BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

STATIC ANALYSIS USING FACEBOOK INFER TO FIND ATOMICITY VIOLATIONS

STATICKÁ ANALÝZA V NÁSTROJI FACEBOOK INFER ZAMĚŘENÁ NA DETEKCI PORUŠENÍ ATOMIČNOSTI

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **DOMINIK HARMIM**
AUTOR PRÁCE

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

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2018/2019

Zadání bakalářské práce

Student: **Harmim Dominik**
Program: Informační technologie
Název: **Statická analýza v nástroji Facebook Infer zaměřená na detekci porušení atomičnosti**
Static Analysis Using Facebook Infer to Find Atomicity Violations
Kategorie: Analýza a testování softwaru

Zadání:

1. Prostudujte principy statické analýzy založené na abstraktní interpretaci. Zvláštní pozornost věnujte přístupům zaměřeným na odhalování problémů v synchronizaci paralelních procesů.
2. Seznamte se s nástrojem Facebook Infer, jeho podporou pro abstraktní interpretaci a s existujícími analyzátory vytvořenými v prostředí Faceboook Infer.
3. V prostředí Facebook Infer navrhněte a naimplementujte analyzátor zaměřený na odhalování chyb typu porušení atomicity.
4. Experimentálně ověřte funkčnost vytvořeného analyzátoru na vhodně zvolených netriviálních programech.
5. Shrňte dosažené výsledky a diskutujte možnosti jejich dalšího rozvoje v budoucnu.

Literatura:

- Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis, Springer-Verlag, 2005.
- Blackshear, S., O'Hearn, P.: Open-Sourcing RacerD: Fast Static Race Detection at Scale, 2017. Dostupné on-line: https://code.fb.com/android/open-sourcing-racerd-fast-static-race-detection-at-scale/.
- Atkey, R., Sannella, D.: ThreadSafe: Static Analysis for Java Concurrency, Electronic Communications of the EASST, 72, 2015.
- Bielik, P., Raychev, V., Vechev, M.T.: Scalable Race Detection for Android Applications, In: Proc. of OOPSLA'15, ACM, 2015.
- 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.

Pro udělení zápočtu za první semestr je požadováno:

- Body 1, 2 a alespoň začátek návrhu z bodu 3.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Vojnar Tomáš, prof. Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 15. května 2019
Datum schválení: 1. listopadu 2018

Abstract

The goal of this thesis is to propose a *static analyser* that detects *atomicity violations*. The proposed analyser — *Atomer* — is implemented as a module of *Facebook Infer*, which is an open-source and extendable static analysis framework that promotes efficient *modular* and *incremental* analysis. The analyser works on the level of *sequences of function calls*. The proposed solution is based on the assumption that sequences executed *atomically once* should probably be executed *always atomically*. The implemented analyser has been successfully verified and evaluated on both smaller programs created for testing purposes as well as publicly available benchmarks derived from *real-life low-level* programs.

Abstrakt

Cílem této práce je navrhnout *statický analyzátor*, který bude sloužit pro detekci *porušení atomicity*. Navržený analyzátor — *Atomer* — je implementován jako modul pro *Facebook Infer*, což je volně šířený a snadno rozšiřitelný nástroj, který umožňuje efektivní *modulární* a *inkrementální* analýzu. Analyzátor pracuje na úrovni *sekvencí volání funkcí*. Navržené řešení je založeno na předpokladu, že sekvence, které jsou *zavolány atomicky jednou*, by měly být pravděpodobně volány *atomicky vždy*. Implementovaný analyzátor byl úspěšně ověřen a vyhodnocen jak na malých programech, vytvořených pro testovací účely, tak na veřejně dostupných testovacích programech, které vznikly ze *skutečných nízkoúrovňových* programů.

Keywords

static analysis, programs analysis, abstract interpretation, Facebook Infer, atomicity violation, concurrent programs, contracts for concurrency, atomic sequences, atomicity, incremental analysis, modular analysis, compositional analysis, interprocedural analysis

Klíčová slova

statická analýza, analýza programů, abstraktní interpretace, Facebook Infer, porušení atomicity, paralelní programy, kontrakty pro souběžnost, atomické sekvence, atomicita, inkrementální analýza, modulární analýza, kompoziční analýza, interprocedurální analýza

Reference

HARMIM, Dominik. *Static Analysis Using Facebook Infer to Find Atomicity Violations.* Brno, 2019. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor prof. Ing. Tomáš Vojnar, Ph.D.

Rozšířený abstrakt

Softwarové chyby jsou nedílnou součástí počítačových programů od samotného vzniku programování. Naneštěstí jsou často ukryty na nečekaných místech a můžou vést k nečekanému chování, které může způsobit značné škody. Dnes existuje mnoho způsobů, jak mohou vývojáři odhalovat tyto chyby již při vývoji. Často se používají *dynamické analyzátory* nebo nástroje pro automatizované testování, které jsou v mnoha případech dostačující, nicméně stále mohou zanechat spoustu chyb neodhalených, protože jsou schopny analyzovat pouze určité toky běhu programu na základě vstupních dat. Alternativním řešením je *statická analýza*, která má však také svoje nedostatky, kde nejčastějším problémem je *škálovatelnost* při analýze rozsáhlých projektů nebo značně vysoká míra nesprávně hlášených chyb (často se používá anglický výraz „*false alarm*“).

Firma Facebook nedávno představila *Facebook Infer*: nástroj pro tvorbu *vysoce škálovatelných*, *kompozičních*, *inkrementálních* a *interprocedurálních* statických analyzátorů. Facebook Infer značně rozšířil své možnosti, ale je stále aktivně vyvíjen mnoha týmy po celém světě. Je používaný dennodenně nejen v samotné firmě Facebook, ale také v jiných firmách jako např. Spotify, Uber, Mozilla nebo Amazon. Momentálně Facebook Infer nabízí několik analyzátorů, které detekují celou řadu typů softwarových chyb, jako např. chyby typu „buffer overflow“ (přetečení vyrovnávací paměti), „data race“ a různé druhy uváznutí („deadlock“) a stárnutí („starvation“), „null-dereferencing“ (dereference prázdného ukazatele) nebo „memory leak“ (únik paměti). Ale především je Facebook Infer *aplikační rámec* pro rychlou a jednoduchou tvorbu nových analyzátorů. V aktuální verzi nástroje Facebook Infer naneštěstí stále chybí lepší podpora pro detekci chyb v *paralelních programech*. Přestože Facebook Infer nabízí docela pokročilé analyzátory na detekci chyb typu „*data race*“, jsou tyto analyzátory limitovány pouze na programy napsané v jazycích Java a C++ a nejsou navrženy na programy napsané v jazyce C, které manipulují se zámky na *nižší úrovni*.

V paralelních programech se často vyžaduje, aby určité sekvence instrukcí byly provedeny atomicky. Porušení těchto požadavků pak může způsobit různé problémy, jako např. neočekávané chování, výjimky, nepovolené přístupy do paměti („segmentation fault“) nebo jiné selhání. *Porušení atomicity* obvykle není ověřováno překladačem, na rozdíl od syntaktických nebo některých druhů sémantických pravidel. Požadavky na atomicitu navíc většinou ani vůbec nejsou dokumentovány. Takže v konečném důsledku musí samotní programátoři dbát na jejich dodržení, a to obvykle bez jakýchkoliv podpůrných nástrojů. Obecně je náročné vyvarovat se těchto chyb v *atomicky závislých programech*, obzvlášť ve velkých projektech, a ještě těžší a časově náročnější je hledání a opravování těchto chyb.

V této práci je navržen statický analyzátor — *Atomer* — pro hledání chyb určitého typu porušení atomicity, který je implementován jako modul nástroje Facebook Infer. Návrh se konkrétně zaměřuje na *atomické provádění sekvencí volání funkcí*, což je často vyžadováno, např. při použití určitých knihovních volání. Navržený princip je založen na předpokladu, že sekvence provedené *atomicky jednou*, by pravděpodobně měly být provedeny *atomicky vždy*. Návrh je dále založen na konceptu *kontraktů pro souběžnost* (anglicky „*contracts for concurrency*“). Navržená analýza je rozdělena do dvou částí (*fáze analýzy*). **Fáze 1**: detekce *atomických sekvencí*, tj. detekce volání funkcí, které se volají atomicky. **Fáze 2**: detekce *porušení atomicity*, tj. porušení atomické sekvence získané z první fáze.

Analyzátor je implementován v jazyce *OCaml*, což je implementační jazyk nástroje Facebook Infer. Implementace se konkrétně zaměřuje na programy napsané v jazycích C/C++ s použitím zámků typu *PThread*.

Funkčnost analyzátoru byla úspěšně ověřena na menších *ručně vytvořených* programech. Navíc byl analyzátor experimentálně vyhodnocen na veřejně dostupných testovacích programech odvozených od *nízkoúrovňových* programů používaných v praxi. Analýza byla provedena na 9 vybraných nízkoúrovňových programech, které obsahují několik tisíc řádků kódu. Bylo zjištěno, že Atomer je schopný analyzovat i takhle rozsáhlé programy z praxe, ale v těchto případech je nesprávně hlášeno mnoho chyb. Každopádně výsledek této analýzy může být použit jako vstup pro *dynamickou analýzu*, která může být schopna zjistit, jestli tato nahlášená porušení atomicity jsou skutečné chyby.

Atomer má potenciál pro další vylepšování. Budoucí práce se bude zaměřovat především na zlepšování přesnosti použitých metod například tak, že se budou uvažovat *vnořené zámky*, *různé instance použitých zámků*, *parametry funkcí* atd. Budoucí práce se bude dále zaměřovat na zlepšování *škálovatelnosti*, protože Atomer není schopen analyzovat rozsáhlejší a komplexnější programy. Dále by bylo zajímavé rozšířit analyzátor o *další typy zámků* pro synchronizaci souběžných vláken/procesů a otestovat analýzu na dalších v praxi používaných programech.

Vývoj analyzátoru byl diskutován s vývojáři nástroje Facebook Infer a je součástí projektu Aquas (H2020 ECSEL). Zdrojové kódy implementovaného analyzátoru jsou volně dostupné v repositáři na serveru GitHub. Předběžné výsledky práce byly publikovány a prezentovány v článku na studentské konferenci *Excel@FIT*, kde tento článek vyhrál cenu ve dvou kategoriích.

Static Analysis Using Facebook Infer to Find Atomicity Violations

Declaration

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

.....
Dominik Harmim
May 22, 2019

Acknowledgements

I would like to thank my supervisor Tomáš Vojnar. Further, I would like to thank Tomáš Fiedor for providing supplementary information and for his assistance. I would also like to thank my colleagues Vladimír Marcin and Ondřej Pavela for helpful discussions about my thesis. Furthermore, I would like to thank Nikos Gorogiannis and Sam Blackshear from the Infer team at Facebook for useful discussions about the development of my analyser. Lastly, I thank for the support received from the H2020 ECSEL project Aquas.

Contents

Chapter 1

Introduction

Bugs are an integral part of 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 are able to analyse only certain program flows dependent on the input data. An alternative solution is *static analysis* that has its own shortcomings as well, such as the *scalability* on extensive codebases or considerably high rate of incorrectly reported errors (so-called *false positives* or *false alarms*).

Recently, Facebook introduced *Facebook Infer*: a 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. But 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 fairly advanced *data race* analyser, it is limited to Java and C++ programs only and fails for C programs, which use a more *low-level* lock manipulation.

In *concurrent programs*, there are often *atomicity requirements* for 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. So in the end, programmers themselves must abide by these requirements and usually lack any tool support. And in general, it is difficult to avoid errors in *atomicity-dependent programs*, especially in large projects, and even harder and time-consuming is finding and fixing them.

In this thesis, there is proposed the *Atomer* — a static analyser for finding some forms of atomicity violations — which is implemented as a module of Facebook Infer. In particular, the stress is put on the *atomic execution of sequences of function calls*, which is often

required, e.g., when using certain library calls. In fact, the idea of checking atomicity of certain sequences of function calls is inspired by the work of *contracts for concurrency* [11]. In the terminology of [11], atomicity of certain sequences of calls is the simplest (yet very useful in practice) kind of contracts for concurrency. The implementation particularly targets C/C++ programs that use *PThread* locks.

The development of Atomer has been discussed with developers of Facebook Infer, and it is a part of the H2020 ECSEL project Aquas. Parts of this thesis and preliminary results are taken from the paper [13], which was written in collaboration with Vladimír Marcin and Ondřej Pavela.

The rest of the thesis is organised as follows. In Chapter 2 there are described all the topics which are necessary to understand before reading the rest of the thesis. In particular, Section 2.1 deals with *static analysis* based on *abstract interpretation*. Facebook Infer, which uses abstract interpretation, is described in Section 2.2. Finally, in Section 2.3 there is described the concept of *contracts for concurrency*. A proposal of a static analyser for detection of *atomicity violations*, based on this concept, is described in Chapter 3 together with a description of existing analysers of a similar kind. An implementation of the analyser is presented in Chapter 4. Subsequently, Chapter 5 discusses the experimental evaluation of the analyser. Finally, Chapter 6 concludes the thesis. In addition, there are three appendices. Appendix A provides more details of experimental verification results. Appendix B lists contents of the attached memory media, and Appendix C 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. Lastly, the chapter deals with principles which this thesis got inspired by.

In particular, in Section 2.1, there is a brief explanation of *static analysis* itself, and then an explanation of *abstract interpretation* that is used in Facebook Infer, i.e., the tool that is extended in this thesis. Facebook Infer, its principles and features are illustrated in Section 2.2. The concept of *contracts for concurrency* that the thesis gets inspired by is discussed and defined in Section 2.3.

2.1 Static Analysis by Abstract Interpretation

According to [19], *static analysis* of programs is reasoning about the behaviour of computer programs without actually executing them. It has been used since the 1970s in optimising compilers for generating efficient code. More recently, it has proven valuable also for automatic error detection, verification of 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, in other words, a static program analyser is a program that reasons about another 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 fundamental concepts of *formal verification*. It aims to automatically answer questions about a given program, such as, e.g., [19]:

- **Are certain operations executed *atomically*?**
- Does the program terminate on every input?
- Can the program *deadlock*?
- Does there exist an input that leads to a *null-pointer dereference*, a *division-by-zero*, or an *arithmetic overflow*?
- Are all variables initialised before they are used?

- Are arrays always accessed within their bound?
- Does the program contain *dead code*?
- Are all resources correctly released after their last use?

It is well-known that testing, i.e., executing programs with some input data and examining the output, 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 program analysis can prove their absence — with some *approximation* — it can check *all possible executions* of the programs and provide guarantees about their properties. Another advantage of static analysis 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 disadvantage of static analysis is that it can produce many *false alarms*[1], which is often resolved by accepting *unsoundness*[2]. Another major issue is that of ensuring sufficient *scalability* of static analysis: in fact, typically, the more precise the analysis is, the less scalable it becomes.

Various forms of static analysis of programs have been invented, for instance [24]: bug pattern searching, data-flow analysis, constraint-based analysis, type analysis, or symbolic execution. One of the most widely used approaches — *abstract interpretation* — is detailed in Section 2.1.1.

There exist numerous tools for static analysis (often proprietary and difficult to openly evaluate or extend), e.g.: Coverity, Klockwork, CodeSonar, Frama-C, PHPStan, or *Facebook Infer* (described in Section 2.2).

2.1.1 Abstract Interpretation

This section explains and defines the basics of *abstract interpretation*. The description is based on [8, 9, 6, 7, 15, 16, 10, 20, 19, 25]. In these works, there can be found more detailed and more formal explanation.

Abstract interpretation was introduced and formalised by a French computer scientist Patrick Cousot and his wife Radhia Cousot in the year 1977 at POPL (symposium on Principles of Programming Languages) [9]. It is a generic *framework* for static analyses. It allows one to create particular analyses by providing specific components (described later) to the framework. The obtained analysis is guaranteed to be *sound* if certain properties of the components are met. [15, 16]

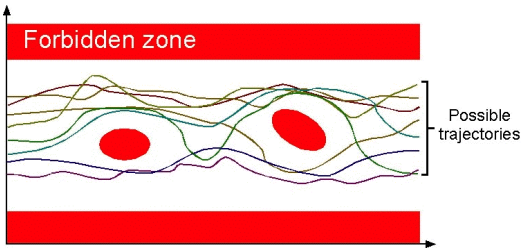
In general, in the set theory, which is independent of the application setting, abstract interpretation is considered a theory for *approximating* sets and set operations. A more restricted formulation of abstract interpretation is to interpret it as a theory of approximation of the behaviour of the *formal semantics* of programs. Those behaviours may be characterised by *fixpoints* (defined below), which is why a primary part of the theory provides efficient techniques for *fixpoint approximation* [20]. So, for a standard semantics, abstract interpretation is used to derive the approximate abstract semantics over an *abstract domain*

[1] **False alarms** – incorrectly reported an error. Also called *false positives*.

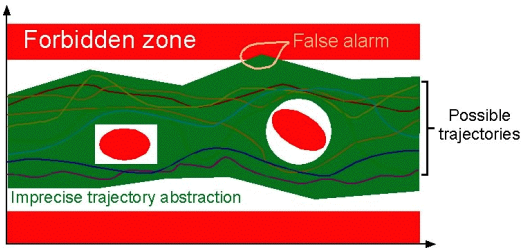
[2] **Soundness** – if a verification method claims that a system is correct according to a given specification, it is truly correct [24].

(defined below). The abstract semantics obtained as a result of program analysis can then be used for verification, optimisation, code generation or transformation, etc. [8]

Patrick Cousot intuitively and informally illustrates abstract interpretation in [6] 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 values, 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 zone is empty may be *undecidable* because the program concrete semantics is, in general, *not computable*. As demonstrates Figure 2.1b, abstract interpretation deals with an *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 zone), the concrete semantics is safe as well. However, the *over-approximation* of the possible program executions causes that inexisting program executions are considered, which may lead to *false alarms*. It is the case when the abstract semantics intersects the forbidden zone, whereas the concrete semantics does not intersect it.



(a) *Concrete semantics* of programs with *forbidden zones*



(b) *Abstract semantics* of programs with imprecise trajectory abstraction

Figure 2.1: Abstract interpretation demonstration [6]. Horizontal axes: time t. Vertical axes: vector $x(t)$ of input, state, and output values of the considered program

Components of Abstract Interpretation

In accordance with [15, 16], the basic components of abstract interpretation are as follows:

- **An Abstract Domain** [7]:
  - An abstraction of the possible concrete program states (or their parts) in the form of *abstract properties*[3] and *abstract operations*[4] [8].
  - Sets of program states at certain locations are represented using *abstract states*.
- **Abstract Transformers**:
  - There is a *transform function* for each program operation (instruction) that represents the impact of the operation executed on an abstract state.

[3] **Abstract properties** approximating *concrete properties behaviours*.

[4] **Abstract operations** include abstractions of the *concrete approximation*, an approximation of the *concrete fixpoint transform function*, etc.

- **The Join Operator** $\circ$:
  - Joins abstract states from individual program branches into a single one.
- **The Widening Operator** $\triangledown$ [20, 10, 15]:
  - Enforces termination of the abstract interpretation.
  - It is used to 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 used, the more accurate the result is (but the analysis takes more time).
- **The Narrowing Operator** $\triangle$ [20, 10, 15]:
  - Using this operator, the approximation obtained by widening can be refined, i.e., it may be used to refine the result of widening.
  - This operator is used when a *fixpoint* is approximated using widening.

Fixpoints and Fixpoint Approximation

A **fixpoint** of a function $f : A \to A$ is an element $a \in A$ if and only if $\boldsymbol{f(a) = a}$ [25].

Computation of the *most precise abstract fixpoint* is not generally guaranteed to terminate, in particular, when a given program contains a loop or recursion. The solution is to approximate the fixpoint using *widening* (over-approximation of a fixpoint) and *narrowing* (improves the approximation of the fixpoint) [15, 16]. Most program properties can be represented as fixpoints. This reduces program analysis to the fixpoint approximation [7]. Further information about fixpoint approximation can be found, e.g., in [20, 10].

Formal Definition of Abstract Interpretation

According to [9, 15], **abstract interpretation** $\boldsymbol{I}$ of a program P with the instruction set S is a tuple

$$\boldsymbol{I = (Q, \circ, \sqsubseteq, \top, \bot, \tau)}$$

where

- $\boldsymbol{Q}$ is the *abstract domain* (domain of *abstract states*),
- $\circ : Q \times Q \to Q$ is the *join operator* for accumulation of abstract states,
- $(\sqsubseteq) \subseteq Q \times Q$ is an *ordering* defined as $x \sqsubseteq y \Leftrightarrow x \circ y = y$ in $(Q, \circ, \top)$,
- $\top \in Q$ is the *supremum* of Q,
- $\bot \in Q$ is the *infimum* of Q,
- $\boldsymbol{\tau} : S \times Q \to Q$ defines *abstract transformers* for specific instructions,
- $(Q, \circ, \top)$ is a *complete semilattice* [25, 15].

Using so-called *Galois connections* [20, 10, 15, 7], one can guarantee the *soundness* of abstract interpretation.

2.2 Facebook Infer — A Static Analysis Framework

This section describes the principles and features of *Facebook Infer*. The description is based on information provided at the Facebook Infer website[5] and in [2, 16]. Parts of this section are taken from the paper [13].

Facebook Infer is an open-source[6] static analysis *framework*, which is able to discover various kinds of software bugs of a given program, with the stress put on *scalability*. The basic usage of Facebook Infer is illustrated in Figure 2.2. A more detailed explanation of its architecture is shown below. Facebook Infer is implemented in *OCaml*[7] – *functional* programming language, also supporting *imperative* and *object-oriented* paradigms. Further details about OCaml can be found in [18] or in official documentation[8], tutorials[9]. Facebook Infer was originally a standalone tool focused on *sound verification* of the absence of *memory safety violations*, which was first published in the well-known paper [5].

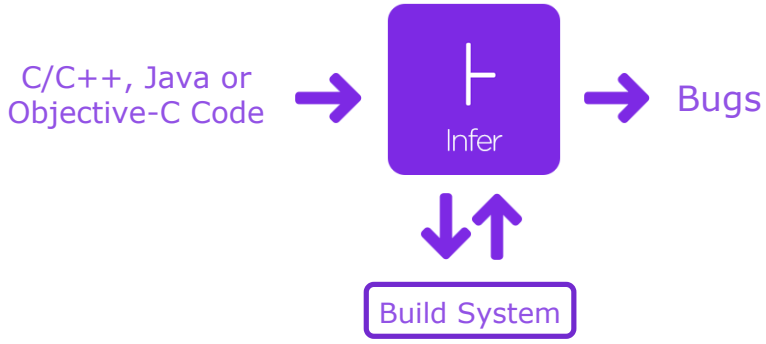


Figure 2.2: Static analysis in Facebook Infer (http://www.codeandyou.com/2015/06/infer-static-analyzer-for-java-c-and.html)

Facebook Infer is able to analyse programs written in several languages. In particular, it supports languages C, C++, Java, and Objective-C. 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* (buffer overruns) [26]; *RacerD* (data races) [3, 4, 12]; and other analyses that check for buffer overflows, some forms of deadlocks and starvation, null-dereferencing, memory leaks, resource leaks, etc.

2.2.1 Abstract Interpretation in Facebook Infer

Facebook Infer is a general framework for static analysis of programs, it is based on *abstract interpretation*. Despite the original approach taken from [5], Facebook Infer aims to find bugs rather than formal verification. It can be used to quickly develop new sorts of *compositional* and *incremental* analysers (*intraprocedural* or *interprocedural* [20]) based on the concept of function *summaries*. In general, a *summary* is a representation of *preconditions* and *postconditions* of a function. However, in practice, a summary is a custom data struc-

[5] **Facebook Infer** website – https://fbinfer.com.
[6] **Open-source repository of Facebook Infer** on GitHub – https://github.com/facebook/infer.
[7] **OCaml** website – https://ocaml.org.
[8] **OCaml documentation** – http://caml.inria.fr/pub/docs/manual-ocaml.
[9] **OCaml tutorials** – https://ocaml.org/learn/tutorials.

ture that may be used for storing any information resulting from the analysis of particular functions. Facebook Infer generally does not compute the summaries in the course of the analysis along the *Control Flow Graph* (**CFG**)[10] as it is done in classical analyses based on the concepts from [21, 22]. Instead, Facebook Infer performs the analysis of a program *function-by-function along the call tree*, starting from its leafs (demonstrated later). Therefore, a function is analysed and a summary is computed without knowledge of the call context. Then, the summary of a function is used at all of its call sites. Since summaries do not differ for different contexts, the analysis becomes more scalable, but it can lead to a loss of accuracy. In order to create a new intraprocedural analyser in Facebook Infer, it is needed to define the following (listed items are described in more detail in Section 2.1.1):

1. The *abstract domain* Q, i.e., a type of an *abstract state*.
2. Operator $\sqsubseteq$, i.e., an *ordering* of abstract states.
3. The *join* operator $\circ$, i.e., the way of joining two abstract states.
4. The *widening* operator $\triangledown$, i.e., the way how to enforce termination of the abstract interpretation of an iteration.
5. *Transfer functions* τ, i.e., a transformer that takes an abstract state as an input and produces an abstract state as an output.

Further, in order to create an interprocedural analyser, it is required to additionally define:

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

An important feature of Facebook Infer improving its scalability is *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.

The Architecture of the Abstract Interpretation Framework in Facebook Infer

The architecture of the abstract interpretation framework of Facebook Infer (**Infer.AI**) may be split into three major parts, as demonstrated in Figure 2.3: a *frontend*, an *analysis scheduler* (and a *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 instructions of SIL. The SIL language consists of the following underlying instructions:

[10] **A control flow graph (CFG)** is a directed graph in which the nodes represent basic blocks and the edges represent control flow paths [1].

- `LOAD`: reading into a temporary variable.
- `STORE`: writing to a program variable, a field of a structure, or an array.
- `PRUNE e` (often called `ASSUME`): evaluation of a condition `e`.
- `CALL`: a function call.

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

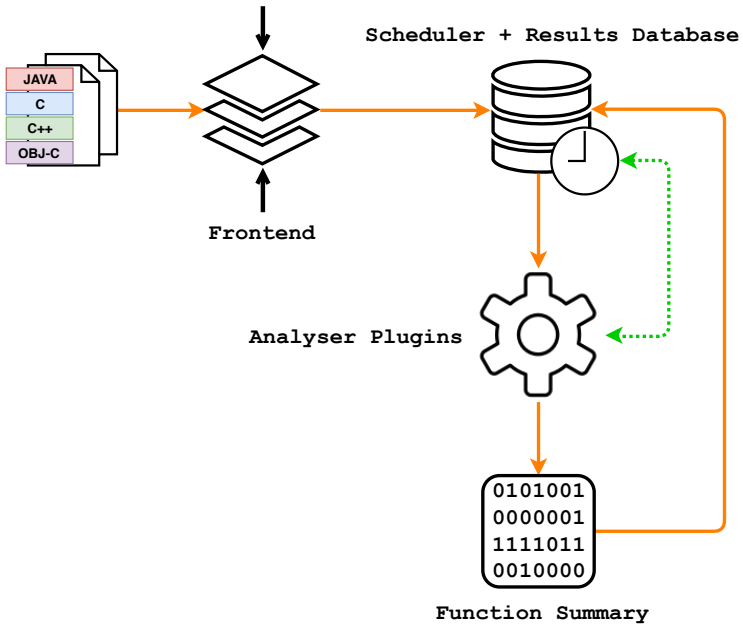


Figure 2.3: The architecture of Facebook Infer's abstract interpretation framework [2, 16]

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*[11]. The scheduler also checks if it is possible to analyse some functions simultaneously, which allows Facebook Infer to run the analysis in parallel.

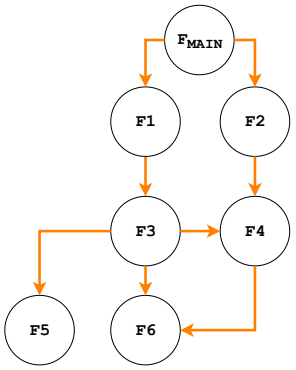


Figure 2.4: A call graph for an illustration of Facebook Infer's analysis process [2, 13, 16]

Example 2.2.1. For demonstrating the order of the analysis in Facebook Infer and its incrementality, assume a call graph in Figure 2.4. 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 taking into consideration 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

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

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 a set of analyser plugins. Each plugin performs some analysis by interpreting of SIL instructions. The result of the analysis of each function (function summary) is stored to the results database. The interpretation of SIL instructions (*commands*) is done using an *abstract interpreter* (also called a *control interpreter*) and *transfer functions* (also called a *command interpreter*). The transfer functions take an actual *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 2.5.

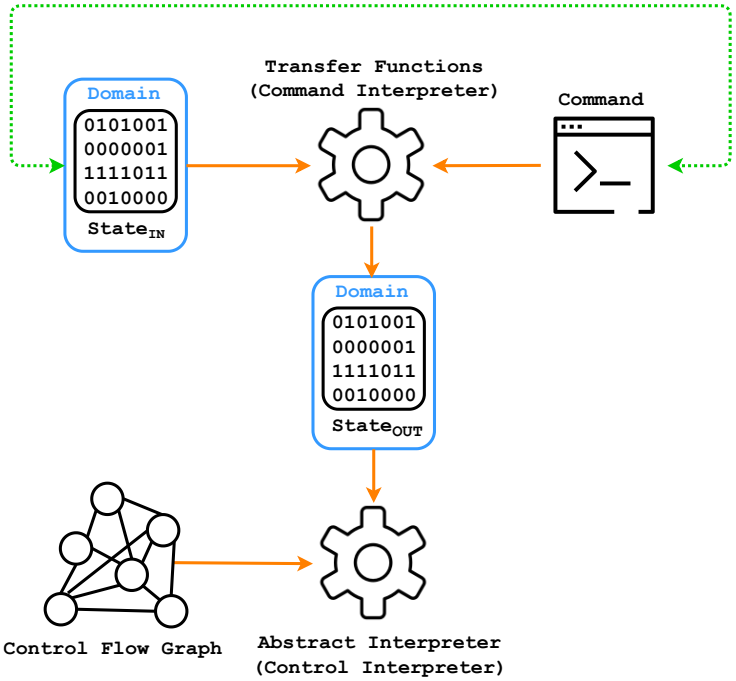


Figure 2.5: Facebook Infer's abstract interpretation process [2, 16]

2.3 Contracts for Concurrency

This section introduces the concept of *contracts for concurrency* described in [23, 11]. Parts of this section are taken from the paper [13]. Listings in this section are pieces of programs written in ANSI C.

Respecting the *protocol* of a software module — 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 function `open` at first, followed by an optional number of functions `read` and `write`, and at last, call function `close`. A program utilising such a module that

does not follow this protocol is erroneous. The methodology of *design by contract* (described in [17]) requires programs to meet such well-defined behaviours. [23]

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 that are called *spoilers* and sets of sequences of calls that are called *targets*, and it is then required that no target overlaps fully with any spoiler. Such contracts may be manually specified by a developer or they may be automatically generated 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 2.3.1 defines the notion of *basic contracts for concurrency*. Further, Section 2.3.2 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). The above mentioned more general contracts for concurrency with spoilers and targets, which essentially extend the basic contracts with some *contextual information*, are not presented here in detail (they are explained in the paper [11]). The reason is that the proposed analyser — *Atomer* — so far concentrates on the basic contracts.

2.3.1 Basic Contracts

In [11, 23], a *basic contract* is formally defined as follows. Let $\Sigma_{\mathbb{M}}$ be a set of all function names of a software module. A contract is a set $\mathbb{R}$ of *clauses* where each clause $\varrho \in \mathbb{R}$ is a *star-free regular expression*[12] 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 the union of *star-free languages*.

Example 2.3.1. Consider the following example from [11, 23]. Assume that there is a module implementing a resizable array implementing the following interface functions:

f_1: `void add(char *array, char element)`

f_2: `bool contains(char *array, char element)`

f_3: `int index_of(char *array, char element)`

f_4: `char get(char *array, int index)`

f_5: `void set(char *array, int index, char element)`

f_6: `void remove(char *array, int index)`

f_7: `int size(char *array)`

[12] **Star-free regular expressions** are regular expressions using only the *concatenation operators* and the *alternative operators* (|), without the *Kleene star operator* (∗).

The module's contract contains the following clauses:

(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 verification of the presence of an element in an array, it can be removed by some *concurrently running process*.

(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 it is applied to the address of an element, because a concurrent modification of an array may shift the position of the element.

(ϱ_3) `size (get | set | remove)`

> The execution of `size` followed by the execution of `get`, `set`, or `remove` should be atomic. Otherwise, the size of an array may be void when accessing an array, 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 (get | index_of)`

> The execution of `add` followed by the execution of `get` or `index_of` should be atomic. Otherwise, the added element needs no longer exist or its position in an array can be changed, when the program tries to obtain information about it.

2.3.2 Contracts with Parameters

The above definition of basic contracts is quite limited in some circumstances and can consider valid programs as erroneous (reports *false alarms*). Hence, in this section, there is defined an extension of basic contracts — *contracts with parameters* — which takes into consideration the data flow within function calls.

Example 2.3.2. Consider the following example from [11, 23], given Listing 2.1. There is a function `replace` that replaces item `a` in an array by item `b`. The implementation of this function comprises two atomicity violations:

(i) when `index_of` is invoked, item `a` does not need to be in the array anymore;

(ii) the acquired index can be obsolete when `set` is invoked.

A basic contract could cover this scenario by the clause ϱ_5:

(ϱ_5) `contains index_of set`

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(char *array, char a, char 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]

In order to respect function call *parameters* and *return values* of functions in contracts, the basic contracts are extended by dependencies among functions in [11, 23] 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` requiring the same objects $o_1/o_2/o_3$ to be used at the positions of `X/Y/Z`):

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

The underscore indicates a *free meta-variable* that does not restrict the contract clause.

With the extension described above, it is possible to extend the contract from Section 2.3.1 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))`

Chapter 3

Atomicity Violations Detector

In this chapter, there are described principles behind the static analyser **Atomer** that have been proposed as a module of *Facebook Infer* (introduced in Section 2.2) for detection of *atomicity violations*. In particular, the Atomer concentrates on checking *atomicity of execution of certain sequences of function calls* that is often required for a correct functioning of concurrent programs. The proposed principle is based on the assumption that sequences executed *atomically once* should probably be executed *always atomically*. The chapter also discusses already existing solutions in this area.

At first, Section 3.1 deals with existing approaches and tools for finding atomicity violations, their advantages, disadvantages, features, availability, and so on. Then, the proposed analysis algorithm that is behind Atomer is introduced in Section 3.2. Parts of this chapter are taken from the paper [13]. Listings in this chapter are pieces of exemplary programs written in ANSI C (assuming *PThread* locks and the existence of an initialised global variable `lock` of a type `pthread_mutex_t`).

3.1 Related Work

The proposed solution is slightly inspired by ideas from [11, 23]. In these papers, there is described a proposal and implementation of a *static validation* for finding some forms of *atomicity violations*, which is based on *grammars* and *parsing trees*. In the paper [11], there is also described and implemented a *dynamic* approach for the validation. The authors of [11, 23] implemented a stand-alone prototype tool[1] for analysing programs written in Java. It led to some promising experimental results but the *scalability* of the tool was still limited. Moreover, the tool from [11, 23] is no more developed. This fact inspired the decision that eventually led to this thesis, namely, to get inspired by the ideas of [11, 23], but reimplement them in *Facebook Infer* redesigning it in accordance with the principles of Facebook Infer (described in Section 2.2), which should make the resulting tool more scalable. In the end, however, due to adapting the analysis for the context of Facebook Infer, the implementation of the analysis within this thesis is significantly different from [11, 23], as it is presented in Chapter 4. Furthermore, unlike [11, 23], the implementation aims

[1] **Gluon** — a tool for static verification of *contracts for concurrency* (see Section 2.3) in Java programs — https://github.com/trxsys/gluon.

at programs written in the C/C++ languages using *POSIX Thread (PThread)* locks for *synchronisation of concurrent threads.*

In Facebook Infer, there is already implemented an analysis called *Lock Consistency Violation*[2]. It is a part of *RacerD* [3, 4, 12]. This analysis finds atomicity violations for writes/reads on single variables that are required to be executed atomically. Atomer is different, it finds atomicity violations for *sequences of functions* that are required to be executed atomically, i.e., it checks whether *contracts for concurrency* (see Section 2.3) hold. Moreover, it is trying to automatically find out which of the sequence should indeed be executed atomically (hence, to *automatically derive the contracts*).

3.2 Analysis and Design

As it has been already said, the proposal of the analyser is based on the concept of *contracts for concurrency* described in Section 2.3. In particular, the proposal considers *basic contracts* described in Section 2.3.1. Neither the contracts extended to *spoilers* and *targets* nor contracts extended by *parameters* (see Section 2.3.2) are so far taken into account.

In general, basic contracts for concurrency allow one to define *sequences of functions* that are required to be *executed atomically*, as it is explained in more detail in Section 2.3. Atomer is able to automatically derive candidates for such contracts, and then to verify whether the contracts are fulfilled. Both of these steps are done statically. The proposed analysis is divided into two parts (*phases of the analysis*):

Phase 1: Detection of (likely) *atomic sequences*, which is described in Section 3.2.1.

Phase 2: Detection of *atomicity violations* (violations of the atomic sequences), which is described in Section 3.2.2.

These phases of the analysis and its workflow are illustrated in Figure 3.1.

This section describes the proposal in general. The concrete types of the *abstract states* and the *summaries*, along with the implementation of all the necessary *abstract interpretation operators* are described in Chapter 4. But in general, the abstract states of both phases of the analysis are proposed as sets. So, in fact, the *ordering* operator is implemented using testing for a *subset*, the *join* operator is implemented as the *union*, and the *widening* operator is implemented using the join operator. Implementation of the *abstract domain* is detailed in Chapter 4.

Function summaries are in below sections reduced to the output parts only. The input parts of summaries are in case of the proposed analysis always empty, because, so far, it is not necessary to have any *preconditions* for analysed functions.

[2] **Lock Consistency Violation** — atomicity violations analysis in Facebook Infer — https://fbinfer.com/docs/checkers-bug-types.html#LOCK_CONSISTENCY_VIOLATION.

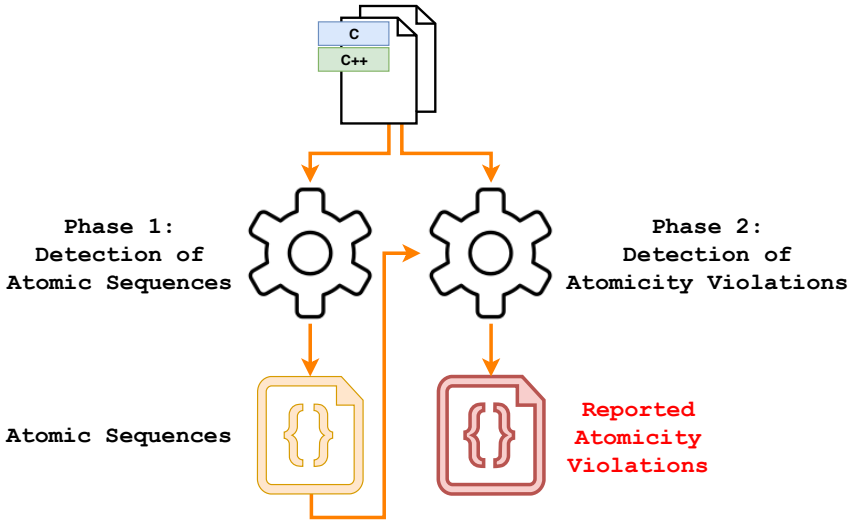


Figure 3.1: Phases of the proposed analyser

3.2.1 Phase 1: Detection of Atomic Sequences

Before the detection of *atomicity violations* may begin, it is required to have contracts introduced in Section 2.3. **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 that are executed atomically 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.

To be able to describe 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. The reason is to ensure *finiteness* of the sequences derived by the analysis and hence termination of the analysis. 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)** of sets of function calls such that: Here, **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 an unlock (or between a lock and the end of the function being analysed).

It would be more precise to generate longer sequences of the type $\mathbf{A_1B_1A_2B_2\ldots}$, instead of the sets of the sequences **(A, B)**. But it would be more difficult to ensure the finiteness of the above longer sequences and the finiteness of sets of these sequences. Moreover, there would be a significantly larger *state explosion problem*. So, the proposed representation of the sets of pairs of sequences has been chosen as a compromise between accuracy and efficiency. But the experiments (described in Chapter 5) show that for appropriate *scalability* will be (in future) needed more pronounced abstraction.

Example 3.2.1. For explanation of the computation of the sets of the sequences **(A, B)**, assume that a state of the analysis of the program P is the following sequence of function calls: `f1 f2`; and a state of the analysis of the program P' is the following sequence of function calls: `f1 f2` (`g1 g2`. The parentheses are used to indicate an *atomic sequence* (closing parenthesis is missing in the second case, which means that the program state is currently inside an *atomic block*). The computed set for the program P is $P_s = \{[\texttt{f1 f2}; \sqcup]\}$, and for the program P', it is $P'_s = \{[\texttt{f1 f2; g1 g2}]\}$. Now, if the next instruction is a call of the function `x`, in the case of the program P, the call will be added to the **A** sequence, and in the case of the program P', the call will be added to the **B** sequence as follows: $P_s = \{[\texttt{f1 f2 x}; \sqcup]\}$, $P'_s = \{[\texttt{f1 f2; g1 g2 x}]\}$. Subsequently, if the next step in the program P is a lock call, the next function calls will be added to the **B** sequence of the set P_s. And if the next step in the program P' is an unlock call, it will be created a new element of the set P'_s and the next function call will be added to the **A** sequence of this element. Finally, if the function `y` is called, the resulting sets will look like follows: $P_s = \{[\texttt{f1 f2 x; y}]\}$, $P'_s = \{[\texttt{f1 f2; g1 g2 x}], [\texttt{y}; \sqcup]\}$.

The *summary* of a function then consists of:

(i) The set of all the **B** sequences that appear on program paths through the function.

(ii) The *concatenation* of all the **A** and **B** sequences with removed duplicates of function calls. In particular, assume that there is the following computed set of the sequences **(A, B)**: $\{(A_1; B_1), (A_2; B_2), \ldots, (A_n; B_n)\}$, then the result of the concatenation is the sequence $A_1 \cdot B_1 \cdot A_2 \cdot B_2 \cdot \ldots \cdot A_n \cdot B_n$. Intuitively, in this component of the summary, it is gathered occurrences of all called functions within an analysed function, which is obtained by concatenation of all the **A** and **B** sequences.

The latter is recorded for the purpose of analysing functions higher in the *call hierarchy* since locks/unlocks can appear in such a *higher-level function*.

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

$$\overbrace{\texttt{f1 }\cancel{\texttt{f1}}}^{\mathbf{A_1}}\overbrace{\texttt{(f1 }\cancel{\texttt{f1}}\texttt{ f2)}}^{\mathbf{B_1}} \mid \overbrace{\texttt{f1 }\cancel{\texttt{f1}}}^{\mathbf{A_2}}\overbrace{\texttt{(f1 f3)}}^{\mathbf{B_2}} \mid \overbrace{\cancel{\texttt{f1}}}^{\cancel{\mathbf{A_3}}}\overbrace{\texttt{(}\cancel{\texttt{f1 f3 f3}}\texttt{)}}^{\cancel{\mathbf{B_3}}}$$

The functions `f1`, `f2`, `f3` are not deeper analysed because it is assumed that these functions are leaf nodes of the *call graph*. The strikethrough of the functions `f1` and `f3` denotes the removal of already recorded function calls in the **A** and **B** sequences. The strikethrough of the entire sequence `f1` (`f1 f3 f3`) means discarding sequences already seen before. The derived summary components for the function `g` are then as follows:

(i) {(`f1 f2`), (`f1 f3`)}, i.e., $\mathbf{B_1}$ and $\mathbf{B_2}$;

(ii) `f1 f2 f3`, i.e., the concatenation of $\mathbf{A_1}$, $\mathbf{B_1}$, $\mathbf{A_2}$, and $\mathbf{B_2}$ from which duplicate function calls were removed.

```
1  void g(void)
2  {
3      f1(); f1();
4
5      pthread_mutex_lock(&lock);
6      f1(); f1(); f2();
7      pthread_mutex_unlock(&lock);
8
9      f1(); f1();
10
11     pthread_mutex_lock(&lock);
12     f1(); f3();
13     pthread_mutex_unlock(&lock);
14
15     f1();
16
17     pthread_mutex_lock(&lock);
18     f1(); f3(); f3();
19     pthread_mutex_unlock(&lock);
20 }
```

Listing 3.1: A code snippet used for 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 into the current **A** or **B** sequence. In particular, assume that (A, B) is the current pair of sequences of the actual state of the analysis. Subsequently, it is called the function `f` with *non-empty summary*, where S is the second component of its summary. If the current state of an analysed function is inside an atomic block, the result of this step of the analysis will be $(A, B \cdot \texttt{f} \cdot S)$, otherwise, the result will be $(A \cdot \texttt{f} \cdot S, B)$. In such cases where a summary is empty, i.e., an analysed function is a leaf node of the call graph, it is appended just the function name to the current pair of sequences of the actual state of the analysis.

Example 3.2.3. This example shows how the function `h` from Listing 3.2 would be analysed using the result of the analysis of the function `g` from Listing 3.1. So the analysis of the function `h` produces the following sequence:

`f1 g ~~f1~~ f2 f3 (g f1 f2 f3)`

The derived summary components for the function `h` are then as follows:

(i) {(`g f1 f2 f3`)};

(ii) `f1 g f2 f3`.

```
1 void h(void)
2 {
3     f1(); g();
4
5     pthread_mutex_lock(&lock);
6     g();
7     pthread_mutex_unlock(&lock);
8 }
```

Listing 3.2: A code snippet used for illustration of the derivation of sequences of functions called atomically with a nested function call (function `g` 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 Listing 3.3 — two solutions have been proposed:

1. At the end of a function, everything is unlocked, i.e., one virtually appends an unlock to the end of the function if it is necessary. Then for the function `x` from Listing 3.3, the first component of its summary (i.e., atomic sequences) would be {(`a`)}. Subsequently, all unlock calls not preceded by a lock are ignored. So the first component of a summary of the function `y` from Listing 3.3 would be the empty set.

2. Addition of two further items to the summaries:

   (a) function calls missing an unlock call,

   (b) function calls missing a lock call.

   For the example from Listing 3.3, this would give:

   - for `x`: {(`f1`},
   - for `y`: {`f2`)}.

   The above sequences would have to be glued to the sequences captured higher in the call hierarchy. Calls of the functions `f1` and `f2` will also appear in the second component of the function summaries (i.e., the sequences of all functions called).

In the end, the first approach of treating such cases described above has been chosen. The reason is that it is much easier for implementation. However, in future, the analysis can be improved by implementing the second approach.

```
1  void x(void)
2  {
3      pthread_mutex_lock(&lock);
4      f1();
5  }
6
7  void y(void)
8  {
9      f2();
10     pthread_mutex_unlock(&lock);
11 }
12
13 void main(void)
14 {
15     x(); y();
16 }
```

Listing 3.3: A code snippet used for illustration of treating cases where lock/unlock calls are not paired in a function

Summary of the Detection of Atomic Sequences and Future Work

The above detection of atomic sequences has been implemented, as it is described in Section 4.1. Furthermore, it has been successfully verified on a set of sample programs created for testing purposes. The verification is presented in Chapter 5 and in Section A.1. The derived sequences of calls assumed to execute atomically, i.e., the **B** sequences, from the summaries of all analysed functions are stored into a file, which is used during **Phase 2**, described below in Section 3.2.2. There are some possibilities for further extending and improving **Phase 1**, e.g., working with *nested locks*; distinguishing the *different locks* used (currently, it is not distinguished between the locks at all); considering *contracts for concurrency with parameters* defined in Section 2.3.2 or other extensions of contracts for concurrency discussed in Section 2.3; or extending the detection for *other types of locks* for synchronisation of concurrent threads/processes. On the other hand, to further enhance the *scalability*, it seems promising to replace working with the **A** and **B** sequences by working with sets of calls: sacrificing some precision but gaining the speed.

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) while this is not the case on some path through the CFG. The pairs of a function whose calls are to be checked for atomicity are obtained as follows: For each function in a given program, it is taken the first component of its *summary* $\{B_1, B_2, \ldots, B_l\}$ and it is taken every pair of functions `f1 f2` that appears as a *substring* in some of the B_i sequences, i.e., $B_i = w_1\texttt{f1f2}w_2$ for some sequences w_1 and w_2. Moreover, if some B_i consists of a single function, it is checked that this function is always called under a lock. The implementation of an algorithm for this detection is described in Section 4.2, particularly, in Algorithm 4.4.

Example 3.2.4. For example, assume that the result of the first phase is the following set of functions called atomically (all the atomic sequences from all functions in an analysed program):

$$\{(\texttt{f1 f2 f3}), (\texttt{f1 f3 f4})\}$$

Then the analysis will look for the following pairs of functions that are not called atomically:

- `f1 f2`
- `f2 f3`
- `f1 f3`
- `f3 f4`

The analysis of functions with nested function calls and cases where lock/unlock calls are not paired in a function are handled analogically as in **Phase 1**. For detailed examples see verification experiments in Section A.2.

Example 3.2.5. For a demonstration of the detection of an atomicity violation, assume the functions `a` and `b` from Listing 3.4. The set of atomic sequences of the function `a` is $\{(\texttt{f2 f3})\}$. In the function `b`, an atomicity violation is detected because the functions `f2` and `f3` are not called atomically (under a lock).

Summary of the Detection of Atomicity Violations and Future Work

Like in the first phase of the analysis, **Phase 2** has been implemented, as it is described in Section 4.2. The implementation has been also successfully verified on a set of sample purposeful programs as discussed in Chapter 5 and in Section A.2. **Phase 2** also has the potential for further enhancing. It is possible to extend this phase with all the improvements discussed in Section 3.2.1. The next idea is to consider atomic sequences from the first phase only if they appear in an *atomic block* more than, e.g., three times. It would strengthen the certainty that such a sequence should be called atomically.

```
1  void a(void)
2  {
3      f1();
4
5      pthread_mutex_lock(&lock);
6      f2(); f3();
7      pthread_mutex_unlock(&lock);
8
9      f4();
10 }
11
12 void b(void)
13 {
14     f1(); f2(); f3(); f4();
15 }
```

Listing 3.4: Example of an atomicity violation

Chapter 4

Implementation

This chapter describes the implementation of the *static analyser* — *Atomer* — proposed in Chapter 3. The analyser is implemented as a module of *Facebook Infer* introduced in Section 2.2. The implementation is demonstrated using algorithms in *pseudocode* and using listings with codes written in *OCaml*, which is an implementation language of Facebook Infer. Sections 4.1 and 4.2 describes the implementation of the *detection of atomic sequences* defined in Section 3.2.1 and an implementation of the *detection of atomicity violations* defined in Section 3.2.2, respectively.

The implementation of the analyser can be found publicly on GitHub[1]. The implementation is done in OCaml and it exploits both *functional* and *imperative* paradigm. Facebook Infer supports analysis of programs written in Java, C, C++, and Objective-C. However, the implementation aims at programs written in the C/C++ languages using *POSIX Thread (PThread)* locks, which is a *low-level* mechanism for *synchronisation of concurrent threads*. So, the analyser understands the `pthread_mutex_lock` as the lock function and the `pthread_mutex_unlock` as the unlock function. It is also possible to run the analysis on programs written in Java or Objective-C languages but the result of the analysis would be likely wrong since these languages use a different mechanism for synchronisation.

Phase 1, i.e., the detection of atomic sequences and **Phase 2**, i.e., the detection of atomicity violations are implemented as separate analysers in Facebook Infer. The output of the first phase is the input of the second phase (as shown in Figure 3.1). Both of these analysers are registered as modules of Facebook Infer in a file `infer/src/checkers/registerCheckers.ml`. These analyses run only if a particular command line argument of Facebook Infer is specified. Implementations of individual phases are discussed below (Section 4.1 and Section 4.2).

In order to make the analysis *interprocedural*, it is necessary to define a type of function *summaries* for each phase. The types of summaries are defined in *abstract domains* of each phase. However, the summaries are stored and accessed using the module `Payloads` (called as the *summary payload*). Fields of the payload that refer to the summaries of the analysis are defined in the file `infer/src/backend/Payloads.ml[i]`.

[1] **The implementation of the analyser** in a GitHub repository, which is a *fork* of the official repository of Facebook Infer, in a branch `atomicity` – https://github.com/harmim/infer/tree/atomicity.

For both phases of the analysis, the analyser is implemented as an *abstract interpreter* using the `LowerHil` module which transforms *SIL instructions* into *HIL instructions*. HIL instructions just wrap SIL instructions, mentioned in Section 2.2.1, and simplify their utilisation. For representing functions, *forward CFGs with no exceptional control-flow* are used. This type of the CFG corresponds to the `ProcCfg.Normal` module in Facebook Infer. *Transfer functions* of both phases are implemented using the same *interface* of an abstract domain, as illustrated in Listing 4.1. In general, a transfer function takes an *abstract state* as its input and produces an abstract state as the output executing the appropriate instruction. In this case, the transfer function (defined on line 1 in Listing 4.1) modifies the abstract state only if a function is called (line 3) (`CALL` instruction). When the called function is a lock or an unlock (lines 6, 8), the abstract state is appropriately updated in the abstract domain of the analysis (lines 7, 9), which is detailed later. Otherwise, the abstract state is updated by appending the called function (line 13) and then, if the called function has already been analysed (line 16), its summary is used to update the abstract state (line 18).

```
1  let exec_instr astate procData _ instr =
2    match instr with
3    | Call (_, Direct procName, _, _, _) ->
4      let procNameS = Procname.to_string procName in
5
6      if is_lock procNameS then
7        Domain.update_astate_on_lock astate
8      else if is_unlock procNameS then
9        Domain.update_astate_on_unlock astate
10     else
11     (
12       let astate =
13         Domain.update_astate_on_function_call astate procNameS
14       in
15
16       match Payload.read procData.pdesc procName with
17       | Some summary ->
18         Domain.update_astate_on_function_call_with_summary astate summary
19       | None -> astate
20     )
21   | _ -> astate
```

Listing 4.1: *Transfer functions* of the analysers

The abstract domains of both phases are quite different. However, the essential *operators* of the abstract domains are practically the same, because for both phases the abstract state is a type of a set. The implementation of these operators is shown in Listing 4.2, where `TSet` is a module representing a **set of structures**, where the fields of these structures are defined differently in each phase. So, each phase of the analysis defines its own `TSet`. The abstract state is then of a type of `TSet`. So particular operators are defined as follows (see also Listing 4.2):

- The *ordering* operator $\sqsubseteq$ (in Facebook Infer, it is denoted `<=`) is defined as follows. Let `lhs` be the left-hand side of this operator and `rhs` the right-hand side of this operator. Then, `lhs <= rhs` (`lhs` is less or equal to `rhs`) if and only if `lhs` is a *subset* of `rhs`.
- The *join* operator $\circ$ (in Facebook infer, it is denoted `join`) is defined simply as the *union* of two abstract states.
- The *widening* operator $\triangledown$ (in Facebook Infer, it is denoted `widen`) is defined as `join` of the previous and next abstract states. Hence, there is no acceleration of the computation currently. Note, however, that due to the working with the *reduced sequences* and due to having only *pairs of sequences* of calls with/without locks (instead of longer sequences of alternating calls with/without locks), it is guaranteed that the computation will terminate.

```
1 let ( <= ) ~lhs:leftSide ~rhs:rightSide =
2   TSet.is_subset leftSide ~of_:rightSide
3
4 let join astate1 astate2 =
5   TSet.union astate1 astate2
6
7 let widen ~prev:prevAstate ~next:nextAstate ~num_iters:_ =
8   join prevAstate nextAstate
```

Listing 4.2: Essential *operators of the abstract domains* of the analysers

4.1 Implementation of the Detection of Atomic Sequences

The main ideas behind the detection of sequences of calls that are likely to be required to execute atomically have been presented in Section 3.2.1. The first phase of the analysis is started using the command line argument `--atomic-sequences`. The detected sequences that are expected to *execute atomically* are printed into a file, as explained in Section 4.1.2.

The main function of the analyser of this phase is `analyse_procedure`, which is shown in Listing 4.3. Facebook Infer invokes this function for every function in an analysed program. It produces a *summary* for the given function. The `analyse_procedure` function computes an *abstract state* for the analysed function using the created abstract interpreter `Analyser` on an *abstract domain*. As the *precondition*, the initial abstract state `initialAstate` from the abstract domain is used. If the computation succeeds, the abstract state is appropriately

updated and converted to the function summary by applying functions in the abstract domain. In the end, the *summary payload* is updated with the resulting summary.

```
let analyse_procedure args =
  let procData = ProcData.make_default args.proc_desc args.tenv in

  match Analyser.compute_post procData ~initial:Domain.initialAstate with
  | Some astate ->
    let summary =
      let astate = Domain.update_astate_at_the_end_of_function astate in
      Domain.convert_astate_to_summary astate
    in

    Payload.update_summary summary args.summary
  | None -> Logging.(die InternalError) "Analysis failed."
```

Listing 4.3: The analysis of a function in the analyser of **Phase 1**

The abstract domain of **Phase 1** of the analysis is described in Section 4.1.1. It includes the definition of an abstract state, summary, and functions working with them. The *ordering* of abstract states, the *join operator*, and the *widening operator* were already defined above.

4.1.1 The Abstract Domain for the Detection of Atomic Sequences

In this section, it is first described how *abstract states* of the *abstract domain* used in **Phase 1** of the analysis look like. Also there are described the functions used to work with these abstract states. Furthermore, there are described summaries of functions used in this phase of the analysis together with functions designed for dealing with these summaries.

Abstract States Used for the Detection of Atomic Sequences

The abstract state is of the type `TSet`. `TSet` is a module representing a **set of structures**. The structure have the following fields:

- `firstOccurrences`: a **list of strings** that captures the *first occurrences* of function calls in the **A** or **B** sequences defined in Section 3.2.1. In other words, it captures the first occurrences of function calls inside or outside *atomic blocks*.
- `callSequence`: a **list of strings** that is used for storing the **A** sequences followed by the **B** sequences. In other words, it stores function calls outside atomic blocks followed by function calls inside atomic blocks. For instance, `f1 f2 (f3)`.
- `finalCalls`: a **set of lists of strings** that is used for storing a set of sequences of calls `callSequence`. For instance, {`f1 f2 (f3)`, `f2 (f1 f3)`}.
- `isInLock`: a **boolean** that determines whether the current state of a function is inside or outside an atomic block, i.e., it is or it is not under a lock.

The *initial abstract state* is then a set with a single empty element. The empty element is an element where `firstOccurrences` and `callSequence` are empty strings, `finalCalls` is the empty set, and `isInLock` is `false`.

In the code of the analysis shown in Listings 4.1 and 4.3, several functions designed for dealing with the abstract states are used. The functions are described below (all of these functions modify all elements of the abstract state):

- `update_astate_on_function_call`: this function is invoked when any function (except a lock or an unlock) is called. It captures the first occurrence of the called function.
- `update_astate_on_lock`: this function is invoked when the locking function is called. When the state is not under a lock, it sets the flag indicating the start of an atomic sequence. Moreover, capturing the first occurrences of function calls inside the atomic sequence begins.
- `update_astate_on_unlock`: this function is invoked when the unlocking function is called. When the state is under a lock, it unsets the flag indicating the start of an atomic sequence. Moreover, capturing of the first occurrences of function calls followed this unlock call begins, and the last captured function calls, i.e., the last captured **A** and **B** sequences, are moved into the set of all such captured sequences within an analysed function.
- `update_astate_at_the_end_of_function`: this function is invoked at the end of the analysis of a function. It moves the last captured function calls, i.e., the last captured **A** and **B** sequences, into the set of all such captured sequences within an analysed function.

Function Summaries of the Domain used for the Detection of Atomic Sequences

The summary is a **structure** with the following fields:

- `atomicSequences`: a **list of lists of strings** that contains all captured atomic sequences within an analysed function, for instance, (`f3`) (`f1 f3`). This field accumulates the derived sequences assumed to be executed atomically, which will be a part of the output of the analysis.
- `allOccurrences`: a **list of strings** that contains all functions called within an analysed function. It is used for the purpose of analysing functions higher in the *call hierarchy*.

In the code of the analysis shown in Listings 4.1 and 4.3, several functions designed for dealing with the summaries are used. In particular, the following functions are used there:

- `update_astate_on_function_call_with_summary`: this function is invoked when the called function `f` has already been analysed so that the abstract state can be updated with its summary. Therefore, occurrences of all functions called from `f` are appended to the first occurrences of an analysed function. As shown in Algorithm 4.1.
- `convert_astate_to_summary`: this function is invoked at the end of the analysis of a function. It transforms the abstract state of the given function to the summary. In particular, it derives all atomic sequences and all called functions within the analysed function from the abstract state, as shown in Algorithm 4.2.

Algorithm 4.1: Updating abstract state with the summary of a called function

```
1  def update_astate_on_function_call_with_summary(astate, sum):
2      if sum.allOccurrences ≠ [] then
3          for e ∈ astate do
4              for o ∈ sum.allOccurrences do
5                  e.firstOccurrences ← AddUniq(e.firstOccurrences, o);
6              end
7          end
8      end
9      return astate;
10 end
```

Algorithm 4.2: Converting an abstract state to the function summary

```
1  def convert_astate_to_summary(astate):
2      atomicSeq ← [];
3      allOccur ← [];
4      for e ∈ astate do
5          for c ∈ e.finalCalls do
6              atomicSeq ← AddUniq(atomicSeq, GetAtomicSeq(c));
7              allOccur ← AddUniq(allOccur, GetAllCalls(c));
8          end
9      end
10     return (atomicSeq, allOccur);
11 end
```

4.1.2 Output of the Detection of Atomic Sequences

The output of **Phase 1** are sequences of functions that were identified as those that are likely to be executed always atomically. These sequences are given separately for each function in which they were identified. These sequences are derived from the summaries of all analysed functions. At the end of the entire analysis, the sequences are printed into the file `infer-atomicity-out/atomic-sequences` in the following format. Each line of the file contains a list of the detected atomic sequences within a particular function. It starts by the name of a function followed by a colon and a whitespace. Then, there are listed the atomic sequences (function names separated by a whitespace) that were derived in the given function, separated by a whitespace. Here is example of the output:

```
functionA: (f1 f2) (f3 f1)
functionB:
functionC: (f3 f4) (f6)
```

The derivation of the atomic sequences and their printing is described on Algorithm 4.3. The atomic sequences are then further processed in the second phase of the analysis, see Section 4.2.

Algorithm 4.3: Printing atomic sequences from the summaries of all analysed functions

```
   Input: A set F of all analysed functions
1  for f ∈ F do
2  |  printf('%s: ', GetFunName(f));
3  |  S ← ReadSummary(f);
4  |  for q ∈ S.atomicSequences do
5  |  |  printf('(%s) ', SeqToString(q));
6  |  end
7  end
```

4.2 Implementation of the Detection of Atomicity Violations

The main ideas of this phase are described in Section 3.2.2. It is started by the command line argument `--atomicity-violations`. It detects *atomicity violations*, i.e., violations of the *atomic sequences* obtained from **Phase 1**. The atomic sequences are read from the file `infer-atomicity-out/atomic-sequences` (see Section 4.1.2). If this file does not exist, i.e., the previous phase of the analysis has not run yet, **Phase 2** will fail.

As in the first phase of the analysis, the main function of the analyser of this phase is the function `analyse_procedure`, which is shown in Listing 4.4. Facebook Infer invokes this function for every single function in an analysed program. The `analyse_procedure` function then produces a summary for the analysed function. The function first initialises the abstract domain of this phase, and then it computes the abstract state that the analysed function reaches at the end of its execution. For that, it uses the abstract interpreter `Analyser` running on the abstract domain designed for the second phase of the analysis.

As the *precondition* of each analysed function, the initial abstract state `initialAstate` from the abstract domain is used. If the computation succeeds, the final abstract state is converted to the function summary. Further, atomicity violations within the analysed function are reported based on the abstract state. This reporting is in more detail described in Section 4.2.2. At the end, the *summary payload* is updated with the resulting summary.

```
1  let analyse_procedure args =
2    Domain.initialise true;
3    let procData = ProcData.make_default args.proc_desc args.tenv in
4
5    match Analyser.compute_post procData ~initial:Domain.initialAstate with
6    | Some astate ->
7      let summary = Domain.convert_astate_to_summary astate in
8
9      Domain.report_atomicity_violations astate ( fun loc msg ->
10       Reporting.log_error
11         args.summary ~loc:loc IssueType.atomicity_violation msg );
12
13     Payload.update_summary summary args.summary
14   | None -> Logging.(die InternalError) "Analysis failed."
```

Listing 4.4: The analysis of a function in the analyser of **Phase 2**

The abstract domain of this phase is described in Section 4.2.1. The description includes the initialisation of the domain, the definition of an abstract state, summaries, and functions working with them. The *ordering* of abstract states, the *join operator*, and the *widening operator* are defined at the beginning of Chapter 4.

4.2.1 The Abstract Domain for the Detection of Atomicity Violations

In this section, at first, it is explained how the abstract domain of this phase is initialised. Then it is described the definition of an abstract state along with functions working with it. At the end, it is explained how the summaries of functions look like in this phase of the analysis together with functions working with the summaries.

Initialisation of the Domain of the Detection of Atomicity Violations

Before analysing each function, i.e., at the beginning of the function `analyse_procedure`, the abstract domain is initialised. The initialisation servers for processing the input file with atomic sequences and storing these sequences into the internal data structures in the appropriate format. In the abstract domain, there is a reference to a global data structure `globalData`. The structure contains the following fields:

- `initialised`: this is a **boolean** value used for determining whether the input file has already been processed.
- `atomicPairs`: a **set of pairs of strings** that stores pairs of functions that should be called atomically, for instance, {(`f1 f2`), (`f2 f3`)}.

The initialisation process is then done in the following way. The input file with atomic sequences is read and parsed. Each pair of functions that should be called atomically, i.e., any pair of functions in any of the atomic sequences from the input file, is stored into the field `atomicPairs` of the structure `globalData`. Single functions may also be stored into this structure when the atomic sequence contains just one function call. This structure is globally accessible throughout the analysis.

Abstract States Used for the Detection of Atomicity Violations

The abstract state is of the type `TSet`. `TSet` is a module representing a **set of structures**. The structure have the following fields:

- `firstCall`: the value of this field is a **string** that captures the first function call within an analysed function. It is used for detection of an atomicity violation due to a pair (a, b) of calls, where a is the last call of a function higher in the *call hierarchy* when calling the analysed function and b is the first function call of the analysed function.
- `lastPair`: the value of this field is a **pair of strings** that captures last two function calls. And it is used for detecting whether this pair violates atomicity. For instance, (`f1 f2`).
- `nastedLastCalls`: a **list of strings** that captures the possible last function calls of the last nested function. It is used for detection of atomicity violation of pair (a, b), where a is one of the last calls of the last nested function and b is the analysed function.
- `atomicityViolations`: a **set of pairs of strings** that is used to record pairs of function calls that violate atomicity so that the violations can be reported at the end of the analysis of a function, for instance, {(`f1 f2`), (`f2 f3`)}.
- `isInLock`: a **boolean** that determines whether the current state of a function is inside or outside an atomic block, i.e., whether it is or it is not under a lock.

The *initial abstract state* is then a set with a single empty element. The empty element is an element where `firstCall` is the empty string, `lastPair` is a pair of two empty strings, `nastedLastCalls` is the empty list, `atomicityViolations` is the empty set, and `isInLock` is `false`.

According to Listing 4.1, there are several functions working with the abstract state of **Phase 2**. The functions are described below (all of these functions modify all elements of the abstract state):

- `update_astate_on_function_call`: this function is invoked when any function (except a lock or an unlock) is called. When the state is not under a lock, the current pair of the last two function calls is updated, and it is checked whether this pair (or any pair created from `nastedLastCalls`) violates atomicity. A simplified implementation of the function is shown in Algorithm 4.4.
- `update_astate_on_lock`: this function is invoked when the locking function is called. It sets the flag indicating the start of an atomic sequence and it clears the stored last function calls.
- `update_astate_on_unlock`: this function is invoked when the unlocking function is called. It unsets the flag indicating the start of an atomic sequence and it clears the stored last function calls.

Algorithm 4.4: A simplified algorithm of updating the abstract state by the called function and a check for atomicity violations

```
Require: An initialised global data structure globalData with the field
         atomicPairs containing pairs of functions that should be called
         atomically
1  def update_astate_on_function_call(astate, f):
2      for e ∈ astate do
3          if ¬(e.isInLock) then
4              (x, y) ← e.lastPair;
5              e.lastPair ← (y, f);
6              if e.lastPair ∈ globalData.atomicPairs then
7                  e.atomicityViolations ← Add(e.atomicityViolations, e.lastPair);
8              end
9          end
10     end
11     return astate;
12 end
```

Function Summaries of the Domain for the Detection of Atomicity Violations

The summary is a **structure** containing the following fields:

- `firstCalls`: a **list of strings** that contains all possible first function calls within the analysed function.
- `lastCalls`: a **list of strings** that contains all possible last function calls within the analysed function.

Both of the summary fields are used for the purpose of detecting atomicity violating pairs across nested function calls.

In the code of the analysis shown in Listings 4.1 and 4.4, several functions designed for dealing with the summaries are used. In particular, the following functions are used there:

- `update_astate_on_function_call_with_summary`: this function is invoked when the called function has already been analysed. It performs checks for an atomicity violation of pairs across nested function calls.
- `convert_astate_to_summary`: this function is invoked at the end of the analysis of an analysed function. It transforms the abstract state of the given function to the summary. In particular, it derives all the first function calls and all the last function calls within the analysed function from the abstract state.

4.2.2 Reporting Atomicity Violations

As shown in Listing 4.4, at the end of the analysis of a function, atomicity violations within the function are reported. The reporting is implemented by the `Reporting` module provided by Facebook Infer. For reporting errors using this module, it is necessary to assign an error to the function summary along with a location of the error (a file and a line). It is also required to specify the type of the error through the `IssueType` module. The error is then printed to the command line as well as logged to logging files.

Reported atomicity violations are deduced from the abstract state of the analysed function. A simplified reporting process is illustrated in Algorithm 4.5.

Algorithm 4.5: Reporting atomicity violations from the abstract state of an analysed function

Input: The abstract state *astate* of an analysed function

```
1 for e ∈ astate do
2 |   for (a, b) ∈ e.atomicityViolations do
3 |   |   LogError('„%s" and „%s" should be called atomically.', a, b);
4 |   end
5 end
```

Chapter 5

Experimental Evaluation

This chapter is devoted to testing and an experimental evaluation of the *Atomer* analyser. Atomer has been tested continuously during its development. As soon as a part of the analyser has been implemented, it has been tested on suitable simple programs created for testing purposes. In the end, the whole analyser has been successfully tested on smaller, specifically created programs, as described in Section 5.1. Furthermore, Section 5.2 shows an experimental evaluation of the analyser on publicly available benchmarks derived from *real-life low-level* programs. Section 5.3 concludes the experimental evaluation and discusses future work.

All the experiments presented in this section are available on the attached memory media, see Appendix B. The way how to run the experiments is described in Appendix C.

5.1 Testing on Smaller Hand-Crafted Examples

Both phases of the analyser were verified separately on simple programs written in ANSI C with *PThread* locks.

For **Phase 1** of the analyser, i.e., the *detection of atomic sequences*, it was created several specifically designed functions. These functions contain sequences of function calls inside and outside *atomic blocks*, *not paired lock/unlock calls*, *iteration*, *selection*, and *nested function calls*. The functions were designed in order to check whether the various parts of the *abstract domain* work well (i.e., they correspond to the proposal from Chapter 3). In particular, the design aims at testing of the *ordering* operator, the *join* operator, and the *widening* operator. Further, it aims at verification of working with all components of the *abstract state* and the *summary*. It was checked that the result of the analysis of these functions (atomic sequences) is correct, with respect to the proposal from Chapter 3. The analysed functions and the result of **Phase 1** can be seen in Section A.1.

In the case of **Phase 2**, i.e., the *detection of atomicity violations*, the verification process is an analogy of the process for **Phase 1**. However, there is also checked that reported atomicity violations are reported correctly (based on the atomic sequences from **Phase 1**), with respect to the proposal from Section 3.2.2. The analysed function and the result of **Phase 2** can be seen in Section A.2.

For all the mentioned testing programs, it was proven the correct behaviour of the analysis of Atomer (with respect to the proposal).

5.2 Evaluation on Real-Life Programs

To find out whether the analyser is able to analyse *real programs*, at first, it was tested on a set of slightly more complex student projects from the *Advanced Operating Systems* course. These projects work with threads and synchronise using *PThreads*. From the beginning, the analysis of some of these student programs was failing due to implementation errors that were thanks to this testing fixed.

Later, Atomer was applied on a subset of *real-life low-level concurrent* C programs from a publicly available benchmark. These programs were derived from the Debian GNU Linux distribution. The entire benchmark was originally used for an experimental evaluation of Daniel Kroening's static deadlock analyser for C/PThreads [14] implemented in the CPROVER framework. For the evaluation, it was selected 9 *deadlock-free* programs. The experiments were run on MacBook Pro 2015 with a 2.7 GHz Intel Core i5 processor and 8 GB RAM running the macOS Mojave 10.14.4 operating system. However, the running time of the experiments is not relevant because all the experiments were done in less than a few seconds. The results of the experiments are listed in Table 5.1. The table states the number of lines of code of an analysed program, the number of detected atomic sequences, and the number of detected atomicity violations.

Program	Lines of Code	Atomic Sequences	**Atomicity Violations**
alsa-utils 1.1.0	7,735	1	**1**
c-icap 0.4.2	24,923	11	**174**
glfw 2.7.9	10,230	9	**13**
libgroove 4.3.0	7,307	34	**294**
npth 1.2	1,593	1	**26**
qrencode 3.4.4	7,006	6	**88**
rt-tests 0.96	1,795	1	**0**
signing-party 2.2	1,023	1	**1**
sslsplit 0.4.11	22,457	18	**344**

Table 5.1: Experimental results of the analyser on *real-life low-level* programs

As one can see in Table 5.1, in larger real-life programs, quite some atomicity violations were reported. Many of them are probably *false alarms*. But, a proper classification whether these reported atomicity violations are real errors is quite challenging and it goes beyond the scope of this thesis. However, the results of the analyser can be used as an input for *dynamic analysis*, which can be able to check whether the atomicity violations are real errors. For example, one could use the *ANaConDA*[1] dynamic analyser which uses *noise-based testing* with *extrapolated checking* for violations of contracts for concurrency. ANaConDA could be instructed to concentrate its analysis and *noise injection* to those sequences whose atomicity was found broken.

[1] **ANaConDA Framework** website – http://www.fit.vutbr.cz/research/groups/verifit/tools/anaconda.

5.3 Summary of the Evaluation and Future Work

The correctness of the analyser was successfully verified on smaller *hand-crafted* example programs. However, when analysing larger *real-life* programs, many *false alarms* are usually reported. To reduce the number of false alarms, it seems promising to work with *nested locks*, distinguish between the *different locks* used, or consider extensions for *contracts for concurrency* introduced in Section 2.3, i.e., consider *function parameters* and *contextual information* of function calls. The considerable number of false alarms is caused by atomic sequences that contain only one function call and by atomic sequences which do not have to be called atomically always. The solution could be ignoring *atomic blocks* where there is just one function call, and consider atomic sequences only if they appear in an atomic block more than, e.g., three times.

Another issue is that analysing of more complex programs with extensive *control structures* and a lot of function calls inside atomic blocks is time and memory consuming and the analysis either fails or it runs for an extremely long time. The solution could be to replace working with the **A** and **B** sequences in **Phase 1** of the analysis by working with sets. However, this speed gain would be received at the cost of precision.

Chapter 6

Conclusion

This thesis started by discussing principles of *static analysis* and *abstract interpretation*. Further, it described a concrete static analysis framework that uses abstract interpretation — *Facebook Infer* — its features, architecture, and existing analysers implemented in this tool. Furthermore, it introduced the concept of *contracts for concurrency*. The major part of the thesis then aimed at the proposal of a static analyser for detecting *atomicity violations* — *Atomer* — and its implementation as a module of Facebook Infer. Lastly, it is described the experimental evaluation of the implemented analyser and discussed possible future work.

The proposed analyser works on the level of *sequences of function calls*. The proposed solution is based on the assumption that sequences executed *atomically once* should probably be executed *always atomically*. It is also inspired by the concept of contracts for concurrency. Atomer is divided into two phases of the analysis: **Phase 1** — detection of function calls executed atomically; and **Phase 2** — detection of violations of the *atomic sequences* obtained from the first phase.

Atomer has been successfully tested on smaller *hand-crafted* programs. Moreover, it has been experimentally evaluated on publicly available benchmarks derived from *real-life low-level* programs from the Debian distribution. It has been found out that Atomer is able to analyse such extensive real-life programs, however, at the cost of quite high *false alarms* ratio. Anyway, a result of the analyser can be used as an input for *dynamic analysis* which can determine whether the reported atomicity violations are real errors.

Atomer shows a potential for further improvements. The future work will focus mainly on increasing the accuracy of the methods used by, e.g., considering *nested locks*, *different locks* used, *function parameters*, etc. The future work will also focus on enhancing the *scalability* because Atomer still fails to analyse more extensive and complex programs. Further, it would be interesting to extend the analysis for *other types of locks* for synchronisation of concurrent threads/processes and testing the analysis on other real-world programs.

The code of Atomer is available on GitHub as an *open-source repository*. The *Pull Request* to the `master` branch of Facebook Infer's repository is currently the work under progress. The preliminary results of the thesis were published and presented in the *Excel@FIT'19* paper [13], where the paper won an award in two categories.

Bibliography

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

[2] Blackshear, S.; Distefano, D.; Villard, J.: Building your own compositional static analyzer with Infer.AI [online]. PLDI'17. 2017 [cit. 2019-05-03]. Retrieved from: https://fbinfer.com/downloads/pldi17-infer-ai-tutorial.pdf

[3] Blackshear, S.; Gorogiannis, N.; O'Hearn, P. W.; Sergey, I.: RacerD: Compositional Static Race Detection. *Proceedings of ACM Programming Languages*. vol. 2, no. OOPSLA'18. October 2018: pp. 144:1 – 144:28. ISSN 2475-1421. doi:10.1145/3276514.

[4] Blackshear, S.; O'Hearn, P. W.: Open-sourcing RacerD: Fast static race detection at scale [online]. 2017-10-19 [cit. 2019-05-03]. Retrieved from: https://code.fb.com/android/open-sourcing-racerd-fast-static-race-detection-at-scale

[5] Calcagno, C.; Distefano, D.; O'Hearn, P. W.; 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*. POPL'09. Savannah, GA, USA: ACM, New York, NY, USA. January 2009. ISBN 978-1-60558-379-2. pp. 289 – 300. doi:10.1145/1480881.1480917.

[6] Cousot, P.: Abstract Interpretation in a Nutshell [online]. [cit. 2019-05-02]. Retrieved from: https://www.di.ens.fr/~cousot/AI/IntroAbsInt.html

[7] Cousot, P.: Abstract Interpretation [online]. 2008-08-05 [cit. 2019-05-02]. Retrieved from: https://www.di.ens.fr/~cousot/AI

[8] Cousot, P.: Abstract Interpretation Based Formal Methods and Future Challenges, invited paper. In *« Informatics — 10 Years Back, 10 Years Ahead »*, *Lecture Notes in Computer Science*, vol. 2000, edited by R. Wilhelm. Springer-Verlag. March 2001. pp. 138 – 156. doi:10.1007/3-540-44577-3_10.

[9] Cousot, P.; 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*. POPL'77. Los Angeles, California: ACM Press, New York, NY. 1977. pp. 238 – 252. doi:10.1145/512950.512973.

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

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

[12] Gorogiannis, N.; O'Hearn, P. W.; Sergey, I.: A True Positives Theorem for a Static Race Detector. *Proceedings of ACM Programming Languages*. vol. 3, no. POPL'19. January 2019: pp. 57:1 – 57:29. ISSN 2475-1421. doi:10.1145/3290370.

[13] Harmim, D.; Marin, V.; Pavela, O.: Scalable Static Analysis Using Facebook Infer. In *Excel@FIT*. Brno University of Technology, Faculty of Information Technology. 2019.

[14] Kroening, D.; Poetzl, D.; Schrammel, P.; Wachter, B.: Sound static deadlock analysis for C/Pthreads. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ASE'16. Singapore, Singapore: ACM, New York, NY, USA. September 2016. ISBN 978-1-4503-3845-5. pp. 379 – 390. doi:10.1145/2970276.2970309.

[15] Lengál, O.; Vojnar, T.: Abstract Interpretation. Lecture Notes in Formal Analysis and Verification. Brno University of Technology, Faculty of Information Technology. 2018.

[16] Marcin, V.: Static Analysis of Concurrency Problems in the Facebook Infer Tool. Project practice. Brno University of Technology, Faculty of Information Technology. 2018.

[17] Meyer, B.: Applying „Design by Contract". *Computer*. vol. 25, no. 10. October 1992: pp. 40 – 51. ISSN 0018-9162. doi:10.1109/2.161279.

[18] Minsky, Y.; Madhavapeddy, A.; Hickey, J.: *Real world OCaml*. Sebastopol, CA: O'Reilly Media. first edition. 2013. ISBN 144932391X.

[19] Møller, A.; Schwartzbach, I. M.: *Static Program Analysis*. Department of Computer Science, Aarhus University. October 2018.

[20] Nielson, F.; Nielson, R. H.; Hankin, C.: *Principles of Program Analysis*. Berlin: Springer-Verlag. 2005. ISBN 3-540-65410-0.

[21] Reps, T.; Horwitz, S.; Sagiv, M.: Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL'95. ACM, New York, NY, USA. 1995. ISBN 0-89791-692-1. pp. 49 – 61. doi:10.1145/199448.199462.

[22] Sharir, M.; Pnueli, A.: Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, edited by S. S. Muchnick; N. D. Jones. chapter 7. Prentice Hall Professional Technical Reference. 1981. ISBN 0137296819. pp. 189 – 211.

[23] Sousa, D. G.; Dias, R. J.; Ferreira, C.; Lourenço, J. M.: Preventing Atomicity Violations with Contracts. *CoRR*. vol. abs/1505.02951. 2015. 1505.02951.

[24] Vojnar, T.: Different Approaches to Formal Verification and Analysis. Lecture Notes in Formal Analysis and Verification. Brno University of Technology, Faculty of Information Technology. 2018.

[25] Vojnar, T.: Lattices and Fixpoints for Symbolic Model Checking. Lecture Notes in Formal Analysis and Verification. Brno University of Technology, Faculty of Information Technology. 2018.

[26] Yi, K.: Inferbo: Infer-based buffer overrun analyzer [online]. 2017-02-06 [cit. 2019-05-04].
Retrieved from:
https://research.fb.com/inferbo-infer-based-buffer-overrun-analyzer

Appendix A

Experimental Verification Results

This appendix illustrates results of experimental verification of the implemented analyser. The verification process and the results, in general, are in more detail discussed in Chapter 5. Section A.1 shows experimental results of **Phase 1** of the analyser, i.e., the *detection of atomic sequences*. Section A.2 then shows experimental results of **Phase 2**, i.e., the *detection of atomicity violations*.

Both sections demonstrate the analysis on programs written in ANSI C and assume *PThread* locks and the existence of an initialised global variable `lock` of a type `pthread_mutex_t`.

Moreover, the below experiments are available on the attached memory media, see Appendix B. Instructions how to run these experiments can be found in Appendix C.

A.1 Detection of Atomic Sequences

For the verification of the detection of atomic sequences are used functions defined in Listing A.1. The result of this detection, i.e., *sequences of function that should be called atomically*, is shown in Listing A.2.

```
1  void f1(void) {}
2  void f2(void) {}
3  void f3(void) {}
4  void f4(void) {}
5  void f5(void) {}
6  void ff(void) { f1(); f2(); }
7
8  void test1(void)
9  {
10     f1(); f1();
11
12     pthread_mutex_lock(&lock);
13     f1(); f1(); f2();
14     pthread_mutex_unlock(&lock);
15
16     f1(); f1();
```

```
17
18      pthread_mutex_lock(&lock);
19      f1(); f3();
20      pthread_mutex_unlock(&lock);
21
22      f1();
23
24      pthread_mutex_lock(&lock);
25      f1(); f3(); f3();
26      pthread_mutex_unlock(&lock);
27  }
28
29  void test2(void)
30  {
31      f1(); f1();
32
33      pthread_mutex_lock(&lock);
34      f1(); f1(); f2();
35      pthread_mutex_unlock(&lock);
36
37      f3(); f3();
38
39      pthread_mutex_lock(&lock);
40      f1(); f4(); f4();
41      pthread_mutex_unlock(&lock);
42  }
43
44  void test_only_lock(void)
45  {
46      pthread_mutex_lock(&lock);
47      f1();
48  }
49
50  void test_only_unlock(void)
51  {
52      f2();
53      pthread_mutex_unlock(&lock);
54  }
55
56  void test_iteration(void)
57  {
58      int c;
59      f1(); f2();
60      while (c > 0) { f3(); f5(); }
61
62      pthread_mutex_lock(&lock);
63      f1(); f2();
64      pthread_mutex_unlock(&lock);
```

```
65
66      pthread_mutex_lock(&lock);
67      while (c > 0) f3();
68      pthread_mutex_unlock(&lock);
69
70      f4();
71  }
72
73  void test_selection(void)
74  {
75      int c;
76      f1(); f2();
77      if (c > 0) { f3(); f5(); }
78      else
79      {
80          pthread_mutex_lock(&lock);
81          f1();
82          pthread_mutex_unlock(&lock);
83      }
84
85      pthread_mutex_lock(&lock);
86      f2();
87      if (c > 42) f3();
88      else if (c > 0) f4();
89      pthread_mutex_unlock(&lock);
90
91      f4();
92  }
93
94  void test_nested(void)
95  {
96      pthread_mutex_lock(&lock);
97      ff(); f3();
98      pthread_mutex_unlock(&lock);
99
100     pthread_mutex_lock(&lock);
101     f4(); f5(); ff();
102     pthread_mutex_unlock(&lock);
103 }
```

Listing A.1: Functions to be analysed for the detection of atomic sequences

```
f1:␣
f2:␣
f3:␣
f4:␣
f5:␣
ff:␣
test1:␣(f1␣f2)␣(f1␣f3)
```

```
test2:␣(f1␣f2)␣(f1␣f4)
test_only_lock:␣(f1)
test_only_unlock:␣
test_iteration:␣(f1␣f2)␣(f3)
test_selection:␣(f1)␣(f2)␣(f2␣f3)␣(f2␣f4)
test_nested:␣(ff␣f1␣f2␣f3)␣(f4␣f5␣ff␣f1␣f2)
```

Listing A.2: The result of **Phase 1** (atomic sequences) of the analysis of functions from Listing A.1

A.2 Detection of Atomicity Violations

For the verification of the detection of atomicity violations are used functions defined in Listing A.3. The result of the detection of atomic sequences is shown in Listing A.4. The result of the detection of atomicity violations, i.e., *functions that should be called atomically but they are not*, is stated in Listing A.3 using comments.

```
1  void f1(void) {}
2  void f2(void) {}
3  void f3(void) {}
4  void f4(void) {}
5  void g(void) {}
6  void ff(void) { f3(); f1(); f4(); } // (f3 f1) (f1 f4)
7
8  void atomic_sequences(void)
9  {
10     pthread_mutex_lock(&lock);
11     f1(); f2(); f3();
12     pthread_mutex_unlock(&lock);
13
14     pthread_mutex_lock(&lock);
15     f4(); f2();
16     pthread_mutex_unlock(&lock);
17
18     pthread_mutex_lock(&lock);
19     f1(); f3();
20     pthread_mutex_unlock(&lock);
21
22     pthread_mutex_lock(&lock);
23     ff(); f3();
24     pthread_mutex_unlock(&lock);
25 }
26
27 void test1(void)
28 {
29     f1(); f2(); g(); // (f1 f2)
30     f1(); g(); f2(); g();
```

```
31      f1(); f1(); f2(); g(); // (f1 f2)
32      f1(); f2(); f3(); g(); // (f1 f2) (f2 f3)
33      f1(); g(); f2(); g(); f3();
34  }
35
36  void test2(void)
37  {
38      f4(); f2(); g(); // (f4 f2)
39      f2(); f4(); g();
40
41      f4();
42      pthread_mutex_lock(&lock);
43      pthread_mutex_unlock(&lock);
44      f2();
45      g();
46
47      f3(); f4();
48  }
49
50  void test_only_lock(void)
51  {
52      pthread_mutex_lock(&lock);
53      f1(); f2();
54  }
55
56  void test_only_unlock(void)
57  {
58      f1(); f2(); // (f1 f2)
59      pthread_mutex_unlock(&lock);
60  }
61
62  void test_iteration(void)
63  {
64      int c;
65      while (c > 0) { f1(); f2(); } // (f1 f2)
66
67      f1();
68      while (c > 0) f2(); // (f1 f2)
69      f3(); // (f1 f3) (f2 f3)
70
71      for (; c > 0; f1()) f3(); // (f3 f1) (f1 f3)
72
73      pthread_mutex_lock(&lock);
74      f1(); f2();
75      while (c > 0) { f2(); f3(); }
76      pthread_mutex_unlock(&lock);
77  }
78
```

```
79  void test_selection(void)
80  {
81      int c;
82      f1();
83      if (c > 0) f2(); // (f1 f2)
84      else f3(); // (f1 f3)
85      f3(); // (f2 f3)
86
87      g();
88
89      f4();
90      if (c > 42) f4();
91      else if (c > 0) f2(); // (f4 f2)
92      f2(); // (f4 f2)
93  }
94
95  void test_nested(void)
96  {
97      pthread_mutex_lock(&lock);
98      ff();
99      pthread_mutex_unlock(&lock);
100
101     ff(); g(); // (ff f3)
102     ff(); f2(); // (ff f3) (f4 f2)
103 }
```

Listing A.3: Functions to be analysed for the detection of atomic sequences and for the subsequent detection of atomicity violations (detected atomicity violations are stated in comments)

```
f1:␣
f2:␣
f3:␣
f4:␣
g:␣
ff:␣
atomic_sequences:␣(f1␣f2␣f3)␣(f4␣f2)␣(f1␣f3)␣(ff␣f3␣f1␣f4)
test1:␣
test2:␣
test_only_lock:␣(f1␣f2)
test_only_unlock:␣
test_iteration:␣(f1␣f2)␣(f1␣f2␣f3)
test_selection:␣
test_nested:␣(ff␣f3␣f1␣f4)
```

Listing A.4: The result of **Phase 1** (atomic sequences) of the analysis of functions from Listing A.3

Appendix B

Contents of the Attached Memory Media

This appendix lists contents of the attached memory media.

The attached memory media particularly contains the following:

- `/experiments/`
  - The results of the experimental evaluation and experimental programs to analyse.
- `/infer/`
  - Source codes of Facebook Infer.
  - The created source files of the implemented analyser are located mainly in a subdirectory `infer/src/checkers`.
  - In this directory, there are `.md` files with the official instructions on how to install and build this tool. But such information is primarily provided in Appendix C.
- `/text/`
  - LaTeX source codes of this thesis.
  - It can be compiled into the PDF using `pdflatex` with the command `make`.
- `xharmi00.pdf`
  - This thesis in the PDF.

Appendix C

Installation and User Manual

This appendix serves as an installation and user manual.

Further, it is assumed that the *working directory* contains all the files from the attached memory media, see Appendix B.

Installation Manual

The whole installation process may be time-consuming.

At first, it is required to install Facebook Infer's dependencies and then to compile Facebook Infer. Here are the prerequisites to be able to compile Facebook Infer on Linux:

- opam >= 2.0.0
- python 2.7
- pkg-config
- gcc >= 5.X or clang >= 3.4
- autoconf >= 2.63
- automake >= 1.11.1

Installation of Facebook Infer can be done using the following commands:

```
cd infer
./build-infer.sh clang
sudo make install
```

The official installation manual can be found in https://github.com/harmim/infer/blob/master/INSTALL.md.

Furthermore, it is needed to build Facebook Infer using these commands:

```
cd infer
make -j BUILD_MODE=default
```

The official building manual can be found in https://github.com/harmim/infer/blob/master/CONTRIBUTING.md.

Facebook Infer now should be installed and executable by the command `infer`. It is installed in `infer/infer/bin`.

User Manual

In general, to analyse a C/C++ program with Facebook Infer, it can be done using the following command (for a single file):

```
infer run -- gcc -c sourc_file.c
```

Another option is to analyse the entire project with `Makefile` using the following:

```
infer run -- make <target>
```

Many other build systems may be used, see https://fbinfer.com/docs/analyzing-apps-or-projects.html.

The implemented **atomicity violations analyser** is deactivated by default. The analysis have to be executed twice (it has two phases). Each phase runs with a different command line option. The first phase derives sequences of functions that should be executed atomically into the file `infer-atomicity-out/atomic-sequences`. The second phase then detects atomicity violations according to this file. So, the analysis can be triggered using the following commands:

```
infer run --atomic-sequences-only -- gcc -c sourc_file.c
infer run --atomicity-violations-only -- gcc -c sourc_file.c
```

Or it can be triggered along with other analyses that Facebook Infer provides:

```
infer run --atomic-sequences -- gcc -c sourc_file.c
infer run --atomicity-violations -- gcc -c sourc_file.c
```