Advanced Static Analysis of Atomicity in Concurrent Programs through Facebook Infer

Excel@FIT 2021 — Paper No. 57

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

6th May 2021

Motivation

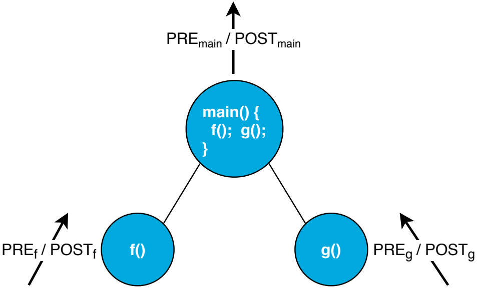
- Detecting and checking desired atomicity of function call sequences.
  - Often required in concurrent programs.
  - Violation may cause nasty errors.

```
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.

- 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, C#.

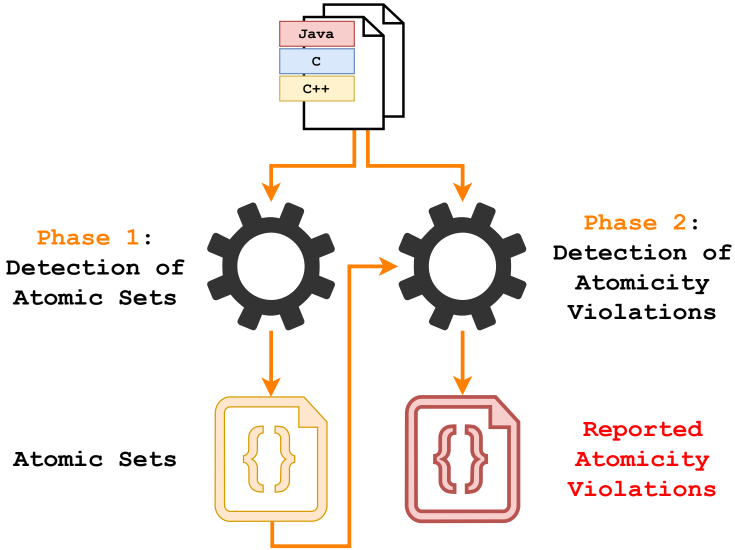


- Facebook Infer plugin created within the author's BSc thesis:

  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



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

```
void f() {
  lock(L);
  x(); y(); z(); // x.y.z -> {x,y,z}
  unlock(L);
  a();
  lock(L);
  z(); y(); x(); // z.y.x -> {x,y,z}
  unlock(L);
}
```

$$\chi_{\mathbf{f}} = (\{\mathbf{a}, \mathbf{x}, \mathbf{y}, \mathbf{z}\}, \{\{\mathbf{x}, \mathbf{y}, \mathbf{z}\}\})$$
$$\chi'_{\mathtt{f}} = (\mathtt{x} \cdot \mathtt{y} \cdot \mathtt{z} \cdot \mathtt{a}, \{\mathtt{x} \cdot \mathtt{y} \cdot \mathtt{z}, \mathtt{z} \cdot \mathtt{y} \cdot \mathtt{x}\})$$

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();
}
```

$$\Omega = \{(\mathbf{x}, \mathbf{y}), (\mathbf{x}, \mathbf{z}), (\mathbf{y}, \mathbf{x}), (\mathbf{y}, \mathbf{z}), (\mathbf{z}, \mathbf{x}), (\mathbf{z}, \mathbf{y})\}$$
$$\Omega' = \{(\mathtt{x}, \mathtt{y}), (\mathtt{y}, \mathtt{z}), (\mathtt{z}, \mathtt{y}), (\mathtt{y}, \mathtt{x})\}$$
$$(\mathtt{x}, \mathtt{y}) \in \Omega \implies \chi_{\mathbf{g}} = \{(\mathbf{x}, \mathbf{y})\}$$

Further Atomer's Enhancements

- Support for C++ and Java.
  - Working with advanced locks: re-entrant locks, monitors, lock guards, etc.
- Distinguishing different lock instances.
  - Approximating lock objects using syntactic access paths — a representation of heap locations via the paths used to access them.
- Analysis's parametrisation:
  - ignoring generic functions — concentrating on critical functions;
  - limiting the number of calls or the depth of nested calls in critical sections.

Experimental Evaluation

- Scalability evaluated on 54 real-life complex C programs.
  - 806,431 LOC in total.
- Double acceleration in average.

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

- Experiments with Apache Cassandra and Apache Tomcat (both $\sim$250 KLOC).
  - Successfully rediscovered already fixed reported real bugs.
  - The number of reported bugs was significantly reduced ($\sim 4\times$).
  - Still hard to say which of the bugs are real — the accuracy needs to be further improved.

Summary

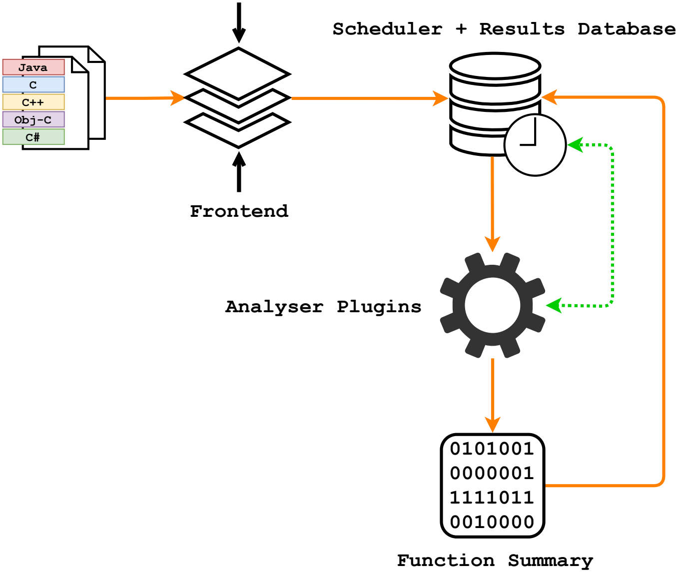
Atomer's Extensions:

- Proposed and implemented extensions for Atomer:
  - approximation with sets, support for C++ and Java, distinguishing different lock instances, parametrisation of the analysis.
- Successfully tested and experimentally evaluated.
  - Both scalability and accuracy were increased.
- Experiments with real-life programs.

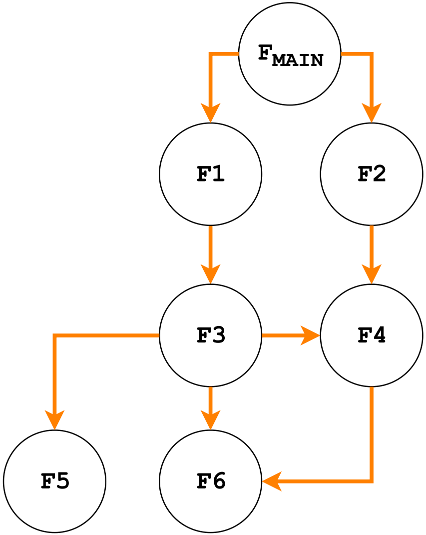
Future goals:

- Increase accuracy/reduce the number of false alarms:
  - Combining with dynamic analysis.
  - Statistic ranking of atomic functions/reported errors.
  - Considering formal parameters of function.
  - Machine learning of analysis parameter values.

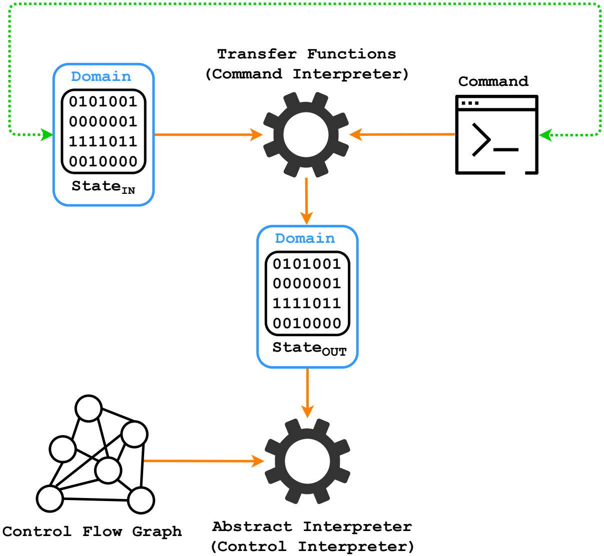
Facebook Infer's Architecture



Demonstration of Facebook Infer's Analysis



Abstract Interpretation in Facebook Infer



Real-life bug in package `org.apache.catalina.core.StandardContext`.

```
public void addParameter(String name, String value) {
  ...
  if (parameters.get(name) != null)
    throw new IllegalArgumentException
      (sm.getString("standardContext.parameter.duplicate", name));

  // Add this parameter to our defined set
  synchronized (parameters) {
    parameters.put(name, value);
  }
  fireContainerEvent("addParameter", name);
}
```

Advanced Manipulation with Locks

- Access path used for locks identification: $\pi \in \Pi ::= Var \times Field^*$
  - *Var* is a set of all variables,
  - *Field* is a set of field names.
- Identification of critical sections: $(\pi, l) \in \Pi \times \mathbb{N}^\top$
  - π is an access path that identifies the lock object that locks the section,
  - l is the number of locks of the lock object identified by π,
  - $\mathbb{N}^\top$ denotes $\mathbb{N} \cup \{\top\}$
    - $\top$ represents a number larger than some upper bound $t \in \mathbb{N}$.
- Representation of lock guards: $(\pi_g, L) \in \Pi \times 2^\Pi$
  - π_g is is an access path that identifies the lock guard,
  - L is a set of access paths that identify the lock objects associated with the guard identified by π_g.