Finding Incorrect Compositions of Atomicity
Peng Liu, Julian Dolby, Charles Zhang
Seminar Presentation

Claudio Corrodi

March 11, 2014
Problem Statement

- Concurrent APIs provide atomic operations.
- User composes APIs to new atomic functionality.

```
1: x = pos.getX();
2: y = pos.getY();
// (x, y) only valid if called atomically
```

- Problematic interleaving:

```
P1: pos = new Pos(0, 0);
P1: x = pos.getX();
P2: pos.move(5, 5);  
P1: y = pos.getY();  // P1 sees (0, 5)
```

- Problem: Identify compositions and find out whether they need to be implemented atomically.
Contributions

- Identify the problem of **incorrect compositions of atomic library APIs**.
- **Automatic** approach to find incorrect compositions.
- **Extensive evaluation** of the approach.
Algorithm

1. Inferring atomic sets.
2. Identifying library and client using atomic sets.
3. Inferring atomic compositions.
4. Exhibiting synchronization errors.
Atomic Sets

- Definition in the paper:
  “In object-oriented programs, where objects form reference hierarchies via field references, an atomic set is a set of instance fields, each of which is reachable from the root object along a field chain.”

- Atomic sets are inferred statically for each synchronized block or method.

- Sets that share fields are merged.
Atomic Sets

- Example:

```java
Account acc
Owner myOwner
String name
String address
int balance
```

```java
{acc.balance, acc.myOwner, acc.myOwner.name, ... }
```
Library Example

```java
public class Account {
    private int balance;
    private Owner myOwner;

    public synchronized void deposit() {
        /* ... */
    }

    public synchronized int withdraw(int amount) {
        /* ... */
    }

    public synchronized int getBalance() {
        return balance;
    }
}
```
public class Client {
    private Account account;

    /* Uses atomic APIs withdraw and getBalance from Account library. */
    public static void main(String[] args) {
        int balance, cash;
        balance = account.checkBalance();
        cash = account.withdraw(100);
    }
}
Library and Client

- Library module: Classes which declare the fields in the atomic set.
  Example: \{Account, Owner\}

- Client module: The class of which the methods invoke the atomic methods of the library module.
  Example: class Client

- Library and client are inferred **statically**.
Atomic Compositions

- **Atomic API**: Methods that use fields of the atomic set in a synchronized block.

- **Atomic Composition**: Using multiple atomic APIs in a single method.

- Does not always need to be atomic:

  ```java
cash = someAccount.withdraw(100); otherAccount.deposit(cash);
```

- Which compositions **need to be atomic**?
USE Symptom

- If a program dependence exists between two atomic calls, they should be called atomically.
- Example:

```java
// withdraw everything if balance < 100
int balance, cash;
balance = account.checkBalance();
if (balance < 100) {
    cash = account.withdraw(balance);
}
```
Complementation Symptom

If two invocations dominate and post-dominate each other, they should be called atomically.

Example:

```c
/* withdraw 100 after checking the balance */
int balance, cash;
balance = account.checkBalance();
cash = account.withdraw(100);
```
Dynamic Checking

- Uses existing atomicity violation detection analyses to find **buggy interleavings**.
- Monitor a normal run (trace) and then try to find violating interleavings.
- Prunes a lot of false positives.
- For example: If a composition is only executed by one thread, it does not need to be atomic.
The Big Picture

1. **Infer atomic sets.**
   Simple in our example:
   \[
   \{ \text{Account.balance}, \text{Account.owner}, \text{Account.owner.name}, \cdots \}.
   \]

2. **Identify library and client.**
   Library: \{Account, Owner\}.
   Clients: *main* method in class *Client*.

3. **Infer atomic compositions.**
   For example:
   ```java
   balance = account.checkBalance();
   cash = account.withdraw(balance);
   ```

4. **Find buggy interleavings (dynamic checking).**
   E.g. bug if another thread can withdraw from my account in between calls.
Evaluation

- Compare with state of the art approach: MUVI (statistics based).
- Static part: Number of compositions found comparable to MUVI.
- Dynamic checking: Prunes most compositions.
- Case study with Tomcat: Inspect reported violations and (try to) determine if they are true/false positives.
Remarks: What I did not like

- Some explanations not very detailed.
- Evaluation seems to leave out “uncomfortable” data.
- Some statements in the evaluation are a bit vague and/or useless.
  - “Our evaluation on a set of large scale applications shows, the static analysis finds up to 391 atomic compositions for an application, while half would be missed by the previous statistic-based approach.”
- Poor documentation of the provided program.
- Dynamic checking not included in provided program.
Remarks: What I did like

- First half (algorithm description) well written and easy to understand.
- Impressive results (e.g. 5 of 12 compositions in Tomcat were actual bugs).
- The tool can be useful in practice.
  - Few actual reports (less than 20 in each of their cases).
  - But: Execution time may be an issue with larger project.