Abstract

Atomer is a *static analyser* based on the idea that if some *sequences of functions* of a *multi-threaded program* are executed *under locks* in some runs, likely, they are *always intended to execute atomically.* Atomer thus strives to look for such sequences and then detects for which of them the atomicity may be broken in some other program runs. The author of this master's thesis proposed and implemented the first version of Atomer as a plugin of the *Facebook Infer framework* in his bachelor's thesis. In the master's thesis, a new and *significantly improved* version of Atomer is proposed. The improvements aim at both increasing *scalability* as well as *precision.* Moreover, support for several initially not supported programming features has been added (including, e.g., the possibility of analysing *C++ and Java programs* or support for *re-entrant locks* or *lock guards*). Through a number of experiments (including experiments with *real-life code* and *real-life bugs*), it is shown that the new version of Atomer is indeed much *more general, scalable, and precise.*

možná bez emph

Abstrakt

Nástroj *Atomer* je *statický analyzátor* založený na myšlence, že pokud jsou některé *sekvence funkcí vícevláknového programu* prováděny v některých bězích *pod zámky*, je pravděpodobně zamýšleno, že mají být *vždy provedeny atomicky.* Analyzátor Atomer se tudíž snaží takové sekvence hledat a poté zjišťovat, pro které z nich může být v některých jiných bězích programu porušena atomicita. Autor této diplomové práce ve své bakalářské práci navrhl a implementoval první verzi nástroje Atomer jako zásuvný modul *aplikačního rámce Facebook Infer.* V této diplomové práci je navržena nová a *výrazně vylepšená* verze analyzátoru Atomer. Cílem vylepšení je zvýšení jak *škálovatelnosti*, tak *přesnosti.* Kromě toho byla přidána podpora pro několik původně nepodporovaných programovacích vlastností (včetně např. možnosti analyzovat *programy napsané v jazycích C++ a Java* nebo podpory pro *reentrantní zámky* nebo *stráže zámků*, tzv. „lock guards"). Prostřednictvím řady experimentů (včetně experimentů s *reálnými programy* a *reálnými chybami*) se ukázalo, že nová verze nástroje Atomer je skutečně *mnohem obecnější, přesnější a lépe škáluje*

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, function calls sequence, multi-threaded programs

Klíčová slova

Facebook Infer, statická analýza, abstraktní interpretace, kontrakty pro paralelismus, porušení atomicity, paralelní programy, analýza programů, atomicita, inkrementální analýza, modulární analýza, kompoziční analýza, interprocedurální analýza, škálovatelnost, Atomer, sekvence volání funkcí, vícevláknové programy

Reference

HARMIM, Dominik. *Advanced Static Analysis of Atomicity in Concurrent Programs through Facebook Infer.* Brno, 2021. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor prof. Ing. Tomáš Vojnar, Ph.D.

Rozšířený abstrakt

Advanced Static Analysis of Atomicity in Concurrent Programs through Facebook Infer

Declaration

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

.....
Dominik Harmim
15th May 2021

Acknowledgements

I thank my colleagues from VeriFIT for their assistance. I would particularly like to thank my supervisor Tomáš Vojnar. I also wish to acknowledge Nikos Gorogiannis from the Infer team at Facebook for valuable discussions about the analyser's development. Lastly, I acknowledge the financial support received from the H2020 ECSEL projects AQUAS, Arrowhead Tools, and VALU3S.

Contents

1 Introduction **1**

2 Preliminaries **4**

2.1 Concepts in Program Analysis 5

2.2 Abstract Interpretation 11

2.3 Facebook Infer — Static Analysis Framework 15

2.4 Contracts for Concurrency 19

3 Atomer — Atomicity Violations Detector **24**

3.1 Related Work 24

3.2 Design of Atomer and Its Principles 25

3.3 Atomer's Limitations 32

4 Proposal of Enhancements for Atomer **35**

4.1 Approximation of Sequences by Sets 35

4.2 Advanced Manipulation with Locks 39

4.3 Analysis's Parametrisation 45

4.4 Local/Global Atomicity Violations 46

5 Implementation of a New Version of Atomer **49**

5.1 Phase 1 — Detection of Atomic Sets 49

5.2 Phase 2 — Detection of Atomicity Violations 49

5.3 Support for C++ and Java Languages 49

5.4 Additional Locking Mechanisms 49

6 Experimental Evaluation of the New Version of Atomer **50**

6.1 Testing on Hand-Crafted Examples 50

6.2 Scalability Benchmark 50

6.3 Evaluation on Validation Programs Derived from Gluon 50

6.4 Experiments with Real-Life Programs 50

6.5 Summary of the Evaluation and Future Work 50

7 Conclusion **51**

Bibliography **52**

A Contents of the Attached Memory Media **56**

B Installation and User Manual **57**

Chapter 1

Introduction

which

Bugs have been present in 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* (~~despite it~~, of course, suffers from some problems too—such as the possibility of reporting many *false alarms*, i.e., *spurious errors*). Quite some 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*. Moreover, the only available checker of *atomicity of call sequences* is the first version of *Atomer* [15] proposed in the bachelor's thesis of the author.

At the same time, in *concurrent programs*, there are often *atomicity requirements* for the 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. Therefore, 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

Je to dost dlouhý monoblok – dal bych sem nějaké pojmenované odstavce: sem možná \paragraph{Atomer.}?

of calls

laborious and time-consuming is finding and fixing them. The papers [11, 13, 33, 37] discuss the importance of *atomicity-related bugs*, and they also show some bugs in *real-world programs*. Unfortunately, tool support for automatically discovering such kinds of errors is currently minimal.

Atomer puts stress

As already mentioned, within the author's bachelor's thesis [15], *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. For example, assume the function `replace` from Listing 1.1 that replaces item `a` in an array by item `b`. It contains an atomicity violation — the index obtained may be outdated when `set` is executed (because, e.g., a *concurrent thread* can modify the array), i.e., `index_of` and `set` should be executed atomically. The analysis is based on the assumption that sequences of function calls executed *atomically once* should probably be executed *always atomically*. Hence, the checker naturally works with sequences. In fact, the idea of checking the atomicity of certain sequences of function calls is inspired by the works of *contracts for concurrency* [11, 33]. In the terminology of [11, 33], the atomicity of specific sequences of calls is the most straightforward (yet very useful in practice) kind of contracts for concurrency. However, while the idea of using sequences in the given context is indeed natural and rather exact, it quite severely limits the *scalability* of the analysis (indeed, even with a few functions, there can appear numerous different orders in which they can be called). Moreover, the implementation of the first version of Atomer targets mainly *C programs* using *PThread locks*. Consequently, there was no support for other languages and their locking mechanisms in the first version of Atomer.

rozbít odstavec?

```
1 void replace(int *array, int a, int b)
2 {
3     int index = index_of(array, a);
4     if (index >= 0) set(array, index, b);
5 }
```

Listing 1.1: An example of an *atomicity violation*

\paragraph{New developments of Atomer}?

Within this thesis, Atomer has been *significantly improved and extended*. In particular, to improve scalability, working with *sequences* of function calls was *approximated* by working with *sets* of function calls. Furthermore, several new features were implemented: support for *C++ and Java*, including various advanced kinds of *locks* these languages offer (such as *re-entrant locks* or *lock guards*); or a more precise way of *distinguishing between different lock instances*. Moreover, the analysis has been *parameterised* by function names to concentrate on during the analysis and limits of the number of functions in *critical sections*. These parameters aim to reduce the number of false alarms. Their proposal is based on the author's analysis of false alarms produced by the first Atomer's version. Lastly, new experiments were performed to test capabilities of the new version of Atomer.

The development of the original Atomer started under the H2020 ECSEL projects AQUAS and Arrowhead Tools. The development of its new version is supported by the H2020 ECSEL project VALU3S. It has been discussed with the developers of Facebook Infer too. Parts of the thesis concerning the preliminaries and the basic version of Atomer are partially

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

– \paragraph{Acknowledgment}?

– Dal bych to až jako poslední odstavec

introduces basics of all..

taken from the thesis [15]. Moreover, some preliminary results were also published in the Excel@FIT'21 paper [16] written by the author.

The rest of the thesis is organised as follows. Chapter 2 ~~describes all~~ the topics related to and essential to this thesis (including *static analysis*, *abstract interpretation*, *Facebook Infer*, and *contracts for concurrency*). The original version of *Atomer*, its limitations, and related work are described in Chapter 3. Subsequently, Chapter 4 presents all the proposed extensions and improvements. The implementation of these extensions is then covered in Chapter 5. The experimental evaluation of the new Atomer's features and other experiments performed within this thesis are discussed in Chapter 6 together with the future work. Finally, the thesis is concluded in Chapter 7. Besides, there are included the following appendices. The content of the attached memory media is listed in Appendix A. In the end, Appendix B serves as an installation and user manual.

\paragraph {Plan of the thesis}
+ dal bych to před podkapitolu

toto bych dal asi jako samostatný odstavec za plan of the thesis.

\para {Plan of the rest..
—
—
Parts of ...

\para {Ack...}

However, we still tried to improve the description when we found this appropriate.

Chapter 2

Preliminaries

Not only theory!

This chapter ~~explains the theoretical~~ provides the background that the thesis builds on. It also explains and describes the existing tools used in the thesis. ~~Also,~~ Note that the author already partially published the contents of the sections in this chapter in his bachelor's thesis [15] and paper [16]. — Indeed, the background of the original and new versions of Atomerace the same.

Multi-Threaded Programs [13, 26] *Multiple threads* of control are commonly used in the software development process because they help reduce *latency*, increase *throughput*, ~~and better utilise *multiple processor computers*.~~ Threads or *processes* ~~are~~ contain independent *sequences of instructions* that may be performed simultaneously. A process represents a single ~~running program~~ with threads sometimes called light-weight processes. It has its own address space and a unique identifier. One process can consist of multiple threads, ~~i.e., a thread is a so-called *lightweight process*.~~ All threads of a single process share the same address space ~~(i.e., code and data)~~ for... (not stack!!). However, reasoning about the behaviour and correctness of a multi-threaded system is complex due to the need to consider all possible interleavings of the executions of different threads. An integral part of parallel programming is the *synchronisation* of individual threads. Usually, synchronisation mechanisms ensure the *mutual exclusion* of shared resources or synchronise actions that threads perform. Operating systems provide basic synchronisation primitives that can be often used in programming languages in a higher-level manner. Such fundamental mechanisms are *semaphores* (binary semaphores are called *mutexes*), *barriers*, *read-copy-update* techniques, or *monitors*. In this work (and also in practice), for simplicity, ~~it will be used~~ we will use a notion of *locks* for semaphores, mutexes, monitors, etc. as a mechanism that mutually excludes access to a *critical section*.

The use of locks and access to shared data in parallel programs involves the risk of errors not known in sequential programming. Much previous work on checking *thread interference* has focused on *data races* ~~(or, in general, on *race conditions*)~~. A data race occurs when two threads simultaneously access the same data variable, and, at least, one of the access is a write. In practice, data races are commonly avoided by guarding each data structure with a lock. Unfortunately, the absence of data races is insufficient to guarantee the absence of errors due to unexpected interference between threads. [13, 37]

Atomicity Violations [13] Another possible source of errors due to unexpected interference between threads are *atomicity violations*. A method (or, in general, a code block)

race conditions include deadlocks ...

makes it

number

is *atomic* iff, for every (arbitrarily interleaved) program execution, there is an equivalent execution with the same overall behaviour where the atomic method is executed serially. In other words, the method's execution is not interleaved with instructions of other threads. Also, atomicity provides a powerful, indeed maximal, guarantee of non-interference among threads. In short, atomicity is a generally applicable and fundamental *correctness property* of multi-threaded code. Nevertheless, commonly used *testing approaches* are deficient in verifying atomicity. While testing may reveal a concrete interleaving in which an atomicity violation causes erroneous behaviour, the exponentially large ~~amount~~ of possible interleavings ~~does~~ fundamentally impossible to get suitable *test coverage*. Research results [11, 13, 33, 37] have shown that defects related to atomicity are common, even in well-tested libraries.

In order to be able to check for atomicity violations, it must of course be known which blocks ...

It has to be described which blocks of code should be executed atomically to detect atomicity violations. Algorithms for the detection of ~~this error~~ often deal only with the atomicity of operations on variables. Examples of a complex description of *atomicity requirements* are *contracts* that require atomicity of the execution of certain methods/functions (defined in Section 2.4). [26]

! Never outside of sentence. | atomicity violations

Further, Section 2.1 outlines the fundamental notions and approaches in *program analysis*. In Section 2.2, there is an explanation of *static analysis* by *abstract interpretation*, which is used in *Facebook Infer*, i.e., the key framework used in this thesis. Facebook Infer, its principles, and features are covered in Section 2.3. The concept of *contracts for concurrency* is discussed and defined in Section 2.4.

An approach that allows for describing complex usage of ...

2.1 Concepts in Program Analysis

This section provides a fundamental intuition about the main principles of *program analysis*. It discusses several standard techniques for reasoning about programs. The section is based on a few first chapters from the book [31] and the overview paper [19].

2.1.1 What to Analyse

the

The first question is *what programs to analyse*. An obvious characterisation of programs to analyse is the programming language in which the programs are written. Moreover, certain specific families of input programs may be distinguished. However, besides the language and the family of programs to consider, the way input programs are processed can also differ and affects how the analysis works. The most straightforward way to handle input programs is working directly with ~~a~~ *source code* just like a compiler would. Nevertheless, different representations of programs are used in program analysis likewise. The two below types of techniques are customarily distinguished.

? often ?

Program-Level Analysis The first possibility is to run the analysis on ~~a~~ the *source code* of input programs (e.g., programs written in conventional programming languages like C or Java; or hardware described in VHDL, Verilog, etc.) or on *executable binaries*. This technique typically involves some *front-end* comparable to a compiler's one that constructs the *syntax trees* of input programs.

(and possibly other representations such as control flow graphs)

details?

Nějak mi to sem nesedí.

to extract them from source code

Model-Level Analysis An alternative option is to consider a different input language that aims at modelling the semantics of programs. Then, the analysis takes as an input a description that *models* the program to analyse. Such models either need to be created manually, or some specialised tools are used. These models may hide implementation difficulties or inaccuracies. Examples of the models are automata of any kind, UML diagrams, Petri nets, Markov chains, or specialised modelling languages like, e.g., Promela.

2.1.2 Static vs Dynamic Approaches

Another critical question in program analysis is *when* the analysis is made. In particular, whether it operates *during* or *before* the program's execution.

One solution is to analyse programs at *run-time*, i.e., *during* the program's execution. This approach is called *dynamic analysis*. It takes place while the program computes, often, over a number of executions.

~~A second~~ Another approach is to analyse the program *before* execution, which is called *static analysis*. It is done independently from any execution. (at least under the original semantics).

Static and dynamic approaches are significantly distinct in many ways. They come with different benefits and weaknesses. While dynamic approaches are often simpler to design and implement, they often have problems with performance at run-time. They do not force developers to fix bugs before program execution. Moreover, some properties cannot be checked (or, at least, it is very challenging) dynamically. For instance, dynamically detecting whether an execution does not *terminate* would require an infinite program run. Particular static and dynamic techniques are further discussed in Section 2.1.5.

BETTER mention that: Dynamic approaches do have limited coverage of program behaviour since they build on what they see in program executions that were performed.

NO! One can take snapshots of memory & check for loops — explore real paths.

⊕ They may have more information about concrete states (aliasing...) & thus avoid some false alarms.

2.1.3 Automation and Scalability

Automation is a further relevant aspect in program analysis. It would be ideal if program analysis methods were *fully automated* (i.e., no human help is needed). Unfortunately, this is not always possible due to the consequences of the *Rice's theorem* [30]. Thus, sometimes, it is needed to give up on automation and let program analyses ask some *user input* (i.e., some human help is required). In that case, the user is asked to give some information to the analysis, e.g., *invariants*[1]. That is, the analysis is partially manual since users need to compute parts of the results themselves. However, having to provide this information may be unwieldy because input programs can be huge or complex.

Scalability is another essential characteristic of program analysis algorithms. Even if a program analysis is fully automatic, it is not guaranteed that it will generate a result within a reasonable time, depending on the complexity of the algorithms. A program analysis tool may not *scale* to extensive programs due to time costs or, e.g., memory constraints.

2.1.4 Soundness and Completeness

In order to preserve automation and/or scalability, the conditions about program analysis may be relaxed. Namely, the analysis can be proposed to return *inaccurate* results (altern-

[1] An **invariant** is a *logical property* that can be proved to be inductive for a given program.

atively, it can return a non-conclusive "do not know" answer). For this purpose, two dual properties (forms of *approximations* or *inaccuracies*) are used. To express these notions, let $\mathcal{L}$ be a *Turing-complete language*, φ be a *non-trivial semantic property* of interest of programs of $\mathcal{L}$, and T be an analysis tool that decides whether φ holds in a given program. Ideally, if T were absolutely precise, it would be such that:

$$\text{for every program } P \in \mathcal{L} : T(P) = \mathit{true} \iff P \models \varphi$$

The above, of course, can be decomposed into a pair of implications:

$$\begin{cases} \forall P \in \mathcal{L} : T(P) = \mathit{true} \implies P \models \varphi \\ \forall P \in \mathcal{L} : T(P) = \mathit{true} \impliedby P \models \varphi \end{cases}$$

Soundness A *sound* program analysis satisfies the first implication.

Definition 2.1.1 (Soundness [31]). The program analyser T is **sound** w.r.t. property φ whenever, for any program $P \in \mathcal{L}$, $T(P) = \mathit{true} \implies P \models \varphi$.

When a sound analysis terminates and claims that the analysed program has property φ, it guarantees that the program indeed satisfies φ, i.e., no errors are missed. In other words, a sound analysis will refuse all programs that do not satisfy φ. In terms of errors in *binary classification*, there are not *false-negative* errors[2] when using sound analysis. On the other hand, there is a chance of *false-positive* errors[3].

Completeness A *complete* program analysis satisfies the second implication.

Definition 2.1.2 (Completeness [31]). The program analyser T is **complete** w.r.t. property φ whenever, for any program $P \in \mathcal{L}$, $P \models \varphi \implies T(P) = \mathit{true}$.

A complete program analysis will accept every program that satisfies property φ. In other words, when a complete analysis refuses an analysed program, it is guaranteed that the program indeed fails to satisfy φ, i.e., there are no false-positive errors. However, there can be false-negative errors.

Due to the *computability* barrier [30], it is not possible to design a general analysis to determine which programs satisfy any non-trivial property for a Turing-complete language that is sound, complete, fully automated, and scalable at the same time. Some of these have to be sacrificed, or the analysis has to be proposed to operate only on a specific set of input programs.

2.1.5 Program Analysis Techniques

This section describes several standard *program analysis techniques*. Although this thesis focuses on *static analysis* (or maybe rather *bug finding* based on static analysis), it is essential to see the differences between other analysis techniques. Hence, a brief overview is presented in the below sections. Finally, Table 2.1 compares the techniques (how they are usually used) based on the earlier criteria.

[2] A **false-negative** error is a real error that is undetected by an analysis tool.

[3] A **false-positive** error (also called a **false alarm**) is a *spurious error*, i.e., it is detected by an analysis tool, but the error does not exist in the real program.

* Note, however, that this needs not hold for extrapolation based dynamic analyses such as the already mentioned Eraser & FastTrack

[ ] a citation would be nice. [ ]

Testing and Dynamic Analysis The *testing* approach checks a finite set of finite program executions. In the development process, several levels of testing are used at various stages of the software/hardware life-cycle, e.g., *unit testing* or *integration testing*. In general, it is challenging to achieve suitable *test coverage* because of infinite program paths when using, e.g., *random testing*. However, several more advanced techniques that improve the coverage have been introduced. These techniques are often combined with other verification approaches. For instance, *concolic testing* combines testing with *symbolic execution*, *dynamic analysis* that observes behaviour in a testing run (such behaviour can be *extrapolated* to behaviour not seen in the given testing run), or *search-based techniques* that can generate test data or parameters. Moreover, to test programs with non-deterministic semantics (i.e., concurrent programs), techniques like *noise injection* are applied. In particular, many advanced dynamic analysers have been proposed to detect data races or deadlocks, including, e.g., Eraser or FastTrack.

Testing has the following features. In general, it is simple to automate. It is unsound in almost all cases (besides programs that have a finite number of finite paths). Since failed testing runs provide incorrect concrete executions, the testing is complete. *

Other analysis techniques mentioned below use a static approach. The significant difference between static and dynamic approaches is the following. It is well-known that testing 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 approaches may be able to prove their absence — with some *approximation* — they can check *all possible executions* of a program and provide guarantees about its properties. Another static approaches' benefit 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 drawback of static approaches, in general, is that they can produce a lot of false alarms (though this is not the case, e.g., for *theorem proving*), which is often resolved by accepting unsoundness. Another crucial issue of static approaches (this is, however, also an issue of dynamic analyses) is ensuring sufficient scalability — in fact, typically, the more precise the analysis, the less scalable it becomes.

or model checking

(usually)

Deductive Verification The *semi-automated* approach of *deductive verification* (or *theorem proving*) uses *inference systems* for inferring theorems about the analysed system from the facts known about the system and from general theorems of different logical theories. The approach falls under *machine-assisted* techniques, which means that users may be required to provide extra information to the analysis (usually *loop invariants*, procedure *pre-conditions/post-conditions*, assertions, ~~or~~ some other invariants). This can be demanding and require some level of expertise. However, a substantial part of the verification can usually still be carried out in a fully automated fashion. This approach is very general, but there is a problem with generating diagnostic information for incorrect systems. There exist a number of *interactive theorem proving* tools like Coq, Isabelle/HOL, PVS, ACL2, etc. The user usually guides the inference process in these tools.

predicate

These techniques also involve *automated decision procedures* (or *satisfiability solvers*) for various logical theories. Such solvers are often used as back-end components for higher-level verification methods, such as *symbolic execution* or *predication abstraction* in model checking. Commonly used solvers are *SAT-solvers* (e.g., CaDiCaL and Glucose) and *SMT-*

a very high

, or they may be required to guide the tool by selecting which inference should be done from what facts and when)

To sum up,

I am not sure what you want to say here.

solvers (e.g., Z3 and CVC4). Various tools allow the user to provide some *logical annotations* in a code and then automatically attempt to prove specific properties using decision procedures. Examples of such tools are VCC and Dafny.

(usually)

Theorem proving techniques have the following properties. They are not fully automatic, i.e., *high user expertise* is often needed. They are sound w.r.t. the model of the program used in the proof, and they are usually complete up to the capabilities of the proof assistant.

primarily

Well, currently, many model checkers work on the source-code level.

Model Checking Another technique called (*finite-state*) *model checking* aims at finite systems.* It automatically verifies whether a system or its model satisfies a particular property based on an *algorithmic exploration of the system's state space.* Unfortunately, the biggest issue here is the *state space explosion* problem [35]. However, in practice, model checking tools use effective data structures (such as *binary decision diagrams* or *hierarchical storage of states*) to describe program behaviours and avoid enumerating all executions thanks to approaches that reduce the search space. In addition, other techniques are used to cope with this problem. For instance, various abstractions are used (e.g., *predicate abstraction*) or *bounded model checking*[4]. Properties are usually defined using *temporal logics*, such as LTL, CTL, or PCTL. Finite model checking has the following characteristics. It is automatic (up to the need of modelling the system or its environment). It is sound and usually complete, w.r.t. the model. Other advantages are that it is pretty general and provides diagnostic data for incorrect systems. ???

Really? Not sure – I'd better remove this.

often

needs not

Model checking is ~~typically~~ done at the *model level*, i.e., a model of the program needs to be built, either manually or automatically. In practice, model checking tools usually implement a front-end for that purpose. The problem is that the model ~~generally cannot~~ precisely capture the input program's behaviours ~~since programs are usually infinite systems~~. Thus, the checking of the ~~synthesised~~ model may be either incomplete or unsound, w.r.t. the input program. Some model checking techniques can automatically *refine* the model when they fail to prove a property due to a *spurious counterexample*, ~~although a *termination* must be ensured~~. In practice, model checking tools are often sound and incomplete w.r.t. the input program.

Even if the program itself needs not be modelled, one usually needs a model of its environment

???

CEGAR usually does not have any term. guarantees!

Model checking has found many successful applications, including hardware verification, verification of *concurrent* and *distributed* systems, *probabilistic* systems, *biological* systems, etc. Examples of hardware model checkers are RuleBase, Incisive Verifier, or NuSMV. Model checkers for concurrent and distributed systems include Spin or DIVINE. Tools, such as CPAchecker or BLAST, are model checkers that use predicate abstraction. PRISM and Storm are state-of-the-art tools for model checking probabilistic systems. Finally, Uppaal is a model checker that verifies temporal logic formulas on *timed automata.*

Tady by to asi chtělo cíleně (i v detailech) probabilistic → odstranit

Static Analysis The last technique (and the most important one for the thesis) is *static analysis.* It relies on other techniques to compute *conservative* (sound but incomplete) descriptions of program behaviours using finite resources rather than building a finite model. The fundamental idea is to finitely *over-approximate* the set of all program behaviours using a particular set of properties.

[4]**Bounded model checking** explores models up to a *fixed depth* only. This technique is often referred to as *bug finding* because it sacrifices completeness and often also soundness. Examples of tools that implement such technique are CMBC, LLBMC, or JBMC.

* (extensions to infinite-state systems have also been considered – e.g., m.c. with predicate abstraction, regular m.c., ...)

According to [27], static analysis of programs is reasoning about the behaviour of computer programs without really executing them. It has been used since the 1960s in *optimising compilers* for generating efficient code. More recently, it has proven valuable also for automatic error detection, verification of the 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 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 leading concepts in *formal verification*. It can even automatically answer considerably complicated questions about a given program, such as [27]:

- Does the program *terminate* on every input?
- Do two pointers refer to *disjoint data structures* in memory?
- Are *data races* possible? Can the program *deadlock*?
- Does there exist an input that leads to a *null-pointer dereference*, *division-by-zero*, or *arithmetic overflow*?
- Are arrays always accessed *within their bounds*?
- **Are certain operations executed *atomically*?**

In general, conservative static analysis has the following characteristics. It is automatic, it can often handle large systems, and it does not ordinarily need a model of the environment. It produces sound results, and it is generally incomplete, i.e., it can produce many false alarms. However, it is possible to drop soundness to minimise the number of false alarms and preserve automation, which, in fact, is done by many tools. Such techniques are often instead called *bug finding* or *bug hunting*, which is based not only on static analysis but also, e.g., on model checking. Since the primary motivation of these approaches is to discover bugs (and not to prove their absence), they are neither sound nor complete, and they aim at swiftly catching bugs.

The above explanation of static analysis is quite general. Thus, even model checking or deductive verification may be considered as static analysis. However, when setting these approaches aside, the most traditional techniques in the static analysis include:

- *Syntactic checks* looking for various *bug patterns* (*anti-patterns*) — implemented, e.g., in Lint, Cppcheck, or in analyses in VisualStudio, Clang, GCC.
- *Data-flow analysis* — it is often combined with bug pattern searching, and it is usually unsound. Many successful tools of this kind have been implemented. They are often proprietary and hard to openly evaluate or extend. Examples of some well-known tools are Coverity, CodeSonar, Klocwork, FindBugs/SpotBugs, SonarQube.
- *Constraint-based analysis*, *type-based analysis* (type and effect systems).
- *Symbolic execution* — implemented, e.g., in KLEE, Pex, Symbiotic.

Are you sure? Isn't it more about abstract interpr.?

can be sound but it is often combined with... and used in an unsound way in the end.

Extrapolate
Dynamic analysis Yes No No Program + test suite Dynamic

- *Abstract interpretation* (see Section 2.2) — examples of sound tools are Polyspace, AbsInt/Astrée, or Sparrow. State-of-the-art frameworks that allow creating sound analysers include Facebook Infer (see Section 2.3), Facebook SPARTA, or Frama-C. These tools also offer already implemented sound as well as unsound checkers.

Table 2.1: A summary of *program analysis techniques* [31]

extended by the line on dyn. analysis and also in the object column.

Technique	Automatic	Sound	Complete	Object	Approach
Testing	Yes	No	Yes	Program	Dynamic
Deductive verification	No	Yes	Yes/No	Model/Program	Static
Model checking	Yes	Yes	Yes/No	Model/Program	Static
Conservative static analysis	Yes	Yes	No	Program	Static
Bug finding	Yes	No	No	Program	Static

Dodat veku odkazů na tabulku. Properties of ... are summarised in table...

+ test suite
+ test suite/ model of env.

2.2 Abstract Interpretation

the

This section explains and defines the basics of static analysis technique *abstract interpretation*. The description is based on [6, 7, 8, 9, 10, 19, 21, 27, 28, 31]. In these works, there can be found a more detailed and formal explanation.

the of

Abstract interpretation was introduced and formalised by French computer scientist Patrick Cousot and his wife Radhia Cousot at POPL'77 [9]. It is a generic *framework* for static analyses. It allows one to create particular analyses by providing specific *components* (discussed in Section 2.2.1) to the framework. The obtained analysis is guaranteed to be *sound* if specific properties of the components are met.

In general, in *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. *Fixpoints* may characterise those behaviours (see Section 2.2.3), which is why the primary part of the theory provides efficient techniques for *fixpoint approximation* [28]. ~~Therefore, for standard semantics,~~ abstract interpretation is used to derive the approximate abstract semantics over an *abstract domain* (explained in Section 2.2.1). The abstract semantics obtained from program analysis can then be used for verification, optimisation, code generation, etc. [8].

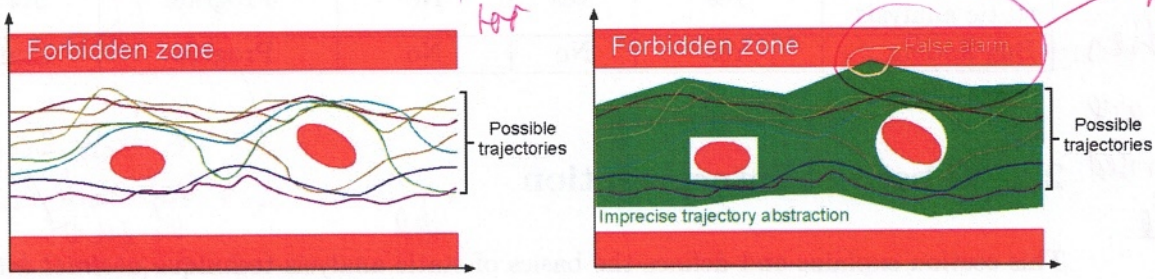
⊕ sometimes test suite/ model of env.

Menu

To be sound, it is essential that an approximation performed using abstract interpretation should be an *over-approximation*. This means that the considered abstract semantics should be a *superset* of the concrete semantics, i.e., the abstract semantics should cover all possible cases. Whence, if the abstract semantics satisfies a given property, then the concrete semantics satisfies it too. Moreover, the consequence of the over-approximation of the possible executions is that inexisting executions are considered, leading to *false alarms*. Thus, due to a lack of accuracy, abstract interpretation is usually *incomplete*. [7]

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, 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 zones is empty may be *undecidable* because the program concrete semantics is, in general, *not computable*. As demonstrated in Figure 2.1b, abstract interpretation deals with the 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 zones — the concrete semantics is safe as well. However, the over-approximation of the possible program executions causes that inexisting program executions are considered, which leads to false alarms, as demonstrates the figure. It is the case when the abstract semantics intersects the forbidden zone, whereas the concrete semantics does not intersect it.



(a) The *concrete semantics* of programs with *forbidden zones*.

(b) The *abstract semantics* of programs with *imprecise abstraction*.

Figure 2.1: An *abstract interpretation* demonstration [7] (horizontal axes: the time t; vertical axes: the vector $x(t)$ of input, state, and output values of the considered program).

2.2.1 Components of Abstract Interpretation

Before the formal definition of abstract interpretation is given, below are intuitively described basic components of the framework in accordance with [6, 8, 21, 28]:

- **Abstract Domain Q**:
  - An abstraction of the possible concrete program states (or their parts) in the form of *abstract properties*[5].
  - In other words, it is a set of *abstract states* (or *abstract contexts*), where an abstract state represents a set of program states reachable at a particular program location.
  - For instance, several practical domains have been defined: numerical intervals, polyhedra, octagons, congruences, or various heap domains.
- **Abstract Transformers τ**:
  - There is a *transform function* τ for each program operation (instruction) representing the impact of the operation executed on an abstract state.

[5] **Abstract properties** approximate *concrete properties* behaviours. [8]

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

Note that neither the widening operator nor narrowing operator are required. However, at least the widening operator is frequently used. The narrowing operator can be sometimes dropped.

2.2.2 Formal Definition of Abstract Interpretation

The definitions below consider notions from *lattice theory*. More information about lattices, functions on lattices, *partial orders*, and set theory can be found, e.g., in [27, 28, 31].

Definition 2.2.1 (Abstract Interpretation). According to [9, 21], **abstract interpretation** I of a program P with the instruction set $Instr$ is a tuple

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

where

- Q is the *abstract domain* (the set 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 $x \sqsubseteq y \iff x \sqcup y = y$ where
  - $\langle Q, \sqsubseteq \rangle$ is a *complete* $\sqcup$*-semilattice*,
- $\top \in Q$ is the *supremum* of $\langle Q, \sqsubseteq \rangle$,
- $\bot \in Q$ is the *infimum* of $\langle Q, \sqsubseteq \rangle$ — thus, $\langle Q, \sqsubseteq \rangle$ is, in fact, a *complete lattice*,
- $\tau : Instr \times Q \to Q$ defines the *abstract transformers* for specific instructions,
  - τ must be *monotone*[6] on Q for each instruction from $Instr$.

[6] Assume two *partially-ordered sets* $\langle E, \preceq_E \rangle$ and $\langle F, \preceq_F \rangle$, and a function $f : E \to F$. f is **monotone** iff $\forall x, y \in E : x \preceq_E y \implies f(x) \preceq_F f(y)$. [31]

Using a so-called *Galois connection* [9, 10, 21, 27, 28, 31], one can ensure the *soundness* of the abstract interpretation, i.e., the correspondence between the *concrete semantics* and its *abstract semantics* can be formalised by a Galois connection (also a so-called pair of *adjoined functions*).

referred to as a

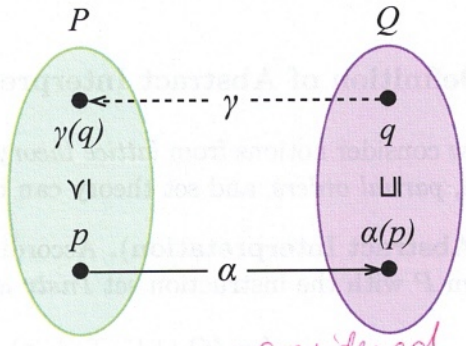
Definition 2.2.2 (Galois Connection). If $\mathcal{P} = \langle P, \preceq \rangle$ and $\mathcal{Q} = \langle Q, \sqsubseteq \rangle$ are *partially-ordered sets*, then a quadruple $\pi = (\mathcal{P}, \alpha, \gamma, \mathcal{Q})$ is a **Galois connection**, written

$$\langle P, \preceq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle Q, \sqsubseteq \rangle$$

iff $\alpha : P \to Q$ and $\gamma : Q \to P$ are *monotone* functions such that

$$\forall p \in P : \forall q \in Q : \alpha(p) \sqsubseteq q \iff p \preceq \gamma(q)$$

where P is the concrete domain and Q is the abstract domain. Furthermore, α is an *abstraction function*, and γ is a *concretisation function*. Consequently, $\alpha(p)$ is the abstraction of p, i.e., the most precise approximation of $p \in P$ in Q, and $\gamma(q)$ is the concretisation of q, i.e., the most imprecise element of P which $q \in Q$ can soundly approximate. A Galois connection is illustrated in Figure 2.2.



considered

Figure 2.2: An illustration of a *Galois connection*

If the abstraction and concretisation functions of the abstract interpretation form a Galois connection, then applying the abstraction function and concretising the result back yield a less (or equally) precise result. However, it is a safe (*conservative*) approximation. That is, formally written, $\forall p \in P : p \preceq \gamma(\alpha(p))$. Finally, the abstract interpretation may only over-approximate the concrete semantics (i.e., it is sound) if for each *concrete transformer* (instruction) $i : P \to P$ from the instruction set $Instr$ and the appropriate *abstract transformer* $\tau : Instr \times Q \to Q$ respects a Galois connection. This is $\forall p \in P : \alpha(i(p)) \sqsubseteq \tau(i, \alpha(p))$.

the

2.2.3 Fixpoint Approximation [9, 10, 21, 27, 28, 31]

Most program properties can be represented as *fixpoints*[7]. This reduces the program analysis to the *fixpoint approximation*. The complete analysis of a program using abstract interpretation can be then viewed as finding the *least/greatest* fixpoint of the equation

[7] Consider a function $f : L \to L$ on a set L. A **fixed point** (or **fixpoint**) of f is an element $l \in L$ iff $f(l) = l$. The set of all fixpoints of f is denoted as $Fix(f) = \{l \in L \mid f(l) = l\}$. [28]

such that

(provided that $\overline{\tau}$ is monotone & we work with complete lattices)

$\overline{Q} = \overline{\tau}(\overline{Q})$, where $\overline{Q}$ is a vector of abstract states and $\overline{\tau}$ is an extension of τ to the whole program. This fixpoint equation is then solved iteratively. *Knaster-Tarski theorem* [34] guarantees these fixpoints' existence.

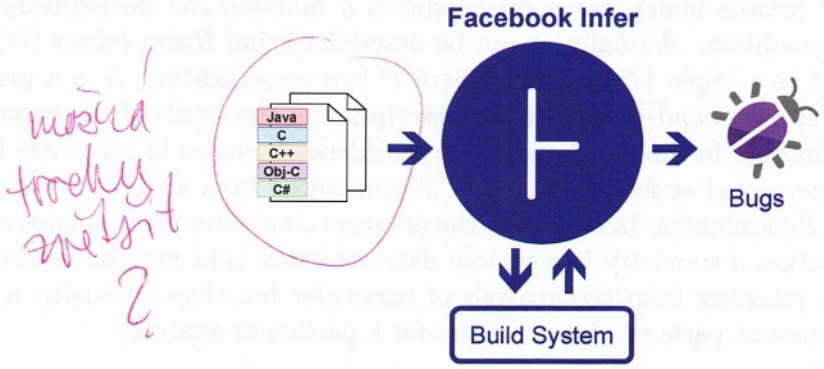
The computation of the most precise abstract fixpoint is not generally guaranteed to *terminate*, in particular, when a given program contains a loop (or recursion) and uses an infinite domain (or even finite but very large). In order to enforce or accelerate the convergence, the fixpoint is often over-approximated using the *widening* operator ∇. The approximation may be later refined using the *narrowing* operator Δ. These two operators are defined below. In practice, the analysis is usually done by iterating the abstract transformers over the *control flow graph*[8] (CFG). The join operator $\sqcup$ is applied at program locations where different branches come across. Moreover, the widening operator ∇ is applied at loop junctions (and afterwards, the narrowing operator Δ may be used).

Definition 2.2.3 (Widening). Assume the abstract interpretation from Definition 2.2.1. The **widening operator** over the abstract domain Q is a binary operator $\nabla : Q \times Q \to Q$ such that $\forall x, y \in Q : x \sqcup y \sqsubseteq x \nabla y$, and for all *increasing chains* $x_0 \sqsubseteq x_1 \sqsubseteq \ldots$, the increasing chain defined by $y_0 = x_0, \ldots, y_{i+1} = y_i \nabla x_{i+1}, \ldots$ is not *strictly increasing*. Note that the chain eventually stabilises since the result of ∇ is an *upper bound*.

Definition 2.2.4 (Narrowing). Assume the abstract interpretation from Definition 2.2.1. The **narrowing operator** over the abstract domain Q is a binary operator Δ: $Q \times Q \to Q$ such that $\forall x, y \in Q : y \sqsubseteq x \Longrightarrow y \sqsubseteq x \Delta y \sqsubseteq x$, and for all *decreasing chains* $x_0 \sqsupseteq x_1 \sqsupseteq \ldots$, the decreasing chain defined by $y_0 = x_0, \ldots, y_{i+1} = y_i \Delta x_{i+1}, \ldots$ is not *strictly decreasing*. Note that the chain eventually stabilises since the result of Δ is a *lower bound*.

2.3 Facebook Infer — Static Analysis Framework

This section describes the principles and features of *Facebook Infer*. The description is based on information provided at the Facebook Infer's website[9] and in [2].



mozná trochu zvětšit ?

Figure 2.3: Static analysis in *Facebook Infer*

[8] A **control flow graph** (**CFG**) is a *directed graph* in which the nodes represent *basic blocks* of the program, and the edges represent control flow paths among them [1]. A basic block is a *maximal sequence of statements* such that all transfers to it are to the first statement in the sequence, and all statements in the sequence are executed sequentially [28].

[9] The **Facebook Infer's** website: `https://fbinfer.com`.

Facebook Infer is an *open-source*[10] *static analysis framework*, which can discover various kinds of software bugs and which stress the *scalability* of the analysis. The Facebook Infer's basic usage is illustrated in Figure 2.3. A more detailed explanation of its architecture is given in Section 2.3.2. Facebook Infer is implemented in *OCaml*[11] — a *functional* programming language, also supporting *imperative* and *object-oriented* paradigms. Further details about OCaml can be found in the book [25]. Infer has initially been a rather specialised tool focused on *sound verification* of the absence of *memory safety violations*, which was first published in the well-known paper [5]. Once Facebook has purchased it, its scope significantly widened and abandoned the focus on sound analysis only.

Facebook Infer can analyse programs written in the following languages: C, C++, Java, Objective-C (and support for C# has been recently implemented [36]). 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* [38] (buffer overruns); *RacerD* [3, 4, 14] (data races); and other analyses that check for ~~buffer overflows~~, some forms of deadlocks and starvation, null-dereferencing, memory leaks, resource leaks, etc. Since Facebook Infer is a relatively popular and open-source framework, many experimental analysers arise pretty often. For instance, the promising experimental deadlock checker L2D2 [23] has been implemented at FIT BUT not long ago.

Inferbo!

2.3.1 Abstract Interpretation in Facebook Infer

Facebook Infer is a general framework for static analysis of programs, and it is based on *abstract interpretation*. Despite the original approach taken from [5], Facebook Infer aims to find bugs rather than perform *formal verification*. It is still possible to propose *sound* and *complete* analyses in the framework. However, the majority of the checkers already implemented in Facebook Infer are both unsound and incomplete.

It can be used to develop new sorts of *compositional* and *incremental* analysers quickly (both *intraprocedural* and *interprocedural* [28]) based on the concept of function *summaries*. In general, a summary χ represents a set of *pre-condition/post-condition* pairs for a function. In particular, it records under which pre-condition a function can be performed leading to a given post-condition. Formally, it can be described using *Hoare triples* [17], because it can be viewed as a triple $\{P\}\ S\ \{R\}$, where P is a pre-condition, S is a program (or a single statement/command)[12], and R is a description of the result of the execution of S (i.e., a post-condition). In theory, P and R are considered formulae in a suitable logic. The triple may be interpreted as follows. Suppose S is executed from a state satisfying P, and the execution of S terminates. In that case, the program state after S terminates satisfies R. 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. Usually, a summary consists of the relevant parts of *abstract states* for a particular analysis.

Facebook Infer generally does not compute the summaries during the analysis along the CFG as it is done in classical analyses based on the concepts from [29, 32]. Instead, Facebook Infer performs the analysis of a program *function-by-function along the call-tree*, starting from its leaves (demonstrated later in Example 2.3.1). Therefore, a function is

[10] The **Facebook Infer's open-source repository** at GitHub: `https://github.com/facebook/infer`.
[11] The **OCaml's** website: `https://ocaml.org`.
[12] In Facebook Infer's summaries considered as Hoare triples, S is usually one function.

* footnote? There is also an older internal representation called MIL which is now deprecated.

analysed, and a summary is computed without knowledge of the call context. Then, the summary of the function is used at all its call sites. It is needed to deduce under which pre-conditions a function can produce post-conditions appropriate for the given analysis. Since the summaries do not differ for different contexts, each function is analysed precisely once, and the analysis becomes more scalable, but it can lead to a loss of accuracy. However, of course, it is more troublesome to propose such a *bottom-up* analysis.

In order to create a new intraprocedural analyser in Facebook Infer, it is required to define the following (the listed items are described in more detail in Section 2.2):

1. The *abstract domain* Q, i.e., the type of *abstract states*.
2. The *ordering 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 computation.
5. The *transfer functions* τ, i.e., transformers that take an abstract state as an input and produce an abstract state as an output.

Furthermore, to create an interprocedural analyser, it is required to define additionally:

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

An important Facebook Infer's feature, which improves its scalability, is the *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, as shown in Example 2.3.1 later on.

2.3.2 Architecture of the Infer.AI Framework

The architecture of the abstract interpretation framework of Facebook Infer (**Infer.AI**) may be split into three major parts, as demonstrated in Figure 2.4: the *frontend*, an *analysis scheduler* (and the *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 SIL instructions. The SIL language consists of the following underlying instructions:

- `LOAD` — reading into a temporary variable;
- `STORE` — writing to a program variable, field of a structure, or array;
- `CALL` — a function call;

- PRUNE e — the evaluation of an expression e in a condition, cycle, etc.

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

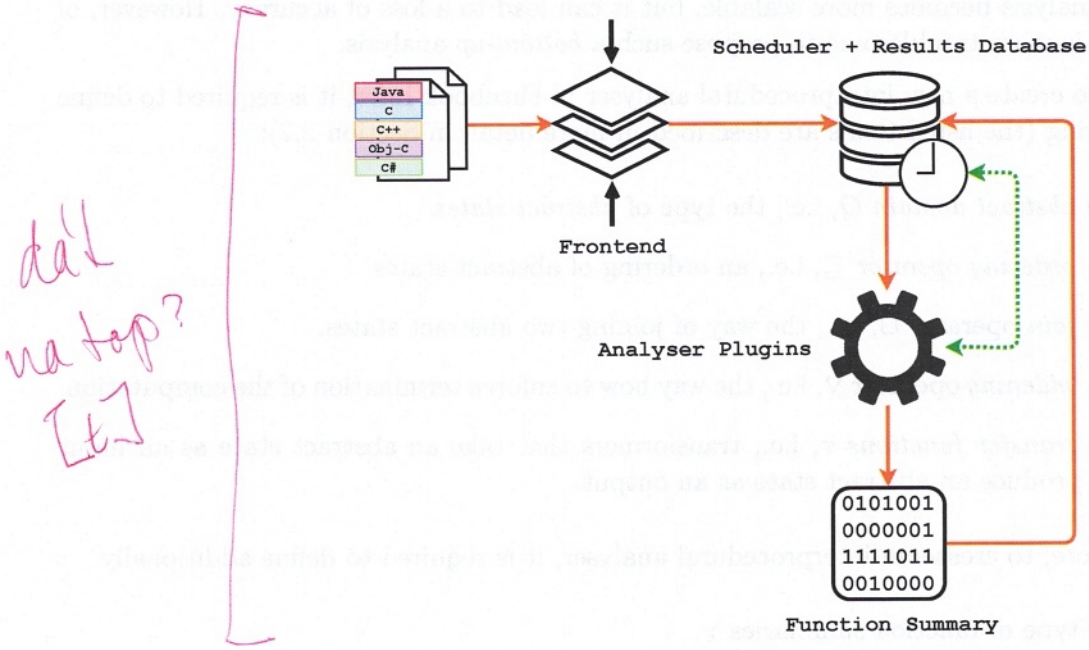


Figure 2.4: The *architecture* of the Facebook Infer's *abstract interpretation framework* [15]

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

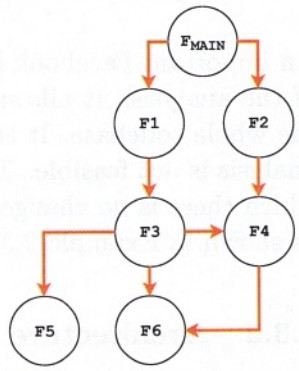


Figure 2.5: A *call graph* for an illustration of Facebook Infer's *analysis process* [15]

Example 2.3.1. For demonstrating the order of the analysis in Facebook Infer and its incrementality, assume the call graph given in Figure 2.5. At first, leaf functions F5 and F6 are analysed. Further, the analysis goes on towards the root of the call graph — F_{MAIN}, while considering the dependencies denoted by the edges. This order ensures that a summary is available once a nested function call is abstractly interpreted within the analysis. When there is a subsequent code change, only directly changed functions and all the functions up the call path are re-analysed. For instance, if there is a change of source code of function F4, Facebook Infer triggers re-analysis of functions F4, F2, and F_{MAIN} only.

The last part of the architecture consists of analyser plugins. Each plugin performs some analysis by interpreting SIL instructions. The result of the analysis of each function (func-

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

tion summary) is stored in the results database. The interpretation of SIL instructions (*commands*) is made using the *abstract interpreter* (also called the *control interpreter*) and *transfer functions* (also called the *command interpreter*). The transfer functions take a previously generated 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 2.6.

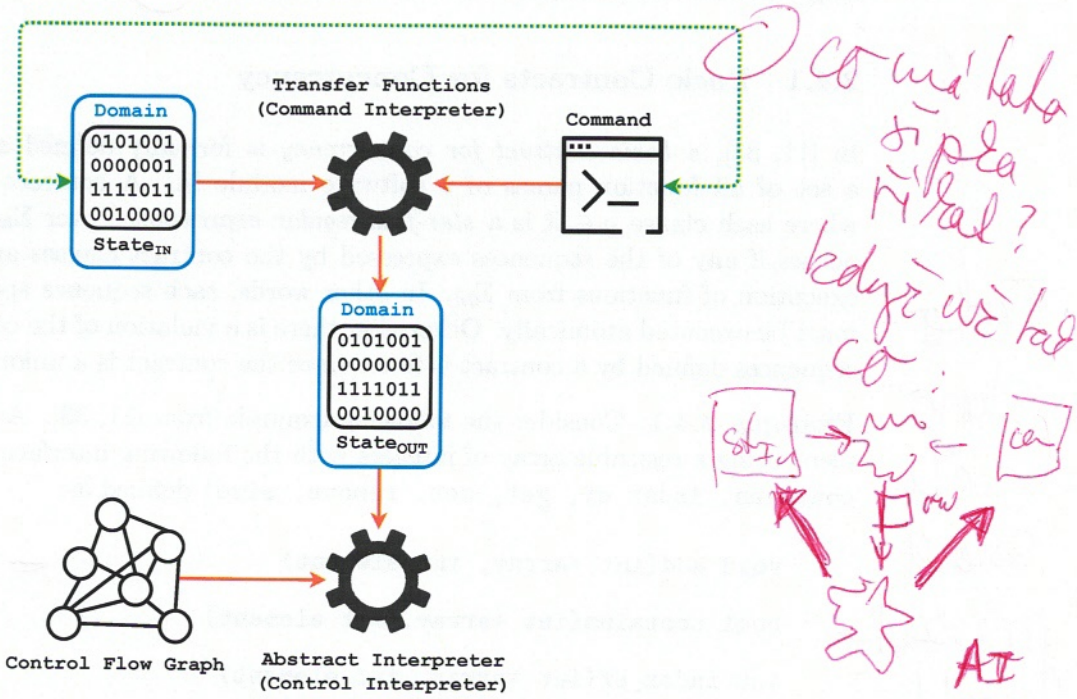


Figure 2.6: The Facebook Infer's *abstract interpretation process* [15]

2.4 Contracts for Concurrency

This section introduces the concept of *contracts for concurrency* [11, 33]. Examples and listings in the section are pieces of programs written in the C language.

Respecting the *protocol* of a software module — that 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 a function `open` at first, followed by an optional number of functions `read` and `write`, and at last, call a function `close`. A program utilising such a module that does not follow this protocol is erroneous. The methodology of *design by contract* [24] requires programs to meet such well-defined behaviours. [33]

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 called *spoilers* and sets of sequences of calls called *targets*. It is then required that no target overlaps fully with any spoiler. A developer may manually specify such contracts

Tady si nejsem jistý, zda bylo třeba ty rozšířené kontrakty zavádět – ale když už tam jsou OK,

or automatically generate them 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 2.4.1 defines the notion of *basic contracts for concurrency*. Further, Section 2.4.2 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. Finally, Section 2.4.3 presents those mentioned above more general contracts for concurrency with spoilers and targets, which essentially extend the basic contracts with some *contextual information*.

PAK se na to ale aspoň někde odkazuj

2.4.1 Basic Contracts for Concurrency

In [11, 33], a *basic contract for concurrency* is formally defined as follows. Let $\Sigma_{\mathbb{M}}$ be a set of all function names of a software module $\mathbb{M}$. A contract is a set $\mathbb{R}$ of *clauses*, where each clause $\varrho \in \mathbb{R}$ is a *star-free regular expression*[14] 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 a union of *star-free languages*.

Toto skoro nečtu, sněd přerušeně korektně (ale ne doslova)

Example 2.4.1. Consider the following example from [11, 33]. Assume a module $\mathbb{M}$ implementing a resizable array of integers with the following interface functions $\Sigma_{\mathbb{M}} = \{$`add`, `contains`, `index_of`, `get`, `set`, `remove`, `size`$\}$ defined as:

```
void add(int *array, int element)
bool contains(int *array, int element)
int index_of(int *array, int element)
int get(int *array, int index)
void set(int *array, int index, int element)
void remove(int *array, int index)
int size(int *array)
```

← např. v úvodu sekce 3 říci, co podporuje Gluon a co Anaconda ve future work

The module's contract contains clauses $\mathbb{R} = \{\varrho_1, \varrho_2, \varrho_3, \varrho_4\}$ such that:

NA TO POZOR !!!

(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 checking the presence of an element in an array, it can be removed by some concurrent thread.

(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 applied to address an array element because a concurrent modification of the array may shift the element's position.

Posunout blíže třeba k tomu odkazu s vysvětlením, alespoň stručně

[14] **Star-free regular expressions** are regular expressions that use only the *concatenation operator* and the *alternative operator* (|), without the *Kleene star operator* (*).

(ϱ_3) `size (get | set | remove)`

The execution of `size` followed by the execution of `get`, `set`, or `remove` should be atomic. Otherwise, an array may be empty when accessing it 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 index_of`

The execution of `add` followed by the execution of `index_of` should be atomic. Otherwise, the added element needs no longer exist in an array.

2.4.2 Contracts for Concurrency with Parameters

The above definition of basic contracts for concurrency is quite limited in some circumstances and can consider valid programs as erroneous (i.e., *false alarms* may be reported). Hence, in this section, there is introduced an extension of basic contracts — *contracts for concurrency with parameters* (defined in [11, 33]) — which takes into consideration the *data flow* within function calls.

Example 2.4.2. Consider the following example from [11, 33], given Listing 2.1. There is the function `replace` that replaces item `a` in an array by item `b`. The implementation of this function comprises two atomicity violations:

1. when `index_of` is invoked, item `a` does not need to be in the array anymore; and
2. the acquired index can be obsolete when `set` is executed.

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

(ϱ_5) `contains index_of set`

It can be obtained from the composition of clauses ϱ_1 and ϱ_2. 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(int *array, int a, int 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, 33]

To respect function call *parameters* and *return values* of functions in contracts, the basic contracts are extended by dependencies among functions in [11, 33] as follows. Function

is required to be respected

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` requires the same objects $o_1/o_2/o_3$ to be used at the positions of `X/Y/Z`, respectively; and the underscore indicates a *free meta-variable* that does not restrict the contract clause):

(ϱ_5') `contains(X,Y) Z=index_of(X,Y) set(X,Z,_)`

Example 2.4.3. With the extension described above, it is possible to extend the contract from Example 2.4.1 to capture the dependencies between function calls 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))`

(ϱ_3') `size(X) (get(X,_) | set(X,_,_) | remove(X,_))`

(ϱ_4') `add(X,Y) index_of(X,Y)`

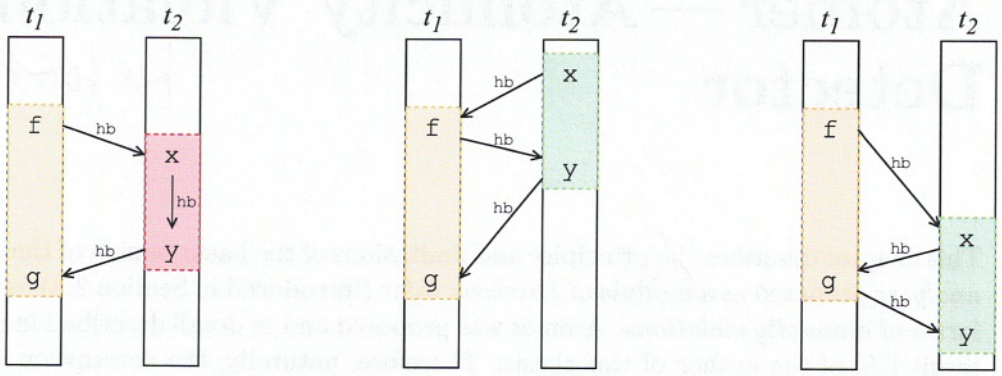
2.4.3 Contracts for Concurrency with Spoilers

Interleaving a sequence of function calls from a contract clause (considering the basic contracts for concurrency or contracts with parameters) with some function calls of a given module can cause atomicity violations. At the same time, this is not the case for some other function calls from the module. However, this is not possible to describe in the contracts introduced so far. For instance, clause (ϱ_1) `contains index_of` requires that this sequence of calls must always be performed atomically, i.e., it does not matter which functions are executed by other threads. Thus, interleaving the execution of this sequence with, e.g., `remove` or `get` is a contract violation. However, in effect, only the execution of `remove` may be problematic, while the execution of `get` may not.

The paper [11] proposes a solution to the above issue — an extension of the basic contracts for concurrency with *contextual information* — allowing one to describe in which context the contract clauses should be enforced. Each clause of the basic contract is now called *target*. For each target, there is a set of so-called *spoilers* that restrict its application. That is, a spoiler is a sequence of function calls that can violate its target. In the end, it has to be ensured that each target is executed atomically w.r.t. its spoilers. Assuming the earlier example, let `contains index_of` be the target clause. Then, a possible spoiler is `remove`. The syntax for this description is as follows: `contains index_of` $\leftsquigarrow$ `remove`.

Formally, let $\Sigma_{\mathbb{M}}$ be a set of all function names of a software module $\mathbb{M}$. Further, let $\mathbb{R}$ be a set of *target clauses*, where each clause $\varrho \in \mathbb{R}$ is a star-free regular expression over $\Sigma_{\mathbb{M}}$. Let $\mathbb{S}$ be a set of *spoiler clauses*, where each clause $\sigma \in \mathbb{S}$ is a star-free regular expression over $\Sigma_{\mathbb{M}}$. Moreover, let $\Sigma_{\mathbb{R}} \subseteq \Sigma_{\mathbb{M}}$ and $\Sigma_{\mathbb{S}} \subseteq \Sigma_{\mathbb{M}}$ be the alphabets of function names used in the target or spoiler clauses, respectively. Then, a contract is a relation $\mathbb{C} \subseteq \mathbb{R} \times \mathbb{S}$, which states for each target the spoilers that can cause atomicity violations. One spoiler may violate more than one target, and, on the contrary, one target may be violated by more

than one spoiler. A contract is violated iff any executed sequence expressed by a target $\varrho \in \mathbb{R}$ is *completely interleaved* with the execution of the sequence representing its spoiler, i.e., $\sigma \in \mathbb{C}(\varrho)$. A target sequence r is completely interleaved by a spoiler sequence s iff the execution of r *starts before* the execution of s, and the execution of s *ends before* the execution of r. *Partial interleavings* of targets and spoilers are here not taken into account ~~to cause~~ an error. However, if needed, this can be resolved by adding a new target clause with an appropriate spoiler. Whether a sequence is executed before another one ~~is~~, is defined using the so-called *"happened before" relation* ($\xrightarrow{\texttt{hb}}$) [20]. Complete interleaving is illustrated in Figure 2.7a, and partial interleavings are shown in Figures 2.7b and 2.7c.



(a) *Complete interleaving*: the contract is violated

(b) *Partial interleaving*: the contract is not violated (*s started before r*)

(c) *Partial interleaving*: the contract is not violated (*s ended after r*)

Figure 2.7: A *contract violation* demonstration with *target and spoiler interleavings*. In each sub-figure, a thread t_1 executes a target sequence $r = \texttt{f g}$ and a thread t_2 executes a spoiler sequence $s = \texttt{x y}$. Consequently, ~~there is~~ the following contract: $\texttt{f g} \rightsquigarrow \texttt{x y}$ [26].

Example 2.4.4. With the extension of spoilers, the contract from Example 2.4.1 can be refined with contextual information to refuse *unsafe interleavings* as follows (all other interleavings, not described by spoilers, are deemed safe):

(ϱ_1'') `contains index_of` $\rightsquigarrow$ `remove | set`

(ϱ_2'') `index_of (get | set | remove)` $\rightsquigarrow$ `remove | set`

(ϱ_3'') `size (get | set | remove)` $\rightsquigarrow$ `remove`

(ϱ_4'') `add index_of` $\rightsquigarrow$ `remove | set`

Lastly, it is possible to combine the extension of contracts for concurrency with spoilers and with parameters. The following clause can demonstrate it:

```
Y=index_of(X,_) get(X,Y) ⇝ remove(X,_)
```

The clause requires sequences of `index_of` and `get` to be performed atomically, but only when working with the same element `X` and index `Y`, and only w.r.t. the concurrent execution of `remove`. Note that the argument `index` of `remove` is not restricted because any concurrent removal may produce an atomicity violation — by either removing an element on index `Y` or by shifting its position.

(*) Tuto poznámku nechápu! Říkáš, že Gluon má omezenou škálovatelnost a tu poznámku pak vlastně naznačuje, že totéž platí pro Atomer? [Spíše bych zde citoval to, co oni sami mají někde jako poznámku v kódu ne?]

[Ale to tam máš v Eus dál... tak bych ten footnote jen vytrhl.

Chapter 3

An

Atomer — Atomicity Violations Detector

This chapter describes the principles and limitations of the basic version of the *Atomer static analyser* proposed as a module of *Facebook Infer* (introduced in Section 2.3) for finding some forms of *atomicity violations*. Atomer was proposed and in detail described in the bachelor's thesis [15] of the author of this thesis. Therefore, naturally, the description in Section 3.2 is based on the mentioned thesis. Already existing solutions in this area (besides Atomer) are discussed in Section 3.1. In particular, it deals with other existing approaches and tools for finding atomicity violations, their advantages, disadvantages, features, availability, and so on. In Section 3.3, there are discussed limitations and shortcomings of Atomer. Some of the thoughts mentioned in this section are taken into consideration already in [15].

teď vidím, že tam něco máš později, ale já bych to uvedl i sem (ale pozor na opakování)

The approach targets the basic contracts

3.1 Related Work

Atomer is slightly inspired by ideas from [11, 33]. In these papers, there is described a proposal and implementation of a *static* approach for finding *atomicity violations of sequences of function calls*, which is based on *grammars* and *parsing trees*. Note that 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 static analyser called *Gluon*[1] for analysing programs written in Java. To the best author's knowledge, Gluon is the only static analyser that tries to go in a similar direction as Atomer does. Gluon led to some promising experimental results, but the *scalability* of Gluon was still limited.[2] Moreover, Gluon is no more actively developed, and it is not easy to use. Despite all author's efforts, it was not put into operation. Above that, the authors themselves note that the code of Gluon is very ad hoc, and many things are hard-coded in it. These facts, in fact, inspired the decision that eventually led to the implementation of the first version of Atomer, namely, to get inspired by the ideas of [11, 33] but reimplement them in *Facebook Infer*, redesigning it in accordance with the principles of Facebook Infer, which should make

extended with data (???) as described in Section

[1] **Gluon** is a tool that implements a *static* approach for finding *atomicity violations of sequences of function calls* 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, which is discussed in Sections 6.3, 6.4.

which targets the case of contracts with both targets & spoilers (but so far without data) an extension

Lolo myslím neplatí !!!!! to umí jen Anaconda

the resulting tool more scalable. In the end, however, due to adapting the analysis to the context of Facebook Infer, the implementation of Atomer's analysis is significantly different from [11, 33], as is presented in Chapter 4 of [15]. Furthermore, unlike Gluon, the new version of Atomer is capable of analysing a much wider range of programs because it also supports other languages than Java, and it supports more advanced *locking mechanisms*. On the other hand, Gluon implements *extended contracts for concurrency* (see Sections 2.4.2, 2.4.3) that consider *data flow* within functions and *contextual information*, while Atomer implements only the idea of *basic contracts for concurrency* (see Section 2.4.1). These extended contracts should improve the precision of the analysis. Nonetheless, it is the author's future work to implement the extended contracts in Atomer as well.

In Facebook Infer, there is already implemented an analysis called *Lock Consistency Violation*[3]. It is a part of *RacerD* [3, 4, 14]. 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. Moreover, Atomer tries to automatically determine which sequences should indeed be executed atomically.

3.2 Design of Atomer and Its Principles

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

The proposal of Atomer is based on the concept of *contracts for concurrency* described in Section 2.4. In particular, the proposal considers the *basic contracts* described in Section 2.4.1. Neither the contracts extended by *parameters* explained in Section 2.4.2 nor the contracts extended by *spoilers* and *targets* described in Section 2.4.3 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 2.4. Atomer is able to automatically derive candidates for such contracts and then verify whether the contracts are fulfilled. In other words, Atomer can both automatically derive those sequences that are sometimes executed atomically as well as subsequently check whether they are indeed always executed atomically. Both of these steps are done statically. The proposed analysis is thus divided into two parts (*phases of the analysis* that are in-depth described in the sections below and illustrated in Figure 3.1):

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

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

This section provides a high-level view of the *abstract interpretation* underlying Atomer. The concrete types of the *abstract states* (i.e., elements of the *abstract domain* Q) and

[3] **Lock Consistency Violation** is an *atomicity violations* analysis implemented in *Facebook Infer*. It is described at `https://fbinfer.com/docs/all-issue-types#lock_consistency_violation`.

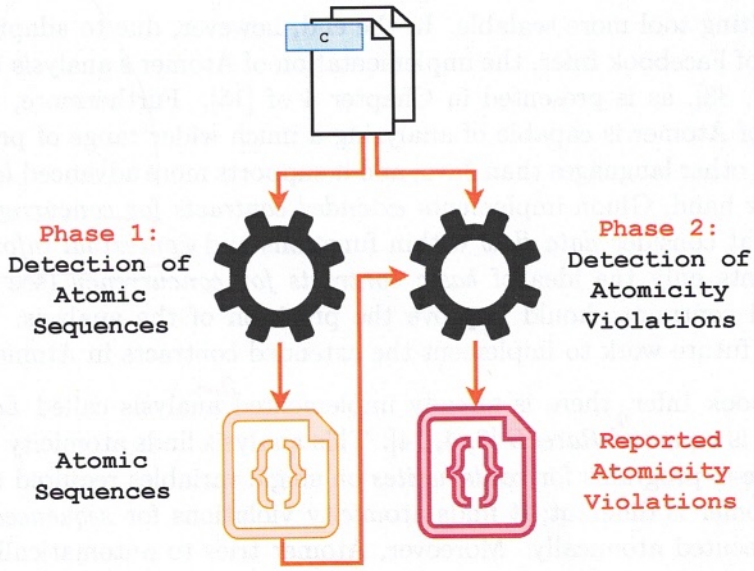


Figure 3.1: *Phases* of the analysis of the Atomer's first version and the analysis *high-level process* illustration [15]

the *summaries* χ, along with the implementation of all necessary *abstract interpretation operators* are stated in Chapter 4 of [15]. However, actually, the abstract states $s \in Q$ 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., $s \sqsubseteq s' \iff s \subseteq s'$, where $s, s' \in Q$), the *join operator* $\sqcup$ is implemented as the *set union* (i.e., $s \sqcup s' \iff s \cup s'$), and the *widening operator* $\triangledown$ is implemented using the join operator (i.e., $s \triangledown s' \iff s \sqcup s'$) since the domains are *finite*. Furthermore, it is essential to note that the proposed analysis is neither *sound* nor *complete*. Its goal is to effectively find bugs rather than formally verify the absence of atomicity violations.

Function summaries are in the below sections reduced to the output parts only (*post-conditions* R). The input parts of summaries (*pre-conditions* P) are in case of the proposed analysis always empty because, so far, it is not necessary to have any pre-conditions for analysed functions. Thus, in this case, *Hoare triples* — $\{true\}\ S\ \{R\}$ — are used, where S is an analysed function, i.e., $P = true$.

Listings in the below sections are pieces of programs written in the C language (assuming `lock`/`unlock` functions for *mutual exclusion* to *critical sections*).

3.2.1 Phase 1 — Detection of Atomic Sequences

Before detecting *atomicity violations* may begin, it is required to have *contracts for concurrency* introduced earlier. 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, particularly, under some *lock*, 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*.

On the other hand, will be created

first B sequence as follows: $s_S = \{\{(\mathtt{f} \cdot \mathtt{g} \cdot \mathtt{x},\ \varepsilon)\}\}$, $s_{S'} = \{\{(\mathtt{f} \cdot \mathtt{g},\ \mathtt{m} \cdot \mathtt{n} \cdot \mathtt{x})\}\}$. Subsequently, if the next step in the program S is a lock call, the next function calls will be added to the first B sequence of the set s_S. ~~And~~ if the next step in the program S' is an unlock call, ~~it will be created~~ a new element of the first set of the set $s_{S'}$, and the next function calls will be added to the A sequence of this element. Finally, if a function $\mathtt{y}$ is called, the resulting sets will look like follows: $s_S = \{\{(\mathtt{f} \cdot \mathtt{g} \cdot \mathtt{x},\ \mathtt{y})\}\}$, $s_{S'} = \{\{(\mathtt{f} \cdot \mathtt{g},\ \mathtt{m} \cdot \mathtt{n} \cdot \mathtt{x}), (\mathtt{y},\ \varepsilon)\}\}$. Note that the final sequences of function calls look like follows: $\mathtt{f} \cdot \mathtt{g} \cdot \mathtt{x}\ [\mathtt{y}$ and $\mathtt{f} \cdot \mathtt{g}\ [\mathtt{m} \cdot \mathtt{n} \cdot \mathtt{x}]\ \mathtt{y}$ for the programs S and S', respectively.

Tato notace nebyla zavedena! Zvyklý bych si na zavedení fčí, předp. že se analýza techto 2 programů. Pak bych řekl

non-empty

A *summary* $\chi_{\mathtt{f}} \in 2^{\Sigma^*} \times \Sigma^*$ of a function $\mathtt{f}$ is then a pair $\chi_{\mathtt{f}} = (\mathcal{B}, \overline{AB})$, where:

NEBO C

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

  $\{B \in \Sigma^+ \mid \exists p \in S_f : \exists A \in \Sigma^* : (A,B) \in p\}$

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

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

  $\odot$ AB $AB \in \overline{AB}$

  jinak se to plete s A, B!!!

  $\{AB \in \Sigma^* \mid \exists p \in S_f : (A,B) \in p\}$

  where $AB' = \{ab \in \Sigma^* \mid \exists p \in s_{\mathtt{f}} : \exists (A, B) \in p : ab = A \cdot B\}$, $\bigcup$ concatenates all sequences of a set, and *reduce* is a function that removes duplicates of function calls. Intuitively, in this component of the summary, the analysis gathers occurrences of all called functions within the analysed function obtained by a concatenation of all the A and B sequences. AB is recorded to facilitate the derivation of atomic call sequences that show up higher in the *call hierarchy*. Indeed, while locks/unlocks can appear in such a *higher-level* function, parts of the call sequences can appear lower in the call hierarchy.

  $\odot$

  Tato notace musí být zavedena!

  nějak bych to odlišil od AB nebo \mathcal{AB} a $\overline{AB}$ dát sem

Example 3.2.2. For instance, the analysis of the function $\mathtt{f}$ from Listing 3.1 produces the following sequences:

$$\overbrace{\mathtt{x} \cdot \cancel{\mathtt{x}} \cdot \mathtt{y}}^{A_1}\ \overbrace{[\mathtt{a} \cdot \mathtt{b} \cdot \cancel{\mathtt{b}}]}^{B_1}\ \overbrace{\mathtt{y} \cdot \cancel{\mathtt{y}}}^{A_2}\ \overbrace{[\mathtt{a} \cdot \mathtt{c}]}^{B_2}\ \overbrace{\cancel{\mathtt{y}}}^{A_3}\ \overbrace{\cancel{[\mathtt{a} \cdot \mathtt{a} \cdot \mathtt{c}]}}^{B_3}$$

The functions $\mathtt{a}$, $\mathtt{b}$, $\mathtt{c}$, $\mathtt{x}$, $\mathtt{y}$ are not deeper analysed because it is assumed that these functions are leaf nodes of the call graph. The strikethrough of the functions $\mathtt{b}$, $\mathtt{x}$, $\mathtt{y}$ denotes removing already recorded function calls in the A and B sequences to get the reduced form. The strikethrough of the entire sequence $\mathtt{y}\ [\mathtt{a} \cdot \mathtt{a} \cdot \mathtt{c}]$ means discarding sequence already seen before. For the above, the abstract state at the end of the abstract interpretation of the function $\mathtt{f}$ is $s_{\mathtt{f}} = \{\{(\mathtt{x} \cdot \mathtt{y},\ \mathtt{a} \cdot \mathtt{b}), (\mathtt{y},\ \mathtt{a} \cdot \mathtt{c}), (\varepsilon,\ \varepsilon)\}\}$. The derived summary $\chi_{\mathtt{f}}$ for the function $\mathtt{f}$ is $\chi_{\mathtt{f}} = (\mathcal{B}, AB)$, where $\mathcal{B} = \{\mathtt{a} \cdot \mathtt{b},\ \mathtt{a} \cdot \mathtt{c}\}$, i.e., B_1 and B_2; and $AB = \mathtt{x} \cdot \mathtt{y} \cdot \mathtt{a} \cdot \mathtt{b} \cdot \mathtt{c}$, i.e., the concatenation of A_1, B_1, A_2, B_2 from which duplicate function calls were removed.

(*) Ten příklad je třeba předělat — přijde mi nepochopitelný a užívá něco, co není zavedeno?

This

odstavce

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. It is needed 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 the set of functions of a given program) of reduced sequences of function calls for each path such that: 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 unlock (or between a lock and the end of the function being analysed). Thus, the abstract states of the analysis are elements of the set $2^{2^{\Sigma^* \times \Sigma^*}}$ because there is a set of the (A, B) pairs for each program path.

Sg — a new stav?

TOTO NELZE!

spíše current

It would be more precise to generate longer sequences of type $A_1 \cdot B_1 \cdot A_2 \cdot B_2 \cdot \ldots$ 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* [35]. 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 [15] show that it needs even more pronounced abstraction for appropriate *scalability*.

normální začátek odstavce

(viz X.1)

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:

normální začátek odstavce?

musí to být leaf?

$$
\begin{aligned}
s_{\mathtt{g}} = \{p' \in 2^{\Sigma^* \times \Sigma^*} \mid \exists p \in s_{\mathtt{g}} : p' = \{(A', B') \in \Sigma^* \times \Sigma^* \mid \exists (A, B) \in p : \\
[\neg actual(p, (A, B)) \wedge (A', B') = (A, B)] \vee [actual(p, (A, B)) \\
\wedge [(lock \wedge (A', B') = (A, B \cdot \mathtt{f})) \vee (\neg lock \wedge (A', B') = (A \cdot \mathtt{f}, B))]]\}\}
\end{aligned}
$$

Co je to za program?!!

We consider 2 programs where...

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*. Further, 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:

NEZAVEDENO

(+) raději P a P'-program?

$$
\begin{aligned}
s_{\mathtt{g}} = \{p' \in 2^{\Sigma^* \times \Sigma^*} \mid \exists p \in s_{\mathtt{g}} : p' = \{(A, B) \in \Sigma^* \times \Sigma^* \mid \\
[(A, B) = (\varepsilon, \varepsilon) \wedge setActual(p, (A, B))] \vee (A, B) \in p\}\}
\end{aligned}
$$

Nekontrolujeme — snad přenesl!

(*)

Example 3.2.1. For an explanation of the computation of the sets of the pairs (A, B), assume that a state of the analysis of a program S is the following sequence of function calls: $\mathtt{f} \cdot \mathtt{g}$; and a state of the analysis of a program S' is the following sequence of function calls: $\mathtt{f} \cdot \mathtt{g}\,[\mathtt{m} \cdot \mathtt{n}$. The square brackets are used to indicate an *atomic sequence* (the closing square bracket is missing in the case of the program S', which means that the program state is currently inside an atomic block). The computed abstract state for the program S is $s_S = \{\{(\mathtt{f} \cdot \mathtt{g},\ \varepsilon)\}\}$, and for the program S', it is $s_{S'} = \{\{(\mathtt{f} \cdot \mathtt{g},\ \mathtt{m} \cdot \mathtt{n})\}\}$. Now, if the next instruction is a call of a function `x`, in the case of the program S, the call will be added to the first A sequence, and in the case of the program S', the call will be added to the

! viz X.1

Stav je sekvence? Nebylo to zavedeno jako (A,B) ?!?

X.1

- Původně jsem si myslel, že ty vstahy jsou převzaté, tak jsem je neprocházel. Teď jsem je prošel a vidím, že je to špatně !!! Nelze mít predikáty jako loc a actual, které mluví o částech stavu, který ale v tom stavu není !!!!!!!

- Ty abstraktní stavy by měly mít podobu asi:

$$\underbrace{2^{2^{\Sigma^* \times \Sigma^*}}}_{\text{již uzavřené dvojice}} \times \underbrace{2^{\Sigma^* \times \Sigma^* \times \{0,1\}}}_{\text{rozpracované dvojice + příznak zaměnění.}}$$

- Volání f-ce f na straně 27 by pak vedlo na něco jako:

$$S_g = \bigcup_{p \in S_g} \{ (p \cap \Sigma^* \times \Sigma^* \times \{0\} \times \{0,1\}) \cup \{ (A, B \boxdot f, 1, 1) \mid (A, B, 1, 1) \in p \} \cup \{ (A \boxdot f, B, 1, 0) \mid (A, B, 1, 0) \in p \} \}$$

ne-aktivní se jen kopírují

kde $\boxdot$ – reduced concatenation a $\cup$ – sjednocení reflektující redukci (to tam teď také není).

- Pro unlock na straně 27 by to dalo:

$$S_g = \bigcup_{p \in S_g} \{ \{ (A, B, 0, \ell) \mid \exists\, c \in \{0,1\} : (A, B, c, \ell) \in p \} \cup \{ (\varepsilon, \varepsilon, 1, 0) \} \}.$$

- Při té příležitosti mě napadlo, jak se řeší double unlock? A double lock? Ignoruje se? Možno uvést?
- Jak by vypadal lock? Pro které dvojice se jen nastaví příznak zamknutí? Tedy:

$$\acute{S}g = \bigcup_{p \in Sg} \{ (p \cap \Sigma^* \times \Sigma^* \times \{0\} \times \{0,1\}) \sqcup \{(A,B,1,1) \mid \exists l \in \{0,1\} : (A,B,1,l) \in p\}\} \quad ?$$

- Co by byl počáteční stav?

$$\{\{(\varepsilon,\varepsilon,1,0)\}\}$$

↑ odemknuto?

- Pro volání f na straně by to pak dalo:

$$\acute{S}g = \bigcup_{p \in Sg} \{ (p \cap \Sigma^* \times \Sigma^* \times \{0\} \times \{0,1\}) \sqcup \{(A,B \boxdot f \boxdot C) \mid (A,B,1,1) \in p\} \sqcup \{(A \boxdot f \boxdot C, B) \mid (A,B,1,0) \in p\}\},$$

kde C je summary f.

Nebezpečné — není zřejmé, proč — mění se pořadí.

```
1  void f()
2  {
3      x(); x(); y();
4      lock(&L); // a . b
5      a(); b(); b();
6      unlock(&L);
7      y(); y();
8      lock(&L); // a . c
9      a(); c();
10     unlock(&L);
11     y();
12     lock(&L); // a . c
13     a(); a(); c();
14     unlock(&L);
15 }
```

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

Further, ~~it is~~ we demonstrated how the results of the analysis of *nested functions* are used during the detection of atomic sequences. The result of the analysis of a nested function is used as follows. When calling an already analysed function, one plugs the sequence from the second component of its summary (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` is called ~~with a~~ non-empty summary (i.e., $AB \neq \varepsilon$). If the current program state of the 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, just the function name is appended to the most recent A or B sequences of all 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:

2x toleč?

zmatek — není jasné, není kdo A/B patří — co třeba C?

$$
\begin{aligned}
s_{\mathtt{g}} = \{p' \in 2^{\Sigma^* \times \Sigma^*} \mid \exists p \in s_{\mathtt{g}} : p' = \{&(A', B') \in \Sigma^* \times \Sigma^* \mid \exists (A, B) \in p : \\
&[\neg actual(p, (A, B)) \wedge (A', B') = (A, B)] \vee [actual(p, (A, B)) \\
&\wedge [(lock \wedge (A', B') = (A, B \cdot \mathtt{f} \cdot AB)) \vee (\neg lock \wedge (A', B') = (A \cdot \mathtt{f} \cdot AB, B))]]\}\}
\end{aligned}
$$

Example 3.2.3. 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. The second component of $\chi_{\mathtt{f}}$ is $AB = \mathtt{x} \cdot \mathtt{y} \cdot \mathtt{a} \cdot \mathtt{b} \cdot \mathtt{c}$. The analysis of the function `g` produces the following sequence:

$$\mathtt{x} \cdot \mathtt{f} \cdot \cancel{\mathtt{x}} \cdot \mathtt{y} \cdot \mathtt{a} \cdot \mathtt{b} \cdot \mathtt{c} \cdot \mathtt{z}\ [\mathtt{f} \cdot \mathtt{x} \cdot \mathtt{y} \cdot \mathtt{a} \cdot \mathtt{b} \cdot \mathtt{c}]$$

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

```
1  void g()
2  {
3      x(); f(); z();
4      lock(&L); // f . x . y . a . b . c
5      f();
6      unlock(&L);
7  }
```

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

Cases Where Lock/Unlock Calls Are Not Paired in a Function

For treating cases where *lock/unlock calls are not paired* in a function — as demonstrated in Listing 3.3 — the following solution is implemented in the basic version of Atomer.

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 `m` from Listing 3.3, the atomic section is virtually closed. Hence, there is detected an atomic sequence `a`. In particular, the summary is as follows: $\chi_{\mathtt{m}} = (\{\mathtt{a}\},\ \mathtt{a})$.

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

```
1  void m()
2  {
3      lock(&L); // a
4      a();
5  }
6  void n()
7  {
8      a();
9      unlock(&L);
10 }
```

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

Storing the results of

~~Summary~~ of Phase 1 The derived sequences of calls assumed to execute atomically — the $\mathcal{B}$ sequences — from the summaries of all analysed functions are stored into a file used during Phase 2, which is described later on.

3.2.2 Phase 2 — Detection of Atomicity Violations

In the second phase of the analysis, i.e., when *detecting violations* of the atomic sequences obtained from Phase 1, the analysis looks for *pairs of functions* that *should be called atomically* (or just for single functions if there is only one function call in an atomic sequence) and that are not executed atomically (i.e., under a lock) on some path through the CFG. The pairs of function calls to be checked for atomicity are obtained as follows. For each function `f`

(As a particular case, if the atomic seq consists of a single f, it is checked well...)

Vytvořil bych jako samostatnou věta až za konec toho. Je to jiná příběh komplikovaně!