Advanced Static Analysis of Atomicity in Concurrent Programs through Facebook Infer

Author: **Dominik Harmim** | Supervisor: prof. Ing. Tomáš Vojnar, Ph.D.

Brno University of Technology, Faculty of Information Technology

`https://github.com/harmim/infer`

Motivation and Goals

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. Furthermore, it is generally challenging to avoid errors in **atomicity-dependent programs**, especially in large projects, and finding and fixing them is even more laborious and time-consuming. The paper [1] discusses the importance of **atomicity-related bugs** and shows some bugs in **real-world programs**. Unfortunately, tool support for automatically discovering such kinds of errors is currently minimal.

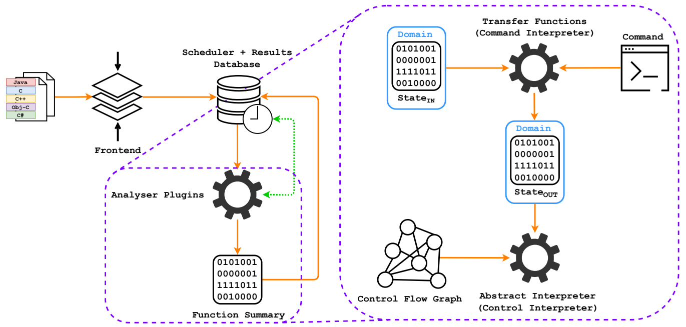
For example, assume the below code snippet where `method` is invoked on `server` only if it is registered on it. Methods `isRegistered` and `invoke` should be **executed atomically**. If **not locked**, `method` can be unregistered by a **concurrent thread**.

```
if (server.isRegistered(method))
   server.invoke(method);
```

Goal of the Thesis

The thesis aims to **improve the detection of atomicity violations** within **Atomer** [2] implemented in the **Facebook Infer framework**. The improvements aim at both increasing **scalability** as well as **precision**.

Facebook Infer Framework



- Open-source **static analysis framework** for **interprocedural analyses**.
  - Based on **abstract interpretation**.
  - Provides analysers that check, e.g., for buffer overflows, data races, null-dereferencing, memory leaks, or some forms of deadlocks and starvation.
  - Lacks better support for **concurrency bugs**. **Atomer** [2] is the only available checker of **atomicity of call sequences**.
- **Highly scalable**.
  - Follows principles of **compositionality** and **incrementality**.
  - Computes function **summaries bottom-up** on call-trees.
- Supports C, C++, Java, Objective-C, and C#.

Atomer: Atomicity Violations Analyser

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. In fact, the idea of checking the atomicity of certain sequences of function calls is inspired by the work of **contracts for concurrency** [1].

The first version of Atomer was proposed and implemented as a **plugin** of the **Facebook Infer framework** within the author's BSc thesis [2]. The implementation targets **C programs** that use **PThread locks**. However, the **scalability** and **precision** of the first Atomer's version are limited on **large codebases**.

Atomer can both **automatically derive** sequences of functions 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 analysis is thus divided into two parts (**phases of the analysis**):

1. Detection of **atomic call sets**.
   - **Approximates sequences by sets.**
   - **Summary**: $\chi \in 2^{2^{\Sigma}}$ (set of atomic call sets)

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

$$\chi_f = \{\{x, y\}\}$$

2. Detection of **atomicity violations**.
   - Derives "**atomic pairs**" from the first phase: $\Omega \in 2^{\Sigma \times \Sigma}$.
   - Looks for "**non-atomic pairs**" of calls assumed to **run atomically**.
   - **Summary**: $\chi \in 2^{\Sigma \times \Sigma}$ (set of atomicity violations)

```
void g() {
  a(); x(); y(); b();
}
```

$$\Omega = \{(x, y), (y, x)\}$$

Atomicity Violation!

$$(x, y) \in \Omega \Longrightarrow \chi_g = \{(x, y)\}$$

Proposed Enhancements

Within this 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. In particular, the following enhancements were implemented:

- **Approximating sequences** of calls by **sets** of calls (improves scalability).
- Support for **C++** and **Java**.
  - **Advanced manipulation with locks**: re-entrant locks, monitors, lock guards, etc.
- Distinguishing **different lock instances**.
  - **Approximating locks** using **syntactic access paths** [3] – a representation of **heap locations** via the paths used to access them.
  - Formally, an access path is defined as follows: $\pi \in \Pi ::= Var \times Field^{*}$ where Var is a set of all variables and $Field$ is a set of all field names.
- Analysis's **parametrisation** (aims to reduce the number of **false alarms**):
  - possibility to **ignore generic functions** and/or **concentrate on critical functions**;
  - **limiting** the **number of calls** and/or the **depth of nested calls** in **critical sections**.
- Considering **interprocedural locks** when checking for atomicity violations.

Experimental Evaluation

The **scalability** of the analysis has been evaluated on 61 **real-life complex concurrent** C programs (806,431 LOC in total) derived from the Debian GNU/Linux distribution. The table below shows aggregated results of the evaluation. There are times of analyses for both phases of the analysis for both the first/new version of Atomer, i.e., v1.0.0/v2.0.0, resp. On average, the new version of Atomer is about **twice faster**.

	v1.0.0		v2.0.0	
	Phase 1	Phase 2	Phase 1	Phase 2
Average Time (s)	70.98	109.11	37.96	50.93
Total Time (s)	4,117	5,892	2,164	2,750

Furthermore, two **open-source real-life extensive** (both ~250 KLOC) Java programs were analysed – **Apache Cassandra** 3.11 and **Apache Tomcat** 8.5. Atomer successfully **rediscovered already fixed reported real atomicity-related bugs** (they were originally discovered in [1]). The number of reported bugs by the Atomer's new version was **significantly reduced** (~4×).

Summary

Within this thesis, Atomer's **scalability** was improved using the **approximation of call sequences by sets**. Furthermore, several new features were implemented in the new version of Atomer, e.g., support for **C++** and **Java** (including various **advanced kinds of locks**, such as re-entrant locks or lock guards), a more precise way of **distinguishing between different lock instances**, or the analysis **parametrisation**.

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**.

Future Work

The future work will focus mainly on further increasing **accuracy**/reducing the number of **false alarms**, e.g., by:

- combining with a **dynamic analysis**;
- **statistic ranking** of atomic functions/reported errors;
- considering **formal parameters** of functions/methods involved in the contracts;
- or **machine learning** of appropriate values of the analysis' parameters.

References

[1] 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.

[2] 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.

[3] 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.

Acknowledgements

This work was supported by the H2020 ECSEL projects *AQUAS: Aggregated Quality Assurance for Systems*, *Arrowhead Tools for Engineering of Digitalisation Solutions*, and *VALU3S: Verification and Validation of Automated Systems' Safety and Security*.