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

Klíčová slova

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

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.

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 2, 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 [6], which I wrote together with Vladimír Marcin and Ondřej Pavela. In [6], there was 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*. Proposal of a static analyser for detection *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 existing solutions and 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. The 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 [9], 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 [3]. 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. [9]:

- **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?

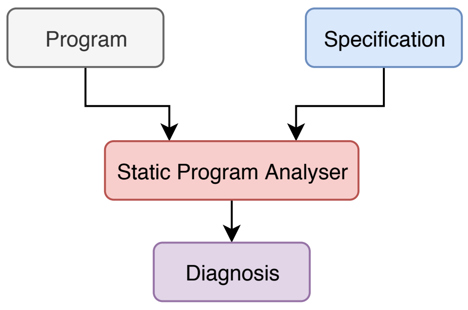


Figure 2.1: Static program analysis (inspired by [3])

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 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 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 [11]: 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 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 [3], [4], [1], [2], [7], [8], [5], [10], [9], [12]. 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] [4]. 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*.

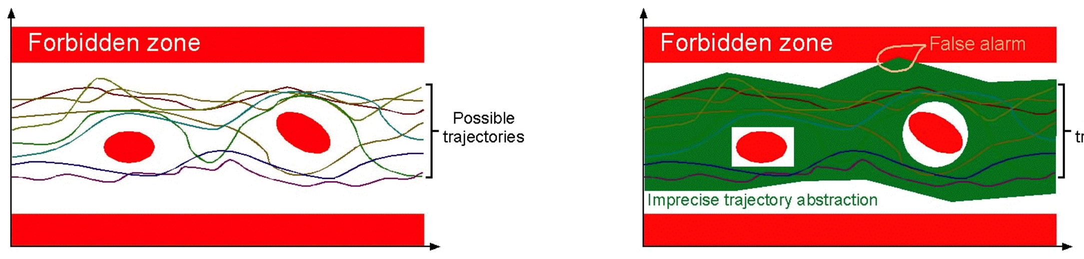
[2] *Soundness* – if a verification method claims that a system is correct according to a given specification, it is truly correct. [11]

[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. [7], [8]

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* [10]. 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. [3]

Patrick Cousot intuitively and informally illustrates abstract interpretation in [1] 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 [1]. Horizontal axes: the time t. Vertical axes: the vector $x(t)$ of the input values of programs

Components of Abstract Interpretation

In accordance with [7], [8], basic components of abstract interpretation are as follows:

- **Abstract Domain** [2]

  - An abstraction of the *concrete semantics* in the form of *abstract properties*[4] and *abstract operations*[5]. [3]
  - 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** $\triangledown$ [10], [5], [7]
  - 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** $\triangle$ [10], [5], [7]
  - 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

In [12], there is a *fixpoint* defined as:

- let $(A, \leq_A)$ be a *lattice* [12],
- an element $a \in A$ is a **fixpoint** of a function $f : A \rightarrow A$ 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) [7], [8]. Most program properties can be represented as fixpoints. This reduces program analysis to the fixpoint approximation [2]. Further information about fixpoint approximation can be found in [10], [5].

[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

According to [4], [7], **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 the *abstract transformers* for specific instructions,
- $(Q, \circ, \top)$ is a *complete semilattice* [12], [7].

Using so-called *Galois connections* [10], [5], [7], [2] can be guaranteed the *soundness* of abstract interpretation.

2.2 Facebook Infer – Static Analysis Framework

2.3 Contracts for Concurrency

Chapter 3

Proposal of Static Analyser for Detecting Atomicity Violations

Chapter 4

Implementation of the Analyser in Facebook Infer

Chapter 5

Experimental Verification and Evaluation of the Analyser

Chapter 6

Conclusion

Bibliography

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

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

[3] 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. 2001. pp. 138 – 156.

[4] 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*. Los Angeles, California: ACM Press, New York, NY. 1977. pp. 238 – 252.

[5] 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. 1992. pp. 269 – 295.

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

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

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

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

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

[11] 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.

[12] 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.

Appendix A

Contents of Attached Memory Media

Appendix B

Installation and User Manual