BRNO UNIVERSITY OF TECHNOLOGY

FACULTY OF INFORMATION TECHNOLOGY

DEPARTMENT OF INTELLIGENT SYSTEMS

STATIC ANALYSIS IN FACEBOOK INFER FOCUSED ON ATOMICITY

PROJECT PRACTICE

AUTHOR **Bc. DOMINIK HARMIM**

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

BRNO 2021

Abstract

This project practice aims to improve and extend *Atomer*. A further goal is to perform new experiments with it. Atomer is a *static analyser* that detects *atomicity violations*, created within the bachelor's thesis of the author as a module of *Facebook Infer* — an open-source static analysis framework that promotes efficient analysis. The original analysis assumes that *sequences of function calls* executed *atomically once* should probably be executed *always atomically*, and it naturally works with sequences. In the project, to improve *scalability*, the use of sequences was *approximated* by *sets*. Further, two new features were implemented: support for *C++* and *Java* languages; and distinguishing *multiple locks used*. The new features were verified and evaluated on smaller programs created for testing purposes. Furthermore, new experiments on *real-life programs* were made, and already fixed and reported *real bugs* were rediscovered. Several other improvements were proposed (*parametrisation* of the analysis, support for *interprocedural locks*, or combination with a *dynamic analysis*) and their implementation and evaluation is the work in progress.

Keywords

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

Reference

HARMIM, Dominik. *Static Analysis in Facebook Infer Focused on Atomicity.* Brno, 2021. Project practice. Brno University of Technology, Faculty of Information Technology. Supervisor prof. Ing. Tomáš Vojnar, Ph.D.

Static Analysis in Facebook Infer Focused on Atomicity

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 analyser's development. Lastly, I thank for the support received from the H2020 ECSEL project Arrowhead Tools.

Contents

Chapter 1

Introduction

Bugs are an integral part of computer programs ever since the inception of the programming discipline. Unfortunately, they are often hidden in unexpected places, and they can lead to unexpected behaviour, which may cause significant damage. Nowadays, developers have many possibilities of catching bugs in the early development process. *Dynamic analysers* or tools for *automated testing* are often used, and they are satisfactory in many cases. Nevertheless, they can still leave too many bugs undetected, because they 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 a considerably high rate of incorrectly reported errors (so-called *false positives* or *false alarms*). Several efficient tools for static analysis were implemented, e.g., Coverity or CodeSonar. However, they are often proprietary and difficult to openly evaluate and extend.

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

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.

Within the author's bachelor's thesis [13], *Atomer*[1] was proposed — a *static analyser* for finding some forms of *atomicity violations* implemented as a Facebook Infer's module. In particular, the stress is put on the *atomic execution of sequences of function calls*, which is often required, e.g., when using specific library calls. It is based on the assumption that sequences of function calls executed *atomically once* should probably be executed *always atomically*, and it naturally works with sequences. In fact, the idea of checking atomicity of certain sequences of function calls is inspired by the works of *contracts for concurrency* [11, 25]. In the terminology of [11, 25], atomicity of specific sequences of calls is the most straightforward (yet very useful in practice) kind of contracts for concurrency. The implementation of the first version of Atomer mainly targets *C programs* that use *POSIX thread*, i.e., *PThread locks*.

Within this project practice, Atomer was improved and extended. Further, other experiments were performed. In particular, to improve *scalability*, working with sequences of function calls was *approximated* by working with *sets of function calls*. Furthermore, two new features were implemented: support for *C++* and *Java* languages; and a way to distinguish *multiple locks used*. Moreover, several other improvements were proposed to reduce the number of false alarms, namely, *parametrisation of the analysis*, support for *interprocedural locks*, or combinations with a *dynamic analysis*. Their implementation and evaluation is currently the work in progress.

The development of Atomer and its extensions have been discussed with developers of Facebook Infer, and it is a part of the H2020 ECSEL project Arrowhead Tools. Parts of this report are taken from the thesis [13], project practice [14], and the paper [15] written by the author.

The rest of the report is organised as follows. In Chapter 2, there are described all the topics related to and essential to this project. In particular, Section 2.1 deals with *static analysis* based on *abstract interpretation*. The *Facebook Infer* framework, which uses abstract interpretation, is described in Section 2.2. Finally, in Section 2.3, there is described the concept of *contracts for concurrency*. *Atomer* is described in Chapter 3, together with its limitations. All the extensions and improvements proposed within the project are described in Chapter 4. Finally, the project is concluded in Chapter 5.

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

Chapter 2

Preliminaries

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

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

2.1 Static Analysis by Abstract Interpretation

According to [21], *static analysis* of programs is reasoning about the behaviour of computer programs without actually executing them. It has been used since the 1970s in optimising compilers for generating efficient code. More recently, it has proven valuable also for automatic error detection, verification of correctness of programs, and it is used in other tools that can help programmers. Intuitively, a static program analyser is a program that reasons about the behaviour of other programs, in other words, a static program analyser is a program that reasons about another programs by looking for some *syntactic patterns* in the code and/or by assigning the program statements some *abstract semantics* and then deriving a characterisation of the behaviour in terms of the abstract semantics. Nowadays, static analysis is one of the fundamental concepts of *formal verification*. It aims to automatically answer questions about a given program, such as, e.g., [21]:

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

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

It is well-known that testing, i.e., executing programs with some input data and examining the output, may expose errors, but it cannot prove their absence. (It was also famously stated by Edsger W. Dijkstra: "*Program testing can be used to show the presence of bugs, but never to show their absence!*".) However, static program analysis can prove their absence — with some *approximation* — it can check *all possible executions* of the programs and provide guarantees about their properties. Another advantage of static analysis is that the analysis can be performed during the development process, so the program does not have to be executable yet and it already can be analysed. The biggest disadvantage of static analysis is that it can produce many *false alarms*[1], which is often resolved by accepting *unsoundness*[2]. Another major issue is that of ensuring sufficient *scalability* of static analysis: in fact, typically, the more precise the analysis is, the less scalable it becomes.

Various forms of static analysis of programs have been invented, for instance [27]: bug pattern searching, data-flow analysis, constraint-based analysis, type analysis, or symbolic execution. One of the most widely used approaches — *abstract interpretation* — is detailed in Section 2.1.1.

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

2.1.1 Abstract Interpretation

This section explains and defines the basics of *abstract interpretation*. The description is based on [8, 9, 7, 6, 17, 18, 10, 22, 21, 28]. In these works, there can be found more detailed and more formal explanation.

Abstract interpretation was introduced and formalised by a French computer scientist Patrick Cousot and his wife Radhia Cousot in the year 1977 at POPL (symposium on Principles of Programming Languages) [9]. It is a generic *framework* for static analyses. It allows one to create particular analyses by providing specific components (described later) to the framework. The obtained analysis is guaranteed to be *sound* if certain properties of the components are met. [17, 18]

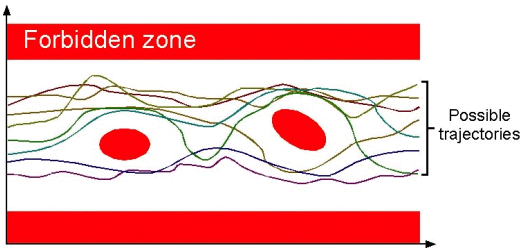
In general, in the set theory, which is independent of the application setting, abstract interpretation is considered a theory for *approximating* sets and set operations. A more restricted formulation of abstract interpretation is to interpret it as a theory of approximation of the behaviour of the *formal semantics* of programs. Those behaviours may be characterised by *fixpoints* (defined below), which is why a primary part of the theory provides efficient techniques for *fixpoint approximation* [22]. So, for a standard semantics, abstract interpretation is used to derive the approximate abstract semantics over an *abstract domain* (defined

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

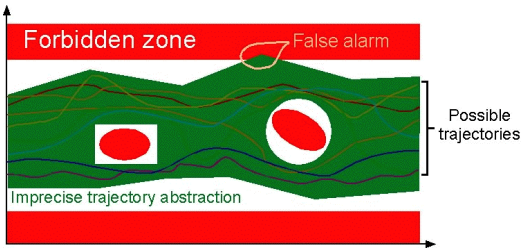
[2] **Soundness** – if a verification method claims that a system is correct according to a given specification, it is truly correct [27].

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

Patrick Cousot intuitively and informally illustrates abstract interpretation in [7] as follows. Figure 2.1a shows the *concrete semantics* of a program by a set of curves, which represents the set of all possible executions of the program in all possible execution environments. Each curve shows the evolution of the vector $x(t)$ of input values, state, and output values of the program as a function of the time t. *Forbidden zones* on this figure represent a set of erroneous states of the program execution. Proving that the intersection of the concrete semantics of the program with the forbidden zone is empty may be *undecidable* because the program concrete semantics is, in general, *not computable*. As demonstrates Figure 2.1b, abstract interpretation deals with an *abstract semantics*, i.e., the *superset* of the concrete program semantics. The abstract semantics includes all possible executions. That implies that if the abstract semantics is safe (i.e. it does not intersect the forbidden zone), the concrete semantics is safe as well. However, the *over-approximation* of the possible program executions causes that inexisting program executions are considered, which may lead to *false alarms*. It is the case when the abstract semantics intersects the forbidden zone, whereas the concrete semantics does not intersect it.



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



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

Figure 2.1: Abstract interpretation demonstration [7]. Horizontal axes: time t. Vertical axes: vector $x(t)$ of input, state, and output values of the considered program

Components of Abstract Interpretation

In accordance with [17, 18], the basic components of abstract interpretation are as follows:

- **An Abstract Domain** [6]:
  - An abstraction of the possible concrete program states (or their parts) in the form of *abstract properties*[3] and *abstract operations*[4] [8].
  - Sets of program states at certain locations are represented using *abstract states*.
- **Abstract Transformers**:
  - There is a *transform function* for each program operation (instruction) that represents the impact of the operation executed on an abstract state.

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

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

- **The Join Operator** $\sqcup$:
  - Joins abstract states from individual program branches into a single one.
- **The Widening Operator** $\triangledown$ [22, 10, 17]:
  - Enforces termination of the abstract interpretation.
  - It is used to approximate the *least fixed points* of program semantics (it is performed on a sequence of abstract states at a certain location).
  - Usually, the later in the analysis this operator is used, the more accurate the result is (but the analysis takes more time).
- **The Narrowing Operator** $\triangle$ [22, 10, 17]:
  - Using this operator, the approximation obtained by widening can be refined, i.e., it may be used to refine the result of widening.
  - This operator is used when a *fixpoint* is approximated using widening.

Fixpoints and Fixpoint Approximation

A **fixpoint** of a function $f : A \to A$ is an element $a \in A$ if and only if $\boldsymbol{f(a) = a}$ [28].

Computation of the *most precise abstract fixpoint* is not generally guaranteed to terminate, in particular, when a given program contains a loop or recursion. The solution is to approximate the fixpoint using *widening* (over-approximation of a fixpoint) and *narrowing* (improves the approximation of the fixpoint) [17, 18]. Most program properties can be represented as fixpoints. This reduces program analysis to the fixpoint approximation [6]. Further information about fixpoint approximation can be found, e.g., in [22, 10].

Formal Definition of Abstract Interpretation

According to [9, 17], **abstract interpretation** $\boldsymbol{I}$ of a program P with the instruction set S is a tuple

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

where

- $\boldsymbol{Q}$ is the *abstract domain* (domain of *abstract states*),
- $\sqcup : Q \times Q \to Q$ is the *join operator* for accumulation of abstract states,
- $(\sqsubseteq) \subseteq Q \times Q$ is an *ordering* defined as $x \sqsubseteq y \Leftrightarrow x \sqcup y = y$ in $(Q, \sqcup, \top)$,
- $\top \in Q$ is the *supremum* of Q,
- $\bot \in Q$ is the *infimum* of Q,
- $\boldsymbol{\tau} : S \times Q \to Q$ defines *abstract transformers* for specific instructions,
- $(Q, \sqcup, \top)$ is a *complete semilattice* [28, 17].

Using so-called *Galois connections* [22, 10, 17, 6], one can guarantee the *soundness* of abstract interpretation.

2.2 Facebook Infer — a Static Analysis Framework

This section describes the principles and features of *Facebook Infer*. The description is based on information provided at the Facebook Infer website[5] and in [2, 18]. Parts of this section are taken from the paper [15].

Facebook Infer is an open-source[6] static analysis *framework*, which is able to discover various kinds of software bugs of a given program, with the stress put on *scalability*. The basic usage of Facebook Infer is illustrated in Figure 2.2. A more detailed explanation of its architecture is shown below. Facebook Infer is implemented in *OCaml*[7] – *functional* programming language, also supporting *imperative* and *object-oriented* paradigms. Further details about OCaml can be found in [20] or in official documentation[8], tutorials[9]. Facebook Infer was originally a standalone tool focused on *sound verification* of the absence of *memory safety violations*, which was first published in the well-known paper [5].

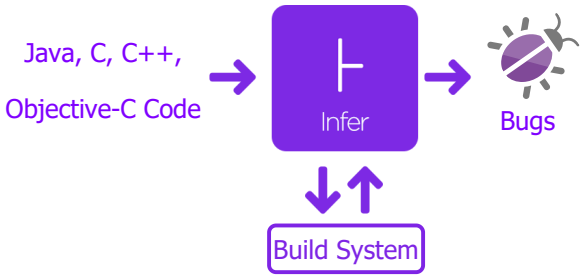


Figure 2.2: Static analysis in Facebook Infer (inspired by https://adtmag.com/articles/2015/06/12/facebook-infer.aspx)

Facebook Infer is able to analyse programs written in several languages. In particular, it supports languages C, C++, Java, and Objective-C. Moreover, it is possible to extend Facebook Infer's *frontend* for supporting other languages. Currently, Facebook Infer contains many analyses focusing on various kinds of bugs, e.g., *Inferbo* (buffer overruns) [29]; *RacerD* (data races) [3, 4, 12]; and other analyses that check for buffer overflows, some forms of deadlocks and starvation, null-dereferencing, memory leaks, resource leaks, etc.

2.2.1 Abstract Interpretation in Facebook Infer

Facebook Infer is a general framework for static analysis of programs, it is based on *abstract interpretation*. Despite the original approach taken from [5], Facebook Infer aims to find bugs rather than formal verification. It can be used to quickly develop new sorts of *compositional* and *incremental* analysers (*intraprocedural* or *interprocedural* [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

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

analysis along the *Control Flow Graph* (**CFG**)[10] 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 later). Therefore, a function is analysed and a summary is computed without knowledge of the call context. Then, the summary of a function is used at all of its call sites. Since summaries do not differ for different contexts, the analysis becomes more scalable, but it can lead to a loss of accuracy. In order to create a new intraprocedural analyser in Facebook Infer, it is needed to define the following (listed items are described in more detail in Section 2.1.1):

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

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

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

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

The Architecture of the Abstract Interpretation Framework in Facebook Infer

The architecture of the abstract interpretation framework of Facebook Infer (**Infer.AI**) may be split into three major parts, as demonstrated in Figure 2.3: a *frontend*, an *analysis scheduler* (and a *results database*), and a set of *analyser plugins*.

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

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

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

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

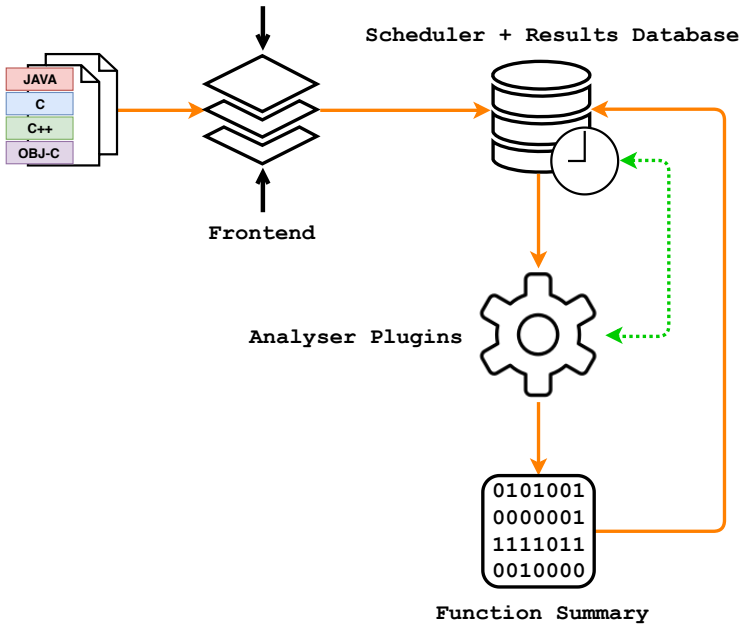


Figure 2.3: The architecture of Facebook Infer's abstract interpretation framework [2, 18]

The next part of the architecture is the scheduler, which defines the order of the analysis of single functions according to the appropriate *call graph*[11]. The scheduler also checks if it is possible to analyse some functions simultaneously, which allows Facebook Infer to run the analysis in parallel.

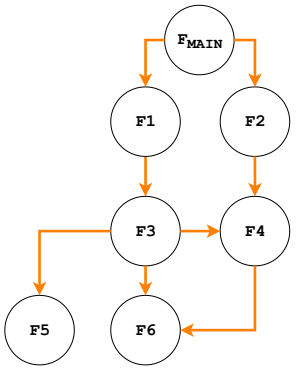


Figure 2.4: A call graph for an illustration of Facebook Infer's analysis process [2, 15, 18]

Example 2.2.1. For demonstrating the order of the analysis in Facebook Infer and its incrementality, assume a call graph in Figure 2.4. At first, leaf functions `F5` and `F6` are analysed. Further, the analysis goes on towards the root of the call graph – $\texttt{F}_{\texttt{MAIN}}$, while taking into consideration the dependencies denoted by the edges. This order ensures that a summary is available once a nested function call is abstractly interpreted within the analysis. When there is

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

a subsequent code change, only directly changed functions and all the functions up the call path are re-analysed. For instance, if there is a change of source code of function `F4`, Facebook Infer triggers re-analysis of functions `F4`, `F2`, and $\texttt{F}_{\texttt{MAIN}}$ only.

The last part of the architecture consists of a set of analyser plugins. Each plugin performs some analysis by interpreting of SIL instructions. The result of the analysis of each function (function summary) is stored to the results database. The interpretation of SIL instructions (*commands*) is done using an *abstract interpreter* (also called a *control interpreter*) and *transfer functions* (also called a *command interpreter*). The transfer functions take an actual *abstract state* of an analysed function as an input, and by applying the interpreting command, produce a new abstract state. The abstract interpreter interprets the command in an *abstract domain* according to the CFG. This workflow is shown in a simplified form in Figure 2.5.

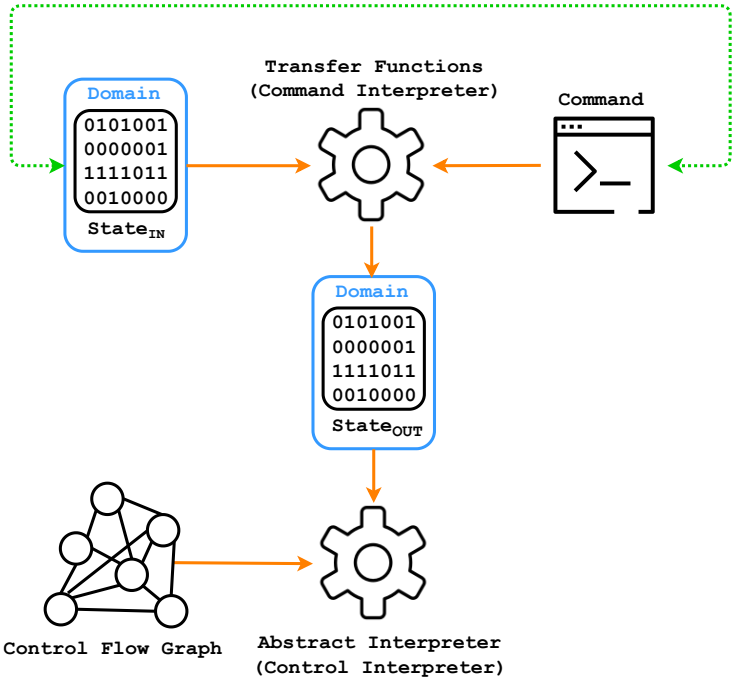


Figure 2.5: Facebook Infer's abstract interpretation process [2, 18]

2.3 Contracts for Concurrency

This section introduces the concept of *contracts for concurrency* described in [25, 11]. Parts of this section are taken from the paper [15]. Listings in this section are pieces of programs written in ANSI C.

Respecting the *protocol* of a software module — defines which *sequences of functions* are legal to invoke — is one of the requirements for the correct behaviour of the module. For example, a module that deals with a file system typically requires that a programmer using this module should call function `open` at first, followed by an optional number of functions `read` and `write`, and at last, call function `close`. A program utilising such a module that

does not follow this protocol is erroneous. The methodology of *design by contract* (described in [19]) requires programs to meet such well-defined behaviours. [25]

In *concurrent programs*, contracts for concurrency allow one — in the simplest case — to specify *sequences of functions* that are needed to be *executed atomically* in order to avoid *atomicity violations*. In general, contracts for concurrency specify sets of sequences of calls that are called *spoilers* and sets of sequences of calls that are called *targets*, and it is then required that no target overlaps fully with any spoiler. Such contracts may be manually specified by a developer or they may be automatically generated by a program (analyser). These contracts can be used to verify the correctness of programs as well as they can serve as helpful documentation.

Section 2.3.1 defines the notion of *basic contracts for concurrency*. Further, Section 2.3.2 defines contracts extended to consider the *data flow* between functions (where a sequence of function calls must be atomic only if they handle the same data). The above mentioned more general contracts for concurrency with spoilers and targets, which essentially extend the basic contracts with some *contextual information*, are not presented here in detail (they are explained in the paper [11]). The reason is that the proposed analyser — *Atomer* — so far concentrates on the basic contracts.

2.3.1 Basic Contracts for Concurrency

In [11, 25], a *basic contract* is formally defined as follows. Let $\Sigma_{\mathbb{M}}$ be a set of all function names of a software module. A contract is a set $\mathbb{R}$ of *clauses* where each clause $\varrho \in \mathbb{R}$ is a *star-free regular expression*[12] over $\Sigma_{\mathbb{M}}$. A *contract violation* occurs if any of the sequences expressed by the contract clauses are interleaved with the execution of functions from $\Sigma_{\mathbb{M}}$, in other words, each sequence specified by any clause ϱ must be executed atomically, otherwise, there is a violation of the contract. The number of sequences defined by a contract is finite since the contract is the union of *star-free languages*.

Example 2.3.1. Consider the following example from [11, 25]. Assume that there is a module implementing a resizable array implementing the following interface functions:

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

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

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

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

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

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

f_7: `int size(char *array)`

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

The module's contract contains the following clauses:

(ϱ_1) `contains index_of`

> The execution of `contains` followed by the execution of `index_of` should be atomic. Otherwise, the program may fail to get the index, because after verification of the presence of an element in an array, it can be removed by some *concurrently running process*.

(ϱ_2) `index_of (get | set | remove)`

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

(ϱ_3) `size (get | set | remove)`

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

(ϱ_4) `add (get | index_of)`

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

2.3.2 Contracts for Concurrency with Parameters

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

Example 2.3.2. Consider the following example from [11, 25], given Listing 2.1. There is a function `replace` that replaces item `a` in an array by item `b`. The implementation of this function comprises two atomicity violations:

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

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

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

(ϱ_5) `contains index_of set`

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

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

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

In order to respect function call *parameters* and *return values* of functions in contracts, the basic contracts are extended by dependencies among functions in [11, 25] as follows. Function call parameters and return values are expressed as *meta-variables*. Further, if a contract should be required to be observed exclusively if the same object emerges as an argument or as the return value of multiple calls in a given call sequence, it may be denoted by using the same meta-variable at the position of all these occurrences of parameters and return values.

Clause ϱ_5 can be extended as follows (repeated application of meta-variables `X/Y/Z` requiring the same objects $o_1/o_2/o_3$ to be used at the positions of `X/Y/Z`):

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

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

With the extension described above, it is possible to extend the contract from Section 2.3.1 as follows:

(ϱ_1') `contains(X,Y) index_of(X,Y)`

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

Chapter 3

Atomer — an Atomicity Violations Detector

This chapter describes the principles and limitations of the *static analyser Atomer* proposed as a module of *Facebook Infer* (introduced in Section 2.2) for finding some forms of *atomicity violations*. Atomer was proposed and in detail described in the bachelor's thesis of the author of this project [13]. Therefore, naturally, the following description in Section 3.1 is based on the mentioned thesis, and further, there is used information from [14, 15].

In Section 3.2, there are discussed *limitations* and *shortcomings* of Atomer. Again, some of the thoughts mentioned in this section are taken into consideration in [13, 14, 15].

3.1 Design of Atomer and Its Principles

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

At first, already existing solutions in this area (besides Atomer) are discussed. In particular, the following deals with other existing approaches and tools for finding atomicity violations, their advantages, disadvantages, features, availability, and so on. Then, finally, the algorithm that is behind Atomer is introduced.

Listings in the below sections are pieces of exemplary programs written in C language (assuming *POSIX thread*, i.e., *PThread locks* and declared and initialised global variable `lock` of a type `phtread_mutex_t`).

3.1.1 Related Work

Atomer is slightly inspired by ideas from [11, 25]. In these papers, there is a proposal and implementation of a *static validation* for finding some forms of *atomicity violations* based on *grammars* and *parsing trees*. In the paper [11], there is also described and implemented a *dynamic* approach for the validation. The authors of these papers implemented a stand-

alone prototype tool called *Gluon*[1] for analysing programs written in Java. It led to some promising experimental results, but the *scalability* of Gluon was still limited.[2] Moreover, Gluon is no more actively developed. This fact inspired the decision that eventually led to Atomer, namely, to get inspired by the ideas of [11, 25], but reimplement them in *Facebook Infer* redesigning it in accordance with the principles of Facebook Infer (described in Section 2.2), which should make the resulting tool more scalable. In the end, however, due to adapting the analysis for the context of Facebook Infer, the implementation of the analysis of Atomer is significantly different from [11, 25], as it is presented in Chapter 4 of [13]. Furthermore, unlike [11, 25], the implementation aims at programs written in the *C language* using *PThread locks* to *synchronise concurrent threads*.

In Facebook Infer, there was already implemented an analysis called *Lock Consistency Violation*[3]. It is a part of *RacerD* [3, 12, 4]. This analysis finds atomicity violations in C++ and Objective C programs for reads/writes on single variables required to be executed atomically. Atomer is different; it finds atomicity violations for *sequences of functions* required to be executed atomically, i.e., it checks whether *contracts for concurrency* (see Section 2.3) hold. Moreover, it tries to automatically determine which sequences should indeed be executed atomically (hence, to *derive the contracts automatically*).

3.1.2 Design of the Atomer Analyser

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

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

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

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

The phases are in-depth described in the sections below. Also, these phases of the analysis and its workflow are illustrated in Figure 3.1.

This section describes the proposal of Atomer in general. The concrete types of the *abstract states* (i.e. elements of the *abstract domain* $\boldsymbol{Q}$) and the *summaries* χ, along with the implementation of all necessary *abstract interpretation operators* are stated in Chapter 4 of [13]. However, actually, the abstract states $s \in \boldsymbol{Q}$ of both phases of the analysis are proposed as *sets*. So, in fact, the *ordering operator* $\sqsubseteq$ is implemented using testing for a *subset* (i.e. $s \sqsubseteq s' \Leftrightarrow s \subseteq s'$, where $s, s' \in \boldsymbol{Q}$), the *join operator* $\sqcup$ is implemented as the

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

[2] Some of the experiments performed by Gluon are also similarly performed by Atomer.

[3] **Lock Consistency Violation** is an *atomicity violations* analysis implemented in *Facebook Infer*. It is described at https://fbinfer.com/docs/checkers-bug-types/#lock-consistency-violation.

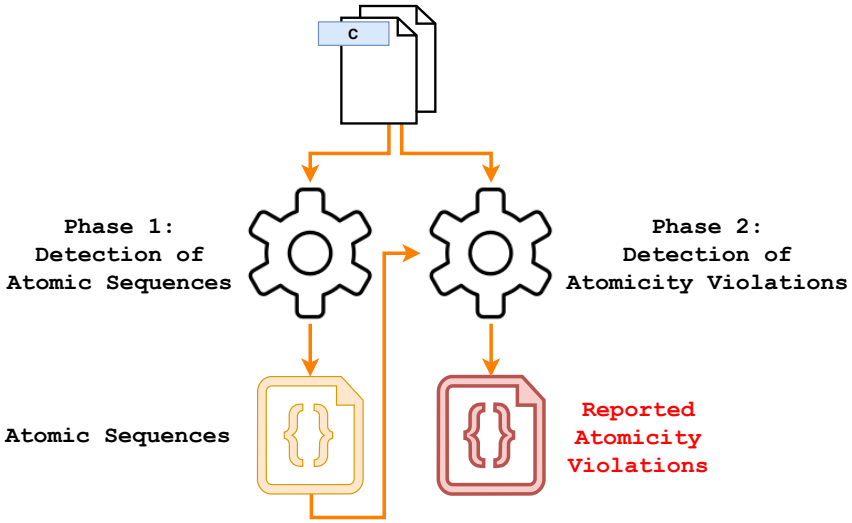


Figure 3.1: *Phases* of the Atomer's analysis and the analysis *high-level process* illustration

set union (i.e. $s \sqcup s' \Leftrightarrow s \cup s'$), and the *widening operator* $\triangledown$ is implemented using the join operator (i.e. $s \triangledown s' \Leftrightarrow s \sqcup s'$).

Function summaries are in below sections reduced to the output parts (*postconditions* R) only. The input parts of summaries (*preconditions* P) are in case of the proposed analysis always empty, because, so far, it is not necessary to have any preconditions for analysed functions. Thus, in this case, the Hoare triple $true\ \{Q\}\ R$ holds, where Q is an analysed program, i.e., $P = true$.

Phase 1: Detection of Atomic Sequences

Before detecting *atomicity violations* may begin, it is required to have *contracts* introduced in Section 2.3. **Phase 1** of Atomer is able to produce such contracts, i.e., it detects *sequences of functions* that should be *executed atomically*. Intuitively, the detection is based on looking for sequences of functions 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*.

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

$s \in \boldsymbol{Q}$ is defined as $2^{2^{\Sigma^* \times \Sigma^*}}$ (because there is a set of the (A, B) pairs for each program path).

It would be more precise to generate longer sequences of type $A_1B_1A_2B_2\ldots$, instead of the sets of the pairs (A, B). Nevertheless, it would be more challenging to ensure the above longer sequences' finiteness and the sets of these sequences' finiteness. Moreover, there would be a significantly larger *state space explosion problem* [26]. So, the proposed representation of the sets of pairs of sequences has been chosen to compromise accuracy and efficiency. However, the experiments described in Chapter 5 of [13] show that it needs even more pronounced abstraction for appropriate *scalability*.

Formally, the *initial abstract state* of a function is defined as $s_{init} = \{\{(\varepsilon, \varepsilon)\}\}$, where ε indicates an empty sequence. To formalise the analysis of a function, let `f` be a called leaf function. Further, let $s_{\mathtt{g}}$ be the abstract state of a function `g` being analysed before the function `f` is called. After the call of `f`, the abstract state will be changed as follows: $s_{\mathtt{g}} = \{p' \in 2^{\Sigma^* \times \Sigma^*} \mid \exists\, p \in s_{\mathtt{g}} : p' = \{(A', B') \in \Sigma^* \times \Sigma^* \mid \exists\, (A, B) \in p : (\neg actual(p, (A, B)) \wedge (A', B') = (A, B)) \vee [actual(p, (A, B)) \wedge [(lock \wedge (A', B') = (A, B \cdot \mathtt{f})) \vee (\neg lock \wedge (A', B') = (A \cdot \mathtt{f}, B))]]\}\}$, where *actual* is a Boolean function that determines whether a given (A, B) pair is the most recent pair of sequences of the current program state for a given program path. Furthermore, *lock* is a predicate indicating whether the current program state is inside an atomic block. Furthermore, let $s_{\mathtt{g}}$ be the abstract state of a function `g` being analysed before an unlock is called. After the unlock is called, a new (A, B) pair is created and labelled as an actual using the function *setActual* as follows: $s_{\mathtt{g}} = \{p' \in 2^{\Sigma^* \times \Sigma^*} \mid \exists\, p \in s_{\mathtt{g}} : p' = \{(A', B') \in \Sigma^* \times \Sigma^* \mid ((A', B') = (\varepsilon, \varepsilon) \wedge setActual(p, (A', B'))) \vee (A', B') \in p\}\}$.

Example 3.1.1. For an explanation of the computation of the sets of the pairs (A, B), assume that a state of the analysis of a program Q is the following sequence of function calls: `f, g`; and a state of the analysis of a program Q' is the following sequence of function calls: `f, g [m, n`. The square brackets are used to indicate an *atomic sequence* (the closing square bracket is missing in the case of the program Q', which means that the program state is currently inside an *atomic block*). The computed abstract state for the program Q is $s_Q = \{\{((\mathtt{f}, \mathtt{g}), \varepsilon)\}\}$, and for the program Q', it is $s_{Q'} = \{\{((\mathtt{f}, \mathtt{g}), (\mathtt{m}, \mathtt{n}))\}\}$. Now, if the next instruction is a call of a function `x`, in the case of the program Q, the call will be added to the first A sequence, and in the case of the program Q', the call will be added to the first B sequence as follows: $s_Q = \{\{((\mathtt{f}, \mathtt{g}, \mathtt{x}), \varepsilon)\}\}$, $s_{Q'} = \{\{((\mathtt{f}, \mathtt{g}), (\mathtt{m}, \mathtt{n}, \mathtt{x}))\}\}$. Subsequently, if the next step in the program Q is a lock call, the next function calls will be added to the first B sequence of the set s_Q. And if the next step in the program Q' is an unlock call, it will be created a new element of the first set of the set $s_{Q'}$, and the next function calls will be added to the A sequence of this element. Finally, if a function `y` is called, the resulting sets will look like follows: $s_Q = \{\{((\mathtt{f}, \mathtt{g}, \mathtt{x}), (\mathtt{y}))\}\}$, $s_{Q'} = \{\{((\mathtt{f}, \mathtt{g}), (\mathtt{m}, \mathtt{n}, \mathtt{x})), ((\mathtt{y}),\ \varepsilon)\}\}$. Note that the final sequences of function calls look like follows: `f, g, x [y` and `f, g [m, n, x] y` for the programs Q, and Q', respectively.

The *summary* $\chi_{\mathtt{f}} \in 2^{\Sigma^*} \times \Sigma^*$ of a function `f` is then defined as $\chi_{\mathtt{f}} = (\boldsymbol{B}, AB)$, where:

- $\boldsymbol{B}$ is a set constructed that contains all the B sequences that appear on program paths through `f`, i.e., $\boldsymbol{B} = \{B' \in \Sigma^* \mid \exists\, p \in s_{\mathtt{f}} : \exists\, (A, B) \in p : B \neq \varepsilon \wedge B' = B\}$, where $s_{\mathtt{f}}$ is the abstract state at the end of an interpretation of `f`. In other words, this component of the summary is a set of sequences of atomic function calls appearing in an analysed function.

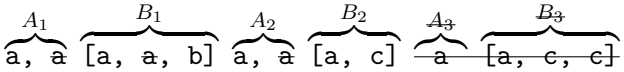
- AB is a *concatenation* of all the A and B sequences with removed duplicates of function calls. In particular, assume that there is the following computed set of (A, B) pairs: $\{(A_1, B_1), (A_2, B_2), \ldots, (A_n, B_n)\}$, then the result is concatenated sequence $A_1 \cdot B_1 \cdot A_2 \cdot B_2 \cdot \ldots \cdot A_n \cdot B_n$ with removed duplicates. Formally:

$$AB = reduce(\dot{\bigcup_{ab \in AB'}} ab)$$

  where $AB' = \{ab \in \Sigma^* \mid \exists p \in s_{\mathtt{f}} : \exists (A, B) \in p : ab = A \cdot B\}$, *reduce* is a function that removes duplicates of function calls, and $\dot{\bigcup}$ concatenates all sequences of a set. Intuitively, in this component of the summary, it is gathered occurrences of all called functions within an analysed function obtained by concatenation of all the A and B sequences.

AB is recorded to analyse functions higher in the *call hierarchy* since locks/unlocks can appear in such a *higher-level* function.

Example 3.1.2. For instance, the analysis of the function `f` from Listing 3.1 produces the following sequences:



The functions `a, b, c` are not deeper analysed because it is assumed that these functions are leaf nodes of the *call graph*. The strikethrough of the functions `a` and `c` denotes removing already recorded function calls in the A and B sequences. The strikethrough of the entire sequence `a [a, c, c]` means discarding sequence already seen before. For the above, the abstract state at the end of an interpretation of the function `f` is $s_{\mathtt{f}} = \{\{((\mathtt{a}), (\mathtt{a}, \mathtt{b})), ((\mathtt{a}), (\mathtt{a}, \mathtt{c})), (\varepsilon, \varepsilon)\}\}$. The derived summary $\chi_{\mathtt{f}}$ for the function `f` is $\chi_{\mathtt{f}} = (\boldsymbol{B}, AB)$, where:

- $\boldsymbol{B} = \{(\mathtt{a}, \mathtt{b}), (\mathtt{a}, \mathtt{c})\}$, i.e., B_1 and B_2;
- $AB = (\mathtt{a}, \mathtt{b}, \mathtt{c})$, i.e., the concatenation of A_1, B_1, A_2, and B_2 from which duplicate function calls were removed.

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

```
1  void f()
2  {
3      a(); a();
4
5      pthread_mutex_lock(&lock); // (a, b)
6      a(); a(); b();
7      pthread_mutex_unlock(&lock);
8
9      a(); a();
10
11     pthread_mutex_lock(&lock); // (a, c)
12     a(); c();
13     pthread_mutex_unlock(&lock);
14
15     a();
16
17     pthread_mutex_lock(&lock); // (a, c)
18     a(); c(); c();
19     pthread_mutex_unlock(&lock);
20 }
```

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

the program paths. To formalise this process, let `f` be a called function that was already analysed, and the second component of its summary is AB. Further, let $s_{\mathtt{g}}$ be the abstract state of a function `g` being analysed before the function `f` is called. After the call of `f`, the abstract state will be changed as follows: $s_{\mathtt{g}} = \{p' \in 2^{\Sigma^* \times \Sigma^*} \mid \exists p \in s_{\mathtt{g}} : p' = \{(A', B') \in \Sigma^* \times \Sigma^* \mid \exists (A, B) \in p : (\neg actual(p, (A, B)) \wedge (A', B') = (A, B)) \vee [actual(p, (A, B)) \wedge [(lock \wedge (A', B') = (A, B \cdot \mathtt{f} \cdot AB)) \vee (\neg lock \wedge (A', B') = (A \cdot \mathtt{f} \cdot AB, B))]]\}\}$.

Example 3.1.3. This example shows how the function `g` from Listing 3.2 would be analysed using the result of the analysis of the function `f` from Listing 3.1. So the analysis of the function `g` produces the following sequence:

```
a, f, a, b, c [f, a, b, c]
```

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

Cases Where Lock/Unlock Calls Are Not Paired in a Function For treating cases where *lock/unlock calls are not paired* in a function — as demonstrated in Listing 3.3 — in Atomer, the following solution is implemented.

Everything is unlocked at the end of a function, i.e., one *virtually* appends an unlock to the end of the function if it is necessary. Then, for the function `x` from Listing 3.3, the atomic

```
1  void g()
2  {
3      a(); f();
4
5      pthread_mutex_lock(&lock); // (f, a, b, c)
6      f();
7      pthread_mutex_unlock(&lock);
8  }
```

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

section is virtually closed. Hence, there is detected an atomic sequence (`a`). In particular, the summary is as follows: $\chi_{\texttt{x}} = (\{(\texttt{a})\}, (\texttt{a}))$.

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

```
1  void x()
2  {
3      pthread_mutex_lock(&lock); // (a)
4      a();
5  }
6  void y()
7  {
8      a();
9      pthread_mutex_unlock(&lock);
10 }
```

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

Summary of the Phase 1 The above detection of atomic sequences was implemented and also validated on a set of sample programs created for testing purposes, which is described in [13]. The derived sequences of calls assumed to execute atomically — the $\boldsymbol{B}$ sequences — from the summaries of all analysed functions are stored into a file used during **Phase 2**, which is described below.

Phase 2: Detection of Atomicity Violations

In the second phase of the analysis, i.e., when *detecting violations* of the atomic sequences obtained from **Phase 1**, the analysis looks for *pairs of functions* that *should be called atomically* (or just for single functions if there is only one function call in an atomic sequence) while this is not the case on some path through the CFG. The pairs of function calls to be checked for atomicity are obtained as follows: For each function `f` with the *summary* $\chi_{\texttt{f}} = (\boldsymbol{B}, AB)$ in a given program Q, it is taken the first component $\boldsymbol{B}$ of the

summary $\chi_{\mathtt{f}}$, i.e., $\boldsymbol{B} = \{B_1, B_2, \ldots, B_n\}$, and it is taken *every pair* $(\mathtt{x}, \mathtt{y}) \in \Sigma \times \Sigma$ of functions that appear as a *substring* in some of the B_i sequences, i.e., $B_i = w \cdot \mathtt{x} \cdot \mathtt{y} \cdot w'$ for some sequences w, w'. Note that $\mathtt{x}$ could be ε (an empty sequence) if some B_i consists of a single function. All these "atomic pairs" are put into the set $\Omega \in 2^{\Sigma \times \Sigma}$. More formally, $\Omega = \{(\mathtt{x}, \mathtt{y}) \in \Sigma \times \Sigma \mid \exists\, (\boldsymbol{B}, AB) \in X_Q : \exists\, B \in \boldsymbol{B} : (|B| = 1 \wedge (\mathtt{x}, \mathtt{y}) = (\varepsilon, B)) \vee (|B| > 1 \wedge \exists\, w, w' \in \Sigma^* : B = w \cdot \mathtt{x} \cdot \mathtt{y} \cdot w' \wedge (\mathtt{x}, \mathtt{y}) \neq (\varepsilon, \varepsilon))\}$, where Σ^* is a set of all possible sequences of functions from Σ from a given program, and $X_Q \in 2^{2^{\Sigma^*} \times \Sigma^*}$ is a set of all summaries of the program Q.

Example 3.1.4. For instance, assume that in **Phase 1**, there was analysed a function `f`. Which produced the summary $\chi_{\mathtt{f}} = (\boldsymbol{B}, AB)$, where $\boldsymbol{B} = \{(\mathtt{a}, \mathtt{b}, \mathtt{c}), (\mathtt{a}, \mathtt{c}, \mathtt{d})\}$, i.e., a set of sequences of functions that should be called atomically. The analysis will then look for the following pairs of functions that are not called atomically: $\Omega = \{(\mathtt{a}, \mathtt{b}), (\mathtt{b}, \mathtt{c}), (\mathtt{a}, \mathtt{c}), (\mathtt{c}, \mathtt{d})\}$.

An element of this phase's *abstract state* is a triple $(\mathtt{x}, \mathtt{y}, \Delta) \in \Sigma \times \Sigma \times 2^{\Sigma \times \Sigma}$, where $(\mathtt{x}, \mathtt{y})$ is a pair of the most recent function calls, and Δ is a *set of pairs that violate atomicity*. Thus, the abstract state $s \in \boldsymbol{Q}$ is defined as $2^{\Sigma \times \Sigma \times 2^{\Sigma \times \Sigma}}$. Whenever a function `f` is called, it is created a new pair $(\mathtt{x}', \mathtt{y}')$ of the most recent function calls from the previous pair $(\mathtt{x}, \mathtt{y})$ (i.e. $(\mathtt{x}', \mathtt{y}') = (\mathtt{y}, \mathtt{f})$). Further, when the current program state is not inside an atomic block, it is checked whether the new pair (or just the last call) violates atomicity (i.e. $(\mathtt{x}', \mathtt{y}') \in \Omega \vee (\varepsilon, \mathtt{y}') \in \Omega$). When it does, it is added to the set Δ of pairs that violate atomicity.

Formally, the *initial abstract state* of a function is defined as $s_{init} = \{(\varepsilon, \varepsilon, \emptyset)\}$. To formalise the analysis of a function, let `f` be a called leaf function. Further, let $s_{\mathtt{g}}$ be the abstract state of a function `g` being analysed before the function `f` is called. After the call of `f`, the abstract state will be changed as follows: $s_{\mathtt{g}} = \{(\mathtt{x}', \mathtt{y}', \Delta') \in \Sigma \times \Sigma \times 2^{\Sigma \times \Sigma} \mid \exists\, (\mathtt{x}, \mathtt{y}, \Delta) \in s_{\mathtt{g}} : (\mathtt{x}', \mathtt{y}') = (\mathtt{y}, \mathtt{f}) \wedge [(\neg \mathit{lock} \wedge \Delta' = \{(\mathtt{x}'', \mathtt{y}'') \in \Sigma \times \Sigma \mid (\mathtt{x}'', \mathtt{y}'') \in \Delta \vee [((\mathtt{x}'', \mathtt{y}'') = (\mathtt{x}', \mathtt{y}') \vee (\mathtt{x}'', \mathtt{y}'') = (\varepsilon, \mathtt{y}')) \wedge (\mathtt{x}'', \mathtt{y}'') \in \Omega]\}) \vee (\mathit{lock} \wedge \Delta' = \Delta)]\}$.

The analysis of functions with *nested function calls* and cases where *lock/unlock calls not not paired* in functions are handled analogically as in **Phase 1**. For detailed examples, see [13].

Example 3.1.5. To demonstrate the detection of an atomicity violation, assume the functions `f` and `g` from Listing 3.4. The set of atomic sequences of the function `f` with the summary $\chi_{\mathtt{f}} = (\boldsymbol{B}, AB)$ is $\boldsymbol{B} = \{(\mathtt{b}, \mathtt{c})\}$, thus $\Omega = \{(\mathtt{b}, \mathtt{c})\}$. In the function `g`, an atomicity violation is detected because the pair of functions `b` and `c` is not called atomically (under a lock).

Summary of the Phase 2 Like in the first phase of the analysis, **Phase 2** was implemented and validated on a set of purposeful sample programs, as it is described in [13]. The sets of atomicity violations Δ from individual functions are the final reported atomicity violations seen by a user.

```
1  void f()
2  {
3      a();
4      pthread_mutex_lock(&lock); // (b, c)
5      b(); c();
6      pthread_mutex_unlock(&lock);
7      d();
8  }
9  void g()
10 {
11     a(); b(); c(); d(); // ATOMICITY_VIOLATION: (b, c)
12 }
```

Listing 3.4: Example of an *atomicity violation*

3.2 Atomer's Limitations

Atomer was proposed as it is detailed in Section 3.1. The analyser was implemented, and it is working as expected. Moreover, it can be used in practice to analyse various kinds of programs, and it may find *real atomicity related bugs*. Nevertheless, there are still several limitations and cases where Atomer would not work correctly, i.e., cases not considered during the original proposal. Some of these cases were briefly discussed in [13, 15], and further described in [14].

So far, Atomer does not work with *nested locks*, i.e., it does not distinguish *different locks* used in a program. Only calls of locks/unlocks are identified, and parameters of these calls (*lock objects*) are not considered. So, if there are several lock objects used, the analysis does not work correctly. Although this may happen in *real-life programs*, insomuch as one could have another (smaller) atomic section inside a current atomic section (this does not have to be evident at first because the *inner atomic section* could be, e.g., included via a macro). For instance: `lock(A); lock(B); ... unlock(B); unlock(A);`. Another possibility is an *alternating sequence of locks*, e.g., two locks are locked at first, and then, they are unlocked in the same order, i.e., `lock(A); lock(B); ... unlock(A); unlock(B);`.

Atomer considers only *basic contracts for concurrency*, which are defined in Section 2.3.1. It is quite limited in some circumstances and therefore, Atomer can report *false alarms*. The basic contracts do not take into consideration the *data flow* within function calls. However, actually, a better idea is to work with the assumption that a sequence of function calls must be atomic only if they *handle the same data*. Assume that there are functions `f, g` manipulating with the same container `c` as follows: `f(c); g(c);`. These are called atomically. Somewhere else — where `f, g` are not called atomically — it does not necessarily mean atomicity violation because they can be invoked with different arguments, which could be valid. This behaviour corresponds to the *extended contracts with parameters* (see Section 2.3.2). Another, a more complex limitation is that basic contracts do not consider any *contextual information*. It would be more precise to consider as atomicity violations such sequences that could be violated only by particular ("dangerous") function calls, not by any calls. For example, assume that there is the following sequence of functions called atomically: `f(); g();`. While somewhere else, these functions are not called atomically,

it does not necessarily mean that it is an atomicity violation because, in this particular context, non of the "dangerous" functions can be executed by any concurrent thread. The *extended contracts with spoilers* formally describe these cases.

Another limitation of Atomer is that it supports only the analysis of programs written in the *C language* that uses *PThread* locks to *synchronise concurrent threads*. Naturally, in practice, many other *types of locks* for synchronisation of concurrent threads or even *synchronisation of concurrent processes* are used. Although the first version of Atomer can analyse C programs with other types of locks, these locks are not recognised as locks. Thus, the analysis would not work as expected. Of course, it would be useful also to analyse other languages than just C. As described in Section 2.2; *Facebook Infer* is capable of analysing programs written in C, C++, Objective C, and Java (and C#). An analysis algorithm could then be the same for all these languages because the Infer's *intermediate language* is analysed, instead of directly analysing the input languages. Again, Atomer should be able to analyse the above languages, but it was not tested in [13]. However, most importantly, other languages might use *very different locks types*, and these would not be recognised.

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

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

Atomer does not consider a locking using *trylock* functions, i.e., functions equivalent to lock functions, except that if the lock object is currently locked, the call of the trylock shall return immediately, i.e., no waiting. It can be determined from the return value of the trylock whether the lock succeeded or not. These types of locks are used (though not so often) in practice as well. In Atomer, trylocks are so far not identified as locks. The question is how to propose an extension of Atomer that would opportunely handle trylocks.

Regarding the *scalability*, Atomer can have problems with more *extensive* and *complex* programs (problems with the *memory* as well as problems with the *analysis time*). A problem is working with the sets of (A, B) pairs of *sequences* in abstract states, and working with *sequences* of atomic calls in summaries. It may be necessary to store many of these sequences, and they could be very long (due to all different paths through the CFG of an analysed program). This leads to the *state space explosion problem* [26].

Solutions for some of the above problems and limitations were proposed in Chapter 4 and further implemented in a new version of Atomer.

Chapter 4

Proposal of Precision/Scalability Enhancements for Atomer

This chapter describes the proposed solutions for Atomer's limitations stated in Section 3.2, i.e., solutions that enhance *precision* and *scalability* of the analysis performed by Atomer. To formally define these enhancements, notions and symbols introduced in Section 3.1 are used. Some of the enhancements were (informally) described already in [14].

In the following sections, to give an intuition, there are used listings with C programs that use *PThread locks* and assume declared and initialised global variables `lock`, `lockA`, `lockB`, ... (of a type `phtread_mutex_lock`).

Section 4.1 proposes an optimisation of Atomer's scalability. Section 4.2, then covers precision improvements, i.e., an extension of Atomer by additional features that improve its ability to cope with cases that were not supported in the first version of Atomer, and that can be seen in *real-life code*.

In the description in the below Sections, the enhancements stated in the preceding Sections of a given Section are considered.

4.1 Approximation of the Use of Sequences by Sets

Because Atomer can have *scalability* problems when analysing more *extensive* and *complex* programs (problems with the *memory* as well as problems with the *analysis time*), it was proposed the following optimisation. It seems promising to *approximate* (*abstraction refinement*) working with the sets of (A, B) pairs of *sequences* of function calls in *abstract states* (during **Phase 1**) by working with the sets of (A, B) pairs of **sets** of function calls. Elements of these pairs are also occurring in *summaries* of the first phase, and they are used during **Phase 2**. Thus, it is needed to make a certain approximation in these structures and algorithms likewise. The approximated phases of the analysis and its collaboration are illustrated in Figure 4.1 (one can compare that with the illustration of the first version of Atomer in Figure 3.1).

In particular, this proposed solution is more scalable because the order of stored function calls is not relevant anymore while working with sets. Therefore, less memory is required

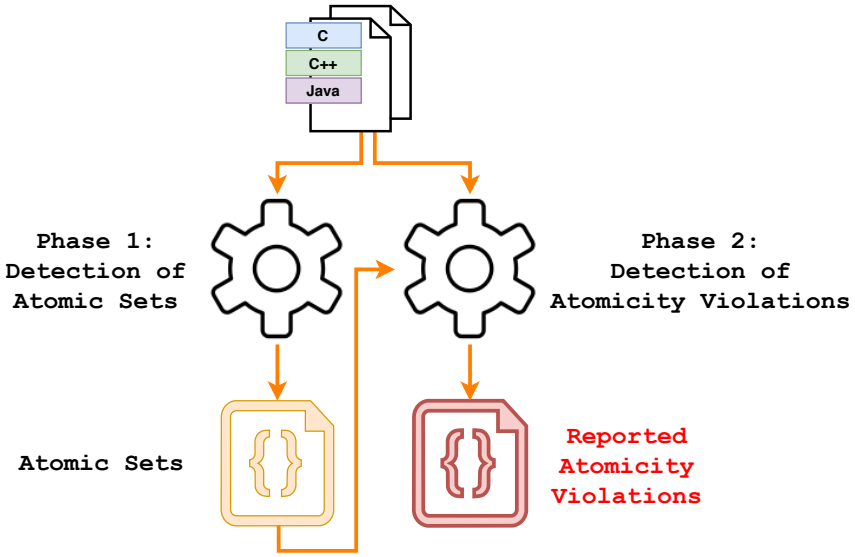


Figure 4.1: An illustration of the *phases* of the Atomer's analysis and the *high-level analysis process* with an *approximation* of working with *sequences* by working with *sets* (moreover, it can be visible that Atomer now accepts programs also written in *C++* and *Java* languages)

because the same sets of function calls are not stored multiple times. The analysis is also faster since there are stored fewer sets of function calls to work with. On the other hand, the analysis is less accurate because the new approach causes some loss of information. In practice, this loss of information could eventually lead to *false alarms*. However, the number of such false alarms should not often be so significant. Moreover, later, there are presented some techniques that try to rid of these false alarms.

4.1.1 Approximation of the Abstract State and the Summary in Phase 1

The *detection of sequences of calls to be executed atomically* is now based on analysing all paths through the CFG of a function and generating all pairs $(A, B) \in 2^{\Sigma} \times 2^{\Sigma}$ (where Σ is a set of all function names in a given program) of **sets** of function calls for each path. Here, A, B are not *reduced sequences* (the notion of a reduced sequence is not needed anymore), but sets, and their semantics is preserved. So, the *abstract state* $s \in \boldsymbol{Q}$ is redefined as $2^{2^{\Sigma} \times 2^{\Sigma}}$.

Further, in all the defined algorithms and definitions, it is sufficient to work with:

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

The above implies that the *initial abstract state* of a function is changed to $s_{init} = \{\{(\emptyset, \emptyset)\}\}$. During the analysis of a function `g` with an abstract state $s_{\mathtt{g}}$, when a leaf function `f` is called, the abstract state's transformation is changed as follows: $s_{\mathtt{g}} = \{p' \in 2^{2^{\Sigma} \times 2^{\Sigma}} \mid \exists\, p \in s_{\mathtt{g}} : p' = \{(A', B') \in 2^{\Sigma} \times 2^{\Sigma} \mid \exists\, (A, B) \in p : (\neg \mathit{actual}(p, (A, B)) \wedge (A', B') = (A, B)) \vee$

$[actual(p,(A,B)) \wedge [(lock \wedge (A',B') = (A, B \cup \{\texttt{f}\})) \vee (\neg lock \wedge (A',B') = (A \cup \{\texttt{f}\}, B))]]\}\}$. Further, when an unlock is called, a new (A,B) pair is created as follows: $s_{\texttt{g}} = \{p' \in 2^{2^\Sigma \times 2^\Sigma} \mid \exists p \in s_{\texttt{g}} : p' = \{(A',B') \in 2^\Sigma \times 2^\Sigma \mid ((A',B') = (\emptyset,\emptyset) \wedge setActual(p,(A',B'))) \vee (A',B') \in p\}\}$. Other definitions (e.g. calling an already analysed *nested* function) will be modified analogically.

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

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

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

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

Whereas, after the approximation, the produced abstract states and summaries are as follows (i.e. there is only one "atomic set", and the summaries and abstract states are the same for both functions because there are the same locked/unlocked function calls, only the order of calls is different):

- $s_{\texttt{f}} = \{\{(\{\texttt{a},\texttt{b}\},\{\texttt{x},\texttt{y}\}),(\emptyset,\emptyset)\}\}$, $\chi_{\texttt{f}} = (\{\{\texttt{x},\texttt{y}\}\},\{\texttt{a},\texttt{b},\texttt{x},\texttt{y}\})$;
- $s_{\texttt{g}} = \{\{(\{\texttt{a},\texttt{b}\},\{\texttt{x},\texttt{y}\}),(\emptyset,\emptyset)\}\}$, $\chi_{\texttt{g}} = (\{\{\texttt{x},\texttt{y}\}\},\{\texttt{a},\texttt{b},\texttt{x},\texttt{y}\})$.

4.1.2 Approximation with Sets in Phase 2

Detecting violations of atomicity works almost the same way as before the approximation. There is only one difference. Before the approximation, it was detected violations of atomic sequences obtained from **Phase 1**. Now, **atomic sets** are obtained; hence, detection of violations of atomic sets is performed. Again, the analysis looks for *pairs of functions that should be called atomically* while this is not the case on some path through the CFG. This algorithm is identical to the algorithm before the approximation.

```
1  void f()
2  {
3      a(); b();
4
5      pthread_mutex_lock(&lock); // (x, y) -> {x, y}
6      x(); y();
7      pthread_mutex_unlock(&lock);
8
9      b(); a();
10
11     pthread_mutex_lock(&lock); // (y, x) -> {x, y}
12     y(); x();
13     pthread_mutex_unlock(&lock);
14 }
15 void g()
16 {
17     b(); a();
18
19     pthread_mutex_lock(&lock); // (y, x) -> {x, y}
20     y(); x();
21     pthread_mutex_unlock(&lock);
22 }
```

Listing 4.1: A code snippet used to illustrate the Atomer's **Phase 1** *approximation* of the analysis with *sets of function calls*

Nevertheless, it is needed to propose a new algorithm that derives the pairs of function calls (from the atomic sets) to be checked for atomicity (i.e. the set $\Omega \in 2^{\Sigma\times\Sigma}$). In order to obtain the pairs, it is taken a union of sets that contain all 2-element *variations* of single atomic sets (i.e. all the possible pairs). Formally, let Q be an analysed program, and let $X_Q \in 2^{2^{2^\Sigma}\times 2^\Sigma}$ be a set of all *summaries* of the program Q. Then, all the atomic pairs (the first item of a pair may be empty if an atomic set consists of a single function) are obtained as follows: $\Omega = \{(\mathtt{x},\mathtt{y}) \in \Sigma\times\Sigma \mid \exists(\boldsymbol{B}, AB) \in X_Q : \exists B \in \boldsymbol{B} : (|B| = 1 \wedge (\mathtt{x},\mathtt{y}) \in \{\varepsilon\}\times B) \vee (|B| > 1 \wedge (\mathtt{x},\mathtt{y}) \in B\times B \wedge \mathtt{x} \neq \mathtt{y})\}$.

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

4.2 Distinguishing Multiple Locks Used

In the first version of Atomer, *different locks* are not distinguished at all. Only calls of locks/unlocks are identified, and parameters of these calls (*lock objects*) are not considered. In order to consider lock objects, it was proposed distinguishing between them using Facebook Infer's built-in mechanism called *access paths*, explained in Section 4.2.1. The analyser does not perform a general *alias analysis*, i.e., it is not performed a precise analysis for saying when arbitrary pairs of accesses to lock objects may alias. During the analysis (both phases), each *atomic section* is identified by an access path of a lock that guards the section, see Sections 4.2.2, 4.2.3. Because *syntactically identical access paths* are used as the intuition for distinguishing atomic sections, some *atomicity violations* could be missed (or some *false alarms* could be reported) due to distinct access paths that refer to the same memory. However, it vastly simplifies the analysis, and the stress is put on finding likely violations.

4.2.1 Access Paths

The *syntactic access paths* [16] represent *heap locations* via the paths used to access them, i.e., a base variable followed by a sequence of fields. More formally, let Var be a set of all variables that can occur in a given program. Let $Field$ be a set of all possible field names that can be used in a given program (e.g. structure fields). Then, an access path π from the set Π of all access paths is defined as follows:

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

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

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

4.2.2 Distinguishing Multiple Locks in Phase 1

The *detection of sets of calls to be executed atomically* is based on generating all pairs $(A, B) \in 2^{\Sigma} \times 2^{\Sigma}$. Now, it is needed to store *access paths* of locks that guard calls executed atomically, i.e., the B sets. Therefore, these pairs are extended to the triples $(A, B, \pi) \in 2^{\Sigma} \times 2^{\Sigma} \times \Pi$, where the third component is an access path that identifies a *lock object* which locks an atomic section that contains the calls from B. Note, that π could also be ε (i.e. $\varepsilon \in \Pi$), which is a special case when there is no lock associated to the (A, B) pair so far, i.e., B is empty as well, and a lock was not called yet. The *abstract state* $s \in \boldsymbol{Q}$ is now defined as $2^{2^{2^{\Sigma} \times 2^{\Sigma} \times \Pi}}$. When a function is called, it is appended to the A set of the triple where $\pi = \varepsilon$, i.e., the triple without an associated lock. Also, it is appended to all the triples that have some lock which is currently locked. When a lock is called, its identifier is associated to the triple without any lock associated to it (which is then labelled as the currently locked lock), and it is created a new triple without a lock. Finally, when an unlock is called, it is created a new triple without a lock, and all the currently locked locks are labelled as unlocked.

Formally, the *initial abstract state* of a function is changed to $s_{init} = \{\{(\emptyset, \emptyset, \varepsilon)\}\}$. During the analysis of a function `g` with an abstract state $s_{\mathtt{g}}$, when a leaf function `f` is called, the abstract state's transformation is changed as follows: $s_{\mathtt{g}} = \{p' \in 2^{2^{\Sigma} \times 2^{\Sigma} \times \Pi} \mid \exists\, p \in s_{\mathtt{g}} : p' = \{(A', B', \pi') \in 2^{\Sigma} \times 2^{\Sigma} \times \Pi \mid \exists\, (A, B, \pi) \in p : (\pi = \varepsilon \wedge (A', B', \pi') = (A \cup \{\mathtt{f}\}, B, \pi)) \vee [\pi \neq \varepsilon \wedge [((A, B, \pi) \in \mathit{locked}(p) \wedge (A', B', \pi') = (A, B \cup \{\mathtt{f}\}, \pi)) \vee ((A, B, \pi) \notin \mathit{locked}(p) \wedge (A', B', \pi') = (A, B, \pi))]]\}\}$, where *locked* is a function that returns (for a given program path) a set of the (A, B, π) triples where the lock identified by π is currently locked. Further, when a lock identified by the access path π_i is called, the abstract state changes as follows: $s_{\mathtt{g}} = \{p' \in 2^{2^{\Sigma} \times 2^{\Sigma} \times \Pi} \mid \exists\, p \in s_{\mathtt{g}} : p' = \{(A', B', \pi') \in 2^{\Sigma} \times 2^{\Sigma} \times \Pi \mid (A', B', \pi') = (\emptyset, \emptyset, \varepsilon) \vee ((A', B', \pi') \in p \wedge \pi' \neq \varepsilon) \vee [(A', B', \varepsilon) \in p \wedge \pi' = \pi_i \wedge \mathit{setLocked}(p, \mathit{locked}(p) \cup \{(A', B', \pi')\})]\}\}$, where *setLocked* is a function that labels triples (for a given program path) as those currently locked by their lock. Furthermore, when an unlock identified by the access path π_i is called, the abstract state changes as follows: $s_{\mathtt{g}} = \{p' \in 2^{2^{\Sigma} \times 2^{\Sigma} \times \Pi} \mid \exists\, p \in s_{\mathtt{g}} : p' = \{(A', B', \pi') \in 2^{\Sigma} \times 2^{\Sigma} \times \Pi \mid (A', B', \pi') = (\emptyset, \emptyset, \varepsilon) \vee ((A', B', \pi') \in p \wedge \pi' \neq \varepsilon \wedge \pi' \neq \pi_i) \vee [(A', B', \pi') \in p \wedge \pi' = \pi_i \wedge \mathit{setLocked}(p, \mathit{locked}(p) \setminus \{(A', B', \pi')\})]\}\}$. Other definitions (e.g. calling and already analysed *nested* function) will be changed analogically.

The *summary* $\chi_{\mathtt{f}} \in 2^{2^{\Sigma}} \times 2^{\Sigma}$ of a function `f` is the same as earlier. Only access paths from abstract states are ignored. I.e. $\chi_{\mathtt{f}} = (\boldsymbol{B}, AB)$, where:

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

Example 4.2.1. Consider two base cases (*nested atomic section* and *alternating sequence of locks*) in function `f` and `g` from Listing 4.2. There are two lock objects `lockA` and `lockB` that are used simultaneously. Further assume, that `a`, `b`, `c` are leaf nodes of the *call graph*. After the extension of the distinguishment of multiple locks used, the produced abstract states and summaries are as follows:

- $s_{\mathtt{f}} = \{\{(\{\mathtt{a}\}, \{\mathtt{b}\}, \mathtt{lockB}), (\emptyset, \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}, \mathtt{lockA}), (\emptyset, \emptyset, \varepsilon)\}\}$,
  $\chi_{\mathtt{f}} = (\{\{\mathtt{b}\}, \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}\}, \{\mathtt{a}, \mathtt{b}, \mathtt{c}\})$;

- $s_{\mathtt{f}} = \{\{(\emptyset, \{\mathtt{a}, \mathtt{b}\}, \mathtt{lockA}), (\{\mathtt{a}\}, \{\mathtt{b}, \mathtt{c}\}, \mathtt{lockB}), (\emptyset, \emptyset, \varepsilon)\}\}$,
  $\chi_{\mathtt{f}} = (\{\{\mathtt{a}, \mathtt{b}\}, \{\mathtt{b}, \mathtt{c}\}\}, \{\mathtt{a}, \mathtt{b}, \mathtt{c}\})$.

```
1  void f()
2  {
3      pthread_mutex_lock(&lockA); // {a, b, c}
4      a();
5      pthread_mutex_lock(&lockB); // {b}
6      b();
7      pthread_mutex_unlock(&lockB);
8      c();
9      pthread_mutex_unlock(&lockA);
10 }
11 void g()
12 {
13     pthread_mutex_lock(&lockA); // {a, b}
14     a();
15     pthread_mutex_lock(&lockB); // {b, c}
16     b();
17     pthread_mutex_unlock(&lockA);
18     c();
19     pthread_mutex_unlock(&lockB);
20 }
```

Listing 4.2: A code snippet used to illustrate *distinguishing multiple locks used* during derivation of *sets of functions called atomically*

4.2.3 Distinguishing Multiple Locks in Phase 2

The pairs Ω of functions that should be called atomically are computed the same way as earlier during the *detection of atomicity violations* in **Phase 2**. The analysis again looks for *pairs of functions that should be called atomically* while this is not the case on some path through the CFG. However, this time, there are stored (in addition to a pair of the most recent function calls) all the most recent pairs of function calls locked under individual locks.

An *abstract state* element is the following: $(\mathtt{x}, \mathtt{y}, \Delta, \Lambda) \in \Sigma \times \Sigma \times 2^{\Sigma \times \Sigma} \times 2^{\Sigma \times \Sigma \times \Pi}$, where $(\mathtt{x}, \mathtt{y})$ and Δ are as before. Λ is a set of *locked pairs* of the most recent function calls with their locks' *access paths*. Thus, the abstract state $s \in \boldsymbol{Q}$ is defined as $2^{\Sigma \times \Sigma \times 2^{\Sigma \times \Sigma} \times 2^{\Sigma \times \Sigma \times \Pi}}$. The analysis works as follows. When a function $\mathtt{f}$ is called, it is created a new pair $(\mathtt{x}', \mathtt{y}')$ of the most recent function calls from the previous pair $(\mathtt{x}, \mathtt{y})$ (i.e. $(\mathtt{x}', \mathtt{y}') = (\mathtt{y}, \mathtt{f})$). This pair is also stored to the locked pairs Λ if there are any locks currently locked. Further, it is checked whether the new pair (or just the last call) violates atomicity, and at the same time, it is not locked by any of the stored locks (i.e. $((\mathtt{x}', \mathtt{y}') \in \Omega \wedge (\mathtt{x}', \mathtt{y}') \notin \Lambda) \vee ((\varepsilon, \mathtt{y}') \in \Omega \wedge (\varepsilon, \mathtt{y}') \notin \Lambda)$). When it holds, it is added to the set Δ of pairs that violate atomicity.

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

abstract state of a function g being analysed before the function f is called. After the call of f, the abstract state will be changed as follows: $s_{\mathtt{g}} = \{(\mathtt{x}', \mathtt{y}', \Delta', \Lambda') \in \Sigma \times \Sigma \times 2^{\Sigma\times\Sigma} \times 2^{\Sigma\times\Sigma\times\Pi} \mid \exists\,(\mathtt{x}, \mathtt{y}, \Delta, \Lambda) \in s_{\mathtt{g}} : (\mathtt{x}', \mathtt{y}') = (\mathtt{y}, \mathtt{f}) \wedge \Lambda' = \{(\mathtt{x}'_\pi, \mathtt{y}'_\pi, \pi') \in \Sigma\times\Sigma\times\Pi \mid \exists\,(\mathtt{x}_\pi, \mathtt{y}_\pi, \pi) \in \Lambda : (\mathtt{x}'_\pi, \mathtt{y}'_\pi, \pi') = (\mathtt{y}_\pi, \mathtt{f}, \pi)\} \wedge \Delta' = \{(\mathtt{x}'', \mathtt{y}'') \in \Sigma \times \Sigma \mid (\mathtt{x}'', \mathtt{y}'') \in \Delta \vee [((\mathtt{x}'', \mathtt{y}'') = (\mathtt{x}', \mathtt{y}') \vee (\mathtt{x}'', \mathtt{y}'') = (\varepsilon, \mathtt{y}')) \wedge (\mathtt{x}'', \mathtt{y}'') \in \Omega \wedge \nexists\,(\mathtt{x}''_\pi, \mathtt{y}''_\pi, \pi'') \in \Lambda' : (\mathtt{x}''_\pi, \mathtt{y}''_\pi) = (\mathtt{x}'', \mathtt{y}'')]\}\}$. Further, when a lock identified by the access path π_i is called, the abstract state is changed as follows: $s_{\mathtt{g}} = \{(\mathtt{x}', \mathtt{y}', \Delta', \Lambda') \in \Sigma \times \Sigma \times 2^{\Sigma\times\Sigma} \times 2^{\Sigma\times\Sigma\times\Pi} \mid \exists\,(\mathtt{x}, \mathtt{y}, \Delta, \Lambda) \in s_{\mathtt{g}} : (\mathtt{x}', \mathtt{y}', \Delta', \Lambda') = (\mathtt{x}, \mathtt{y}, \Delta, \Lambda \cup \{(\varepsilon, \varepsilon, \pi_i)\})\}$. Furthermore, when an unlock identified by the access path π_i is called, the abstract state is changed as follows: $s_{\mathtt{g}} = \{(\mathtt{x}', \mathtt{y}', \Delta', \Lambda') \in \Sigma \times \Sigma \times 2^{\Sigma\times\Sigma} \times 2^{\Sigma\times\Sigma\times\Pi} \mid \exists\,(\mathtt{x}, \mathtt{y}, \Delta, \Lambda) \in s_{\mathtt{g}} : (\mathtt{x}', \mathtt{y}', \Delta', \Lambda') = (\mathtt{x}, \mathtt{y}, \Delta, \Lambda \setminus \Sigma \times \Sigma \times \{\pi_i\})\}$.

Example 4.2.2. Consider the function f from Listing 4.3. There are two lock objects lockA and lockB that are used simultaneously. Further assume, that a, b are leaf nodes of the *call graph*. Then assume, that the result of the first phase of the analysis is that a pair of functions a, b that should be called atomically, i.e., $\Omega = \{(\mathtt{a}, \mathtt{b})\}$. Before the extension of the distinguishment of multiple locks used, the analysis would report an atomicity violation of these functions (line 6). That is because the locks are not distinguished, and the unlock of lockA (line 5) would unlock everything. On the other hand, after the extension, there are not reported any atomicity violations because the pair of functions is locked using the second lock — lockB. The abstract state $s_{\mathtt{f}}$ of the function f before line 7 looks like follows: $s_{\mathtt{f}} = \{(\mathtt{a}, \mathtt{b}, \emptyset, \{(\mathtt{a}, \mathtt{b}, \mathtt{lockB})\})\}$.

```
1 void f()
2 {
3     pthread_mutex_lock(&lockA); // {}
4     pthread_mutex_lock(&lockB); // {a, b}
5     pthread_mutex_unlock(&lockA);
6     a(); b();
7     pthread_mutex_unlock(&lockB);
8 }
```

Listing 4.3: A code snippet used to illustrate *distinguishing multiple locks used* during *detection of atomicity violations*

Chapter 5

Conclusion

This report started by describing the principles of *static analysis* and *abstract interpretation*. Further, *Facebook Infer* was described — a concrete static analysis framework that uses abstract interpretation — its features, architecture, and existing analysers implemented in this tool. Next, there were described *contracts for concurrency*. The major part of the report then aimed at the description of static analyser *Atomer*[1] — proposed and implemented within the author's bachelor's thesis [13] — implemented as a Facebook Infer's module, and that detects *atomicity violations*. It was described its limitations, and thereafter, it was described the proposal of its extensions and improvements.

Atomer works on the level of *sequences of function calls*. It is based on the assumption that sequences of function calls executed *atomically once* should probably be executed *always atomically*, and it naturally works with sequences. In the project practice, to improve *scalability*, the use of sequences was *approximated* by *sets*. Further, two new features were implemented: support for *C++* and *Java* languages; and distinguishing *multiple locks used*.

The introduced enhancements were successfully tested on *hand-crafted* programs. It turned out that such innovations improved the *accuracy* and *scalability*. Moreover, Atomer was experimentally evaluated on additional software. Notably, it was evaluated on *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.

Several other improvements were proposed to reduce the number of false alarms, namely, *parametrisation* of the analysis, support for *interprocedural locks*, or combinations with a *dynamic analysis*. Their implementation and evaluation is currently the work in progress.

Atomer's *accuracy* can be further increased. Some of its limitations and possible solutions are discussed in this report, e.g., considering *formal parameters* and distinguishing the *context* of called functions, *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 to find and report *new bugs*.

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

Bibliography

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

[2] Blackshear, S. Getting the most out of static analyzers. *Speech* [online]. San Jose Convention Center: The @Scale Conference, 2. September 2016 [cit. 2021-01-21]. Available at: https://atscaleconference.com/videos/getting-the-most-out-of-static-analyzers.

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

[4] Blackshear, S. and O'Hearn, P. W. Open-sourcing RacerD: Fast static race detection at scale. *Facebook Engineering* [online]. 19. October 2017 [cit. 2021-01-21]. Available at: https://code.fb.com/android/open-sourcing-racerd-fast-static-race-detection-at-scale.

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

[6] Cousot, P. *Abstract Interpretation* [online]. Revised 5. August 2008 [cit. 2021-01-21]. Available at: https://www.di.ens.fr/~cousot/AI.

[7] Cousot, P. *Abstract Interpretation in a Nutshell* [online]. [cit. 2021-01-21]. Available at: https://www.di.ens.fr/~cousot/AI/IntroAbsInt.html.

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

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

Principles of Programming Languages. Los Angeles, California: ACM Press, New York, NY, January 1977, p. 238–252. POPL'77. DOI: 10.1145/512950.512973. ISBN 9781450373500.

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

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

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

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

[14] Harmim, D. *Static Analysis in Facebook Infer Focused on Atomicity*. Brno, CZ, 2020. Project practice. Brno University of Technology, Faculty of Information Technology. Department of Intelligent Systems. Supervisor Vojnar, T.

[15] Harmim, D., Marin, V. and Pavela, O. Scalable Static Analysis Using Facebook Infer. In: *Excel@FIT*. Brno, CZ: Brno University of Technology, Faculty of Information Technology, 2019. Available at: http://excel.fit.vutbr.cz/submissions/2019/059/59.pdf.

[16] Jones, N. D. and Muchnick, S. S. Flow Analysis and Optimization of LISP-Like Structures. In: *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. San Antonio, Texas: Association for Computing Machinery, New York, NY, USA, January 1979, p. 244—-256. POPL'79. DOI: 10.1145/567752.567776. ISBN 9781450373579.

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

[18] Marcin, V. *Static Analysis of Concurrency Problems in the Facebook Infer Tool*. Brno, CZ, 2018. Project practice. Brno University of Technology, Faculty of Information Technology. Department of Intelligent Systems. Supervisor Vojnar, T.

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

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

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

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

[23] Reps, T., Horwitz, S. and Sagiv, M. Precise Interprocedural Dataflow Analysis via Graph Reachability. In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Francisco, California, USA: ACM, New York, NY, USA, January 1995, p. 49–61. POPL'95. DOI: 10.1145/199448.199462. ISBN 0-89791-692-1.

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

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

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

[27] Vojnar, T. *Different Approaches to Formal Verification and Analysis. Lecture Notes in Static Analysis and Verification*. Brno, CZ: Brno University of Technology, Faculty of Information Technology, 2020. Available at: https://www.fit.vutbr.cz/study/courses/SAV/public/Lectures/sav-lecture-01.pdf.

[28] Vojnar, T. *Lattices and Fixpoints: A Brief Introduction. Lecture Notes in Static Analysis and Verification*. Brno, CZ: Brno University of Technology, Faculty of Information Technology, 2020. Available at: https://www.fit.vutbr.cz/study/courses/SAV/public/Lectures/sav-lecture-05b.pdf.

[29] Yi, K. Inferbo: Infer-based buffer overrun analyzer. *Facebook Research* [online]. 6. February 2017 [cit. 2021-01-21]. Available at: https://research.fb.com/inferbo-infer-based-buffer-overrun-analyzer.