021/057/57.pdf.

[17] Hoare, C. A. R. An Axiomatic Basis for Computer Programming. *Commun. ACM*. New York, NY, USA: Association for Computing Machinery. October 1969, vol. 12, no. 10, p. 576–580. DOI: 10.1145/363235.363259. ISSN 0001-0782.

[18] Kroening, D., Poetzl, D., Schrammel, P. and Wachter, B. Sound Static Deadlock Analysis for C/Pthreads. In: *Proceedings of the 31st IEEE/ACM International*

Conference on Automated Software Engineering. Singapore, Singapore: ACM, New York, NY, USA, August 2016, p. 379–390. ASE'16. DOI: 10.1145/2970276.2970309. ISBN 978-1-4503-3845-5.

[19] Křena, B. and Vojnar, T. Automated Formal Analysis and Verification: An Overview. *International Journal of General Systems*. Taylor & Francis. November 2012, vol. 42, no. 4, p. 335–365. DOI: 10.1080/03081079.2012.757437. ISSN 0308-1079.

[20] Lamport, L. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*. Association for Computing Machinery, New York, NY, USA. July 1978, vol. 21, no. 7, p. 558–565. DOI: 10.1145/359545.359563. ISSN 0001-0782.

[21] Lengál, O. and Vojnar, T. *Abstract Interpretation. Lecture Notes in Static Analysis and Verification*. Brno, CZ: Brno University of Technology, Faculty of Information Technology, 2020. Available at: https://www.fit.vutbr.cz/study/courses/SAV/public/Lectures/sav-lecture-05-ai.pdf.

[22] Lerch, J., Späth, J., Bodden, E. and Mezini, M. Access-Path Abstraction: Scaling Field-Sensitive Data-Flow Analysis with Unbounded Access Paths (T). In: *30th IEEE/ACM International Conference on Automated Software Engineering*. Lincoln, NE, USA: IEEE Computer Society, Los Alamitos, CA, USA, November 2015, p. 619–629. ASE'15. DOI: 10.1109/ASE.2015.9. ISBN 978-1-5090-0025-8.

[23] Marcin, V. *Static Analysis Using Facebook Infer Focused on Deadlock Detection*. Brno, CZ, 2019. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Department of Intelligent Systems. Supervisor Vojnar, T. Available at: https://www.fit.vut.cz/study/thesis/21920.

[24] Meyer, B. Applying "Design by Contract". *Computer*. Washington, DC, USA: IEEE Computer Society Press. October 1992, vol. 25, no. 10, p. 40–51. DOI: 10.1109/2.161279. ISSN 0018-9162.

[25] Minsky, Y., Madhavapeddy, A. and Hickey, J. *Real World OCaml: Functional Programming for the Masses*. 1st ed. Sebastopol, CA: O'Reilly Media, 2013. ISBN 978-1-449-32391-2.

[26] Mužikovská, M. *Towards Parameterized Contract Validator in ANaConDA Framework*. Brno, CZ, 2018. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Department of Intelligent Systems. Supervisor Smrčka, A. Available at: https://www.fit.vut.cz/study/thesis/20642.

[27] Møller, A. and Schwartzbach, I. M. *Static Program Analysis*. Department of Computer Science, Aarhus University, November 2020. Available at: https://cs.au.dk/~amoeller/spa.

[28] Nielson, F., Nielson, H. R. and Hankin, C. *Principles of Program Analysis*. 2nd ed. Berlin, Heidelberg: Springer Berlin Heidelberg, January 2005. ISBN 978-3-642-08474-4.

[29] Reps, T., Horwitz, S. and Sagiv, M. Precise Interprocedural Dataflow Analysis via Graph Reachability. In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Francisco, California, USA: ACM, New

York, NY, USA, January 1995, p. 49–61. POPL'95. DOI: 10.1145/199448.199462. ISBN 0-89791-692-1.

[30] Rice, H. G. Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society*. 1953, vol. 74, no. 2, p. 358–366. DOI: 10.1090/s0002-9947-1953-0053041-6. ISSN 0002-9947.

[31] Rival, X. and Kwangkeun, Y. *Introduction to Static Analysis: An Abstract Interpretation Perspective*. 1st ed. Cambridge: The MIT Press, February 2020. ISBN 978-0-262-04341-0.

[32] Sharir, M. and Pnueli, A. Two Approaches to Interprocedural Data Flow Analysis. In: Muchnick, S. S. and Jones, N. D., ed. *Program Flow Analysis: Theory and Applications*. Prentice Hall Professional Technical Reference, January 1981, chap. 7, p. 189–211. ISBN 0137296819.

[33] Sousa, D. G., Dias, R. J., Ferreira, C. and Lourenço, J. M. Preventing Atomicity Violations with Contracts. *CoRR*. Ithaca, New York, USA: Cornell University Library, arXiv.org. May 2015, abs/1505.02951. ISSN 2331-8422.

[34] Tarski, A. A Lattice-Theoretical Fixpoint Theorem and Its Applications. *Pacific Journal of Mathematics*. June 1955, vol. 5, no. 2, p. 285–309. DOI: 10.2140/pjm.1955.5.285. ISSN 00308730.

[35] Valmari, A. The State Explosion Problem. In: Reisig, W. and Rozenberg, G., ed. *Lectures on Petri Nets I: Basic Models*. Springer, Berlin, Heidelberg, 1998, vol. 1491, p. 429–528. Lecture Notes in Computer Science. DOI: 10.1007/3-540-65306-6_21. ISBN 978-3-540-65306-6.

[36] Villard, J. Facebook Engineering. *Infer powering Microsoft's Infer#, a new static analyzer for C#* [online]. 14. December 2020 [cit. 2021-04-21]. Available at: https://engineering.fb.com/2020/12/14/open-source/infer.

[37] Von Praun, C. and Gross, T. Static Detection of Atomicity Violations in Object-Oriented Programs. *Journal of Object Technology*. Zürich, Switzerland: Laboratory for Software Technology, ETH Zürich. June 2004, vol. 3, no. 6, p. 103–122. DOI: 10.5381/jot.2004.3.6.a5. ISSN 16601769.

[38] Yi, K. Facebook Research. *Inferbo: Infer-based buffer overrun analyzer* [online]. 6. February 2017 [cit. 2021-04-21]. Available at: https://research.fb.com/inferbo-infer-based-buffer-overrun-analyzer.

Appendix A

Contents of the Attached Memory Media

[[Strukturu převzít z bakalářky.]]

Appendix B

Installation and User Manual

[[Převzít z bakalářky a aktualizovat (aktuální informace jsou na Wiki na GitHubu), hlavně přidat použití parametrů analyzátoru. Přidat taky možná instalaci přes Docker.]]

Installation Manual

User Manual