# Static Analysis Using Facebook Infer to Find Atomicity Violations

PP1 – Project Practice 1

Dominik Harmim

Supervisor: prof. Ing. Tomáš Vojnar, Ph.D.

xharmi00@stud.fit.vutbr.cz

Brno University of Technology, Faculty of Information Technology

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

29th June 2020

# Motivation

- Detecting and checking desired atomicity of call sequences.
  - Often required in concurrent programs.
  - Violation may cause nasty errors.

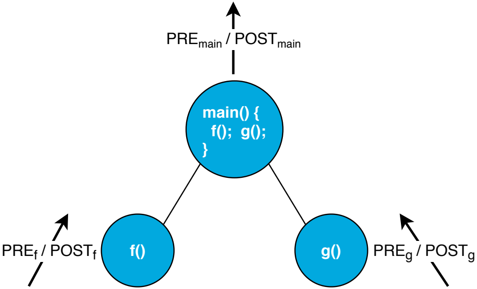
```
void invoke(char *method) {
    ...
    if (server.is_registered(method)) {
        server.invoke(method);
    }
    ...
}
```

The sequence of **`is_registered`** and **`invoke`** should be executed atomically.

If not locked, the method can be unregistered by a concurrent thread.

# Facebook Infer

- An open-source static analysis framework for interprocedural analyses.
  - Based on abstract interpretation.
- Highly scalable.
  - Follows principles of compositionality.
  - Computes function summaries bottom-up on call trees.
- Supports Java, C, C++, and Objective-C.



# Atomer: Atomicity Violations Analyser

- A Facebook Infer plugin created within a bachelor's thesis.
- **Assumption**: Call sequences executed atomically once should be executed always atomically.
- Implemented for C/C++/Java programs that use classical mutual exclusion mechanisms.

# Atomer: Two Phases of the Analysis

1. Detection of atomic call sets.

- Approximates sequences by sets.
- **Summaries**: (set of all calls, set of atomic call sets)

```
void f() {
    lock(L);
    f1(); f1(); f2();
    unlock(L);
    a();
    lock(L);
    b(); c();
    unlock(L);
}
```

**summary**$_f$:

`({f1, f2, a, b, c}, {{f1, f2}, {b, c}})`

2. Detection of atomicity violations.

- Looks for non-atomic pairs of calls assumed to run atomically.
- **Summaries**: (set of first calls, set of last calls, set of atomicity violations)

```
void g() {
    a();
    f1(); f2();
    b();
}
```

**summary**$_g$:

`({a}, {b}, {(f1, f2)})`

# Atomer: New Main Features

- Approximation atomic calls sequences by sets.

- Support for C++ and Java locks (earlier supported only Pthreads).
  - **C++**: `std::mutex`, `std::lock`, `std::lock_guard`, `std::shared_mutex`, `std::timed_mutex`, `std::recursive_mutex`, `std::unique_lock`, `...`
  - **Java**: monitors (`synchronized`), `java.util.concurrent.locks.Lock`, `java.util.concurrent.locks.ReentrantLock`, `...`

- Distinguishes multiple (nested) locks using syntactic access paths.

# Experimental Evaluation

**Earlier Evaluation:**

- The correctness was first verified on hand-crafted programs.
- Real-life low-level concurrent C programs from a Debian-based benchmark suite.
  - Several potential atomicity violations have been found.

**New Evaluation:**

- More had-crafted programs to verify the correctness of the new features.
- Real-life Java programs – Apache Cassandra and Tomcat (∼200k LOC).
  - Successfully rediscovered already fixed reported real bugs.
  - So far quite some false alarms – need to increase accuracy.

# Future Goals

- Further analysis of real-life programs with an effort to find and report new bugs.
  - GNU Coreutils/Binutils, Mozilla, MariaDB, . . .
  - Focus on library containers concurrency restrictions related to method calls.
- Increase accuracy.
  - Distinguishing the context of called functions by considering formal parameters (using syntactic access paths).
  - Ranking of atomic functions.