An Experimental Protocol for Analyzing the Accuracy of Software Error Impact Analysis

Vincenzo Musco
vincenzo.musco@inria.fr
http://www.vmusco.com

CRIStAL GL Seminar – May 2015
Table of Contents

Introduction

Concepts

Protocol

Experiments

Thesis

Conclusion
Table of Contents

Introduction

Concepts

Protocol

Experiments

Thesis

Conclusion
Introduction
Change Impact Analysis

- Change code;

Vincenzo Musco, University of Lille
Introduction
Change Impact Analysis

- Change code;
- Called somewhere else;
Introduction
Change Impact Analysis

- Change code;
- Called somewhere else;
- Like a ripple;

Vincenzo Musco, University of Lille
Introduction

Change Impact Analysis

- Change code;
- Called somewhere else;
- Like a ripple;
- Can lead to crashes;
Introduction

Change Impact Analysis

- Change code;
- Called somewhere else;
- Like a ripple;
- Can lead to crashes;
- Need impact prediction.
Introduction

Usage example

Vincenzo Musco, University of Lille
Introduction

Usage example
Introduction

Usage example
Introduction

State of the art

- General reflection, bug basin [Challet and Lombardoni, Physical Review E 2004];
- Impact analysis using stack traces [Law and Rothermel, ICSE’03];
- Impact analysis on variable alterations/perturbations [Michael and Jones, ACCA’97];
- Similar graph for software concerns [Robillard and Murphy, ICSE’02].
How to evaluate the efficiency of an impact analysis technique (such as static call graph)?
Table of Contents

Introduction

Concepts

Protocol

Experiments

Thesis

Conclusion
Concepts
Software testing

```java
class MyMath{
    int mysum(int a, int b){
        return a+b;
    }
}
class MyTest{
    @Test
    void testSum(){
        MyMath m = new MyMath();
        res = m.mysum(5, 6);
        assertEquals(11, res);
    }
}
```

- defined using source code;
Concepts
Software testing

class MyMath{
    int mysum(int a, int b){
        return a+b;
    }
}
class MyTest{
    @Test
    void testSum(){
        MyMath m = new MyMath();
        res = m.mysum(5, 6);
        assertEquals(11, res);
    }
}

Test case

class MyTest{
    @Test
    void testSum(){
        MyMath m = new MyMath();
        res = m.mysum(5, 6);
        assertEquals(11, res);
    }
}

▶ defined using source code;
▶ defined as special functions;
▶ describes expected behavior;
Concepts
Software testing

class MyMath{
  int mysum(int a, int b){
    return a+b;
  }
}
class MyTest{
  @Test
  void testSum(){
    MyMath m = new MyMath();
    res = m.mysum(5, 6);
    assertEquals(11, res);
  }
}

- defined using source code;
- defined as special functions;
- describes expected behavior;
- can pass or fail;

Test case
Test execution

▶ MyTest.testSum()
Concepts

Mutation testing

class MyMath{
    int mysum(int a, int b){
        return a*b;
    }
}
class MyTest{
    @Test
    void testSum(){
        MyMath m = new MyMath();
        res = m.mysum(5, 6);
        assertEquals(11, res);
    }
}

Test case

class MyTest{
    @Test
    void testSum(){
        MyMath m = new MyMath();
        res = m.mysum(5, 6);
        assertEquals(11, res);
    }
}

Test execution

- copies of a software with one change;
- mutation operator determines the type of change;
- tests are run on those mutants:
  - they can be killed or remain alive.

Vincenzo Musco, University of Lille
boolean foo(int anInt) {
    return anInt == 0;
}

boolean foo() {
    return foo(0);
}

void bar(int nbExec) {
    if (foo(nbExec)) {
        ...  
    }
}

...
### Concepts

#### Call graph

```java
... boolean foo(int anInt) {
    return anInt == 0;
}

boolean foo() {
    return foo(0);
}

void bar(int nbExec) {
    if (foo(nbExec)) { ... }
}
... 
```
Concepts

Call graph

```java
boolean foo(int anInt) {
    return anInt == 0;
}

boolean foo() {
    return foo(0);
}

void bar(int nbExec) {
    if (foo(nbExec)) {
        ...
    }
}
...
Concepts

Call graph

- **Call graph**
  - static;
  - nodes = software methods;
  - directed edge = call from a method to another;
Concepts

Use graph

```java
int NB_EXEC;

... boolean foo(int anInt) {
   return anInt == 0;
}

boolean foo() {
   return foo(0);
}

void bar(int nbExec) {
   if (foo(nbExec)) {
      ... }
}

...
```
Concepts

Use graph

```java
int NB_EXEC;
...
boolean foo(int anInt){
    return anInt == NB_EXEC;
}

boolean foo(){
    return foo(0);
}

void bar(int nbExec){
    if(foo(nbExec)){ ... }
}
...
```
Concepts

Use graph

- **Call graph**
  - static;
  - nodes = software methods;
  - directed edge = call from a method to another;

- **Use graph**
  - extended call graph;
  - nodes also include variables;
  - directed edges also include variables accesses;
Table of Contents

Introduction

Concepts

Protocol

Experiments

Thesis

Conclusion
Goal: determine the accuracy of impact analysis techniques.
Goal: determine the accuracy of impact analysis techniques.

- Mutations inject faults in software;
- Running tests returns the expected impact;
- Use graph determines statically the impact;
Protocol
Introduction

- Generate mutants;
- Generate use graph;

Vincenzo Musco, University of Lille
Protocol

Introduction

▶ Generate mutants;
▶ Generate use graph;
▶ For each compiling mutant...
  ▶ run tests to determine failing ones;
  ▶ explore use graph to predict the impacted tests;
  ▶ compare actual and predicted impacted tests.
int NB_EXEC;
...
boolean foo(int anInt){
  return anInt == NB_EXEC;
}

boolean foo(){
  return foo(0);
}

void bar(int nbExec){
  if(foo(nbExec)){ ... }
}

@Test
void myTest(){
  assertTrue(foo());
}
...

Vincenzo Musco, University of Lille
int NB_EXEC;

... boolean foo(int anInt) {
  return anInt != NB_EXEC;
}

boolean foo() {
  return foo(0);
}

void bar(int nbExec) {
  if (foo(nbExec)) {
    ...
  }
}

@Test
void myTest() {
  assertTrue(foo());
}
...

Vincenzo Musco, University of Lille
propagate if it uses a faulty node; 
follow the use graph edges in reverse direction to determine impacted nodes set; 
can reach a test node.
Protocol Visualization

- Extracted from a mutation on Apache Common Lang;

Vincenzo Musco, University of Lille
Protocol

Mutations sets

Mutant member of:

- if execution and use graph failing tests are the same;
- if execution failing tests set is a subset of use graph one;
- if use graph failing tests set is a subset of execution one;
- if both execution and use graph failing tests are different;

Vincenzo Musco, University of Lille
Protocol

Mutations sets

Mutant member of:

- \( \lor \) if execution and use graph failing tests are the same;
- \( \lor + \) if execution failing tests set is a subset of use graph one;
- \( \lor - \) if use graph failing tests set is a subset of execution one;
- \( \lor o \) if both execution and use graph failing tests are different;
Protocol

Mutations sets

Mutant member of:
- / if execution and use graph failing tests are the same;
- /⁺ if execution failing tests set is a subset of use graph one;
Protocol
Mutations sets

Mutant member of:

- $I$ if execution and use graph failing tests are the same;
- $I^+$ if execution failing tests set is a subset of use graph one;
- $I^-$ if use graph failing tests set is a subset of execution one;
Mutant member of:

- $I$ if execution and use graph failing tests are the same;
- $I^+$ if execution failing tests set is a subset of use graph one;
- $I^-$ if use graph failing tests set is a subset of execution one;
- $I^o$ if both execution and use graph failing tests are different;
Table of Contents

Introduction

Concepts

Protocol

Experiments

Thesis

Conclusion
Experiments

Setup

- Two projects:
  - Apache Common Lang;
  - Apache Common Collections;

<table>
<thead>
<tr>
<th>Project</th>
<th>Version</th>
<th>LOC</th>
<th>Nodes</th>
<th>Edges</th>
<th>Build</th>
<th>Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lang</td>
<td>3.1</td>
<td>52841</td>
<td>5406</td>
<td>10418</td>
<td>8.0s</td>
<td>2015, 7ms</td>
</tr>
<tr>
<td>Collections</td>
<td>4.1</td>
<td>55081</td>
<td>6680</td>
<td>13478</td>
<td>9.7s</td>
<td>367, 49ms</td>
</tr>
</tbody>
</table>

- Two mutation operators:
  - if conditions mutations;
  - return values mutations.

Vincenzo Musco, University of Lille
Experiments

Performance

Perfect match: 30-50%;
Experiments

Performance

- Perfect match: 30-50%;
- Completeness: up to 91%;
Experiments

Performance

- Perfect match: 30-50%;
- Completeness: up to 91%;

Use graph is a good candidate for error impact analysis.
Experiments

Performance

- differ strongly from a project to another;
Vincenzo Musco, University of Lille

Experiments
Performance

- differ strongly from a project to another;
- differ less from a mutation to another.
Experiments

Performance

- differ strongly from a project to another;
- differ less from a mutation to another.

Use graph is more project dependent than error-family dependent.

Vincenzo Musco, University of Lille
Experiments
Performance

Important remark
Results are encouraging, however more investigation (ie. larger dataset and operators) is necessary to strengthen those observations.
Experiments

Case study

Studied three study cases of incorrect prediction. The causes are:

▶ Java reflection;
▶ Java varargs;
▶ Test cases boundary.
Experiments

Case study: Java reflection

- Method invocation via Java Reflection mechanism;
- Resolved on run-time;
- Unable to detect statically;

```java
class Foo{
    void f(){
        ...
    }
}

class Bar{
    void bar(){
        Class c = Class.forName("Foo");
        Method m = c.getDeclaredMethod("f");
        Object t = c.newInstance();
        m.invoke(t);
    }
}
```
Experiments
Case study: Java reflection

- Method invocation via Java Reflection mechanism;
- Resolved on run-time;
- Unable to detect statically;

However:
- can be detected (warning).

```java
class Foo{
    void f(){
        ...
    }
}

class Bar{
    void bar(){
        Class c = Class.forName("Foo");
        Method m = c.getDeclaredMethod("f");
        Object t = c.newInstance();
        m.invoke(t);
    }
}
```
Experiments

Case study: Java varargs

- special Java notation;
- arbitrary number of parameters;
- method signature mismatch;

```java
class Foo{
    void foo(Object...){
        ...
    }
    void bar(){
        Object o1, o2;
        Object[] o3;
        ...
        this.foo(o1, o2);
        this.foo(o3);
    }
}
```

Vincenzo Musco, University of Lille
Experiments
Case study: Java varargs

- special Java notation;
- arbitrary number of parameters;
- method signature mismatch;

Solution:
- specific processing in graph building phase.

```java
class Foo{
    void foo(Object...){
        ...
    }
    void bar(){
        Object o1, o2;
        Object[] o3;
        ...
        this.foo(o1, o2);
        this.foo(o3);
    }
}
```
Experiments
Case study: Test cases boundaries

- mutation occurs in test code;
- hard to determine test boundary.

class abstract AbstractTest{
    void foo(){
        // Mutation occurs here
    }
}
class T1 extends AbstractTest{
    @Test
    void foo(){
        super.foo();
    }
}
Table of Contents

Introduction

Concepts

Protocol

Experiments

Thesis

Conclusion
How can we use graph science to learn more about software?
How can we use graph science to learn more about software?

- Are there common structures of software dependency graphs? Yes
- Can we generate class dependency graphs which fit real software ones? Yes
- See A Generative Model of Software Dependency Graphs to Better Understand Software Evolution (September 2014 – under review).
How can we use graph science to learn more about software?

- Is a basic use graph a good candidate for error impact analysis? **Yes**
- Can we use machine learning to improve the accuracy of impact prediction? **To be done...**
How can we use graph science to learn more about software?

- Are there specific motifs (patterns) in software dependency graph? **To be done**...
- How are connected new nodes in software dependency graph over time? **To be done**...
Conclusion

- Mutation testing and test suite execution offer an experimental protocol for analyzing the accuracy of software error impact analysis;
- Use graph is a good candidate as a simple tool to impact prediction in software (perfect accuracy for 30% to 49%);
- More work has to be done to strengthen those discoveries.