Automatic Testing of Sequential and Concurrent Substitutability

Paper by: Michael Pradel & Thomas R. Gross
Presented by: Erik Henriksson

12 Mars 2014

Organization

- Motivation
- The paper’s approach
- Evaluation
- Limitations

Motivation

```
class Set() {
    Set() { ... }
    ...
}

class BoundedSet extends Set {
    BoundedSet(int bound) { ... }
    ...
}
```

```
Set s = new Set();
s.add(1); // OK
```

```
Set s = new BoundedSet(0);
s.add(1); // Error
```

Safe Substitute

A class *Sub* is a safe substitute of a class *Super* if and only if we can substitute *Sub* with *Super* without changing the visible behavior of the program.

Motivation

```
class Set() {
    Set() { ... }
    ...
}

class BoundedSet extends Set {
    BoundedSet(int bound) { ... }
    ...
}
```

```
Set s = new Set();
s.add(1); // OK
```

```
Set s = new BoundedSet(0);
s.add(1); // Error
```

BoundedSet is not a safe substitute for Set!

Motivation

However, these classes compile fine under Java and an unexperienced programmer will not see this error.

We want an automatic tool for finding such mistakes.

Pradel's and Gross's Approach

- Easy to apply
- Precise
- Incomplete

Test Generator

Generate test cases:

1. Generic tests
2. Constructor mappings
3. Finding good method arguments
4. Concurrent test cases

Generic Tests

- Test both *Super* and *Sub* with same arguments

- Static type is always *Super*, but dynamic type can vary between *Sub* and *Super*.

Constructor Mappings

- Due to classes not inheriting the constructor in Java, we run into problems

```
Set s = Set() OR BoundedSet(?)
```

What should we write here?

Subclass may not have a constructor that takes same number of arguments as the superclass!

Constructor Mappings

- If constructors have the **same signature**, the tool assumes two objects are semantically equivalent after calling the constructors with the same arguments.
- Otherwise the **user** needs to specify a mapping

```
Person p1 = new Person("Foo");
       p2 = new Student("Foo");
```

Constructor Mappings

- Otherwise the **user** needs to specify a mapping

```
class Student {
    //CM super(name) -> Student(name, 0)
    Student(String name, int credits) {
        ...
    }
}
```

Method Arguments

If a method needs arguments, we choose between

1. If there exists a variable of the correct type, use it
2. Call a method that returns the correct type
3. Randomly generate a value if type is primitive

Concurrent Tests

- Only 2 threads are considered
- We use a pair of methods
- All interleavings are checked
- Error if *Sub* is not thread-safe when *Super* is

The Two Oracles

- The Output Oracle

- The Crash Oracle

Evaluation

- Crash Oracle (CO) works well, 96% of reported bugs should be fixed
- Output Oracle (OO) not that well, only 7% of reported bugs is actual bugs
- The tool found 47 bugs in 4 libraries

Limitations

- No evaluation comparison with related work
- Constructor Mappings which are automatically generated is not precise
- User is responsible for giving correct mappings where the tool fails
- The tool is both incomplete and unprecise
- The tool is not completely automatic, but this is stated in the paper

Constructor Mappings

- If constructors have the same signature, the tool assumes two objects are semantically equivalent after calling the constructors with the same arguments.

Is this sound? No!

```
class Person {
    Person(int age) {...}
    ...
}
```

```
class Student extends Person {
    Student(int credits) {...}
    ...
}
```

Constructor Mappings

- Otherwise the user needs to specify a mapping

`Set()` → `BoundedSet(?)`

What mapping should we provide the tool with?

How do we know our mapping is correct?

Questions?