Advanced Static Analysis of Atomicity in Concurrent Programs through Facebook Infer

SEP — Term Project

Dominik Harmim

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

xharmi00@stud.fit.vutbr.cz

Brno University of Technology, Faculty of Information Technology

3rd February 2021

Motivation

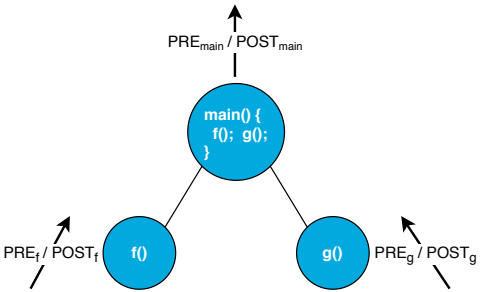
- Detecting and checking desired atomicity of call sequences:
  - Often required in concurrent programs.
  - Violation may cause nasty errors.

```
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.

- 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 C, C++, Java, Obj-C (and C#).



Atomer: Atomicity Violations Analyser

- A Facebook Infer plugin created within the bachelor's thesis:

  HARMIM, D. *Static Analysis Using Facebook Infer to Find Atomicity Violations*. Brno, 2019. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor VOJNAR, T.

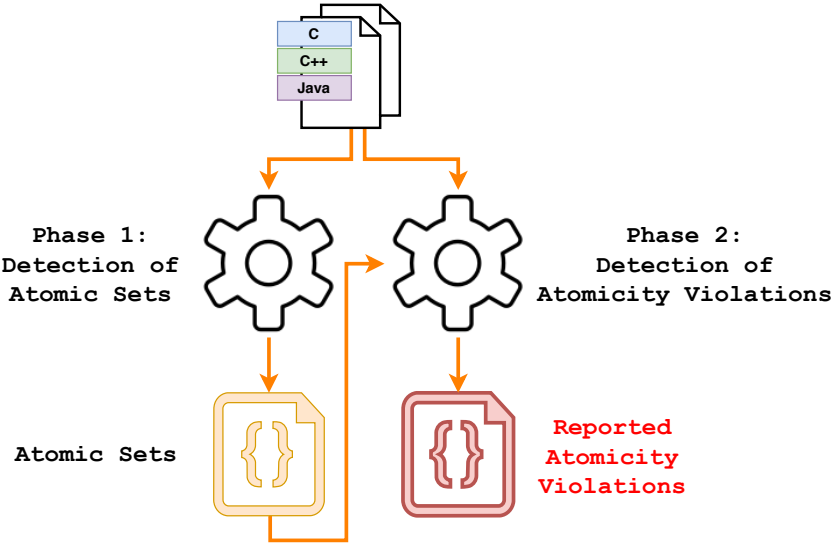
- **Assumption**: Call sequences executed atomically once should (probably) be executed always atomically.

- Implemented for C programs that use PThread locks.

- Limited scalability on extensive codebases.

- Reports many false alarms when analysing real-life code.

High-Level Analysis Process (with Extensions)



Two Phases of the Analysis (Improved Scalability)

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

```
void f() {
    lock(L);
    x(); y(); z();
    unlock(L);
    a();
    lock(L);
    z(); y(); x();
    unlock(L);
}
```

$Summary_f$: **({a, x, y, z}, {{x, y, z}})**

$Summary'_f$: ((x, y, z, a), {(x, y, z), (z, y, x)})

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();
    x(); y();
    b();
}
```

$AtomPairs_f$: **{(x, y), (x, z), (y, x), (y, z), (z, x), (z, y)}**

$AtomPairs'_f$: {(x, y), (y, z), (z, y), (y, x)}

$Summary_g$: **({a}, {b}, {(x, y)})**

Precision Enhancements and Other Extensions

- Support for C++ and Java:
  - **C++**: `std::mutex, std::lock, std::lock_guard, ...`
  - **Java**: monitors (`synchronized`), `Lock, ReentrantLock, ...`
- Distinguishing multiple (nested) locks used:
  - Approximates lock objects using syntactic access paths — a representation of heap locations via the paths used to access them.
- Parametrisation of the analysis by a user:
  - ignore generic functions/concentrate on critical functions,
  - limit the number of calls or the depth of analysing nested calls in critical sections.

Experimental Evaluation

- The correctness of the extensions was verified on hand-crafted programs.
- 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 further increase accuracy.

Summary

Extensions for Atomer:

- Proposed and implemented extensions for Atomer:
  - approximation of sequences by sets, support for C++ and Java, distinguishing multiple locks, parametrisation of the analysis.
- Successfully tested and experimentally evaluated.
- Experiments with real-life programs.

Future goals:

- Further analysis of real-life programs with an effort to find and report new bugs.
- Increase accuracy/reduce the number of false alarms:
  - Support for interprocedural locks.
  - Combination with dynamic analysis.
  - Statistic ranking of atomic functions/reported errors.