

# Adapting Amplified Unit Tests for Human Comprehension

Internship – M2 SIF

Supervisor: Benoit Baudry

KTH, Sweden

Wednesday 7<sup>th</sup> February, 2018–Friday 22<sup>nd</sup> June, 2018

---

Simon Bihel

`simon.bihel@ens-rennes.fr`

Wednesday 27<sup>th</sup> June, 2018

University of Rennes I

École Normale Supérieure de Rennes

# Introduction

---

## Context

- Software projects are now accompanied by strong test suites
- Takes time to write
- Still missing some bugs due to focus on nominal paths when writing test cases

## Context

- Software projects are now accompanied by strong test suites
- Takes time to write
- Still missing some bugs due to focus on nominal paths when writing test cases

## Related works

- Measure the quality of test suites
- Automatically write test suites
- **Amplify** existing test suites

System-Under-Test: function, class, whole program...

**Inputs** E.g. function parameters, method calls to setup and stimulate an object

**Assertions** Used to test whether the function's output is correct, that the object is in the right state

## Test Example

```
1 public class TreeListTest {
2     @Test
3     public void testIterationOrder() {
4         TreeList tl = new TreeList(10);
5         for (int i = 0; i < 10; i++) {
6             tl.add(i);
7         }
8         ListIterator it = tl.listIterator();
9
10        int i = 0;
11        while (it.hasNext()) {
12            Integer val = it.next();
13            assertEquals(i++, val.intValue());
14        }
15    }
16 }
```

# Metrics for Test Suites

## Goal

Detect parts that are not tested.

# Metrics for Test Suites

## Goal

Detect parts that are not tested.

## Code Coverage

Number of instructions or branches executed by the test suite.



# Metrics for Test Suites

## Goal

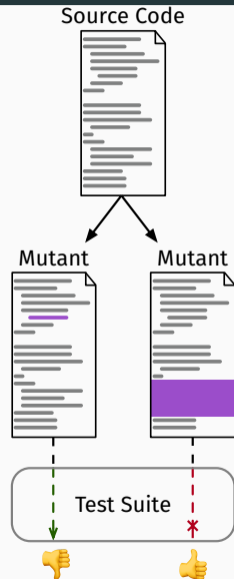
Detect parts that are not tested.

## Code Coverage

Number of instructions or branches executed by the test suite.

## Mutation Testing

1. Create *mutants* (i.e. bugged versions) of the main software (e.g. change a  $>$  with a  $\leq$ ).
2. Count how many mutants for which the test suite fail.



# Automated Test Generation

## Goal

Generate tests from scratch to fulfill a given metric.

Large search space of instructions and values.

## Search-based techniques<sup>1</sup>

Random, iterative and heuristic-based techniques (e.g. Genetic Algorithms, simulated annealing).

---

<sup>1</sup>McMinn, "Search-based software testing: Past, present and future", 2011.

<sup>2</sup>Barr et al., "The oracle problem in software testing: A survey", 2015.

# Automated Test Generation

## Goal

Generate tests from scratch to fulfill a given metric.

Large search space of instructions and values.

## Search-based techniques<sup>1</sup>

Random, iterative and heuristic-based techniques (e.g. Genetic Algorithms, simulated annealing).

## The oracle problem<sup>2</sup>

What should the output of a test be?

→ Avoid this by focusing on regression testing.

<sup>1</sup>McMinn, "Search-based software testing: Past, present and future", 2011.

<sup>2</sup>Barr et al., "The oracle problem in software testing: A survey", 2015.

# Test Suite Amplification

---

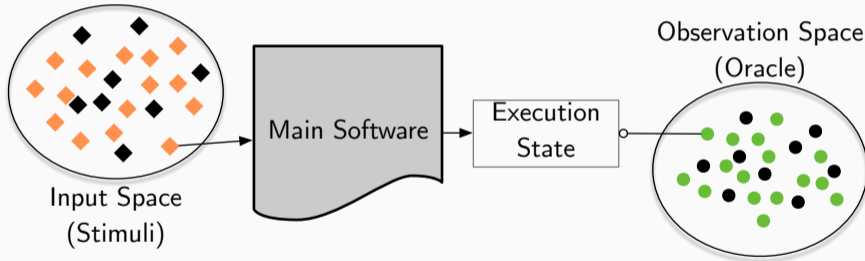
- Reduce search-space by using the existing test suite as a (good) starting population.
- Use knowledge in hand-written tests for a better oracle.

---

<sup>3</sup>Danglot et al., “The Emerging Field of Test Amplification: A Survey”, 2017.

## Goal

Create tests for undetected mutants.



<sup>4</sup>Baudry et al., “DSpot: Test Amplification for Automatic Assessment of Computational Diversity”, 2015.

### Input amplification

**Literals** → replaced with neighbor values.

**Method calls** → duplicated, removed or made-up (with random or default parameters).

### Input amplification

**Literals** → replaced with neighbor values.

**Method calls** → duplicated, removed or made-up (with random or default parameters).

### Assertion amplification

Capture the state of the system after the test's execution.



New tests ought to be approved by the developers.

## Generating Descriptive Messages

---

### Software Artefact Summarization<sup>5</sup>

Includes: documentation for source code, code changes, **test cases**.

---

<sup>5</sup>Nazar, Hu, and Jiang, “Summarizing software artifacts: A literature review”, 2016.

<sup>6</sup>Li et al., “Automatically documenting unit test cases”, 2016.

### Software Artefact Summarization<sup>5</sup>

Includes: documentation for source code, code changes, **test cases**.

#### Example

- *UnitTestScribe*<sup>6</sup> Summarises actions in natural language.

---

<sup>5</sup>Nazar, Hu, and Jiang, “Summarizing software artifacts: A literature review”, 2016.

<sup>6</sup>Li et al., “Automatically documenting unit test cases”, 2016.

Pull Request message.

Explain:

1. the modifications made,
2. the reason why the new test was kept (i.e. mutation score).

**Category** ASSERT, ADD, DEL, MODIFY

**Parent** Type of the parent of modified AST node.

**Role** Role for the parent node (e.g. argument for a method call).

**Old value** Textual representation of the old node.

**New value** Textual representation of the new node.

# Logging Amplifications

**Category** ASSERT, ADD, DEL, MODIFY

**Parent** Type of the parent of modified AST node.

**Role** Role for the parent node (e.g. argument for a method call).

**Old value** Textual representation of the old node.

**New value** Textual representation of the new node.

250 ELoC.

Logging call in each amplifier.

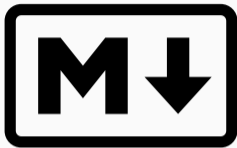
**Location** Modified method and line.

**Description** High-level, sometimes vague, natural language description from PIT<sup>7</sup>.

---

<sup>7</sup>Coles et al., "PIT: a practical mutation testing tool for java", 2016.





Markdown markup language allows for enhanced messages. Code snippet, links to code lines, better rendering, ...

Major platforms use it. **GitHub**  **GitLab**

But not all.  **Bitbucket**

655 Python ELoC.

Demo XWiki

## Performances

Logging:

- no impact on time,
- 6% higher memory usage.

Few seconds for the message generation.

## Performances

Logging:

- no impact on time,
- 6% higher memory usage.

Few seconds for the message generation.

No case study with users. Only discussions with my supervisor.

## Conclusion

---

Too early to tackle the topic of generated tests explanation.

- Lack of precise information (e.g. killed mutants *per* test).
- Small user-base.

## Too early to tackle the topic of generated tests explanation.

- Lack of precise information (e.g. killed mutants *per* test).
- Small user-base.

## What I should have focused solely on.

- Cleaning amplified tests.
- Independent mutation score explanation.





# EVOSUITE<sup>8</sup>

- Test Suite Generation *from scratch*.
- State-of-the-Art & industry grade.

## Differences

- Treats test suites as a whole.
- GA with tests cases as genes.

## Automatic Documentation

### Automatic name generation<sup>9</sup>.

<sup>8</sup>Fraser and Arcuri, “Evosuite: automatic test suite generation for object-oriented software”, 2011.

<sup>9</sup>Daka, Rojas, and Fraser, “Generating unit tests with descriptive names or: Would you name your children thing1 and thing2?”, 2017.

## Is Documentation Essential? — Yes.

Developers surveys<sup>10</sup> and experiments<sup>11</sup>.

Documentation (e.g. JavaDoc, naming) has many advantages<sup>12</sup>, especially for generated tests<sup>13</sup>:

- faster to get familiar with the test;
- faster fault localisation; and
- helps to build trust in a test generator if it can provide a proof for its result.

<sup>10</sup>Daka and Fraser, “A survey on unit testing practices and problems”, 2014; Prado et al., “WAP: Cognitive aspects in unit testing: The hunting game and the hunter’s perspective”, 2015; Prado and Vincenzi, “Advances in the Characterization of Cognitive Support for Unit Testing: The Bug-Hunting Game and the Visualization Arsenal”, 2016; Prado and Vincenzi, “Towards cognitive support for unit testing: a qualitative study with practitioners”, 2018; Li et al., “Automatically documenting unit test cases”, 2016.

<sup>11</sup>Panichella et al., “The impact of test case summaries on bug fixing performance: An empirical investigation”, 2016.

<sup>12</sup>Daka, Rojas, and Fraser, “Generating unit tests with descriptive names or: Would you name your children thing1 and thing2?”, 2017.

<sup>13</sup>Rojas and Fraser, “Is search-based unit test generation research stuck in a local optimum?”, 2017; Shamshiri et al. “How Do Automatically Generated Unit Tests Influence Software Maintenance?” 2018