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

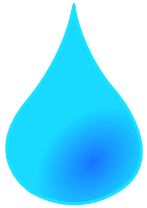
Université de Lille
CRIStAL
Centre de Recherche en Informatique, Signal et Automatique de Lille
cnrs
dépasser les frontières

Table of Contents

Table of Contents

Introduction

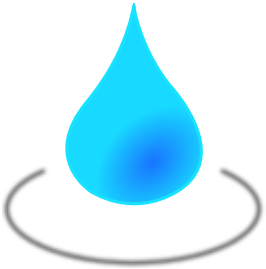
Change Impact Analysis



- Change code;

Introduction

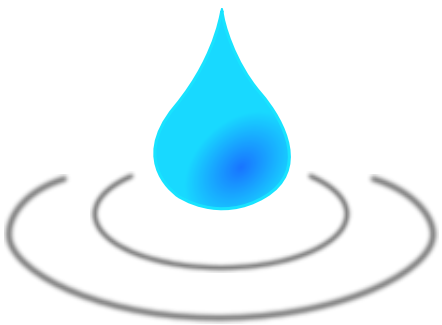
Change Impact Analysis



- Change code;
- Called somewhere else;

Introduction

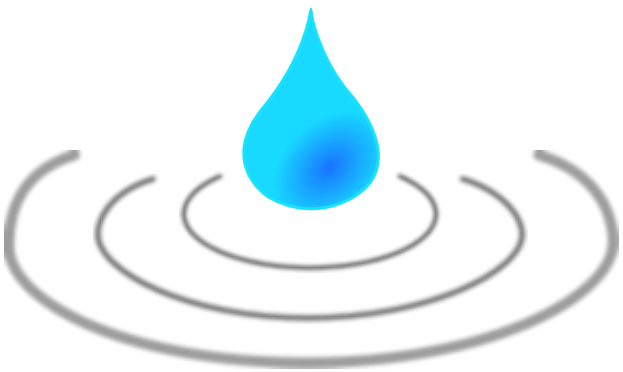
Change Impact Analysis



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

Introduction

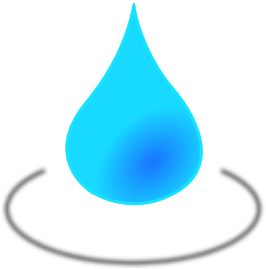
Change Impact Analysis



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

Introduction

Change Impact Analysis

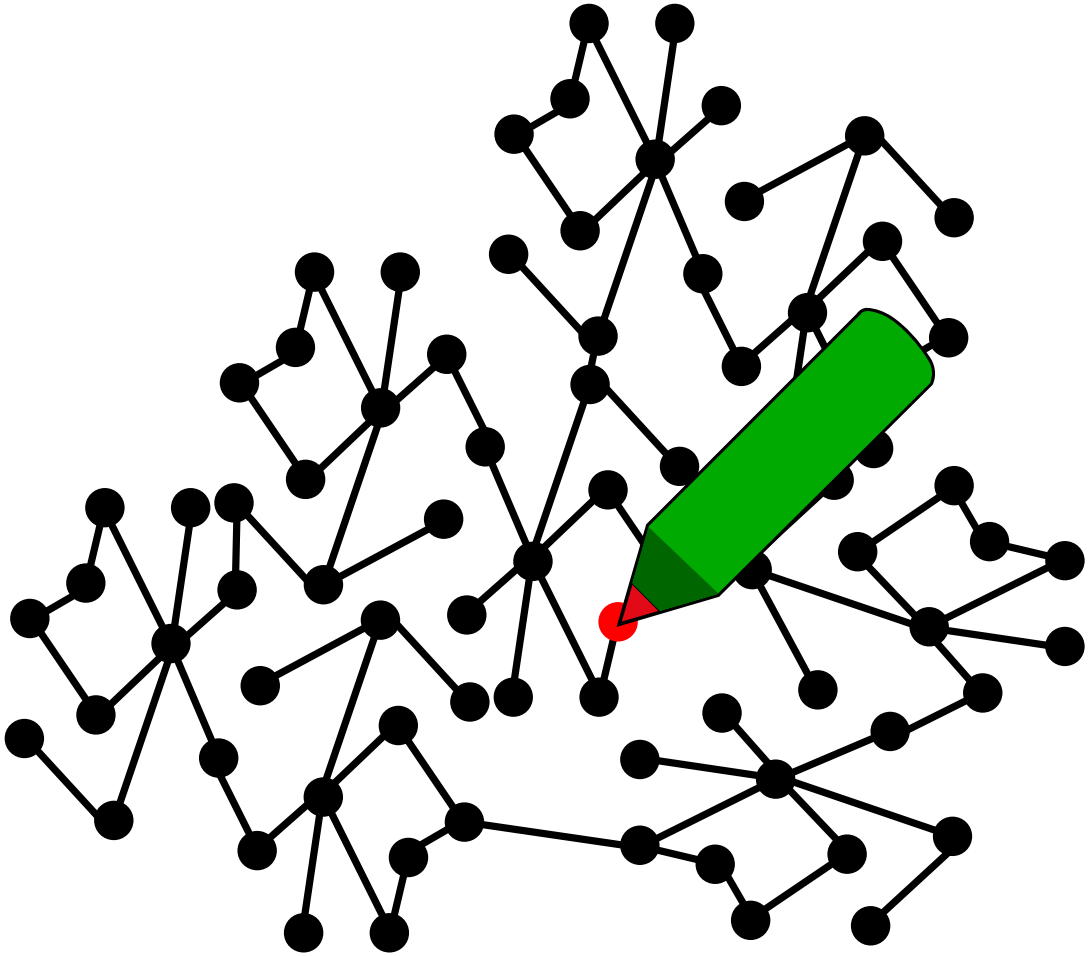


- Change code;
- Called somewhere else;
- Like a ripple;
- Can lead to crashes;

- Need impact prediction.

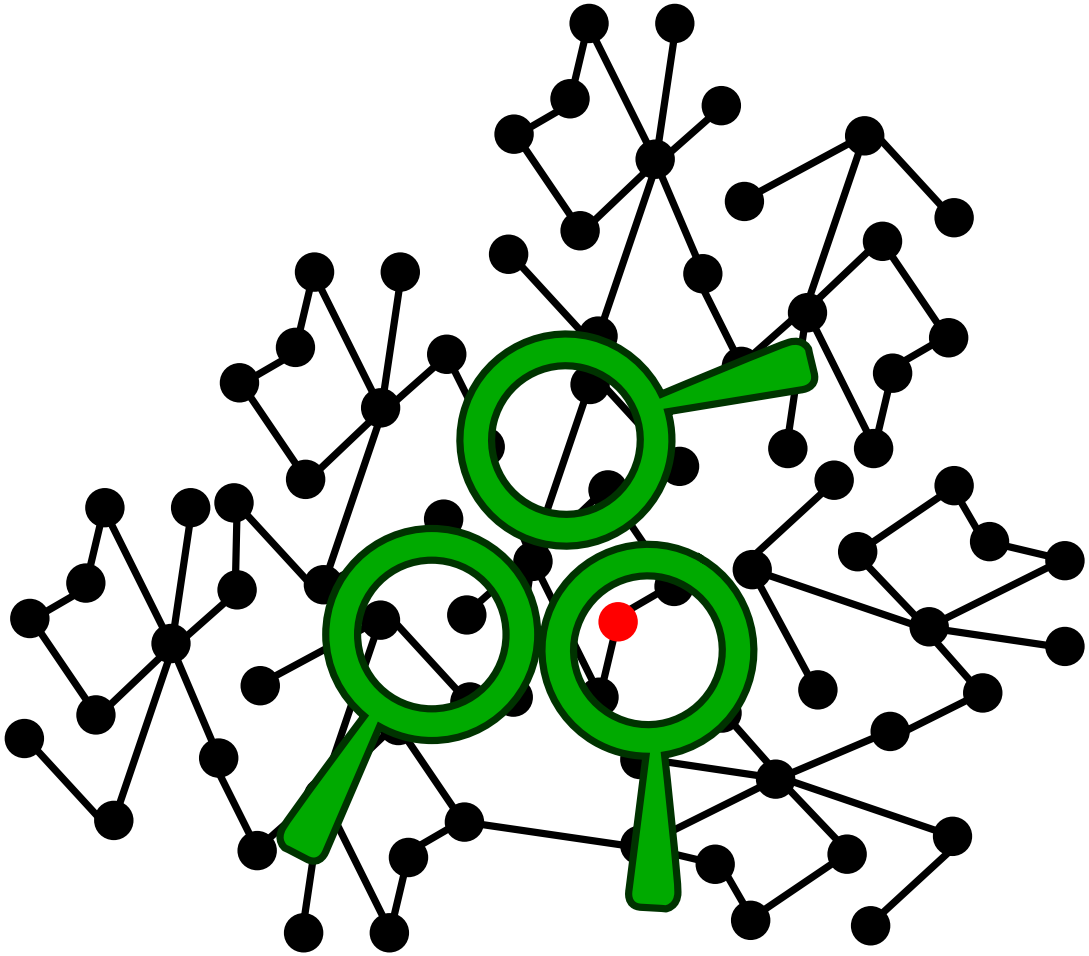
Introduction

Usage example



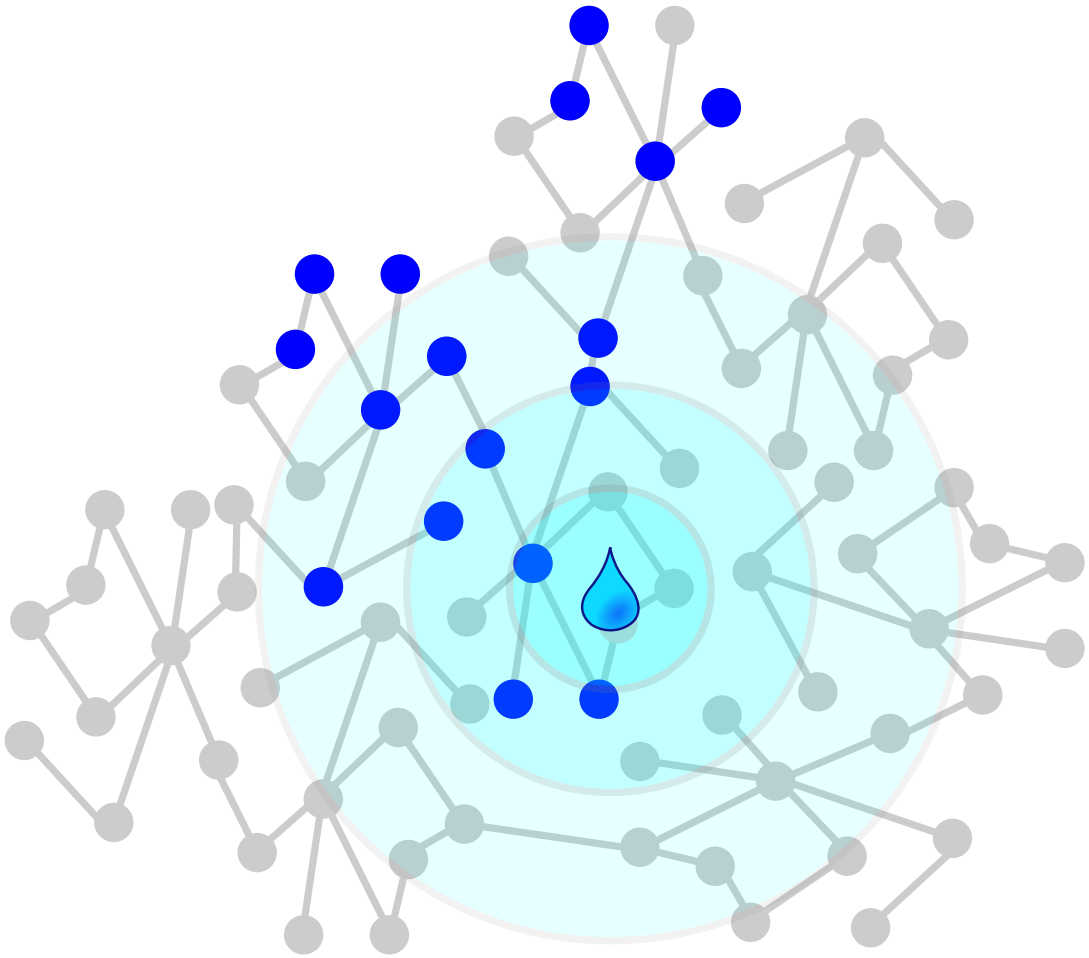
Introduction

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].

Introduction

Question

How to evaluate the efficiency of an impact analysis technique (such as static call graph) ?

Table of Contents

Concepts

Software testing

```
class MyMath{
  int mysum(int a, int b){
    return a+b;
  }
}
```

- defined using source code;

Concepts

Software testing

```
class MyMath{
  int mysum(int a, int b){
    return a+b;
  }
}
```

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;
  }
}
```

Test case

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

Test execution

● MyTest.testSum()

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

Concepts

Mutation testing

```
class MyMath{
  int mysum(int a, int b){
    return a*b;
  }
}
```

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**.

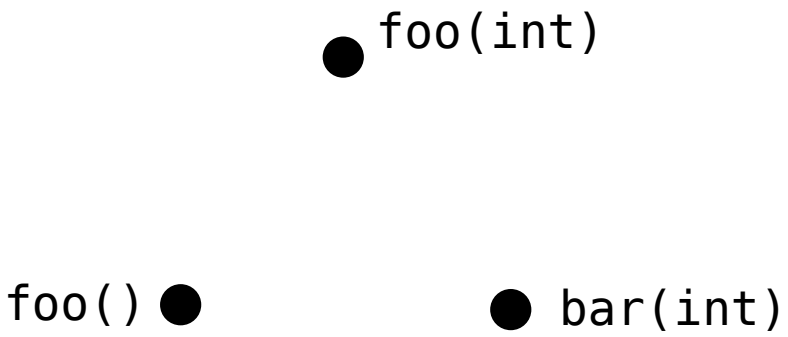
Concepts

Call graph

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

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

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

...
```

Concepts

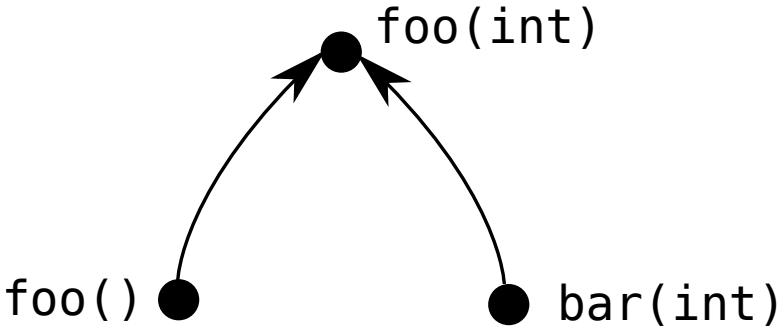
Call graph




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

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

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

...
```

Concepts

Call graph




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

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

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

...
```

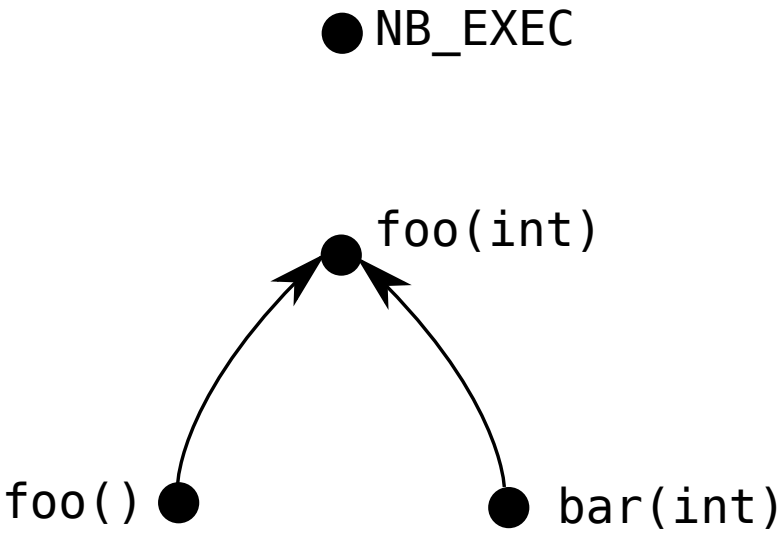
Concepts

Call graph

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

Concepts

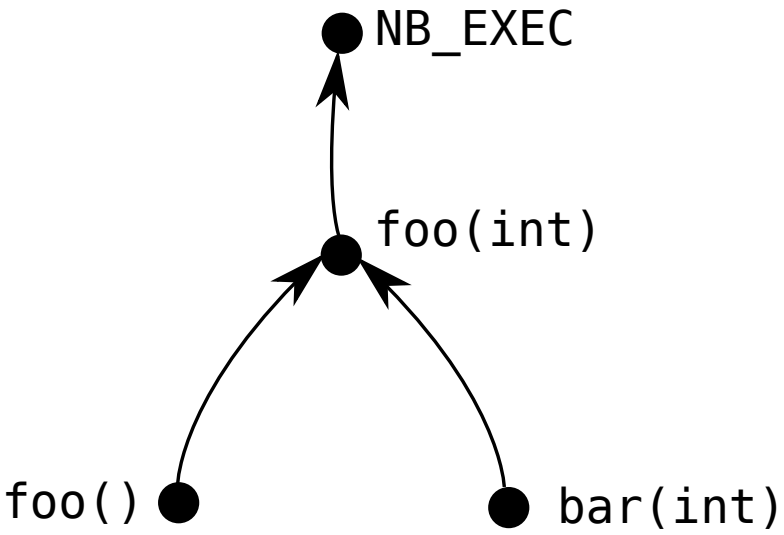
Use graph




```
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




```
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

Protocol

Introduction

Goal: determine the accuracy of impact analysis techniques.

Protocol

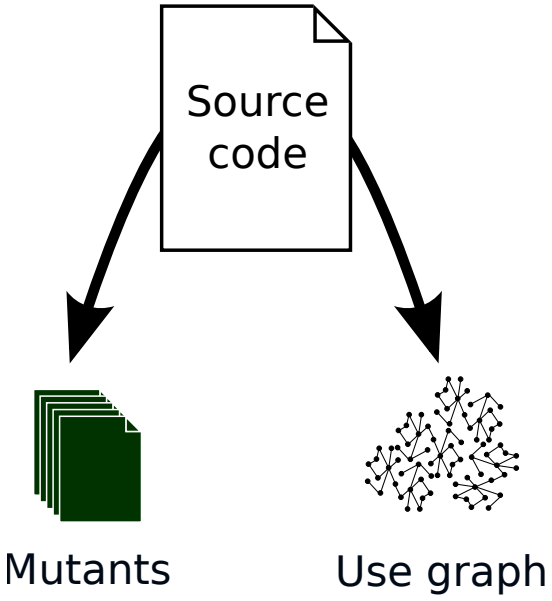
Introduction

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

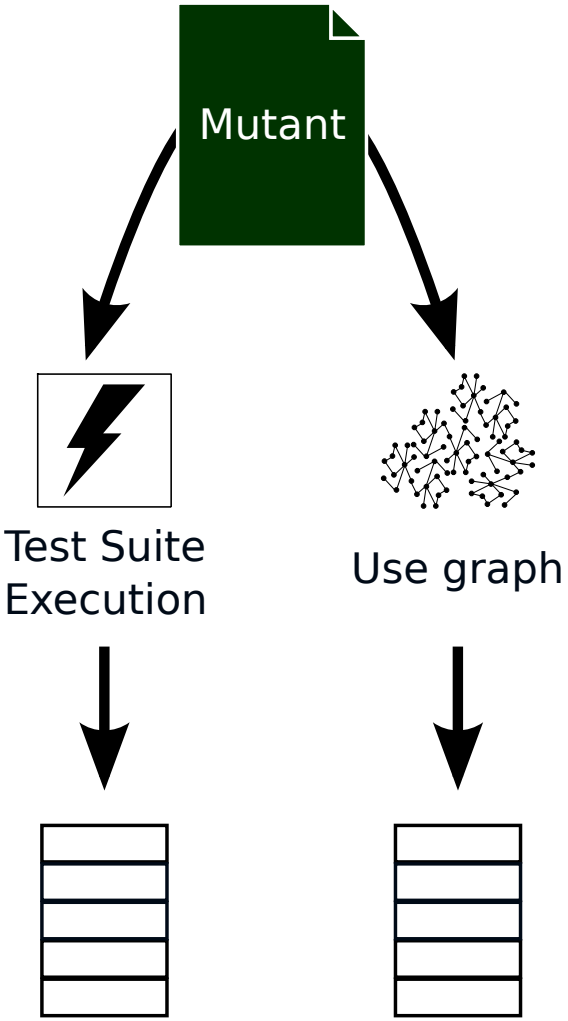
Introduction



- Generate mutants;
- Generate use graph;

Protocol

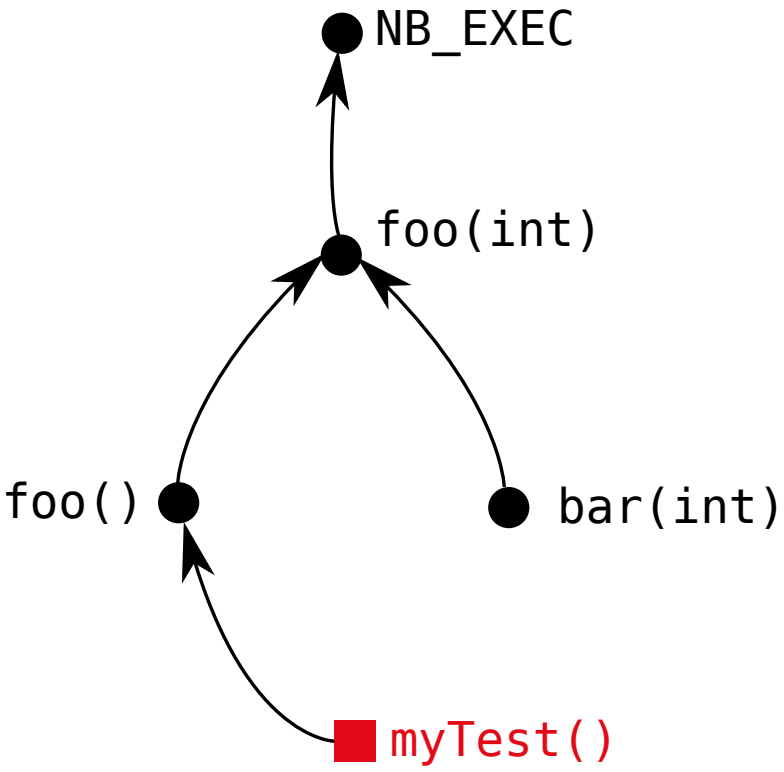
Introduction



- 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.

Protocol

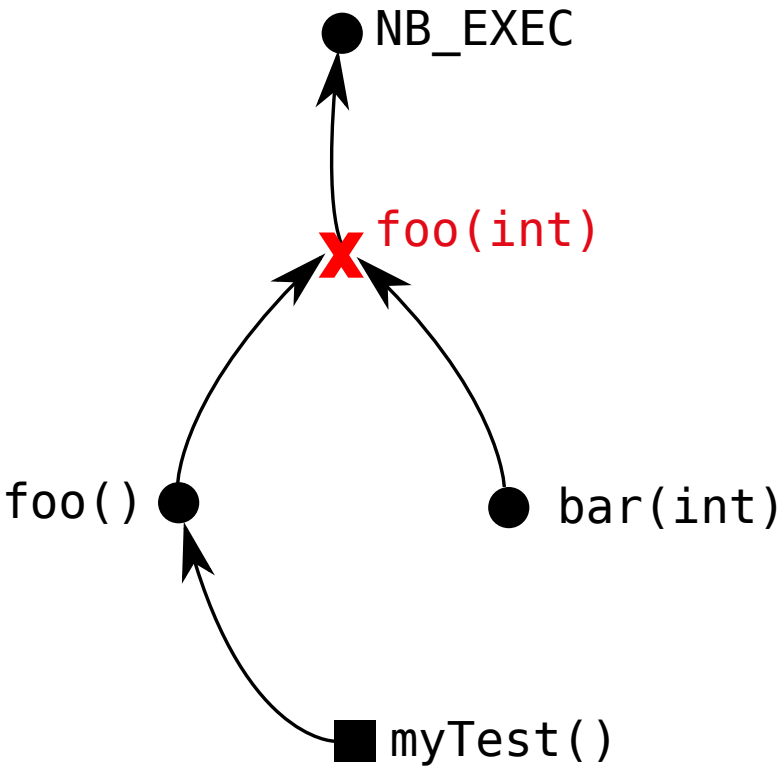
Propagation




```
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());
}
...
```

Protocol

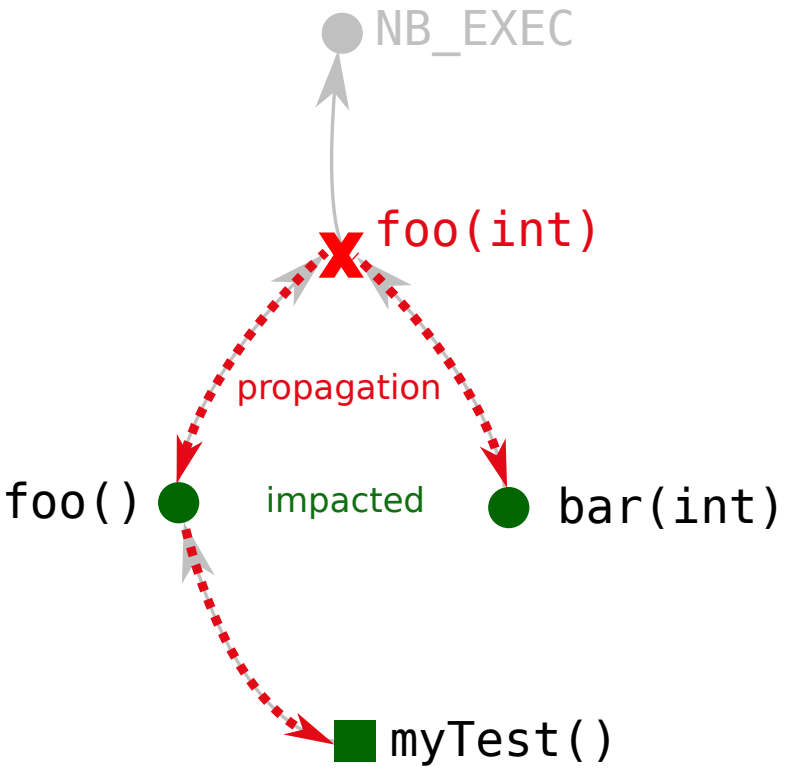
Propagation




```
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());
}
...
```

Protocol

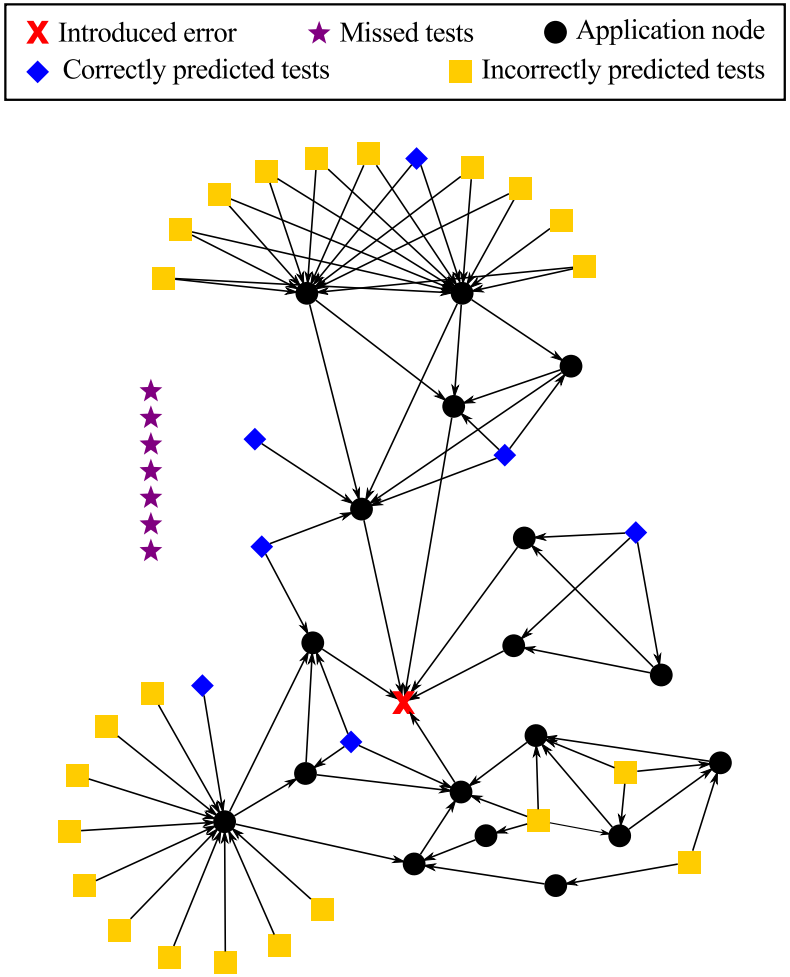
Propagation



- 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

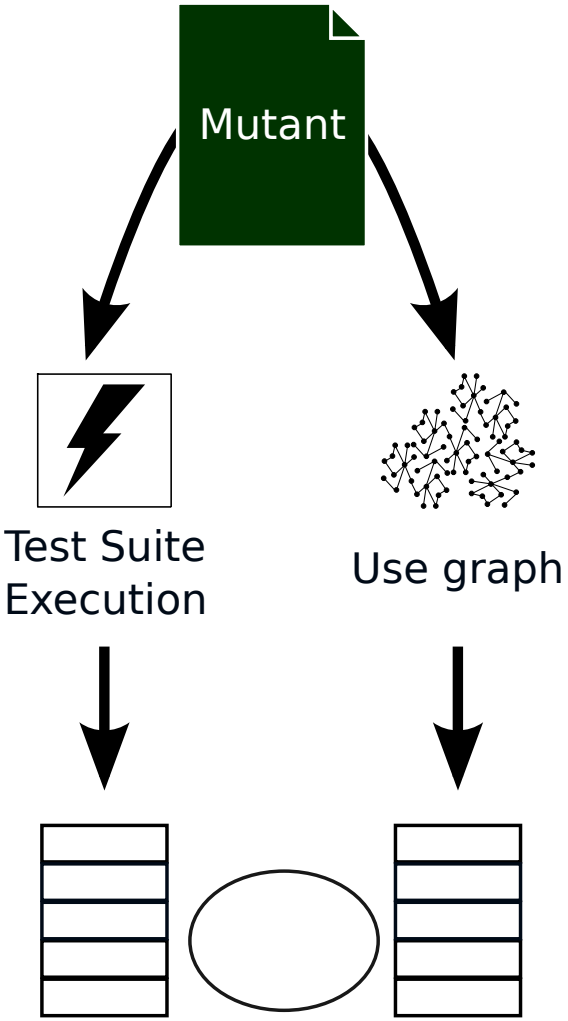
Visualization



- Extracted from a mutation on Apache Common Lang;

Protocol

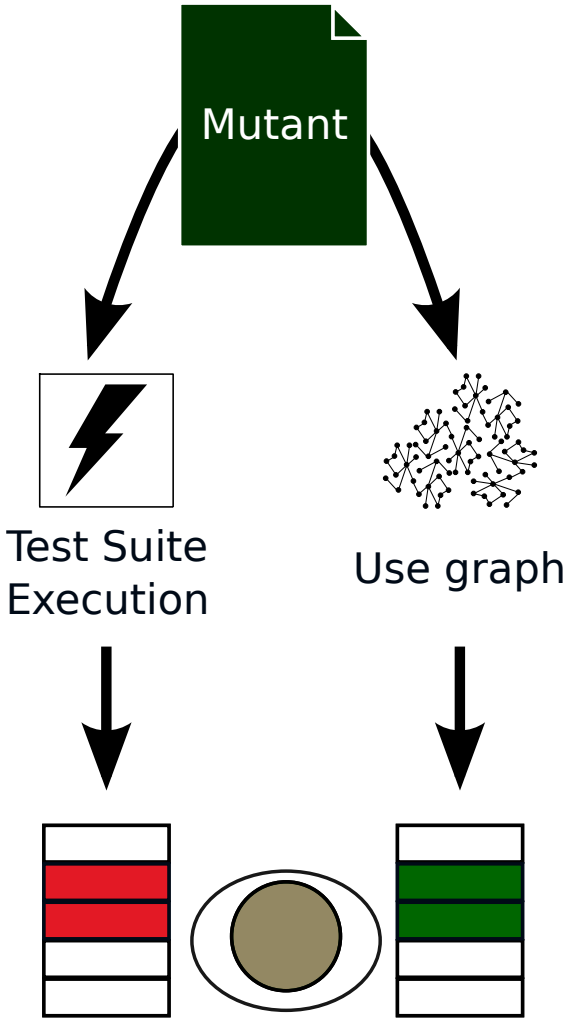
Mutations sets



Mutant member of:

Protocol

Mutations sets

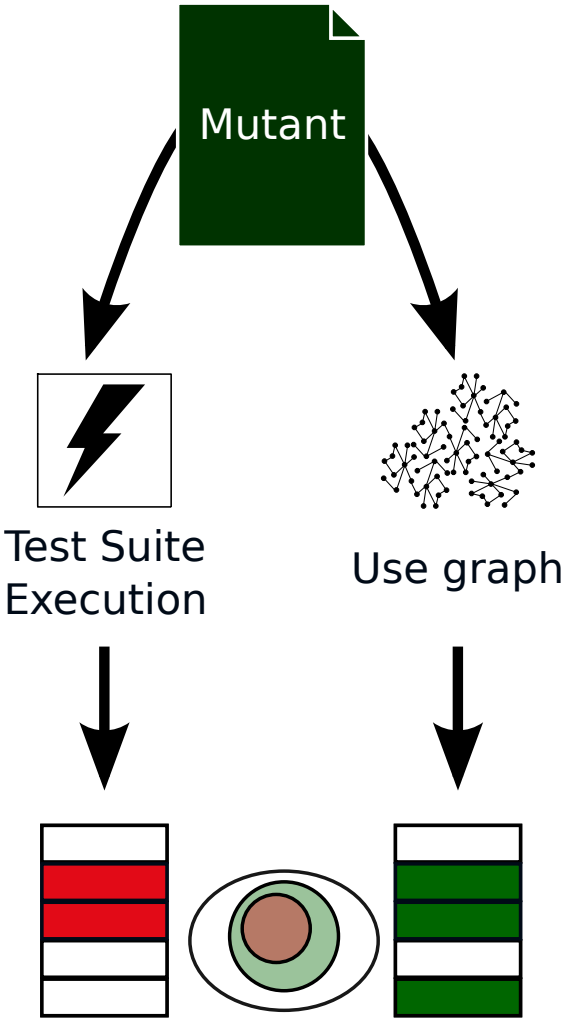


Mutant member of:

- I if execution and use graph failing tests are the same;

Protocol

Mutations sets

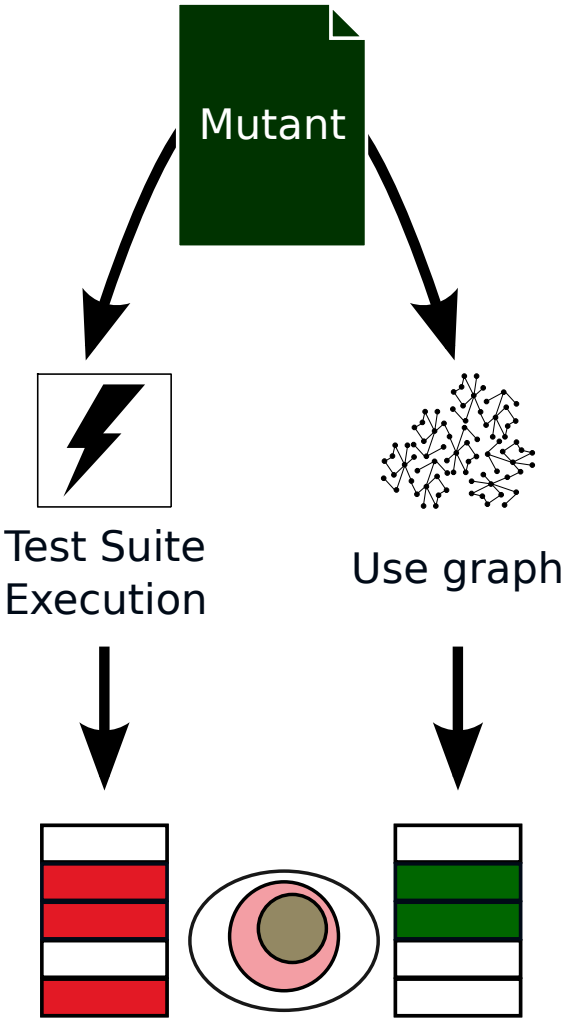


Mutant member of:

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

Protocol

Mutations sets

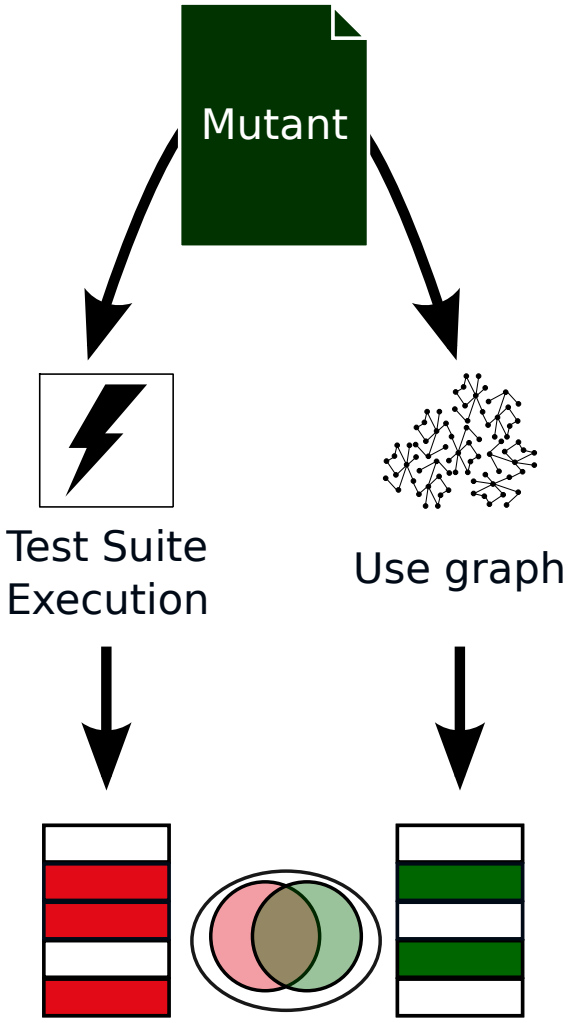


Mutant member of:

- 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;

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;
- I^o if both execution and use graph failing tests are different;

Table of Contents

Experiments

Setup

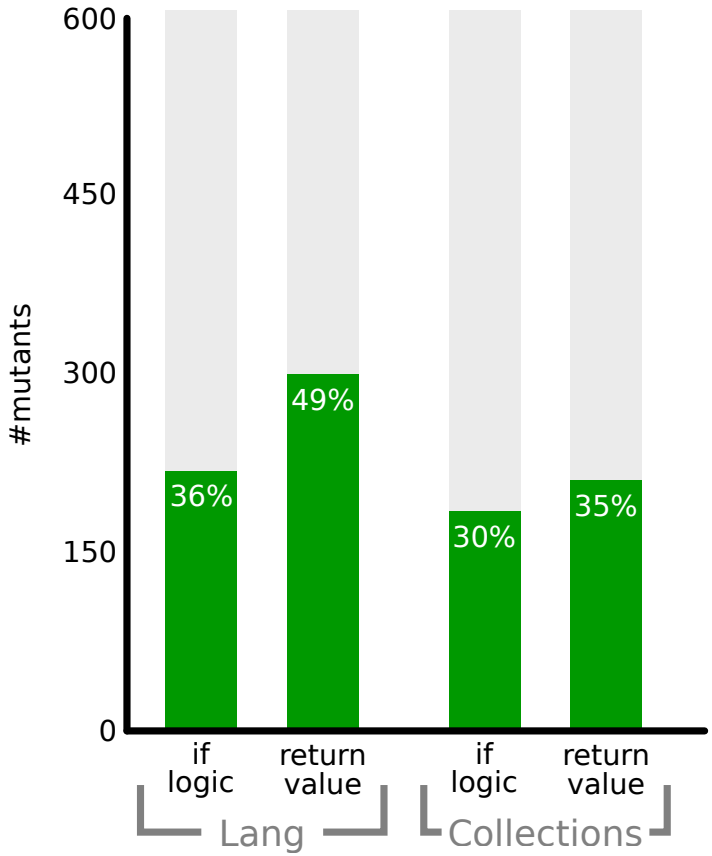
- Two projects:
  - Apache Common Lang;
  - Apache Common Collections;

						Tests	
Project	Version	LOC	Nodes	Edges	Build	Number	Execution
Lang	3.1	52841	5406	10418	8.0s	2015	7ms
Collections	4.1	55081	6680	13478	9.7s	367	49ms

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

Experiments

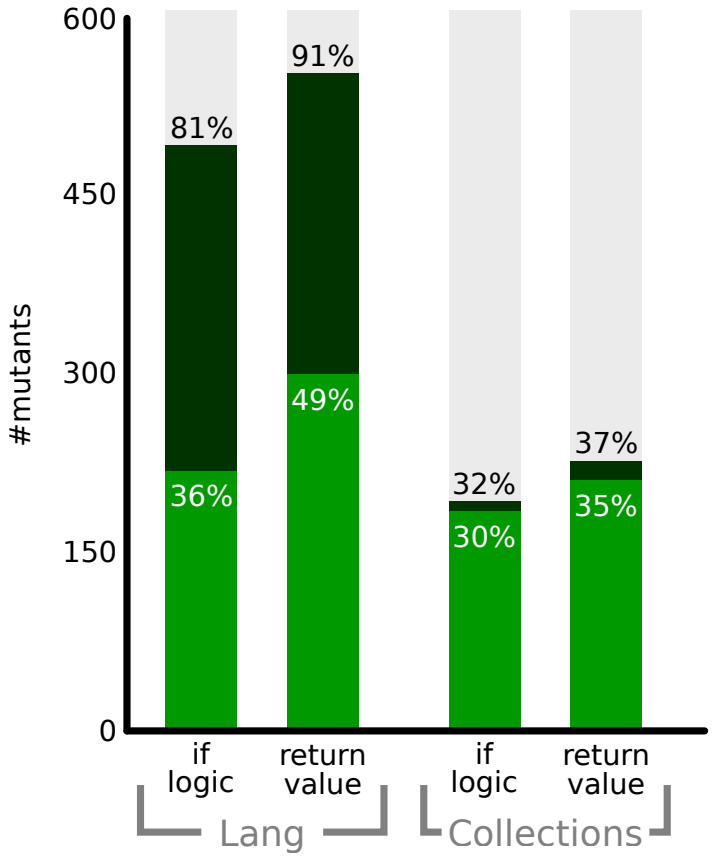
Performance



- Perfect match: 30-50%;

Experiments

Performance

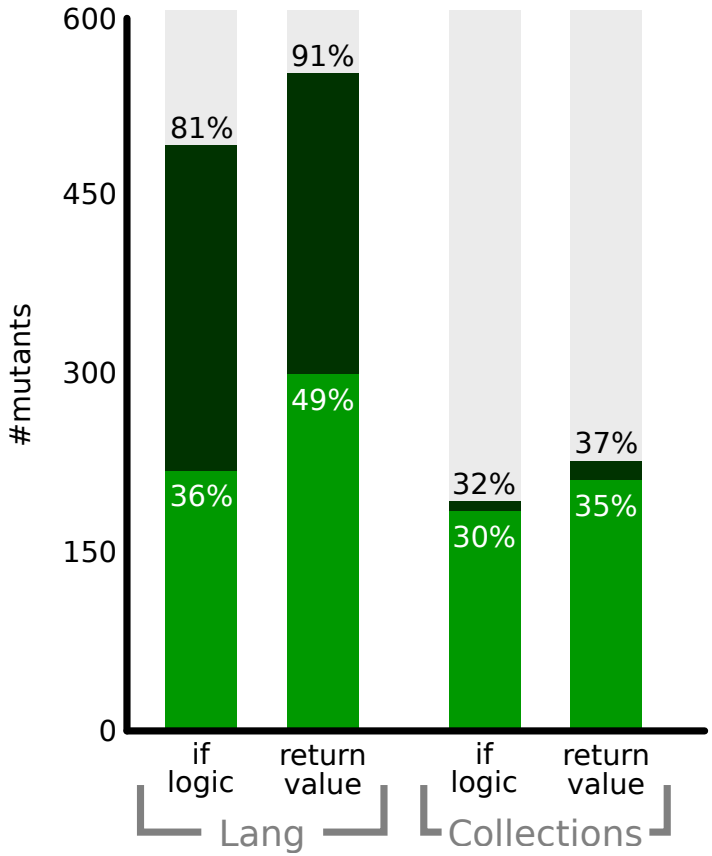


I+I⁺

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

Experiments

Performance

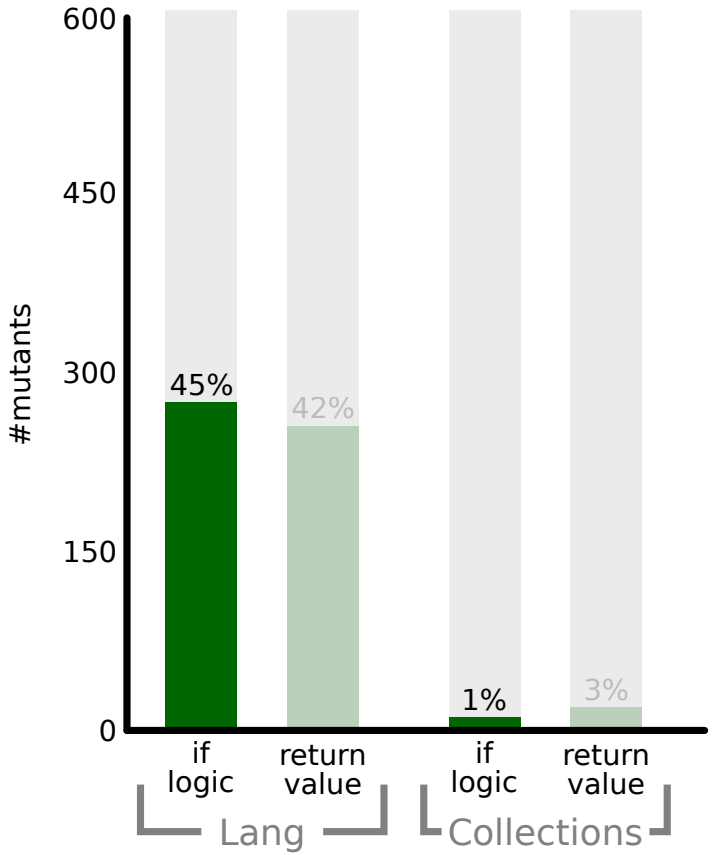


- Perfect match: 30-50%;
- Completeness: up to 91%;

Use graph is a good candidate for error impact analysis.

Experiments

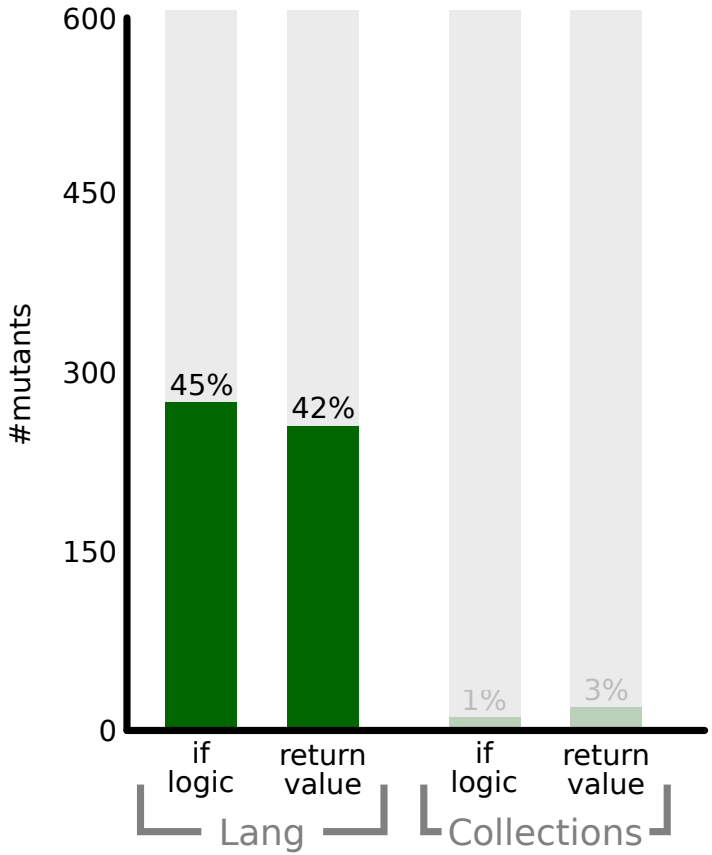
Performance



- differ strongly from a project to another;

Experiments

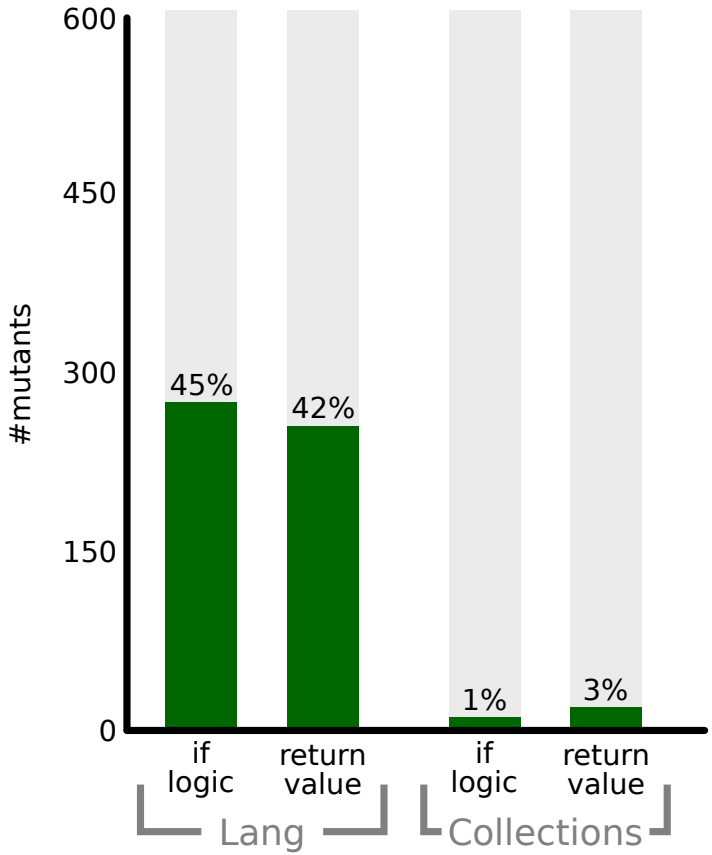
Performance



- 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.

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;

```
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).

```
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;

```
class Foo{
  void foo(Object...){
    ...
  }

  void bar(){
    Object o1, o2;
    Object[] o3;

    ...

    this.foo(o1, o2);
    this.foo(o3);
  }
}
```

Experiments

Case study: Java varargs

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

Solution:

- specific processing in graph building phase.

```
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

Thesis

Research Directions

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

Thesis

Research Directions

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)*.

Thesis

Research Directions

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...**

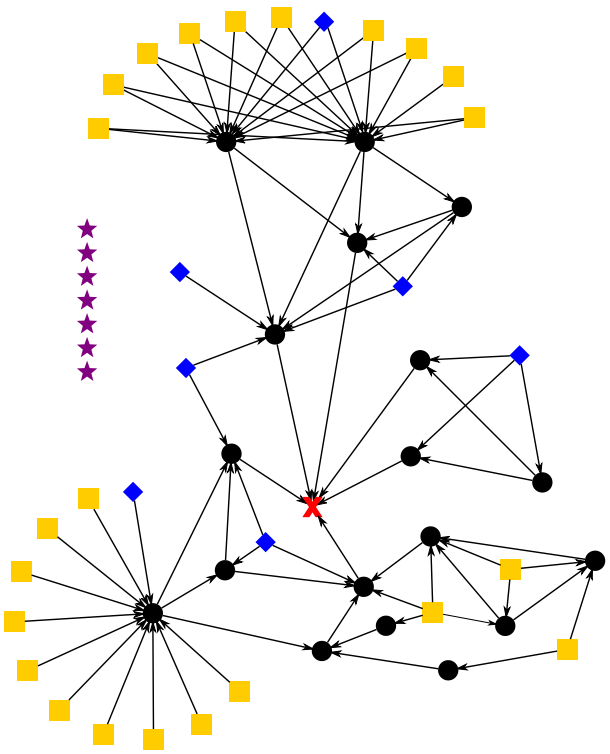
Thesis

Research Directions

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.