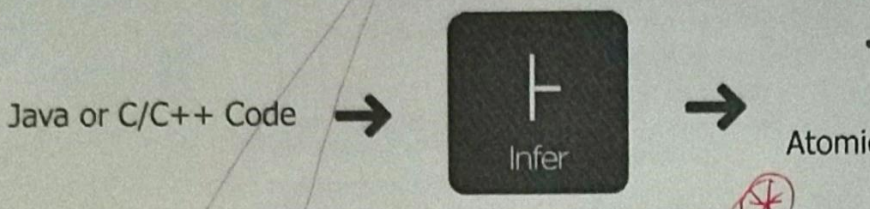
BRNO FACULTY OF INFORMATION TECHNOLOGY UNIVERSITY OF TECHNOLOGY

Project Practice 2019/2020

https://fit.vut.cz

Static Analysis in Facebook Infer Focused on Atomicity

Dominik Harmim*



Abstract

The goal of this project practice is to improve, extend, and perform new experiments with *Atomer* – a *static analyser* that detects *atomicity violations*. Atomer was proposed and implemented within a bachelor's thesis at FIT BUT as a module of *Facebook Infer*, which is an open-source and extendable static analysis framework that promotes efficient *modular* and *incremental* analysis. The original analyser works on the level of *sequences of function calls*, but in this project, it was changed to work on the level of *sets of function calls*. The solution is based on the assumption that sequences executed *atomically once* should probably be executed *always atomically*. Within this project, two new main features were introduced: (i) support for *C++ and Java locks*, (ii) and distinguish *multiple locks used*. The new features were successfully verified and evaluated on smaller programs created for testing purposes. Furthermore, new experiments on *publicly available real-life extensive programs* were made.

Keywords: Facebook Infer — Static Analysis — Abstract Interpretation — Contracts for Concurrency — Atomicity Violation — Concurrent Programs — Program Analysis — Atomic Sets — Atomicity — Incremental Analysis — Modular Analysis — Compositional Analysis — Interprocedural Analysis

Supplementary Material: Atomer Repository: `https://github.com/harmim/infer` — Atomer Wiki: `https://github.com/harmim/infer/wiki` — VeriFIT Static Analysis Plugins: `https://www.fit.vutbr.cz/research/groups/verifit/tools/sa-plugins` — Facebook Infer Repository: `https://github.com/facebook/infer`

*xharmi00@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

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 can analyse only particular program flows dependent on the input data. An alternative solution is *static analysis* that has its shortcomings as well, such as the *scalability* on large 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. However, most importantly, Facebook Infer is a framework for building new analysers quickly and easily. Unfortunately, the current version of Facebook Infer still lacks better support for *concurrency* bugs. While it provides a reasonably advanced *data race* analyser, it is limited to Java and C++ programs only and fails for C programs, which use a 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. Furthermore, in general, it is difficult to avoid errors in *atomicity-dependent programs*, especially in large projects, and even more laborious and time-consuming is finding and fixing them.

In the thesis [12] there was proposed the *Atomer* – a *static analyser* for finding some forms of atomicity violations 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 specific library calls. The idea of checking atomicity of certain sequences of function calls is inspired by the work of *contracts for concurrency* [10]. In the terminology of [10], atomicity of specific sequences of calls is the most straightforward (yet very useful in practice) kind of contracts for concurrency. The implementation mainly targets C/C++ programs that use *PThread* locks. Within this project practice, Atomer was improved, extended, and other experiments were performed. In particular, two new main features were introduced: (i) support for *C++ and Java locks*, (ii) and distinguish *multiple locks used*. Moreover, working with *sequences of function calls* was approximated by working with *sets of function calls* to make the solution more *scalable*.

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 from the thesis [12] and the paper [13] written in collaboration with Vladimír Marcin and Ondřej Pavela.

The rest of the paper is organised as follows. In Section 2, there is described Facebook Infer framework. Atomer is described in Section 3, together with all the extensions and improvements implemented within this project practice. Subsequently, Section 4 discusses the experimental evaluation of the new features and other experiments that were performed in this project. Finally, Section 5 concludes the paper.

2. Facebook Infer

This section describes the principles and features of *Facebook Infer*. The description is based on information provided at the Facebook Infer's website[1] and in [16]. Parts of this section are taken from [12, 13].

Facebook Infer is an *open-source*[2] *static analysis framework* (Brief explanation of static analysis itself can be found in [12] – Section 2.1). In more detail, it is then explained in [21, 26], which can discover various kinds of software bugs of a given program, with the stress put on *scalability*. A more detailed explanation of the architecture of Facebook Infer is shown in Section 2.2. Facebook Infer is implemented in *OCaml*[3] – *functional* programming language, also supporting *imperative* and *object-oriented* paradigms. Further details about OCaml can be found in [18]. 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 [4].

Facebook Infer can 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) [28]; *RacerD* (data races) [2, 3, 11]; and other analyses that check for buffer overflows, some forms of deadlocks and starvation, null-dereferencing, memory leaks, resource leaks, etc.

[1] **Facebook Infer's** website – `https://fbinfer.com`.

[2] **Open-source repository of Facebook Infer** on GitHub – `https://github.com/facebook/infer`.

[3] **OCaml's** website – `https://ocaml.org`.

2.1 Abstract Interpretation in Facebook Infer

Facebook Infer is a general framework for static analysis of programs, and it is based on *abstract interpretation* (Abstract interpretation is explained and formally defined in [12] – Section 2.1.1. Additional description can be found in [7, 8, 6, 5, 15, 16, 9, 22, 21, 27]). Despite the original approach taken from [4], 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* [22]) 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 particular functions. Facebook Infer generally does not compute the summaries in the course of the analysis along the *Control Flow Graph* (**CFG** [1]) as it is done in classical analyses based on the concepts from [23, 24]. Instead, Facebook Infer performs the analysis of a program *function-by-function along the call tree*, starting from its leafs (demonstrated in Example 2.1). 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 [12] – 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 ∇, 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 define additionally:

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.

2.2 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 1: a *frontend*, an *analysis scheduler* (and a *results database*), and a set of *analyser plugins*.

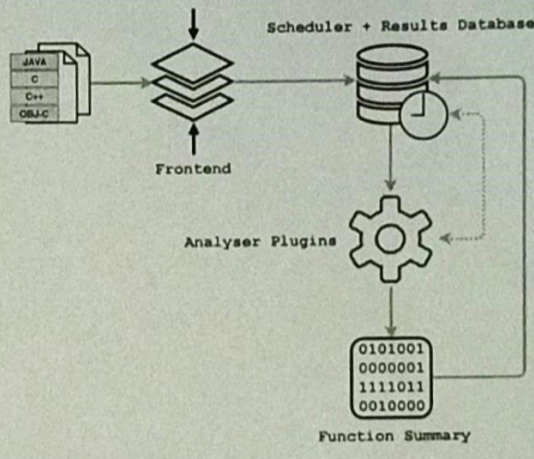


Figure 1. The architecture of Facebook Infer's abstract interpretation framework [12, 16]

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:

- `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 analyses that are *language-independent* (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 func-

tions according to the appropriate *call graph*[4]. The scheduler also checks if it is possible to analyse some functions simultaneously, which allows Facebook Infer to run the analysis in parallel.

Example 2.1. For demonstrating the order of the analysis in Facebook Infer and its incrementality, assume a call graph in Figure 2. At first, leaf functions F5 and F6 are analysed. Further, the analysis goes on towards the root of the call graph – F_{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 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 F_{MAIN} only.

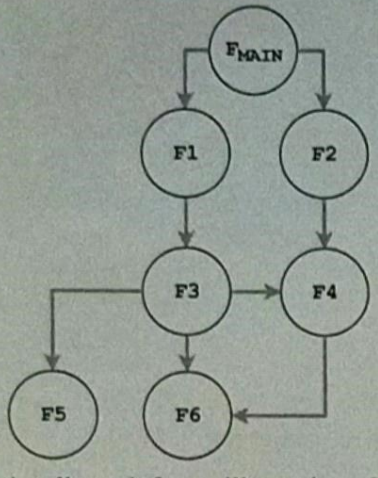


Figure 2. A call graph for an illustration of Facebook Infer's analysis process [12, 13, 16]

The last part of the architecture consists of a set of analyser plugins. Each plugin performs some analysis by interpreting the 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 3.

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

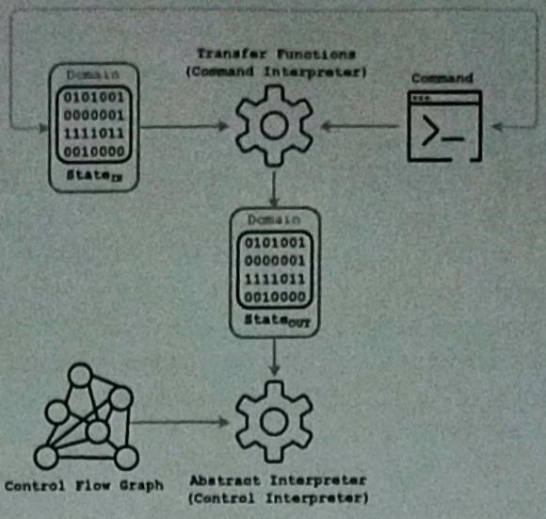


Figure 3. Facebook Infer's abstract interpretation process [12, 16]

3. Atomicity Violations Detector

In the thesis [12] and the paper [13], there was proposed the *Atomer* – a *static analyser* for finding some forms of *atomicity violations* implemented as a module of *Facebook Infer*. The proposed solution is slightly inspired by ideas from [17, 10, 25, 20, 19].

As demonstrated in Figure 4, the proposed analysis is divided into two parts (*phases of the analysis*):

Phase 1: Detection of (likely) *atomic sequences*.
Phase 2: Detection of *atomicity violations* (violations of the atomic sequences).

The design and implementation are in detail described in [12] – Chapters 3 and 4.

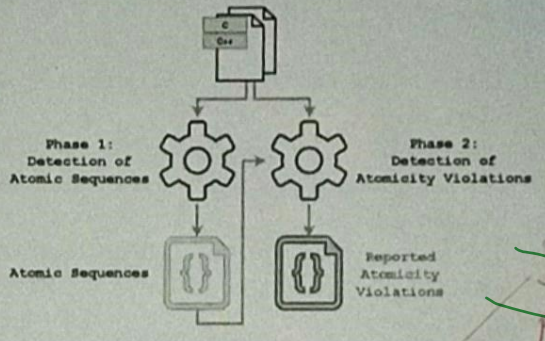


Figure 4. Phases of the proposed analyser

3.1 Approximation with Sets of Function Calls

Working with *sequences of function calls* was approximated by working with *sets of function calls* to make the solution more *scalable*. I.e., the solution is more scalable because the order of stored function calls is not relevant while working with sets. Thus, less memory is required because the same sets of function

calls are not stored multiple times. The analysis is also faster since there are stored fewer function calls to work with. Indeed, the analysis is less accurate because this causes loss of some information. The analysis after the approximation is illustrated in Figure 5.

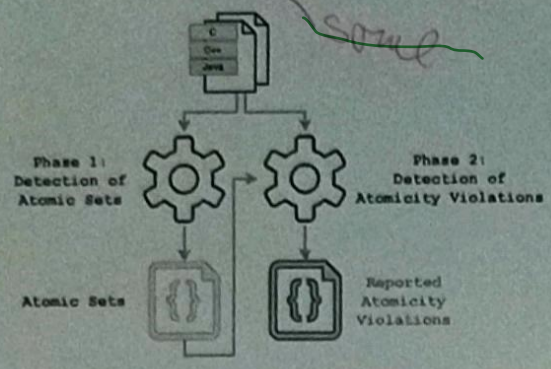


Figure 5. Phases of the analyser approximated with sets of function calls

Example 3.1. For demonstrating the approximation of the analysis with sets of function calls, assume functions from Listing 1. Before the approximation, when the analysis was working with sequences of function calls, **Phase 1** of the analysis produced the following *summery* while analysed function `f`: ((a, b, c), (x, a, b, c)), and the following summary while analysed function `g`: ((c, b, a), (x, c, b, a)), where the first component is a sequence of atomic calls and the second component is a sequence of all calls. Whereas, after the approximation, the produced summary for both functions is the same because the first component is a set of atomic calls and the second component is a set of all calls: ({a, b, c}, {a, b, c, x}).

Example 3.2. In **Phase 2** of the analysis, if the result of the first phase had been the following set of sequences of functions called atomically {(a, b, c)}, the analysis would have looked for the following pairs of functions that are not called atomically: (a, b), (b, c). Since the result of the first phase was changed to the following set of sets of functions called atomically {{a, b, c}}, the analysis now looks for the following pairs of functions that are not called atomically: (a, b), (b, a), (b, c), (c, b), (a, c), (c, a). I.e., it looks for all the possible pairs from a given set.

3.2 Distinguishment of Multiple Locks Used

As it was Atomer designed in [12], different locks were not distinguished at all. Only calls of locks/unlocks were identified, and parameters of these calls (*lock objects*) were not considered. So, if there had been several lock objects used, the analysis would not have been working correctly. Therefore, lock objects are

```
1  void f(void)
2  {
3      x();
4
5      pthread_mutex_lock(&L);
6      a(); b(); c();
7      pthread_mutex_unlock(&L);
8  }
9
10 void g(void)
11 {
12     x();
13
14     pthread_mutex_lock(&L);
15     c(); b(); a();
16     pthread_mutex_unlock(&L);
17 }
```

Listing 1. A code snippet used for illustration of the approximation of the analysis with sets of function calls

now taking into consideration. They are distinguishing using Facebook Infer's built-in mechanism called *access paths*[5]. During the analysis (both phases), each *atomic section* is identified by an access path of a lock that guards that section.

Example 3.3. Consider functions `x` and `y` from Listing 2. There are two lock objects `L1` and `L2` that are used simultaneously. After the extension of the distinguishment of multiple locks used, the analysis works as follows. In function `x`, there is recognised that lock `L1` guards a set of atomic calls `{a, b, c}`, and lock `L2` guards `{b}`. Further, in function `y`, lock `L1` guards `{a, b}`, and lock `L2` guards `{b, c}`.

3.3 Support for C++ and Java

Atomer was extended to support analyses of *C++* and *Java* programs. In general, Facebook Infer can analyse programs written in C, C++, Java, and Objective-C. The Facebook Infer's *frontend* compiles input programs into the *Smallfoot Intermediate Language* (SIL) and represents them as a *CFG*. Individual analyses are then performed on SIL. However, in practice, there are not negligible differences between these languages. Thus, individual non-trivial analysers have to be adapted for specific languages.

Earlier, Atomer supported only C/C++ languages using *PThread locks*. Within this project, it was ad-

[5] Facebook Infer uses **syntactic access paths** for naming heap locations via the paths used to access them. [2]

```
1  void x(void)
2  {
3      pthread_mutex_lock(&L1);
4      a();
5      pthread_mutex_lock(&L2);
6      b();
7      pthread_mutex_unlock(&L2);
8      c();
9      pthread_mutex_unlock(&L1);
10 }
11
12 void y(void)
13 {
14     pthread_mutex_lock(&L1);
15     a();
16     pthread_mutex_lock(&L2);
17     b();
18     pthread_mutex_unlock(&L1);
19     c();
20     pthread_mutex_unlock(&L2);
21 }
```

Listing 2. A code snippet used for illustration of distinguishing multiple locks used

justed to support more complicated *mutual exclusion* C++ mechanisms and to support analysis of Java programs with its mutual exclusion mechanisms. To identification of locks for different languages, it was used and extended the Facebook Infer's built-in module `ConcurrencyModels`.

Currently, Atomer supports these lock mechanisms:

- **C/C++ classic locks:**
  - `pthread_mutex`
  - `std::mutex`
  - `std::lock`
  - `std::recursive_mutex`
  - `std::recursive_timed_mutex`
  - `std::shared_mutex`
  - `std::timed_mutex`
  - `boost::shared_mutex`
  - `boost::mutex`
  - `folly::MicroSpinLock`
  - `folly::RWSpinLock`
  - `folly::SharedMutex`
  - `folly::SharedMutexImpl`
  - `folly::SpinLock`
  - `apache::thrift::concurrency::Mutex`
  - `apache::thrift::concurrency::NoStarveReadWriteMutex`
  - `apache::thrift::concurrency::ReadWriteMutex`
  - `apache::thrift::concurrency::Monitor`
- **C/C++ lock guards:**
  - `std::lock_guard`
  - `std::shared_lock`
  - `std::unique_lock`
  - `folly::SharedMutex::ReadHolder`
  - `folly::SharedMutex::WriteHolder`
  - `folly::LockedPtr`
  - `folly::SpinLockGuard`
  - `apache::thrift::concurrency::Guard`
  - `apache::thrift::concurrency::RWGuard`
- **Java classic locks:**
  - `java.util.concurrent.locks.Lock`
  - `java.util.concurrent.locks.ReentrantLock`
  - `java.util.concurrent.locks.ReentrantReadWriteLock.ReadLock`
  - `java.util.concurrent.locks.ReentrantReadWriteLock.WriteLock`
- **Java lock guards (`synchronized` keyword):**
  - `synchronized` block
  - `synchronized` method

4. Experimental Evaluation

This section is devoted to *testing* and *experimental evaluation* of the new features of the *Atomer* analyser presented in Section 3. Each feature was tested independently as soon as it had been implemented. It was tested on suitable programs created for testing purposes and got inspired by test programs from [12]. In more detail, it is described in Section 4.1. Moreover, Section 4.2 shows performed experiments on test programs derived from the static analyser *Gluon*. Finally, Section 4.3 is about the experimental evaluation of Atomer on *publicly available real-life complex* programs.

4.1 Testing on Hand-Crafted Examples

The correct behaviour of Atomer was already tested in [12], described in Section 5.1, and test programs are available in Appendix A. There are reasonable C programs that use *PThread* locks. These programs contain sequences of function calls inside and outside

atomic blocks, *not paired lock/unlock calls*, *iteration*, *selection*, and *nested function calls*. So, the functions are designed in order to check all the aspects of the analyser. It comprises all the components of the *abstract state* and the *summary*.

Because working with *sequences of function calls* was approximated to working with *sets of function calls*, reference outputs for test programs from [12] (Appendix A) were modified to the appropriate form, and the same tests were performed again[6]. Further, new test programs were derived from those in [12], and they were changed to test the new feature that adds support for distinguish *multiple locks* used[7]. In the end, several new test programs were written in *C++* and *Java* for validation that the support for these languages is working[7]. Various kinds of lock mechanisms were used.

In all the cases above, it was proved the correct behaviour of the analysis of Atomer (concerning the proposal).

4.2 Experiments on Test Programs Derived from the Gluon Tool

Gluon[8] is a tool for static analysis of *contracts for concurrency* in Java programs developed in [25, 10]. Several validation examples[9] in Java were created to validate the functionality of Gluon. These example programs were rewritten to C++[10], and existing *atomicity violations* were successfully found. Moreover, programs were also rewritten to the form where atomicity violations are fixed by adding appropriate locks, and it was shown that Atomer founds no atomicity violations in such correct programs.

4.3 Evaluation on Real-Life Programs

In [12], Atomer was evaluated 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 initially used for an experimental evaluation of Daniel Kroening's static deadlock analyser for *C/PThreads* [14] implemented in the CPROVER framework. Several potential atomicity violations were found. In more detail, the results are listed in [12] – Section 5.2.

Since Atomer supports analysis of Java programs, two *real-life open-source extensive* Java programs were analysed – *Apache Cassandra*[11] and *Apache Tomcat*[12]. In [25, 10], there were reported a few *atomicity-related* bugs[13][14]. It turns out that the reported bugs were real atomicity violation errors, and they were later fixed. Thus, within this project, these programs were analysed by Atomer, and the bugs were successfully *rediscovered*. Unfortunately, so far, quite some *false alarms* were reported. However, make the analyser more *accurate* is currently the work under progress. Besides, the results of the analysis can be used as an input for *dynamic analysis*, which can be able to check whether the atomicity violations are real errors. For instance, one cloud uses *ANaConDA*[15] 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 sets whose atomicity was found broken.

5. Conclusion

This paper started by discussing a *static analysis* framework that uses *abstract interpretation* — *Facebook Infer* — its features, architecture, and existing analysers implemented in this tool. The major part of the paper then aimed at the description of a static analyser for detecting *atomicity violations* — *Atomer* — implemented as a module of Facebook Infer and its extensions and improvements. Lastly, it is described the experimental evaluation of the new features and other experiments performed in this project practice.

The original analyser works on the level of *sequences of function calls*, but in this project, it was changed to work on the level of *sets of function calls*. The solution is based on the assumption that sequences executed *atomically once* should probably be executed *always atomically*. Within this project, two new main

[6] **Atomic sets** testing – https://github.com/harmim/vut-ibt/tree/master/experiments.

[7] **Multiple locks** and **C++/Java** languages support testing – https://github.com/harmim/vut-pp1/tree/master/experiments.

[8] **Static analyser Gluon** – https://github.com/trxsys/gluon.

[9] **Validation examples for Gluon** – https://github.com/trxsys/gluon/tree/master/test/validation.

[10] **Experiments on Gluon** example programs using **Atomer** – https://github.com/harmim/vut-pp1/tree/master/testing-programs/contracts-first-experiments-atomer.

[11] **Apache Cassandra's** repository – https://github.com/apache/cassandra.

[12] **Apache Tomcat's** repository – https://github.com/apache/tomcat.

[13] **Reported atomicity violations in Apache Cassandra** – https://issues.apache.org/jira/browse/CASSANDRA-7757.

[14] **Reported atomicity violations in Apache Tomcat** – https://bz.apache.org/bugzilla/show_bug.cgi?id=56784.

[15] Website of **ANaConDA framework** – http://www.fit.vutbr.cz/research/groups/verifit/tools/anaconda.

features were introduced: (i) support for *C++ and Java locks*, (ii) and distinguish *multiple locks used*.

The introduced extensions and improvements were successfully tested on smaller *hand-crafted* programs. It turned out that such innovations enhanced the *accuracy* and *scalability* of the analysis. Moreover, Atomer was experimentally evaluated on another software. Notably, it was evaluated on *open-source real-life Java programs – Apache Cassandra* and *Tomcat*. Already fixed and reported real bugs were successfully rediscovered. Nevertheless, so far, quite some *false alarms* are reported. However, 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 again shows the potential for further improvements. The future work will focus mainly on increasing the accuracy of the methods used by, e.g., distinguishing the *context of called functions* by considering *formal parameters*, *ranking* of atomic functions, or focusing on *library containers concurrency restrictions* related to method calls. Further, it is needed to perform more experiments on *real-life* programs with an effort to find and report *new bugs*.

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. It is expected that work on this project will continue not only within diploma thesis at FIT BUT.

Acknowledgements

I want to thank my supervisor Tomáš Vojnar for his assistance. Further, I would like to thank other colleagues from VeriFIT. I would also like to thank Nikos Gorogiannis from the Infer team at Facebook for useful discussions about the development of the analyser. Lastly, I thank for the support received from the H2020 ECSEL project Aquas.

References

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

[2] BLACKSHEAR, S.; GOROGIANNIS, N.; O'HEARN, P. W.; SERGEY, I.: RacerD: Compositional Static Race Detection. *Proceedings of ACM Programming Languages*. October 2018, vol. 2, OOPSLA'18, p. 144:1 – 144:28. ISSN 2475-1421. doi:10.1145/3276514.

[3] BLACKSHEAR, S.; O'HEARN, P. W.: *Open-sourcing RacerD: Fast static race detection at scale* [online]. 2017-10-19 [cit. 2020-07-28]. Available at: `https://code.fb.com/android/open-sourcing-racerd-fast-static-race-detection-at-scale`.

[4] 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*. Savannah, GA, USA: ACM, New York, NY, USA, January 2009, p. 289 – 300. POPL'09. ISBN 978-1-60558-379-2. doi:10.1145/1480881.1480917.

[5] COUSOT, P.: *Abstract Interpretation* [online]. 2008-08-05 [cit. 2020-07-28]. Available at: `https://www.di.ens.fr/~cousot/AI`.

[6] COUSOT, P.: *Abstract Interpretation in a Nutshell* [online]. [cit. 2020-07-28]. Available at: `https://www.di.ens.fr/~cousot/AI/IntroAbsInt.html`.

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

[8] 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, p. 238 – 252. POPL'77. doi:10.1145/512950.512973.

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