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

Ústav inteligentních systémů (UITS) Akademický rok 2018/2019

Zadání bakalářské práce

21689

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* of programs, which detects *atomicity violations.* The proposed analyser — *Atomer* — is implemented as an extension for *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 *once atomically* should probably be executed *always atomically.* The implemented analyser has been successfully verified and evaluated on both smaller programs created for this purpose as well as publicly available benchmarks derived from real-life low-level programs.

Abstrakt

Cílem této práce je navrhnout *statický analyzátor* programů, který bude sloužit pro detekci *porušení atomicity.* Navržený analyzátor — *Atomer* — je implementován jako rozšíření pro *Facebook Infer*, což je volně šířený a snadno rozšířitelný 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 *jednou zavolány atomicky*, by měly být pravděpodobně volány *atomicky vždy.* Implementovaný analyzátor byl úspěšně ověřen a vyhodnocen jak na malých programech, vytvořených pro tento účel, tak na veřejně dostupných testovacích programech, které vznikly ze skutečných nízko úrovňových 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

[[TODO]]

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 9, 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. Lastly, I thank for the support received from 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 there are many possible ways of catching bugs in the development process. Dynamic analysis tools or tools for automated testing are often used. These methods 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 its input data. An alternative solution is a *static analysis*. Of course, it has some shortages as well. The big issue is *scalability* on extensive codebases and considerable high rate of incorrectly reported errors (so-called *false positives*, also called *false alarms*).

Not long ago, Facebook introduced *Facebook Infer* – a tool for creating *highly scalable compositional*, *incremental*, and *interprocedural* static analysers. Facebook Infer is a live tool and it is still under the development. Anyway, it is in everyday use in Facebook itself, Spotify, Uber, Mozilla, WhatsApp and other well-known companies. Currently, Facebook Infer provides several analysers implemented as modules in the whole framework. These analysers check for various types of bugs, e.g., buffer overflows, thread-safety, null-dereferencing, or memory leaks. Facebook Infer also aims to create a framework for building new analysers quickly and easily. The current version of Facebook Infer still misses better support for *concurrency* bugs. While it provides a fairly advanced *data race* analyser, it is limited to Java programs only and fails for C programs, which require more through manipulation with locks.

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. Atomicity requirements, in most cases, are not even documented. It means that typically only programmers must take care of following these requirements. In general, it is very difficult to avoid errors in *atomicity-dependent programs*, especially in large projects, and even harder and time-consuming is finding and fixing these errors.

In this thesis, there is described proposal, implementation, and experimental verification and evaluation of *Atomer* — static analyser for finding atomicity violations — which is implemented as an extension for Facebook Infer. In particular, the concentration is put on an *atomic execution of sequences of function calls*, which is often required, e.g., when using certain library calls. The implementation targets to C/C++ programs that use *PThreads* 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 paper are taken over [14], which I wrote together with Vladimír Marcin and Ondřej Pavela. In [14], there were presented preliminary results of my thesis.

The rest of the paper is organised as follows. In Chapter 2, there are described all the topics which are necessary to understand before reading the rest of the paper. In particular, Section 2.1 deals with a *static analysis* based on *abstract interpretation*. Facebook Infer, which uses abstract interpretation, is described in Section 2.2. And 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. Its implementation is in Chapter 4 and experimental results are presented in Chapter 5. Finally, Chapter 6 concludes the paper. Appendix A lists contents of attached memory media and Appendix B serves as an installation and user manual.

Chapter 2

Preliminaries

This chapter explains the theoretical background on which stands the thesis. It also explains and describes the existing tools used in the thesis. Lastly, the chapter deals with principles which this thesis got inspired by.

The aim of this thesis is to propose a *static analyser* and implement it in *Facebook Infer*. So, in Section 2.1, there is a brief explanation of a *static analysis* itself, and then an explanation of *abstract interpretation* that is used in Facebook Infer. Facebook Infer, its principles and features illustrate Section 2.2. A proposal of a solution is based on the concept of *contracts for concurrency*, which is discussed and defined in Section 2.3.

2.1 Static Analysis by Abstract Interpretation

According to [19], a *static analysis* of programs is reasoning about the behaviour of computer programs without actually executing them. It has been used since the 1970s for optimising compilers for generating effective code. More recently, it has proven valuable also for automatic error detection, verification tools 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 checks if the *program semantics* of a given program fulfils the given *specification*, as illustrates Figure 2.1 [8]. Nowadays, a 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 variable 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?

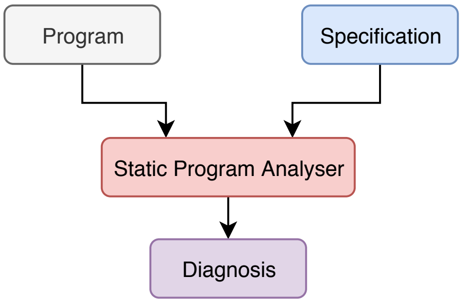


Figure 2.1: A static program analysis [8]

It is well-known that testing, i.e., executing programs with some input data and examining the output, may expose errors, but it can not 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, a 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 a 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 significant issue is how to ensure high precision and *scalability* to be useful in practice. The biggest disadvantage is that a static analysis can produce many *false alarms*[1], but it is often resolved by accepting *unsoundness*[2].

Various forms of a static analysis of programs have been invented, for instance [24]: bug pattern searching, data-flow analysis, constraint-based analysis, type analysis, symbolic execution. And one of the essential concept — *abstract interpretation* — is detailed in Section 2.1.1.

There exist numerous tools for a static analysis (often proprietary and difficult to openly evaluate or extend), e.g.: Coverity, Klockwork, CodeSonar, Loopus, 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 bibliographies, there also can be found more detailed, more formal, and a more theoretical explanation.

The abstract interpretation was introduced and formalised by a French computer scientist Patrick Cousot and his wife Radhia Cousot in the year 1977 at POPL[3] [9]. It is a generic *framework* for static analyses. It is possible to create particular analyses by providing

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

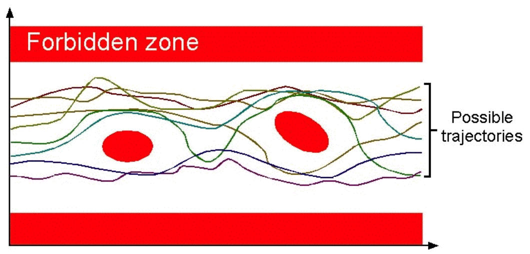
[2] **Soundness** – if a verification method claims that a system is correct according to a given specification, it is truly correct. [24]

[3] **POPL** – symposium on Principles of Programming Languages.

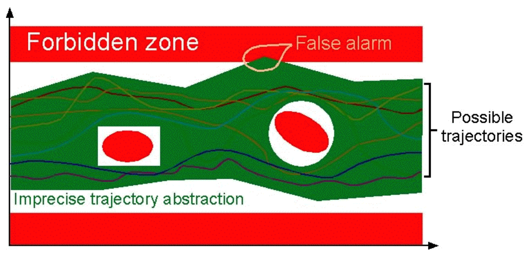
specific components (described later) to the framework. The analysis is guaranteed to be *sound* if certain properties of the components are met. [15], [16]

In general, in the set theory, which is independent on an application setting, abstract interpretation is considered 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), that 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* (defined below), in order to check a given *program specification* using analysation of the abstract semantics. [8]

Patrick Cousot intuitively and informally illustrates abstract interpretation in [6] as follows. Figure 2.2a 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, is undecidable because the program concrete semantics is not computable. As demonstrates Figure 2.2b, 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. does not intersect the forbidden zone), concrete semantics is safe as well. However, the *over-approximation* of the possible program executions causes that inexisting program executions are considered, that 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 2.2: Abstract interpretation demonstration [6]. Horizontal axes: time t. Vertical axes: vector $x(t)$ of input values of programs

Components of Abstract Interpretation

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

- **Abstract Domain** [7]

- An abstraction of the *concrete semantics* in the form of *abstract properties*[4] and *abstract operations*[5]. [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.

- **Join Operator** $\circ$
  - Joins abstract states from individual program branches into a single one.

- **Widening Operator** $\triangledown$ [20], [10], [15]
  - Enforces termination of the abstract interpretation.
  - It is used to approximate the *least fixed points* (it is performed on a sequence of abstract states at a certain location).
  - The later in the analysis is this operator used, the more accurate is the result (but the analysis takes more time).

- **Narrowing Operator** $\triangle$ [20], [10], [15]
  - Encapsulates a termination criterion.
  - Using this operator, the approximation 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

Definition 2.1.1. In [25], there is a *fixpoint* defined as:

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

Computation of the *most precise abstract fixpoint* is not generally guaranteed to terminate in certain cases, such as loops. The solution is to approximate the fixpoint using *widening* (over-approximation of a fixpoint) and *narrowing* (improves an approximation of a fixpoint) [15], [16]. Most program properties can be represented as fixpoints. This reduces a program analysis to the fixpoint approximation [7]. Further information about fixpoint approximation can be found in [20], [10].

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

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

Formal Definition of Abstract Interpretation

Definition 2.1.2. According to [9], [15], **abstract interpretation $\boldsymbol{I}$** of a program P with the instruction set S is a tuple

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

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 the *abstract transformers* for specific instructions,
- $(Q, \circ, \top)$ is a *complete semilattice* [25], [15].

Using so-called *Galois connections* ([20], [10], [15], [7]) can be guaranteed the *soundness* of abstract interpretation.

2.2 Facebook Infer – Static Analysis Framework

This section describes the principles and features of *Facebook Infer*. The description is based on information provided on Facebook Infer website[6] and in [2], [16]. Parts of this section are taken over [14].

Facebook Infer is an open-source static analysis *framework*, which is able to discover various kinds of software bugs of a given program, and the stress is put on *scalability*. Elementary essence of this framework shows Figure 2.3, below is a more detailed explanation of its architecture. Facebook Infer itself 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 has made its breakthrough thanks to a powerful paper [5].

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 another languages. Currently, Facebook Infer contains many analyses focusing on amount sorts of bugs, e.g., *Inferbo* (buffer overruns) [26]; *RacerD* (data races) [3], [4], [13]; *RacerX* (race conditions and deadlocks) [12]; and other analyses checks for buffer overflows, thread-safety, null-dereferencing, memory leaks, resource leaks, etc.

[6]**Facebook Infer** website – https://fbinfer.com.
[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.

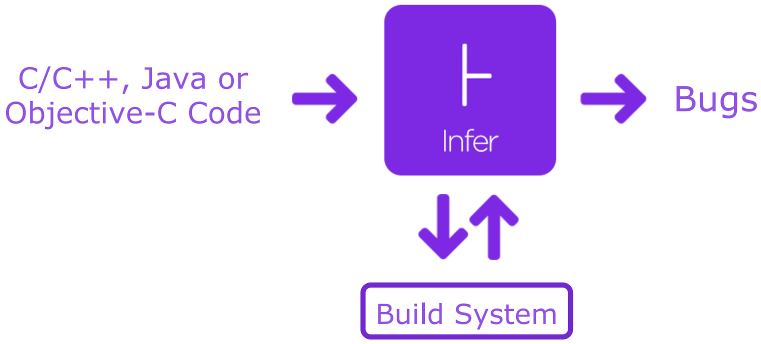


Figure 2.3: A static analysis in Facebook Infer (http://www.codeandyou.com/2015/06/infer-static-analyzer-for-java-c-and.html)

2.2.1 Abstract Interpretation in Facebook Infer

Facebook Infer is a general framework for a static analysis of programs, it is based on *abstract interpretation*, see Section 2.1.1. It 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 structure that may be used for storing any information resulting from the analysis of single functions. Facebook Infer generally does not work out 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. Since summaries worked out in different contexts are equal, this principle makes the analysis more scalable, but it can lead to a loss of accuracy. Then, the summary of a function is used at all of its call sites. In order to create new intraprocedural analyser in Facebook Infer, it is needed to define (listed items are described in more detail in Section 2.1.1):

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

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

1. The type of function summaries.

[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]

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

The next important feature improving the scalability is *incrementality* of the analysis, it allows 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 demonstrates Figure 2.4: a *frontend*, an *analysis scheduler* (and a *results database*), and a set of *analyser plugins*.

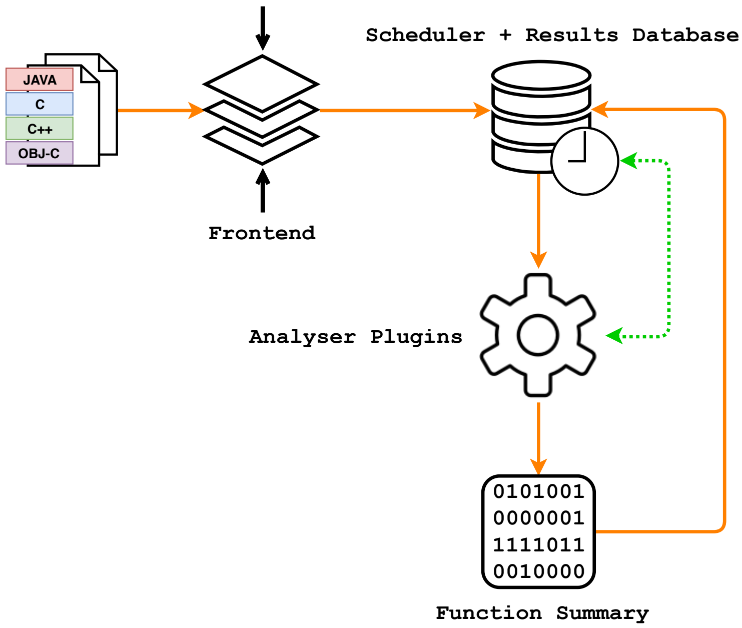


Figure 2.4: The architecture of Facebook Infer's abstract interpretation framework [2], [16]

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

1. `LOAD` – reading into a temporary variable.
2. `STORE` – writing to a program variable, a field of a structure, or an array.
3. `PRUNE e` (often called `ASSUME`) – a condition `e`.
4. `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.

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.

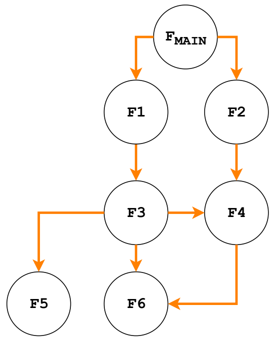


Figure 2.5: A call graph for an illustration of Facebook Infer's analysis process [2], [14], [16]

Example 2.2.1. For demonstrating the order of the analysis in Facebook Infer and its incrementality, assume a call graph in Figure 2.5. At first, leaf functions `F5` and `F6` are analysed. Further, the analysis goes on towards the root of the call graph – $\mathtt{F}_{\mathtt{MAIN}}$, while takes into consideration the dependencies denotes by the edges. This order ensures that a summary is available once a nested function call is abstractly interpreted within the analysis. When there is a subsequent code change, only directly changed functions and all the functions up the call path are re-analysed. For instance, if there is a change of source code of function `F4`, Facebook Infer triggers re-analysation of functions `F4`, `F2`, and $\mathtt{F}_{\mathtt{MAIN}}$ only.

The last part of the architecture consists of the set of analyser plugins. Each plugin performs the analysis by interpretation of SIL instructions. Result of the analysis of each function (function summary) is stored to the results database. 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 input, and by applying the interpreting command produce a new abstract state. Then, the abstract interpreter interprets the command in *abstract domain* according to the CFG. This workflow is simplified in Figure 2.6.

2.3 Contracts for Concurrency

This section introduces and defines the concept of *contracts for concurrency* described in [23], [11]. Parts of this section are taken over [14]. Listings in this section are pieces of programs written in ANSI C[12].

Respecting the *protocol* of a software module — delineates 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 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]

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

[12] **ANSI C** – standard for the C programming language published by the *ANSI* (American National Standards Institute).

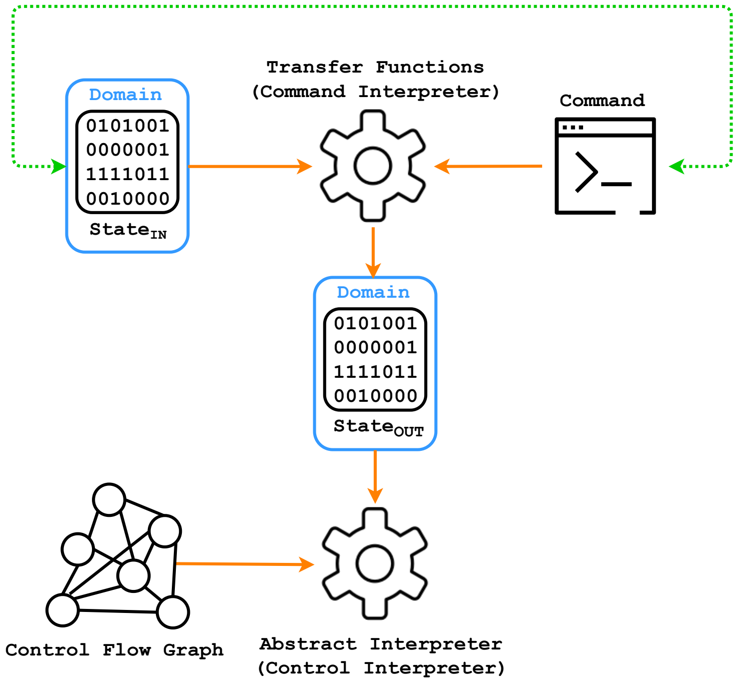


Figure 2.6: Facebook Infer's abstract interpretation process [2], [16]

In *concurrent programs*, contracts for concurrency allow one to specify *sequences of functions* that are needed to be *executed atomically*, in order to avoid *atomicity violations*. Such contracts may be manually specified by a developer or it 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. A program is safe from atomicity violations if the program follows the contract and the contract is well-defined and complete.

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 (i.e., a sequence of function calls must be atomic only if they handle the same data). Above that, paper [11] extends the idea of basic contracts with *spoilers* (i.e., extending by *contextual information*).

2.3.1 Basic Contracts

Definition 2.3.1. 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*[13] 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]. There is a module with the implementation of a resizable array with the listed functions:

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

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)`

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 concurrently, e.g., removed.

(ϱ_2) `index_of (get | set | remove)`

> The execution of `index_of` follow by the execution of `get`, `set`, or `remove` should be atomic. Otherwise, the received index may be outdated when it is applied to address 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 does not have to longer exist or its position in an array can be changed, when the program attempts to obtain information about it.

The above definition of contracts for concurrency is quite limited in some circumstances and can consider valid concurrent programs as erroneous (reports *false alarms*). Hence, in Section 2.3.2, there is defined an extension of contracts for concurrency with *parameters*, which takes into consideration the data flow within function calls. And in [11], [23], there is defined another extension with *spoilers*, which considering contextual information of function calls.

2.3.2 Contracts with Parameters

Example 2.3.2. Consider the following example from [11], [23], as demonstrates Listing 2.1. There is a function `replace` that replaces item `a` in an array by item `b`. 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 defined in Section 2.3.1 could cover this scenario by clause ϱ_5:

(ϱ_5) `contains index_of set`

Nevertheless, it is too restrictive because it is 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 function `set`.

```c
void replace(char *array, char a, char b)
{
    if (contains(array, a))
    {
        int index = index_of(array, a);
        set(array, index, b);
    }
}
```

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 further 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 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`):

(ϱ_5') `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

Proposal of Static Analyser for Detecting Atomicity Violations

This chapter describes a proposal of a static program analyser for detection of *atomicity violations*. The proposed analyser — **Atomer** — has been proposed as an extension for *Facebook Infer*, introduced in Section 2.2. In particular, the proposal concentrates on an *atomic execution of sequences of function calls*, which is often required. The proposed principle is based on the assumption that sequences executed *once atomically* should probably be executed *always atomically*. The chapter also assessments already existing solutions in this area.

At first, Section 3.1 deals with existing approaches and tools for finding atomicity violations, theirs advantages, disadvantages, features, availability, and so on. Then, the proposal itself is introduced in Section 3.2. Parts of this chapter are taken over [14]. Listings in this chapter are pieces of exemplary programs written in ANSI C (assume *PThread* locks and the existence of an initialised global variable `lock` of the type `pthread_mutex_t`).

3.1 Assessment of Existing Analysers for Atomicity-Related Errors

The proposed solution is slightly inspired by ideas from [11], [23]. In these papers, there is described a proposal and an implementation of a *static validation* for finding *atomicity violations*, which is based on *grammars* and *parsing trees*. In paper [11], there is also described and implemented a dynamic approach of this validation. The authors of [11] and [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] and [23] is no more developed. That is why was mad the decision to get inspired by [11] and [23], and reimplement the analysis in *Facebook Infer* redesigning it in accordance with the principles of Facebook Infer (described in Section 2.2), which should make it more scalable. In the end, due to adapting the analysis for the context of Facebook Infer, implementation of the analysis within this thesis is significantly different

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

from [11] and [23], as it is presented in Chapter 4. Furthermore, unlike [11] and [23], the implementation aims at programs written in C/C++ languages using *POSIX Threads (PThreads)* locks for *synchronisation of concurrent threads*.

In Facebook Infer, there is already implemented an analysis called *Lock Consistency Violation*[2]. It is part of *RacerD* [3], [4], [13]. 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.

3.2 Proposal of the Analyser

The proposal of the analyser is based on the concept of *contracts for concurrency* described in Section 2.3. In particular, the proposal considers only *basic contracts* described in Section 2.3.1. *Parameters* of functions and theirs *return values* — expressed in *contracts with parameters* (see Section 2.3.2) — are not taken into consideration.

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 operations are done statically. The proposed analysis is divided into two parts (*phases of the analysis*):

Phase 1: Detection of *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 illustrates Figure 3.1.

3.2.1 Phase 1: Detection of Atomic Sequences

Before the detection of *atomicity violations* (Section 3.2.2) 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.

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: **A** is a *reduced sequence* of function calls that appear between the beginning of the function being analysed and the first lock or between an unlock and a subsequent lock (or between an unlock and the end of the function being analysed), and **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). Here, by

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

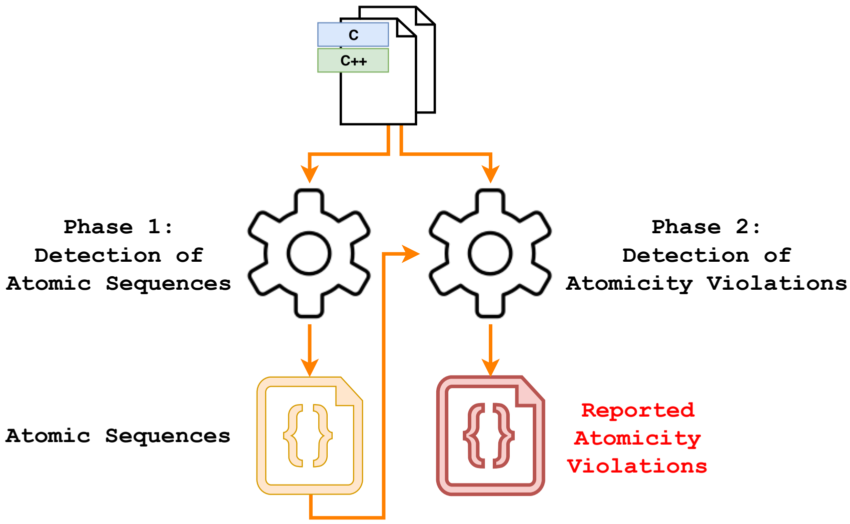


Figure 3.1: Phases of the proposed analyser

a reduced sequence it is meant a sequence in which the first appearance of each function is recorded only. The reason is to ensure *finiteness* of the sequences and of the analysis. The *summary* of a function then consists of:

(i) the set of all the **B** sequences and

(ii) the set of *concatenations* of all the **A** and **B** sequences with removal of duplicate function calls.

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

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

The parentheses are used to indicate an atomic sequence. The strikethrough of the functions `f1` and `f3` denotes a 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 derivated sets 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., concatenation of $\mathbf{A_1}$, $\mathbf{B_1}$, $\mathbf{A_2}$, and $\mathbf{B_2}$ with removal of duplicate function calls.

Analysing Functions Using Results of the Analysis of Nested Functions

Further, it is demonstrated how results of the analysis of *nasted functions* are used durning the detection of atomic sequences. The result of the analysis of a nasted function is used

```c
void g(void)
{
    f1(); f1();

    pthread_mutex_lock(&lock);
    f1(); f1(); f2();
    pthread_mutex_unlock(&lock);

    f1(); f1();

    pthread_mutex_lock(&lock);
    f1(); f3();
    pthread_mutex_unlock(&lock);

    f1();

    pthread_mutex_lock(&lock);
    f1(); f3(); f3();
    pthread_mutex_unlock(&lock);
}
```

Listing 3.1: An example of a code for an illustration of the derivation of sequences of functions called atomically

as follows. When calling an already analysed function, one plugs all the sequences from the second component of its summary into the current **A** or **B** sequence.

Example 3.2.2. 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 derivated sets for the function `h` are then as follows:

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

(ii) {f1 g f2 f3}.

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 demonstrates Listing 3.3 — two solutions have been proposed:

1. At the end of a function, everything is unlocked, i.e., append 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 the function `y` from Listing 3.3 would be an empty set.

```c
void h(void)
{
    f1(); g();

    pthread_mutex_lock(&lock);
    g();
    pthread_mutex_unlock(&lock);
}
```

Listing 3.2: An example of a code for an illustration of the derivation of sequences of functions called atomically with nested function call (function `g` is defined in Listing 3.1)

2. Addition of two further items to the summaries:
   (a) function calls with missing an unlock call,
   (b) function calls with missing a lock call.

   For the example from Listing 3.3, this would give:

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

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

```c
void x(void)
{
    pthread_mutex_lock(&lock);
    f1();
}

void y(void)
{
    f2();
    pthread_mutex_unlock(&lock);
}

void main(void)
{
    x(); y();
}
```

Listing 3.3: An example of a code for an illustration of treating cases where lock/unlock calls are not paired in a function

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

Summary of 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 this purpose. The verification is presented in Section 5.1. 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, we do not distinguish between the locks at all); consider contracts for concurrency with parameters defined in Section 2.3.2 or other extension 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** sequence 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** (see Section 3.2.1), 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.

Example 3.2.3. For example, the result of the first phase is the following set of functions called atomically:

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

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 the analogical way as it is handled in **Phase 1** described in Section 3.2.1. For detailed examples see verification experiments in Section 5.2.

Example 3.2.4. For instance, 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 `a` and `b` are not called atomically (under a lock).

Summary of Detection of Atomicity Violations and Future Work

As well as first phase of the analysis, **Phase 2** has been implemented, as it is described in Section 4.2. And it has been also successfully verified on a set of sample purposeful

```
void a(void)
{
    f1();

    pthread_mutex_lock(&lock);
    f2(); f3();
    pthread_mutex_unlock(&lock);

    f4();
}

void b(void)
{
    f1(); f2(); f3(); f4();
}
```

Listing 3.4: Example of atomicity violation

programs. This verification is described in Section 5.2. Experimental evaluation of the analysis as a whole analyser is then depicts in Section 5.3. **Phase 2** also has a potential for further enhancing. It is possible to extend this phase with all the improvements discussed in Section 3.2.1. Moreover, it is able to improve this phase by working with sets instead of pair, when looking for functions that should be called atomically. 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 certainty that this sequence should be called atomically.

Chapter 4

Implementation of the Analyser in Facebook Infer

[[TODO]]

4.1 Implementation of Detection of Atomic Sequences

[[TODO]]

4.2 Implementation of Detection of Atomicity Violations

[[TODO]]

Chapter 5

Experimental Verification and Evaluation of the Analyser

[[TODO]]

5.1 Experimental Verification of Detection of Atomic Sequences

[[TODO]]

5.2 Experimental Verification of Detection of Atomicity Violations

[[TODO]]

5.3 Experimental Evaluation of the Analyser

[[TODO]]

Chapter 6

Conclusion

[[TODO]]

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 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. 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. New York, NY, USA: ACM. 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)*. Tokyo, Japan: IEEE. March 2017. ISBN 9781509060313. pp. 196 – 206. doi:10.1109/ICST.2017.25.

[12] Engler, D. R.; Ashcraft, K.: RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. SOSP'03. New York, NY, USA: ACM. 2003. ISBN 1-58113-757-5. pp. 237 – 252. doi:10.1145/945445.945468.

[13] 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. January 2019: pp. 57:1 – 57:29. ISSN 2475-1421. doi:10.1145/3290370.

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

[15] Lengál, O.; Vojnar, T.: Abstract Interpretation. In *Formal Analysis and Verification*. Brno University of Technology, Faculty of Information Technology. 2018. lecture notes.

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

[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. New York, NY, USA: ACM. 1995. ISBN 0-89791-692-1. pp. 49 – 61. doi:10.1145/199448.199462.

[22] Sharir, M.; Pnueli, A.: *Two approaches to interproceduraldata flow analysis*. chapter 7. Prentice Hall Professional Technical Reference. 1981. ISBN 0137296819. pp. 189 – 211. eds., Muchnick, Steven S. and Jones, Neil D.

[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. In *Formal Analysis and Verification*. Brno University of Technology, Faculty of Information Technology. 2018. lecture notes.

[25] Vojnar, T.: Lattices and Fixpoints for Symbolic Model Checking. In *Formal Analysis and Verification*. Brno University of Technology, Faculty of Information Technology. 2018. lecture notes.

[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

Contents of Attached Memory Media

[[**TODO**]]

Appendix B

Installation and User Manual

[[TODO]]