BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2021
http://excel.fit.vutbr.cz

Advanced Static Analysis of Atomicity in Concurrent Programs through Facebook Infer

Dominik Harmim*

Abstract
This work aims to improve and extend *Atomer*. A further goal is to perform new experiments with it. Atomer is a *static analyser* that detects *atomicity violations*, created within the bachelor's thesis of the author as a module of *Facebook Infer* — an open-source static analysis framework that ~~promotes~~ efficient analysis. ~~The original analysis assumes that~~ *sequences of function calls* executed *atomically once* should probably be executed *always atomically*, and it naturally works with sequences. In this work, to improve *scalability*, the use of sequences was *approximated by sets*. Further, several new features were implemented, notably: support for *C++ and Java languages*; ~~more advanced and precise *manipulation with locks*;~~ and the analysis's *parametrisation*. The new extensions were verified and evaluated on programs created for testing purposes. Furthermore, new experiments on *extensive real-life programs* were made, ~~and already fixed and reported~~ *real bugs* were rediscovered. The experiments revealed a *significant increase of precision* of the new version of Atomer.

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

Supplementary Material: Atomer Repository — Atomer Wiki — Facebook Infer Repository

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

1. Introduction

Bugs ~~are an integral part~~ of computer programs ever since the inception of the programming discipline. Unfortunately, they are often hidden in unexpected places, and they can lead to unexpected behaviour, which may cause significant damage. Nowadays, developers have many possibilities of catching bugs in the early development process. *Dynamic analysers* or tools for *automated testing* are often used, and they are satisfactory in many cases. Nevertheless, they can still leave too many bugs undetected because they can analyse only *particular program flows* dependent on the input data. An alternative solution is *static analysis* ~~that has its shortcomings as well~~, such as the ~~*scalabil-*~~ *ity* on large codebases or a considerably high rate of incorrectly reported errors (so-called *false positives* or *false alarms*). ~~Several efficient~~ tools for static analysis were implemented, e.g., Coverity or CodeSonar. However, they are often proprietary and difficult to openly evaluate and extend.

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

In *concurrent programs*, there are often *atomicity requirements* for 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 laborious and time-consuming is finding and fixing them.

Within the author's bachelor's thesis [1], *Atomer*[1] was proposed — a *static analyser* for finding some forms of *atomicity violations* implemented as a Facebook Infer's module. In particular, the stress is put on the *atomic execution of sequences of function calls*, which is often required, e.g., when using specific library calls. ~~It~~ is based on the assumption that sequences of function calls executed *atomically once* should probably be executed *always atomically*~~, and it naturally~~ works with sequences. In fact, the idea of checking the atomicity of certain sequences of function calls is inspired by the work of *contracts for concurrency* [2]. In the terminology of [2], the atomicity of specific sequences of calls is the most straightforward (yet very useful in practice) kind of contracts for concurrency.
The implementation of the first version of Atomer ~~mainly~~ ~~targets~~ *C programs* ~~that use~~ *POSIX thread*, ~~i.e.,~~ *PThread locks*.

Within this work, Atomer ~~was~~ improved and extended. Further, ~~other~~ experiments were performed. In particular, to improve *scalability*, working with sequences of function calls was *approximated* by working with *sets of function calls*. Furthermore, several new features were implemented, notably: support for *C++ and Java languages*; more advanced and precise *manipulation with locks*; ~~and~~ the analysis's ~~*paramet-*~~ *risation*.

The development of Atomer and its extensions have been discussed with the developers of Facebook Infer, and it was supported under the H2020 ECSEL project Aquas. Currently, it is a part of the H2020 ECSEL project Arrowhead Tools. Parts of the paper are taken from the thesis [1].

The rest of the article is organised as follows. In Section 2, there is introduced the *Facebook Infer* framework. *Atomer* is described in Section 3. ~~This is followed by~~ Section 4, which considers all the extensions and improvements proposed within this work. The implementation of these extensions, together with the experimental evaluation of the new Atomer's features and other experiments performed within this work, are discussed in Section 5. Finally, the paper is concluded in Section 6.

2. Facebook Infer

This section describes the principles and features of *Facebook Infer*. The description is based on information provided at the Facebook Infer's website[2] and in [3]. Parts of this section are taken from [1].

Facebook Infer is an *open-source*[3] *static analysis framework*[4], which can discover various kinds of software bugs ~~of a given program, emphasising~~ *scalability*. A more detailed explanation of Facebook Infer architecture is given in Section 2.2. Facebook Infer is implemented in *OCaml*[5] — a *functional* programming language, also supporting *imperative* and *object-oriented* paradigms. ~~Facebook~~ Infer was originally a ~~standalone~~ tool focused on *sound verification* of the absence of *memory safety violations*, which was first published in the well-known paper [7].

Facebook Infer can analyse programs written in several languages. In particular, it supports the following languages: C, C++, Java, and Objective-C (~~and~~ C#, see [8]). Moreover, it is possible to extend Facebook Infer's *frontend* for supporting other languages.

[1] The implementation of **Atomer** is available ~~on~~ GitHub as an *open-source* repository. The link can be found among the supplementary materials of this paper.

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

[3] A link to the **Facebook Infer's open-source repository** is in the supplementary materials of the paper.

[4] A ~~brief explanation of **static analysis** itself can be found in [1] — Section 2.1.~~ In more detail, it is then explained in [4, 5, 6].

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

Currently, Facebook Infer contains many analyses focusing on various kinds of bugs, e.g., *Inferbo* (buffer overruns); *RacerD* (data races) [9, 10]; and other analyses that check for buffer overflows, some forms of deadlocks and starvation, null-dereferencing, memory leaks, resource leaks, etc.

2.1 Abstract Interpretation in Facebook Infer

Facebook Infer is a general framework for static analysis of programs, and it is based on *abstract interpretation*[6]. Despite the original approach taken from [7], Facebook Infer aims to find bugs rather than *formal verification*. It can be used to develop new sorts of *compositional* and *incremental* analysers quickly (*intraprocedural* ~~or~~ *interprocedural* [5]) based on the concept of function *summaries*. In general, a *summary* represents *preconditions* and *postconditions* of a function [13]. However, in practice, a summary is a custom data structure that may be used for storing any information resulting from the analysis of particular functions. Facebook Infer generally does not compute the summaries during the analysis along the *Control Flow Graph* (**CFG** [14]) as it is done in classical analyses based on the concepts from [15, 16]. Instead, Facebook Infer performs the analysis of a program *function-by-function along the call tree*, starting from its leaves (demonstrated in Example 2.1). Therefore, a function is analysed, and a summary is computed without knowledge of the call context. Then, the summary of a function is used at all of its call sites. Since the summaries do not differ for different contexts, the analysis becomes more scalable, but it can lead to a loss of accuracy.

In order to create a new intraprocedural analyser in Facebook Infer, it is needed to define the following (the listed items are described in more detail in [1] — Section 2.1.1):

1. The *abstract domain* $\boldsymbol{Q}$, i.e., ~~a~~ type of ~~an~~ *abstract* ~~state,~~
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 ∇, i.e., the way how to enforce ~~the~~ termination of the computation.
5. The *Transfer functions* τ, i.e., ~~a~~ transformer that take~~s~~ an abstract state as an input and produce~~s~~ an abstract state as an output.

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

1. ~~A~~ 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 feature of Facebook Infer improv~~ing~~ 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 *reusing summaries* of functions for which there is no change in them neither in the functions transitively invoked from them, as shown in Example 2.1.

2.2 Architecture of the Abstract Interpretation Framework in Facebook Infer

The architecture of the abstract interpretation framework of Facebook Infer (**Infer.AI**) may be split into three major parts, as demonstrated in Figure 1: ~~a~~ *frontend*, ~~an~~ *analysis scheduler* (and a *results database*), and a set of *analyser plugins*.

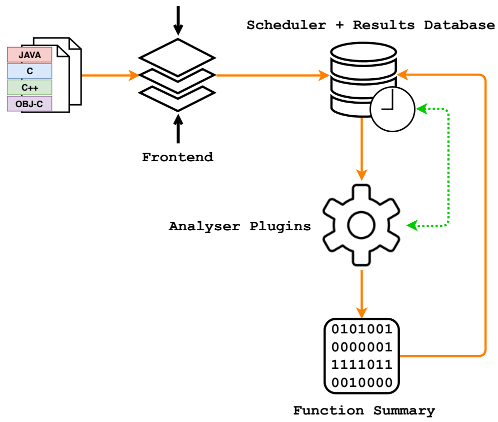

Figure 1. The architecture of Facebook Infer's abstract interpretation framework [1]

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 individual instructions are outlined in [1]). The frontend allows one to propose *language-independent* analyses (to a certain extent) because it supports input programs to be written in multiple languages.

The next part of the architecture is the scheduler, which defines the order of the analysis of single functions according to the appropriate *call graph*[7]. The scheduler also checks if it is possible to simultaneously analyse some functions, allowing Facebook Infer to run the analysis in parallel.

[6] ~~**Abstract interpretation** is explained and formally defined in [1] —~~ Section 2.1.1. Additional description can be found in [11, 12, 5, 6, 4].

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

Example 2.1.

For demonstrating the order of the analysis in Facebook Infer and its incrementality, assume a call graph given in Figure 2. At first, leaf functions `F5` and `F6` are analysed. Further, the analysis goes on towards the root of the call graph — $\texttt{F}_{\texttt{MAIN}}$, while 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 reanalysis of functions `F4`, `F2`, and $\texttt{F}_{\texttt{MAIN}}$ only.

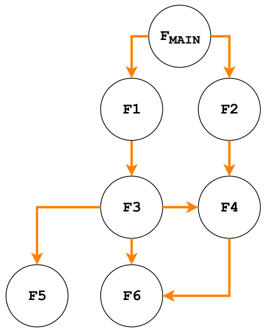

Figure 2. A call graph for an illustration of Facebook Infer's analysis process [1]

The last part of the architecture consists of a set of analyser plugins. Each plugin performs some analysis by interpreting SIL instructions. The result of the analysis of each function (function 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 3.

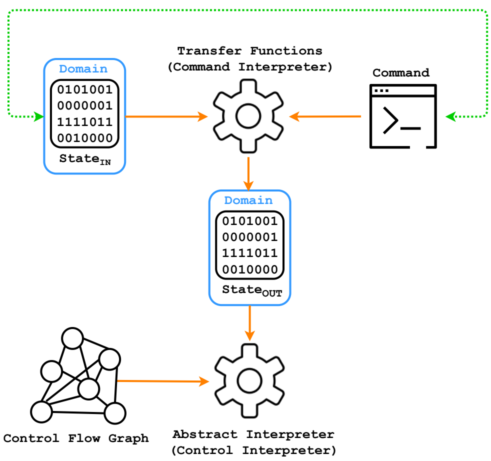

Figure 3. Facebook Infer's abstract interpretation process [1]

3. Atomer: Atomicity Violations Detector

This section introduces the principles of the *static analyser Atomer* proposed as a module of *Facebook Infer* for finding some forms of *atomicity violations*. Atomer was proposed and in detail described in the bachelor's thesis of the author of this paper [1]. Therefore, naturally, the following description is based on the mentioned thesis.

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

The proposal of Atomer is based on *basic contracts for concurrency* [1, 2]. These allow one to define *sequences of functions* required to be *executed atomically*. Atomer can automatically derive candidates for such contracts and then verify whether the contracts are fulfilled. Both of these steps are done statically. The proposed analysis is divided into two parts (*phases of the analysis*):

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

The phases are in depth described in the sections below. Moreover, these phases of the analysis and its workflow are illustrated in Figure 4.

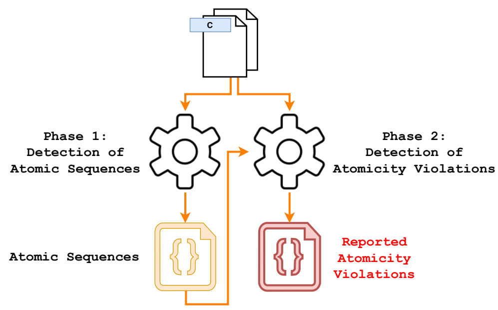

Figure 4. *Phases* of the Atomer's analysis and the analysis *high-level process* illustration

This section describes the proposal of Atomer in general. The concrete types of the *abstract states* (i.e., elements of the *abstract domain* $\boldsymbol{Q}$) and the *summaries* χ, along with the implementation of all necessary *abstract interpretation operators*, are stated in [1]. However, actually, the abstract states $s \in \boldsymbol{Q}$ of both phases of the analysis are proposed as *sets*. So, in fact, the *ordering operator* $\sqsubseteq$ is implemented using testing for a *subset* (i.e., $s \sqsubseteq s' \Leftrightarrow s \subseteq s'$, where $s, s' \in \boldsymbol{Q}$), the *join operator* $\sqcup$ is implemented as the *set union* (i.e., $s \sqcup s' \Leftrightarrow s \cup s'$), and the *widening operator* $\triangledown$ is implemented using the join operator (i.e., $s \triangledown s' \Leftrightarrow s \sqcup s'$).

3.1 Phase 1: Detection of Atomic Sequences

Phase 1 of Atomer detects *sequences of functions* that should be *executed atomically*. Intuitively, the detection is based on looking for sequences of functions executed atomically on some path through a program.

The detection of sequences of calls to be executed atomically is based on analysing all paths through the CFG of a function and generating all pairs $(A,B) \in \Sigma^* \times \Sigma^*$ (where Σ^* is a set of all possible sequences of functions from Σ in a given program) of *reduced sequences* [1] of function calls for each path such that: Here, A is a reduced sequence of function calls that appear between the beginning of the function being analysed and the first lock; between an unlock and a subsequent lock; or between an unlock and the end of the function being analysed. B is a reduced sequence of function calls that follow the calls from A and that appear between a lock and an unlock (or between a lock and the end of the function being analysed). Thus, the *abstract state* $s \in \boldsymbol{Q}$ is defined as $2^{2^{\Sigma^* \times \Sigma^*}}$ (because there is a set of the (A,B) pairs for each program path).

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

- $\boldsymbol{B}$ is a set constructed that contains all the B sequences that appear on program paths through `f`. In other words, this component of the summary is a set of sequences of atomic function calls appearing in an analysed function.
- AB is a *concatenation* of all the A and B sequences with removed duplicates of function calls. In particular, assume that there is the following computed set of (A,B) pairs: $\{(A_1,B_1), (A_2,B_2), \ldots, (A_n,B_n)\}$, then the result is concatenated sequence $A_1 \cdot B_1 \cdot A_2 \cdot B_2 \cdot \ldots \cdot A_n \cdot B_n$ with removed duplicates. Intuitively, in this component of the summary, it is gathered occurrences of all called functions within an analysed function obtained by concatenation of all the A and B sequences. AB is recorded to analyse functions higher in the *call hierarchy* since locks/unlocks can appear in such a *higher-level* function.

Example 3.1. For instance, the analysis of the function `f` from Listing 1 produces the following sequences:

$$\overbrace{\texttt{x, }\cancel{\texttt{x}}\texttt{, y}}^{A}\ \overbrace{\texttt{[a, b, }\cancel{\texttt{b}}\texttt{]}}^{B}$$

The strikethrough of the functions `x` and `b` denotes removing already recorded function calls in the A and B sequences. For the above, the abstract state at the end of an interpretation of the function `f` is $s_{\texttt{f}} = \{\{((\texttt{x},\texttt{y}),(\texttt{a},\texttt{b}))\}\}$. The derived summary $\chi_{\texttt{f}}$ for the function `f` is $\chi_{\texttt{f}} = (\{(\texttt{a},\texttt{b})\},(\texttt{x},\texttt{y},\texttt{a},\texttt{b}))$.

```c
void f() {
    x(); x(); y();
    lock(&L); // (a, b)
    a(); b(); b();
    unlock(&L);
}
```

Listing 1. A code snippet used for an illustration of the derivation of *sequences of functions called atomically*

The derived sequences of calls assumed to execute atomically — the $\boldsymbol{B}$ sequences — from the summaries of all analysed functions are stored into a file used during **Phase 2**, which is described below.

3.2 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) while this is not the case on some path through the CFG. The pairs of function calls to be checked for atomicity are obtained as follows: For each function `f` with the *summary* $\chi_{\texttt{f}} = (\boldsymbol{B}, AB)$ in a given program, it is taken the first component $\boldsymbol{B}$ of the summary $\chi_{\texttt{f}}$, i.e., $\boldsymbol{B} = \{B_1, B_2, \ldots, B_n\}$, and it is taken *every pair* $(\texttt{x},\texttt{y}) \in \Sigma \times \Sigma$ of functions that appear as a *substring* in some of the B_i sequences, i.e., $B_i = w \cdot \texttt{x} \cdot \texttt{y} \cdot w'$ for some sequences w, w'. Note that `x` could be ε (an empty sequence) if some B_i consists of a single function. All these "atomic pairs" are put into the set $\Omega \in 2^{\Sigma\times\Sigma}$.

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

Example 3.2. To demonstrate the detection of an atomicity violation, assume the functions `f` and `g` from Listing 2. The set of atomic sequences of the function `f` with the summary $\chi_{\texttt{f}} = (\boldsymbol{B}, AB)$ is $\boldsymbol{B} = \{(\texttt{a},\texttt{b},\texttt{c})\}$, thus $\Omega = \{(\texttt{a},\texttt{b}),(\texttt{b},\texttt{c})\}$. In the function g, an atomicity violation is detected because the pair of functions b and c is not called atomically.

```
void f() {
    lock(&L); // (a, b, c)
    a(); b(); c();
    unlock(&L);
}
void g() {
    // ATOMICITY_VIOLATION: (b, c)
    x(); b(); c(); y();
}
```

Listing 2. Example of an *atomicity violation*

The sets of atomicity violations Δ from individual functions are the final reported atomicity violations seen by a user.

4. Proposal of Enhancements for Atomer

5. Implementation and Experiments

6. Conclusions

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 project Aquas and H2020 ECSEL project Arrowhead Tools.

References

[1] D. Harmim. *Static Analysis Using Facebook Infer to Find Atomicity Violations*, 2019. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor T. Vojnar.

[2] R. J. Dias, C. Ferreira, J. Fiedor, J. M. Lourenço, A. Smrčka, D. G. Sousa, and T. Vojnar. Verifying Concurrent Programs Using Contracts. In *Proc. of ICST*, 2017.

[3] S. Blackshear. Getting the most out of static analyzers. *Speech at The @Scale Conference* [online], 2. September 2016 [cit. 2021-03-21]. Available at: https://atscaleconference.com/videos/getting-the-most-out-of-static-analyzers.

[4] A. Møller and I. M. Schwartzbach. Static program analysis, 2020. Department of Computer Science, Aarhus University.

[5] F. Nielson, R. H. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer Berlin Heidelberg, 2nd edition, 2005.

[6] X. Rival and Y. Kwangkeun. *Introduction to Static Analysis: An Abstract Interpretation Perspective*. The MIT Press, 1st edition, 2020.

[7] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional Shape Analysis by Means of Bi-Abduction. In *Proc. of POPL*, 2009.

[8] J. Villard. Infer powering Microsoft's Infer#, a new static analyzer for C#. *Facebook Engineering* [online], 14. December 2020 [cit. 2021-03-21]. Available at: https://engineering.fb.com/2020/12/14/open-source/infer.

[9] S. Blackshear, N. Gorogiannis, P. W. O'Hearn, and I. Sergey. RacerD: Compositional Static Race Detection. *Proc. of OOPSLA*, 2018.

[10] N. Gorogiannis, P. W. O'Hearn, and I. Sergey. A True Positives Theorem for a Static Race Detector. *Proc. of POPL*, 2019.

[11] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL*, 1977.

[12] P. Cousot and R. Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation, invited paper. In M. Bruynooghe and M. Wirsing, editors, *Proc. of PLILP*, 1992.

[13] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM*, 1969.

[14] F. E. Allen. Control Flow Analysis. In *Proc. of a Symposium on Compiler Optimization*, 1970.

[15] T. Reps, S. Horwitz, and M. Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proc. of POPL*, 1995.

[16] M. Sharir and A. Pnueli. Two Approaches to Interprocedural Data Flow Analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7. Prentice Hall Professional Technical Reference, 1981.