Advanced Static Analysis of Atomicity in Concurrent Programs through Facebook Infer

Dominik Harmim*

Java or C/C++ Code → ⊢ Infer → Atomicity Violations
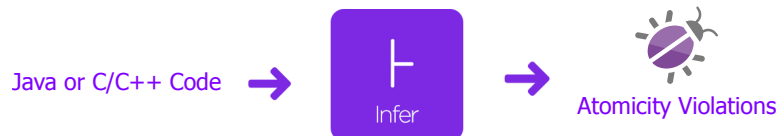
Abstract

This work aims to improve and extend *Atomer* significantly. It is a *static analyser* that detects *atomicity violations*. In particular, it detects *sequences of function calls* that are likely to be intended to execute atomically, but the atomicity is not always ensured. The first basic version of this checker was created within the bachelor's thesis of the author as a module of *Facebook Infer* — an open-source static analysis framework that aims a lot at *scalable* analysis. The idea underlying this approach is that sequences of calls executed *atomically once* should probably be executed *always atomically*. The original implementation hence naturally worked with call 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*; support for various new kinds of *locks* (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*. The new extensions were verified and evaluated on programs created for testing purposes. Furthermore, new experiments on *extensive real-life programs* were made, with the checker re-discovering some previously discovered but hard-to-find *real bugs*. The experiments revealed a *significant increase of the 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 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*[1]). 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*, *compos-*

---

[1]So-called **false alarms** (also *false positives*) are incorrectly reported errors, i.e., actually, they are not errors.

itional, *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* [1] 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 laborious and time-consuming is finding and fixing them.

As already mentioned, within the author's bachelor's thesis [1], *Atomer*[2] 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 from Listing 1 that replaces item `a` in an array by item `b`. It contains atomicity violation — the index obtained may be outdated when `set` is executed, 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 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. 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 *POSIX threads* and POSIX thread locks (*PThread locks*). Consequently, there was no support for other languages and their locking mechanisms in the first version of Atomer, preventing it from being applicable to a broader range of concurrent programs.

```
1  void replace(int a, int b) {
2      int i = array.index_of(a);
3      if (i >= 0) array.set(i, b); }
```

Listing 1. Example of *atomicity violation*

Within this work, 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, notably: support for *C++ and Java languages*, including various advanced kinds of locks these languages offer (such *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*. Lastly, new experiments were performed to test the Atomer's capabilities.

The development of the original Atomer started under the H2020 ECSEL project Aquas. The development of its new version is supported by the H2020 ECSEL project Arrowhead Tools. The development has been discussed with the developers of Facebook Infer too. Parts of the paper concerning the Facebook Infer framework and the basic version of Atomer are partially 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. Subsequently, Section 4 presents all the proposed extensions and improvements. The implementation of these extensions, together with the experimental evaluation of the new

---

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

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[3] and in [3]. Parts of this section are taken from [1].

Facebook Infer is an *open-source*[4] *static analysis framework*[5], which can discover various kinds of software bugs and which stress the *scalability* of the analysis. A more detailed explanation of Facebook Infer architecture is given in Section 2.2. Facebook Infer is implemented in *OCaml* — a *functional* programming language, also supporting *imperative* and *object-oriented* paradigms. 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 [7]. Once Facebook has purchased it, its scope significantly widened and abandoned the focus on sound analysis only.

Facebook Infer can analyse programs written in several languages. In particular, it supports the following languages: C, C++, Java, and Objective-C (as well as C#, see [8]). Moreover, it is possible to extend Facebook Infer's *frontend* for supporting other languages. Currently, Facebook Infer contains many analyses focusing on various kinds of bugs, e.g., *Inferbo* (buffer overruns); *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 perform *formal verification*. It can be used to quickly develop new sorts of *compositional* and *incremental* analysers (both *intraprocedural* and *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, which will be discussed later on. 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 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.

Further, to create an interprocedural analyser, it is required to define additionally: (i) The type of function summaries χ. (ii) 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, 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 *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 Infer AI

The architecture of the abstract interpretation framework of Facebook Infer (**Infer AI**) may be split into three major parts, as demonstrated in Figure 1: 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

---

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

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

[5]For an explanation of the general meaning of **static analysis**, see, e.g., [4, 5, 6].

[6]**Abstract interpretation** is formally defined and further explained in [11, 12, 4, 5, 6].
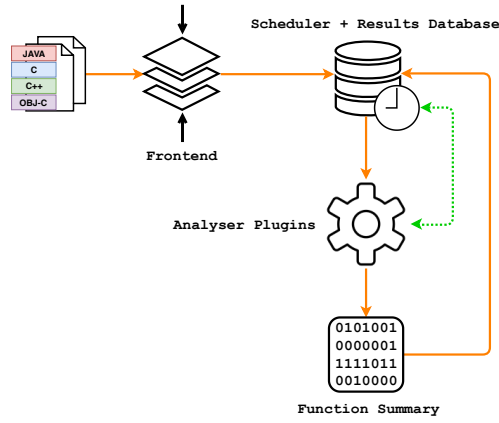
Figure 1. The architecture of Facebook Infer's abstract interpretation framework [1]
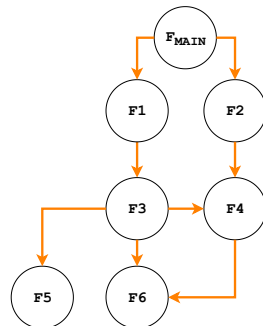
them as a CFG. There is a separate CFG representation for each analysed function. Nodes of this CFG are formed as SIL instructions (the individual instructions are outlined in [1]). The frontend allows one to propose *language-independent* analyses (to a certain extent) because it supports input programs 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.

Example 2.1.
For demonstrating the order of the analysis in Facebook Infer and its incrementality, assume the 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 — 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 reanalysis of functions F4, F2, and F_{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 analyser plugins. Each plugin performs some analysis by

---

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

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.

3. Atomer: Atomicity Violations Detector

This section introduces the principles of the basic version of the *Atomer static analyser* for finding some forms of *atomicity violations* proposed in the bachelor's thesis [1] of the author of this paper. 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. In principle, Atomer is based on the assumption that sequences of function calls executed *atomically once* should probably be executed *always atomically*. Atomer can both automatically derive those sequences that are sometimes executed atomically as well as subsequently check whether they are 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):

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 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., $s \triangledown s' \Leftrightarrow s \sqcup s'$) since the domains are *finite*.

3.1 Phase 1: Detection of Atomic Sequences

Phase 1 of Atomer detects *sequences of functions* that should be *executed atomically*. It is based on looking for sequences executed atomically, in particular, under some lock, on some path through a program.

The detection of such sequences is based on analysing all paths through the CFG of a function and gen-

erating all pairs $(A, B) \in \Sigma^* \times \Sigma^*$ (where Σ is the set of functions of a given program) of *reduced sequences*[8] 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 an 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.

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

- $\boldsymbol{B}$ 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 $\mathtt{f}$. In other words, this component of the summary is a set of sequences of atomic function calls appearing in $\mathtt{f}$.
- 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. Intuitively, in this component of the summary, the analysis gathers occurrences of all called functions within the analysed function. AB are 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.

Example 3.1. For instance, the analysis of the function $\mathtt{f}$ from Listing 2 produces the following sequences:

$$\overbrace{\mathtt{x} \cdot \cancel{\mathtt{x}} \cdot \mathtt{y}}^{A} \overbrace{[\mathtt{a} \cdot \mathtt{b} \cdot \cancel{\mathtt{b}}]}^{B}$$

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

The derived sequences of calls assumed to execute atomically — the $\boldsymbol{B}$ sequences — from the summaries

---

[8]A **reduced sequence** denotes a sequence in which the first appearance of each function is recorded only.

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

Listing 2. A code snippet used for an illustration of the derivation of *sequences of functions called atomically*

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) 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 $\mathtt{f}$ with a *summary* $\chi_{\mathtt{f}} = (\boldsymbol{B}, AB)$ in a given program, where $\boldsymbol{B} = \{B_1, B_2, \ldots, B_n\}$, the analysis considers *every pair* $(\mathtt{x}, \mathtt{y}) \in \Sigma \times \Sigma$ of functions that appear as a *substring* in some of the B_i sequences, i.e., $B_i = w \cdot \mathtt{x} \cdot \mathtt{y} \cdot w'$ for some sequences w, w'. Note that $\mathtt{x}$ could be ε (an empty sequence) if some B_i consists of a single function. All these "atomic pairs" are put into the set $\Omega \in 2^{\Sigma \times \Sigma}$.

An element of this phase's *abstract state* is a triple $(\mathtt{x}, \mathtt{y}, \Delta) \in \Sigma \times \Sigma \times 2^{\Sigma \times \Sigma}$ where $(\mathtt{x}, \mathtt{y})$ is a pair of the most recent calls of functions performed on the program path being explored, and Δ is a set of so far detected *pairs that violate atomicity*. Thus, the abstract states are elements of the set $2^{\Sigma \times \Sigma \times 2^{\Sigma \times \Sigma}}$. Whenever a function $\mathtt{f}$ is called on some path that led to an abstract state $(\mathtt{x}, \mathtt{y}, \Delta)$, a new pair $(\mathtt{x}', \mathtt{y}')$ of the most recent function calls is created from the previous pair $(\mathtt{x}, \mathtt{y})$ such that $(\mathtt{x}', \mathtt{y}') = (\mathtt{y}, \mathtt{f})$. Further, when the current program state is not inside an *atomic block*, the analysis checks whether the new pair (or just the last call) violates the atomicity (i.e., $(\mathtt{x}', \mathtt{y}') \in \Omega \vee (\varepsilon, \mathtt{y}') \in \Omega$). When it does, it is added to the set Δ of pairs that violate atomicity.

Example 3.2. To demonstrate the detection of an atomicity violation, assume the functions $\mathtt{f}$ and $\mathtt{g}$ from Listing 3. The set of atomic sequences of the function $\mathtt{f}$ with the summary $\chi_{\mathtt{f}} = (\boldsymbol{B}, AB)$ is $\boldsymbol{B} = \{\mathtt{a} \cdot \mathtt{b} \cdot \mathtt{c}\}$, thus $\Omega = \{(\mathtt{a}, \mathtt{b}), (\mathtt{b}, \mathtt{c})\}$. In the function $\mathtt{g}$, an atomicity violation is detected because the pair of functions $\mathtt{b}$ and $\mathtt{c}$ is not called atomically.

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

Listing 3. 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

Atomer was implemented in [1] according to the proposal discussed in the previous section. The analyser is working as expected. Moreover, it can be used in practice to analyse various kinds of programs, and it may find *real atomicity related bugs*. Nevertheless, there are still several limitations and cases where Atomer would not work correctly, i.e., cases not considered during the original proposal. Below, there are proposed solutions for some of the limitations. The solutions enhance the analysis's *precision* and *scalability*. Furthermore, Section 5 provides an overview of the implementation of a new version of Atomer.

4.1 Approx. of the Use of Sequences by Sets

Regarding *scalability*, Atomer can have problems with more *extensive* and *complex* programs (problems with *memory* as well as problems with *analysis time*). The problem is working with the sets of (A, B) pairs of *sequences* of function calls in *abstract states* (during **Phase 1**). It may be necessary to store many of these sequences, and they could be very long (due to all different paths through the CFG of an analysed program). It seems promising to *approximate* it (*abstraction refinement*) by working with the sets of (A, B) pairs of **sets** of function calls. Elements of these pairs are also occurring in the the first phase's *summaries*, and they are used during **Phase 2**. Thus, it is needed to make a certain approximation in these structures and algorithms likewise.

In particular, this proposed solution is more scalable because the order of stored function calls is not relevant anymore while working with sets. Therefore, less memory is required because the same sets of function calls are not stored multiple times. The analysis is also faster since there are stored fewer sets of function calls to work with. On the other hand, the analysis is less accurate because the new approach causes some loss of information. In practice, this loss of informa-

tion could eventually lead to *false alarms*. However, the number of such false alarms should not often be so significant.

The *detection of sequences of calls to be executed atomically* now generates all (A, B) pairs of *sets* of function calls for each path instead of pairs of sequences. I.e., $(A, B) \in 2^\Sigma \times 2^\Sigma$. The semantics of the pairs is preserved. So, the *abstract states* are elements of the set $2^{2^{2^\Sigma \times 2^\Sigma}}$. In all the implemented algorithms and definitions, it is sufficient to work with: (i) *sets* 2^Σ of functions, instead of *sequences* Σ^* of functions; (ii) *empty sets* $\emptyset$, instead of *empty sequences* ε; (iii) and a *union* $\cup$ of sets, instead of a *concatenation* $\cdot$ of sequences. Consequently, also the summary is redefined as $\chi \in 2^{2^\Sigma} \times 2^\Sigma$.

Example 4.1. For demonstrating the approximation of the analysis to sets, assume functions f and g from Listing 4. After the approximation, the produced abstract states and summaries are as follows (i.e., they are the same for both functions because there are the same locked/unlocked function calls, only the order of calls differs): $s_f = s_g = \{\{(\{a,b\}, \{x,y\})\}\}$, $\chi_f = \chi_g = (\{\{x,y\}\}, \{a,b,x,y\})$.

```
1   void f() {
2       a(); b();
3       lock(&L); // x.y -> {x, y}
4       x(); y();
5       unlock(&L); }
6   void g() {
7       b(); a();
8       lock(&L); // y.x -> {x, y}
9       y(); x();
10      unlock(&L); }
```

Listing 4. A code snippet used to illustrate the Atomer's **Phase 1** *approximation* of the analysis with *sets of function calls*

Detecting violations of atomicity works almost the same way as before the approximation. There is only one difference. Before, it was detected violations of atomic sequences obtained from **Phase 1**. Now, **atomic sets** are obtained; hence, the detection of violations of atomic sets is performed. It is needed to propose a new algorithm that derives pairs of function calls (from the atomic sets) to be checked for atomicity. To obtain the pairs, it is taken a union of sets containing all *2-element variations* of single atomic sets (i.e., all possible pairs).

Example 4.2. For example, assume that in **Phase 1**, there was analysed a function f, which produced

the summary $\chi_f = (\boldsymbol{B}, AB)$. Assume that a set of sets of functions that should be called atomically is the following: $\boldsymbol{B} = \{\{\mathtt{a},\mathtt{b},\mathtt{c}\}\}$, the analysis now looks for the following pairs of functions that are not called atomically (all 2-element variations): $\Omega = \{(\mathtt{a},\mathtt{b}),(\mathtt{a},\mathtt{c}),(\mathtt{b},\mathtt{a}),(\mathtt{b},\mathtt{c}),(\mathtt{c},\mathtt{a}),(\mathtt{c},\mathtt{b})\}$.

4.2 Advanced Manipulation with Locks

So far, Atomer does not work with *nested locks*, i.e., it does not distinguish *different lock instances* in a program. Only calls of locks/unlocks are identified, and the parameters of these calls (*lock objects*) are not considered. Thus, if there are several lock objects used, the analysis does not work correctly. Although this may happen in *real-life programs*, insomuch as one could have another (smaller) atomic section inside an outer atomic section or an *alternating lock sequence* may be in use.

In order to consider lock objects, it was proposed to distinguish between them using Facebook Infer's built-in mechanism called *access paths* [17]. The analyser does not perform a general *alias analysis*, i.e., it is not performed a precise analysis for saying when arbitrary pairs of accesses to lock objects may alias. During the analysis (both phases), each atomic section is identified by an access path of a lock that guards the section. Because *syntactically identical access paths* are used as the intuition for distinguishing atomic sections, some atomicity violations could be missed (or some false alarms could be reported) due to distinct access paths that refer to the same memory. However, it vastly simplifies the analysis, and the stress is put on finding likely violations.

The *syntactic access paths* represent *heap locations* via the paths used to access them, i.e., a base variable followed by a sequence of fields. More formally, let *Var* be a set of all variables that can occur in a given program. Let *Field* be a set of all possible field names that can be used in the program (e.g., structure fields). An access path π from the set Π of all access paths is then defined as follows: $\pi \in \Pi ::= Var \times Field^*$. Access paths are already implemented in Facebook Infer. For instance, the principle of using access paths is used in an existing analyser in Facebook Infer — RacerD [9] — for data race detection. In general, no sufficiently precise *alias analysis* works *compositionally* and at *scale*. That is the motivation for using access paths in Facebook Infer and Atomer.

Another Atomer's limitation is that it does not count with *re-entrant locks*, i.e., a process can lock the same lock object multiple times without blocking itself, then it unlocks the lock object the same number of times. This approach is, in fact, widespread, e.g.,

in Java, where so-called *synchronised blocks* are used. These blocks are re-entrant by default. To consider re-entrant locks in the analysis, the number of locks of individual lock objects is tracked in the abstract states of both phases of the analysis. A lock is unlocked as soon as this number decreases to 0. Also, an input parameter $t \in \mathbb{N}$ was proposed to limit the upper bound of the number of the same lock is held. This is to ensure the *termination* of the analysis.

The *detection of sets of calls to be executed atomically* is based on generating all (A, B) pairs. Now, it is needed to store access paths and the number of locks of lock objects that guard calls executed atomically, i.e., the B sets. Therefore, these pairs are extended to the tuples $(A, B, \pi, l) \in 2^\Sigma \times 2^\Sigma \times \Pi \times \mathbb{N}_0$, where π is an access path that identifies a lock object which locks an atomic section that contains the calls from B, and l is the number of locks of the lock identified by π. As a consequence, the *abstract states* are elements of the set $2^{2^{2^\Sigma \times 2^\Sigma \times \Pi \times \mathbb{N}_0}}$.

Example 4.3. Consider the function $\mathtt{f}$ from Listing 5. There are two lock objects $\mathtt{L1}$ and $\mathtt{L2}$, that are used simultaneously. Moreover, $\mathtt{L2}$ is locked twice without unlocking in between. After the extension described above, the produced *summary* is the following: $\chi_\mathtt{f} = (\{\{\mathtt{b}\},\{\mathtt{a},\mathtt{b},\mathtt{c}\}\},\{\mathtt{a},\mathtt{b},\mathtt{c}\})$. Moreover, the abstract state after the execution of line 5 is as follows: $s_{\mathtt{f}_5} = \{\{(\emptyset,\{\mathtt{a},\mathtt{b}\},\mathtt{L1},1),(\{a\},\{\mathtt{b}\},\mathtt{L2},2)\}\}$.

```
1  void f() {
2      lock(&L1); // {a, b, c}
3      a();
4      lock(&L2); lock(&L2); // {b}
5      b();
6      unlock(&L2); unlock(&L2);
7      c();
8      unlock(&L1); }
```

Listing 5. A code snippet used to illustrate the *advanced manipulation with locks* during **Phase 1**

While looking for *atomicity violations of pairs of function calls*, from now, the analysis stores (in addition to pairs of the most recent function calls $(\mathtt{x},\mathtt{y})$ and the set Δ of pairs that violate atomicity so far) all the most recent pairs of function calls locked under individual locks. In this phase, the *abstract state* element is $(\mathtt{x},\mathtt{y},\Delta,\Lambda) \in \Sigma \times \Sigma \times 2^{\Sigma \times \Sigma} \times 2^{\Sigma \times \Sigma \times \Pi \times \mathbb{N}_0}$, where Λ is a set of the most recent function calls with their locks' access paths and the number of locks of lock objects of these locks. Thus, the abstract states are elements of the set $2^{\Sigma \times \Sigma \times 2^{\Sigma \times \Sigma} 2^{\Sigma \times \Sigma \times \Pi \times \mathbb{N}_0}}$. The analysis works as follows. When a function $\mathtt{f}$ is called on some

path that led to an abstract state (x, y, Δ, Λ), a new pair (x', y') of the most recent function calls is created from the previous pair (x, y) such that $(x', y') = (y, f)$. This pair is also stored in the locked pairs Λ if there are any locks currently locked. Further, it is checked whether the new pair (or just the last call) violates the atomicity, and at the same time, it is not locked by any of the stored locks (i.e., $((x', y') \in \Omega \wedge (x', y') \notin \Lambda) \vee ((\varepsilon, y') \in \Omega \wedge (\varepsilon, y') \notin \Lambda))$. When it holds, it is added to the set Δ of pairs that violate atomicity.

Example 4.4. Consider the function g from Listing 6. There are two lock objects L1 and L2, that are used simultaneously. Then assume that the result of the first phase of the analysis is that the pair (a, b) should be called atomically, i.e., $\Omega = \{(a, b)\}$. Before the extension for the distinguishment of multiple lock instances, the analysis would report an atomicity violation of these functions (line 5). This is because the locks are not distinguished, and the unlock of L1 (line 4) would unlock everything. On the other hand, after the extension, there are not reported any violations because the pair is still locked using L2. The abstract state of f after the execution of line 5 looks like follows: $s_{f_5} = \{(a, b, \emptyset, \{(a, b, L2, 1)\})\}$.

```
1  void g() {
2      lock(&L1); // {}
3      lock(&L2); // {a, b}
4      unlock(&L1);
5      a(); b();
6      unlock(&L2); }
```

Listing 6. A code snippet used to illustrate the *advanced manipulation with locks* during **Phase 2**

Finally, support for so-called *lock guard objects* was proposed. Lock guards are objects associated with lock objects. One lock guard can be associated with multiple lock objects, and one lock object can be associated with multiple lock guards. When a lock guard is created, all lock objects associated with it are locked. When a lock guard is destroyed (usually, when a *scope of variables* is left), all lock objects associated with it are automatically unlocked. Although, under certain circumstances, lock guards can be locked/unlocked manually. Lock guards are widely used, especially in C++, but they are used, e.g., in Java as well. To cope with them, lock guards are also identified by *access paths*. The association between a lock guard and lock objects is modelled as a pair $(\pi_g, L) \in \Pi \times 2^\Pi$, where π_g is an access path that identifies the lock guard and L is a set of access paths that identify the lock objects associated with the guard identified by π_g. In

the *abstract states* of both phases of the analysis, associations between lock guards and lock objects are maintained as a set that is an element of the set $2^{\Pi \times 2^\Pi}$, i.e., there is a set of associations between lock guards and lock objects. Subsequent locks/unlocks of lock guards are then interpreted as a sequence of locks/unlocks of lock objects associated with these lock guards.

4.3 Analysis's Parametrisation

One of the main reasons that Atomer reports *false alarms* is that in *critical sections*, in practice, there are sometimes called *generic functions* that do not influence atomicity violations (such as functions for printing to the standard output, functions for recasting variables to different types, functions related to iterators, and whatever other "safe" functions for particular program types). Often, to find some atomicity violations, it is sufficient to focus only on certain "critical" functions. In general, usually, calls of *constructor* and *destructor* methods of classes do no effect atomicity violations. Therefore, these calls could be implicitly ignored. Nonetheless, it can be demanding to identify the functions that should be set aside or the functions to focus on because it is application-specific. Although, a user may provide this information to the analysis. For this reason, the following input parameters of the analysis are proposed: (i) a list of functions that will not be analysed; (ii) a list of functions that will be analysed (and all other functions will not be); (iii) a list of functions whose calls will not be considered; (iv) and a list of functions whose calls will be considered (and all other function calls will not be). In other words, there are *black-lists* and *white-lists* of functions to analyse and function calls to consider. It is possible to combine these parameters, and they can be enabled for individual phases of the analysis. The parameters are proposed to be file names of readable text files containing one function name per line. Moreover, suppose the line starts with character R followed by a whitespace. In that case, the function name is considered a *regular expression* so that several function names could be taken into account at once.

Another issue is that in an analysed program, there could be "large" critical sections or critical sections in which appear function calls with a *deep hierarchy of nested function calls*. Both cases could cause massive and "imprecise" atomic sets that are the source of false alarms.

To resolve the problem of large critical sections, input parameter $p \in \mathbb{N}$ is proposed. During the first phase of the analysis, from the (A, B) pairs, there are removed such pairs where $|B| > p$, i.e., removed such

pairs where the number of functions in the set B (functions called atomically) is greater than the limit p.

During the first phase, when calling an already analysed *nested function*, the AB set (i.e., the set of all called functions within a function) from its summary is used. If there is a deep hierarchy of nested function calls, at the top level of the hierarchy, there are used function calls from all lower-level functions, leading to large critical sections. To avoid this problem, the summary $\chi = (\boldsymbol{B}, AB) \in 2^{2^{\Sigma}} \times 2^{\Sigma}$ in **Phase 1** is redefined as $2^{2^{\Sigma}} \times 2^{\mathbb{N}_0 \times 2^{\Sigma}}$, i.e., AB is no longer a set of all functions called within an analysed function. It is a set of pairs where each pair represents called functions at a particular level in the hierarchy of a nested function (0 means the top-level). For instance, the summary $\chi_{\mathtt{f}} = (\emptyset, \{(0, \{\mathtt{a}, \mathtt{b}\}), (1, \{\mathtt{x}, \mathtt{y}\})\})$ of a function $\mathtt{f}$ means that there were called functions $\mathtt{a}, \mathtt{b}$ in $\mathtt{f}$ and that there were called functions $\mathtt{x}, \mathtt{y}$ in functions one level lower in the call tree (i.e., in functions directly invoked from $\mathtt{f}$). During the analysis, the summaries are forwarded among functions in the call hierarchy, and stored function calls are merged in appropriate levels. Furthermore, the input parameter $r \in \mathbb{N}_0$ is proposed to limit the number of levels considered during analysing nested functions.

Example 4.5. Assume function $\mathtt{f}, \mathtt{g}, \mathtt{h}$ from Listing 7. Their summaries for the first phase are as follows: $\chi_{\mathtt{h}} = (\emptyset, \{(0, \{\mathtt{x}, \mathtt{y}\})\})$, $\chi_{\mathtt{g}} = (\emptyset, \{(0, \{\mathtt{c}, \mathtt{d}, \mathtt{h}\}), (1, \{\mathtt{x}, \mathtt{y}\})\})$, $\chi_{\mathtt{f}} = (\emptyset, \{(0, \{\mathtt{a}, \mathtt{b}, \mathtt{g}\}), (1, \{\mathtt{c}, \mathtt{d}, \mathtt{h}\}), (2, \{x, y\})\})$. When the parameter $r = 1$ is used, the summary of the function $\mathtt{f}$ changes as follows: $\chi'_{\mathtt{f}} = (\emptyset, \{(0, \{\mathtt{a}, \mathtt{b}, \mathtt{g}\}), (1, \{\mathtt{c}, \mathtt{d}, \mathtt{h}\})\})$.

```
1  void h() { x(); y(); }
2  void g() { c(); h(); d(); }
3  void f() { a(); g(); b(); }
```

Listing 7. A code snippet used to illustrate the limitation of considered nested functions

5. Implementation and Experiments

[[TODO]]

6. Conclusions

[[TODO]]

Acknowledgements

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 *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 *Program Flow Analysis: Theory and Applications*, chapter 7. Prentice Hall, 1981.

[17] J. Lerch, J. Spath, E. Bodden, and M. Mezini. Access-Path Abstraction: Scaling Field-Sensitive Data-Flow Analysis with Unbounded Access Paths (T). In *Proc. of ASE*, 2015.