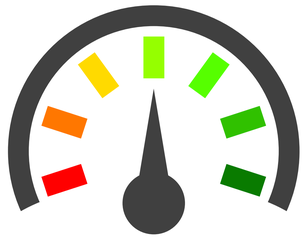
Università della Svizzera italiana

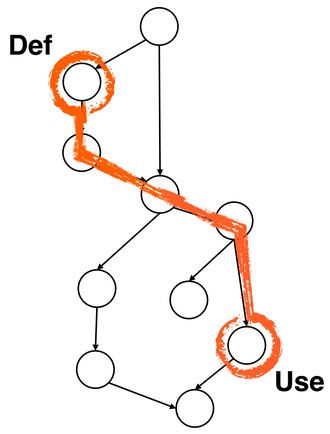
Faculty of Informatics

A Dynamic Monitoring Component of a Data Flow Testing Tool

Student: Simone D'Avico **Advisor:** Prof. Mauro Pezzè **Assistant:** Mattia Vivanti



Code coverage estimates the quality of a test suite by computing the fraction of executed code elements.



Data Flow coverage is based on the idea that, in order to reveal a fault in the code under test, the points of definition and use of faulty values must be identified.

```
public void insert(int coins)
{
     payed += coins;
}

public void getCoffee(){
   if(payed >= Coffee.cost())
   {
      makeCoffee();
   }
}
```

Def

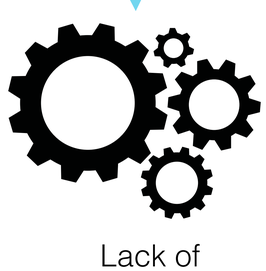
Use

In order to do so, data-flow testing tools compute the fraction of **definition-use pairs** executed (i.e., **covered**) by a test suite.

Although promising, this approach suffers from several **problems:**

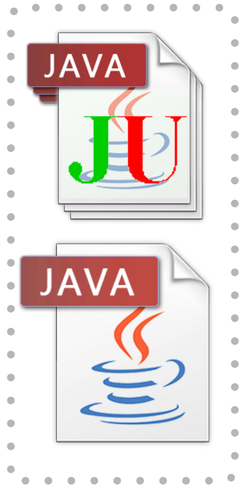
Test creation **complexity**

Unclear **effectiveness**



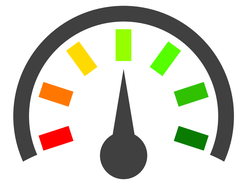
Lack of **automated tools**

I addressed this problems by implementing a **dynamic monitor** for a data flow testing tool called **DaTeC**:



Static Analysis

Dynamic Monitor



This component computes all the possible definition-use pairs by analyzing the source code of the software.

VERIFICATION

I forged a number of **small test cases** that would cover corner cases of the analysis (Singleton `instanceOf()`, abstract classes...) and manually checked the correctness of the obtained coverage;

I tested the tool with a Java graph library called **JGraphT**, providing a good number of test cases. The tool computes the coverage in a reasonable amount of time and is applicable to projects of medium/big size.

The component I implemented traces the active definitions and uses produced at runtime by the software's test suite, computing the fraction of pairs actually covered (**all DU pairs coverage**). For example:

```
public class Person {
   String name;
   int age;

   public Person(int age, String name){
      this.age = age;
      this.name = name;
   }

   public void happyBirthday(){
      age += 1;
   }

   public int getAge() {
      return age;
   }
```

```
public class TestPerson {

   public void aTest(){
      Person simone =
        new Person("Simone",22);

      simone.happyBirthday();

      assertEquals(simone.getAge(),
                   23);
   }
}
```

Defs/**uses** of instance fields are traced through instrumentation, tracking their method context and the owner object's `identityHashCode`.

Defs that are not active anymore (e.g., age is redefined in `happyBirthday()`) are **killed**.

Upon **use** (e.g., `getAge()`) the pair made up of the current active def and the current use of the field is **covered**.