AutoMutation: Towards the Full Automation of Mutation Testing

Prof. Auri Marcelo Rizzo Vincenzi
Full Professor at UFSCar/Brazil
auri@ufscar.br

Agenda

- Software Testing Basic Definitions
- Mutation Testing
- Research Project on Mutation Testing
- Some Recent Publications
- Other Research Topics Under Development
- Final Remarks

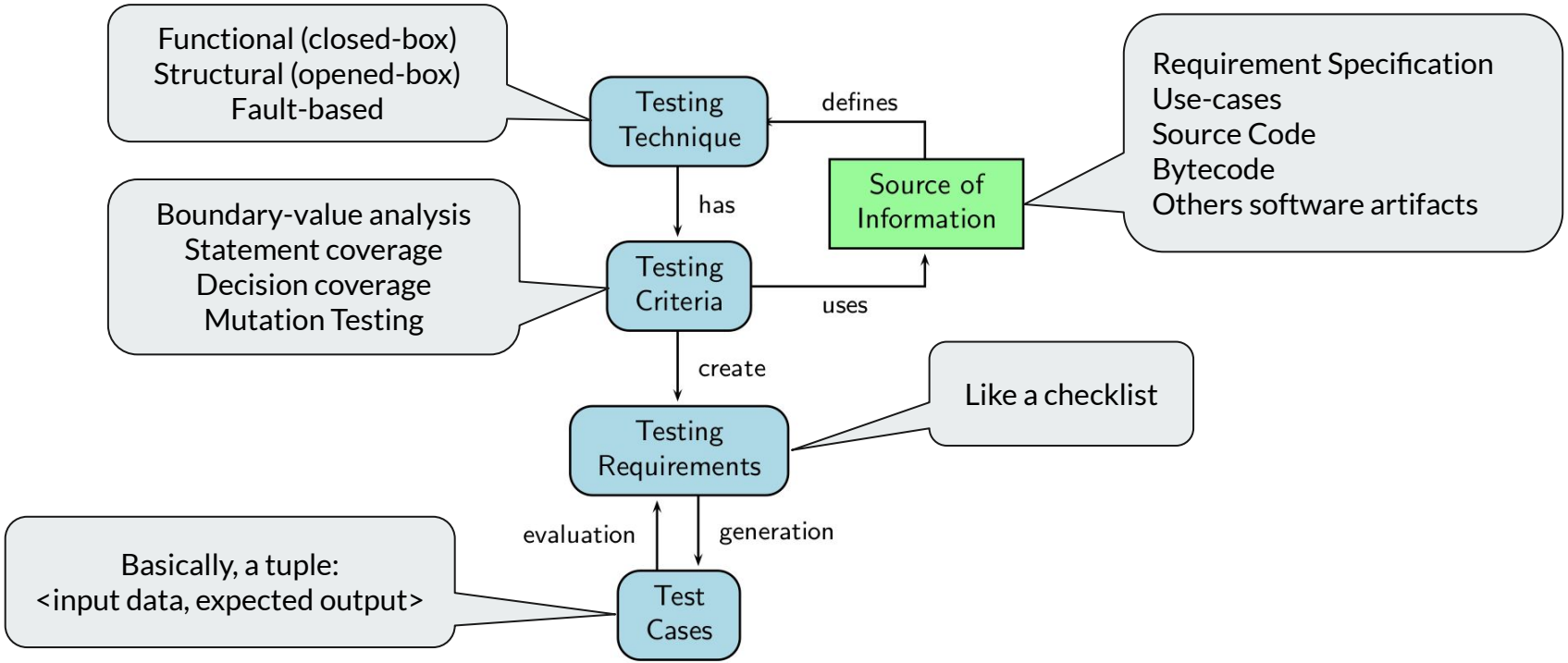
Basic Definitions

Software Testing Definitions

> “Testing shows the presence, not the absence of bugs” (Dijkstra, 1969)
>
> “Testing is the process of executing a program or system with the intent of finding errors” (Myers, 1979)
>
> “The **dynamic verification** of the behavior of a program on **a finite set of test cases**, suitably selected **from the usually infinite executions domain**, against the expected behavior.” (ISO/IEC/IEEE, 2010)

“Testing is sampling” (Ropper, 1994)

Testing Terminology



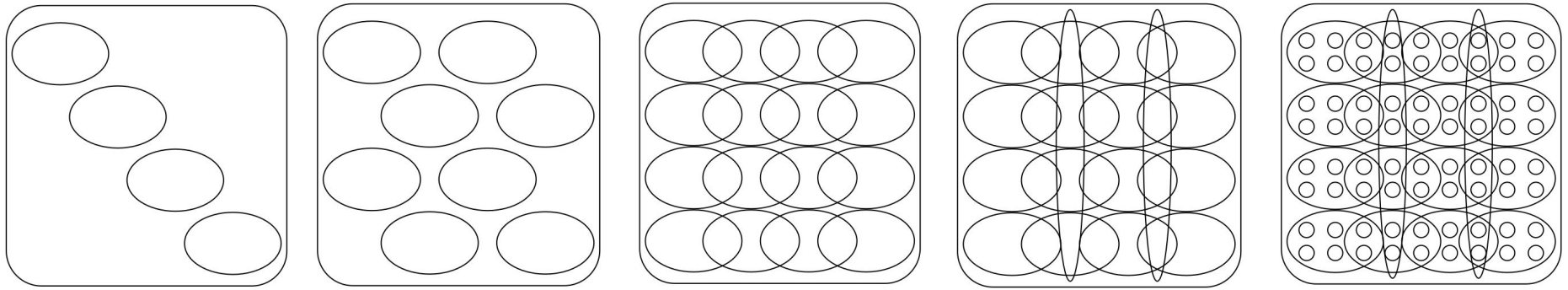
Linux *cal* program documentation

```
NAME
     cal - display a calendar

SYNOPSIS
     cal [ [ month ] year ]

AVAILABILITY
     SUNWesu

DESCRIPTION
     The cal utility writes a Gregorian calendar to standard out-
     put.   If the year operand is specified, a calendar for that
     year is written.  If no operands are specified,  a  calendar
     for the current month is written.

OPERANDS
     The following operands are supported:

     month   Specify the month to be displayed, represented as  a
             decimal  integer  from 1 (January) to 12 (December).
             The default is the current month.

     year    Specify  the  year  for  which  the  calendar  is
             displayed,  represented  as a decimal integer from 1
             to 9999.  The default is the current year.
```

```
ENVIRONMENT
     See environ(5) for descriptions of the following environment
     variables  that  affect  the  execution  of  cal:   LC_TIME,
     LC_MESSAGES, and NLSPATH.

EXIT STATUS
     The following exit values are returned:
     0         Successful completion.
     >0        An error occurred.

SEE ALSO
     calendar(1), environ(5)

NOTES
     An unusual calendar is printed for September 1752.  That  is
     the  month  11 days were skipped to make up for lack of leap
     year adjustments.  To see this calendar, type:
          cal 9 1752

     The command cal 83 refers to the year 83, not 1983.

     The year is always considered to start in January.
```

Purpose of Testing Criteria

Input domain representation



Power of a testing criterion

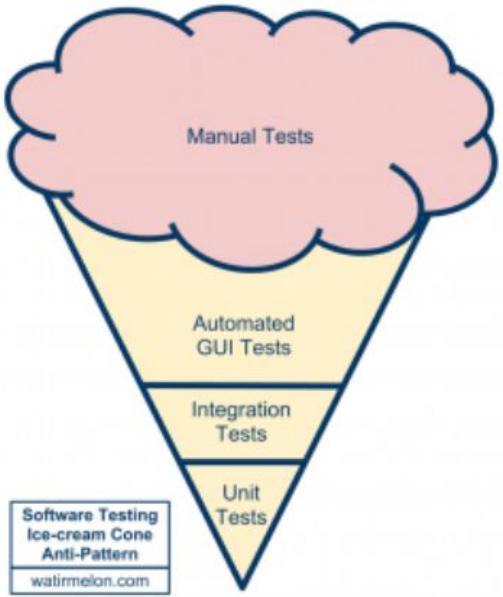
Weak

Strong

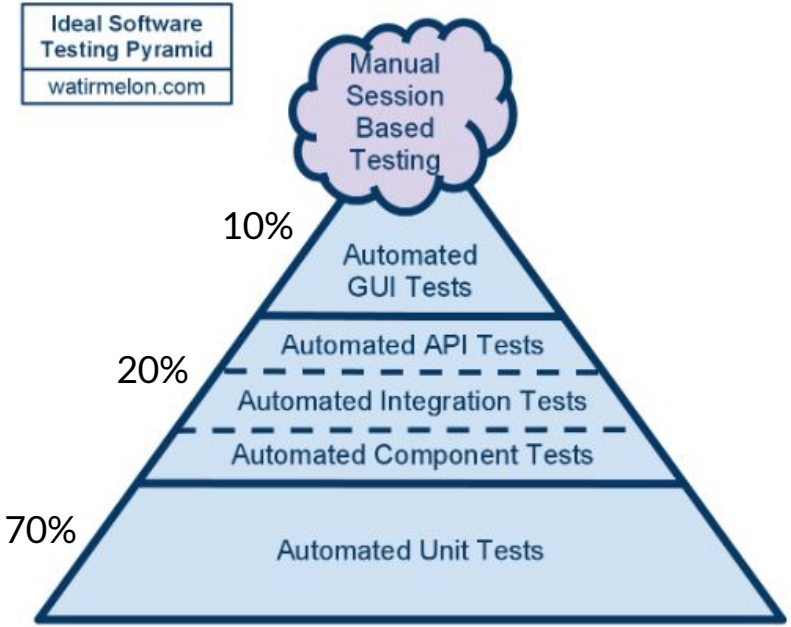
Adapted from Pezzè, M. & Young (2007)

Testing Pyramid and Testing Automation

What many teams do



How it should be done



	Unit	End-toEnd
Fast		
Reliable		
Isolates Failures		
Simulates a Real User		

Source: Mike Wacker. Just Say No to More End-to-End Tests, 2015. Disponível em: https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html

Agenda

- Software Testing Basic Definitions
- Mutation Testing
- Research Project on Mutation Testing
- Some Recent Publications
- Other Research Topics Under Development
- Final Remarks

Some Limitations of Conventional Testing

- Code coverage does not necessarily ensure software quality
- Not all bugs are caught, and some very important ones may be missed
- False sense of security
- In certain circumstances, we need to use more powerful testing criteria

What Is Mutation Testing?

- A criterion to assess the quality of a test set by introducing small changes (mutations) to a program and observing if the test set is able to catch them

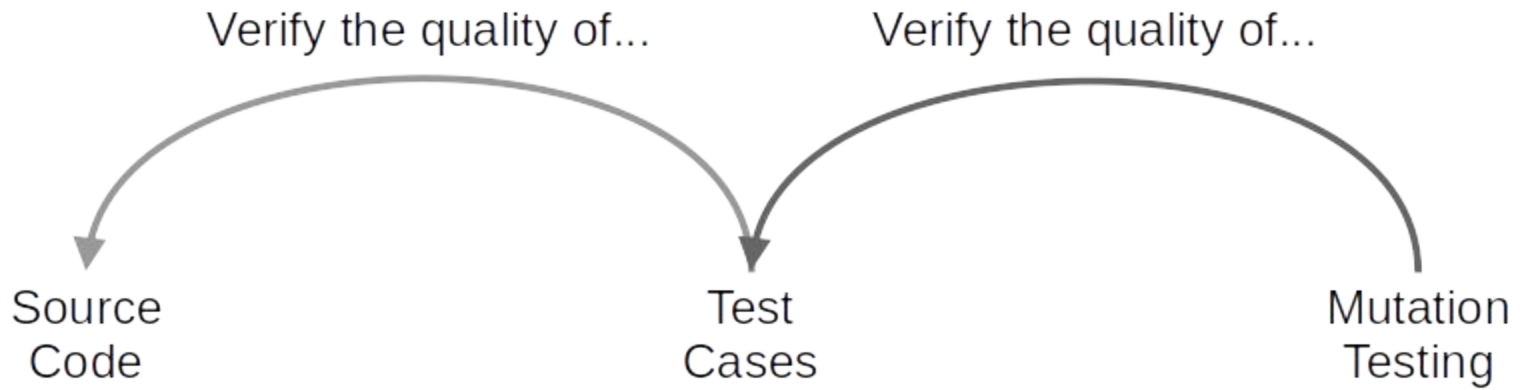


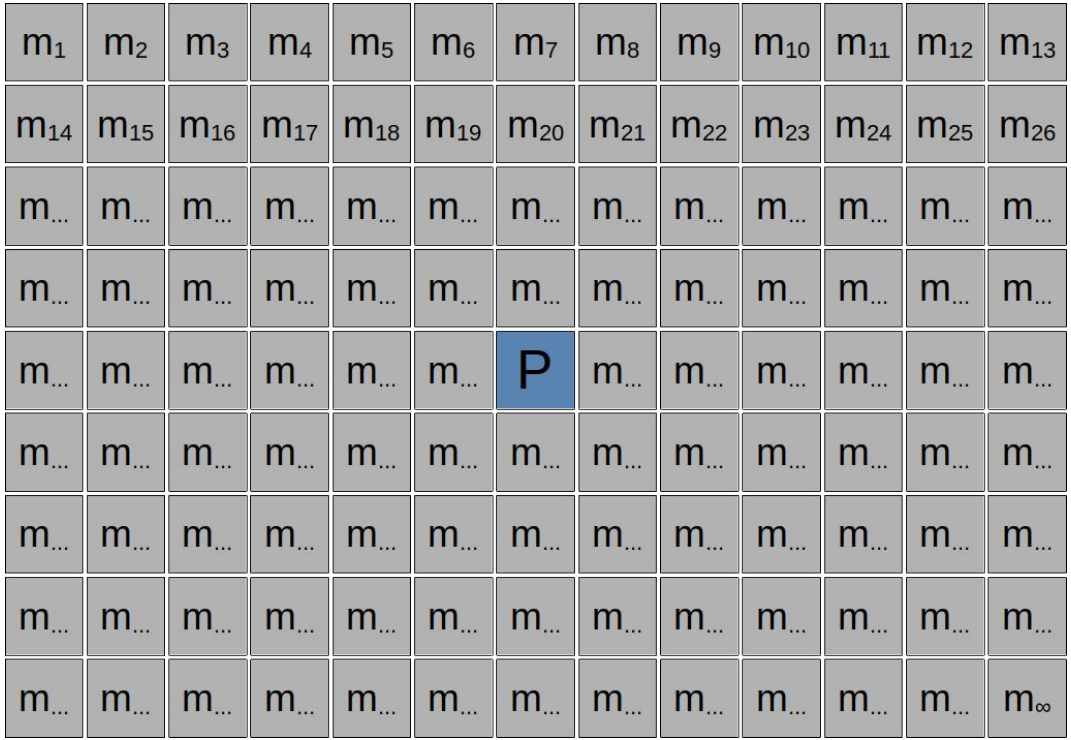
Illustration adapted from: https://www.testingbrain.com/whitebox/mutation-testing.html

Mutation Testing (1)

Absolute Correction

- Consider a Specification S and Program P which implements S
- Consider a test set T
- For each $t \in T$, $P(t) = S(t)$
- Consider $\mathbf{M} = \{m_1, m_2, m_3, \ldots, m_\infty\}$ as the set of **all possible faulty** versions of P
- For each $m \in M$ there is at least one test case $t \in T$ such that $m(t) \neq P(t)$

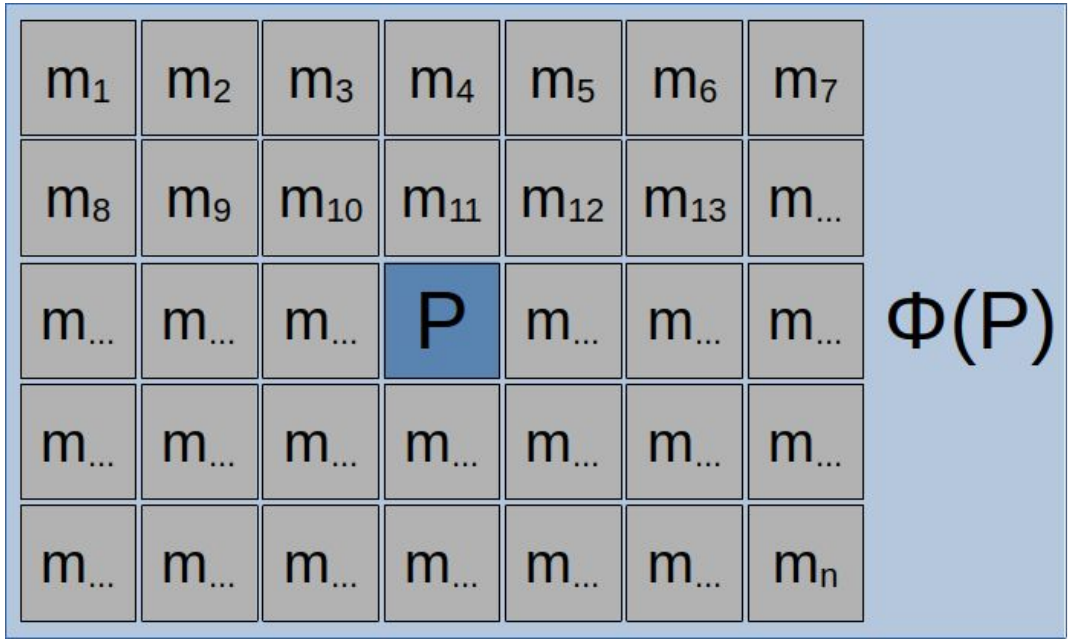
The problem is that M is an infinite neighborhood. It is impossible to execute and compare all m_i



Mutation Testing (2)

One possible solution…

- Establish a finite neighborhood
- Each m_i has small syntax deviation, which represents common faults
- **Mutation Operators** model the syntax deviations
- They implement the syntax changes to create the mutants
- Each mutant represents a “possible” faulty version of P



Mutation Operators

For each language, in general, there is a difference set of Mutation Operators

- Java
  - Pitest - 29 mutation operators (full set)
- C
  - Proteum/IM - 75 mutation operators (unit testing) + 33 operators (integration testing)
- Python
  - MutPy - 20 traditional + 7 experimental (full set)

Example of Mutant (1)

Conditionals Boundary Mutator (CONDITIONALS_BOUNDARY)

Active by default

The conditionals boundary mutator replaces the relational operators `<, <=, >, >=`

with their boundary counterpart as per the table below.

Original conditional	Mutated conditional
<	<=
<=	<
>	>=
>=	>

https://pitest.org/quickstart/mutators/

Original Program

```
public boolean validateIdentifier(String s) {
        char achar;
        boolean valid_id = false;
        achar = s.charAt(0);
        valid_id = valid_s(achar);
        if (s.length() > 1) {
        ...
```

Mutant

```
public boolean validateIdentifier(String s) {
        char achar;
        boolean valid_id = false;
        achar = s.charAt(0);
        valid_id = valid_s(achar);
        if (s.length() >= 1) {
        ...
```

Example of Mutant (2)

Negate Conditionals Mutator (NEGATE_CONDITIONALS)

Active by default

The negate conditionals mutator will mutate all conditionals found according to the replacement table below.

Original conditional	Mutated conditional
==	!=
!=	==
<=	>
>=	<
<	>=
>	<=

https://pitest.org/quickstart/mutators/

Original Program

```
public boolean validateIdentifier(String s) {
      char achar;
      boolean valid_id = false;
      achar = s.charAt(0);
      valid_id = valid_s(achar);
      if (s.length() > 1) {
      ...
```

Mutant

```
public boolean validateIdentifier(String s) {
      char achar;
      boolean valid_id = false;
      achar = s.charAt(0);
      valid_id = valid_s(achar);
      if (s.length() <= 1) {
      ...
```

Example of Mutant (3)

Original Program

```
while (i < s.length() - 1) {
      achar = s.charAt(i);
      if (!valid_f(achar)) {
            valid_id = false;
      }
      i++;
}
```

Mutant

```
while (i < s.length() - 1) {
      achar = s.charAt(i);
      if (!valid_f(achar)) {
            valid_id = false;
      }
      i--;
}
```

Increments Mutator (INCREMENTS)

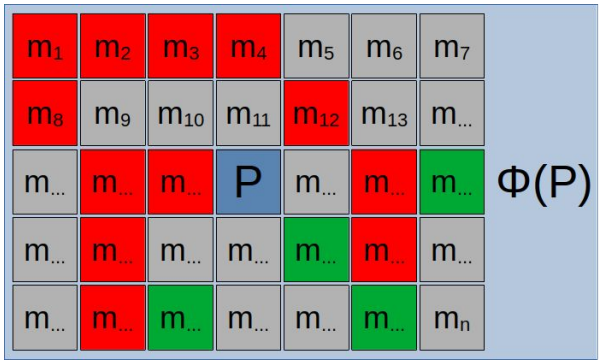
Active by default

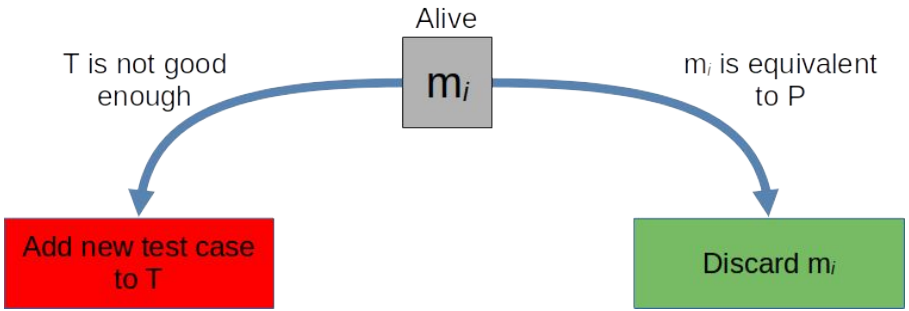
The increments mutator will mutate increments, decrements and assignment increments and decrements of local variables (stack variables). It will replace increments with decrements and vice versa.

https://pitest.org/quickstart/mutators/

Killed and Alive Mutants

- When the behavior of the mutant and the original program differs mutant is **killed**
  - By killing a mutant you ensure your program does not have the fault represented by the mutant
- If a mutant is not killed, it is alive
- A mutant can stay alive for two reasons:
  - The mutant and the original program are **equivalent**
  - The **test set** is **not good enough** to expose the fault (these are the best mutants)





Mutation Score

- The measure of adequacy of T concerning its ability to kill non-equivalent mutants
- Considering
  - **M** - the set of all generated mutants
  - **DM -** the set of all dead mutants
  - **EM** - the set of all equivalent mutants
- Mutations Score (MS) is defined as:

$$MS = \frac{|DM|}{|M| - |EM|}$$

Major Problem with Mutation Testing

- High number of generated mutants
  - More mutants more time to compile and execute
  - More mutants more alive mutants to analyse
- To solve this problem people studies several alternatives:
  - Selective mutation uses only a subset of mutations or mutation operators
  - Parallel mutants execution
  - Multi mutants to reduce compilation time
  - Use of control-flow information to avoid execute mutants not touched by the test case
  - Automatically equivalent mutant detection

Agenda

- Software Testing Basic Definitions
- Mutation Testing
- Research Project on Mutation Testing
- Some Recent Publications
- Other Research Topics Under Development
- Final Remarks

Research Project on Mutation Testing

FAPESP - São Paulo State Research Found Foundation (Grant nº 2019/23160-0)

- **Mutation-Based Software Testing with High Efficiency and Low Technical Debt: Automated Process and Free Support Environment Prototype**
  - **Objective**: Considering the relevance of automating testing activities for the software product industry and the relevance of mutation-based testing from scientific evidence, this project aims to define a mutation-based testing process that can be performed in a 100% automated manner through a supportive test environment, specified, and validated with industrial and open-source applications. **Methods**: Investigate and automate three classic mutation test problems: 1) mutant generation; 2) execution of mutants, and 3) analysis of live and equivalent mutants in the test process. In the generation, it is intended to use static analysis and control and data flow information to select the points that the mutations shall be performed, in addition to selecting specific types of operators to be used depending on the characteristics of the product under test. In the execution of mutants, the selection of good test cases with a high probability of killing non-equivalent mutants is very important. To do this, it is known that different automatic test data generators must be combined and, possibly, new generation algorithms must be developed to kill mutants generated by specific mutation operators. In the analysis of living and equivalent mutants, we need to define and evaluate automated strategies for the determination of equivalent mutants, using heuristics, Bayesian learning, and the frequency of execution of the mutants by the test cases. **Expected results**: a free testing process and a support test environment, as well as experimentation data that allows the generation of benchmarks for testing C, Java, and Python, as well as for the development of new research related to the mutation test. Process and Environment will be applied and evaluated in industrial partners which are formally supporting the present project with interest in its results to be used on production. Technical debt in the context of this proposal will be conducted to characterize the risks associated with the production and release of software products based on mutation testing criterion. **Conclusions**: It is intended to enable the application of mutation testing as an important mechanism for ensuring the quality of software products in a fully automated way, favoring the technological transfer and application of the mutation test to the industry, with the consequent evolution of the software production capacity of the Brazilian software industry.

Agenda

- Software Testing Basic Definitions
- Mutation Testing
- Research Project on Mutation Testing
- Some Recent Publications
- Other Research Topics Under Development
- Final Remarks

Recent Publications

An initial investigation of ChatGPT unit test generation capability*

Vitor H. Guilherme†
vitor.guilherme@estudante.ufscar.br
Federal University of São Carlos
São Carlos, SP, Brazil

Auri M. R. Vincenzi*
auri@ufscar.br
Federal University of São Carlos
São Carlos, SP, Brazil

ABSTRACT

Context: Software testing plays a crucial role in ensuring the quality of software, but developers often disregard it. The use of automated testing generation is pursued with the aim of reducing the consequences of overlooked test cases in a software project. **Problem**: In the context of Java programs, several tools can completely automate generating unit test sets. Additionally, there are studies conducted to offer evidence regarding the quality of the generated test sets. However, it is worth noting that these tools rely on machine learning and other AI algorithms rather than incorporating the latest advancements in Large Language Models (LLMs). **Solution**: This work aims to evaluate the quality of Java unit tests generated by an OpenAI LLM algorithm, using metrics like code coverage and mutation test score. **Method**: For this study, 33 programs used by other researchers in the field of automated test generation were selected. This approach was employed to establish a baseline for comparison purposes. For each program, 33 unit test sets were generated automatically, without human interference, by changing Open AI API parameters. After executing each test set, metrics such as code line coverage, mutation score, and success rate of test execution were collected to evaluate the efficiency and effectiveness of each set. **Summary of Results**: Our findings revealed that the OpenAI LLM test set demonstrated similar performance across all evaluated aspects compared to traditional automated Java test generation tools used in the previous research. These results are particularly remarkable considering the simplicity of the experiment and the fact that the generated test code did not undergo human analysis.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**; **Empirical software validation**; **Software defect analysis**.

KEYWORDS

software testing, experimental software engineering, automated test generation, coverage testing, mutation testing, testing tools

*This work is partially supported by Brazilian Funding Agencies CAPES - Grant 001, FAPESP - Grant n° 2019/23160-0, and CNPq
†All authors contributed equally to the paper content.

SAST 2023, September 25–29, 2023, Campo Grande, MS, Brazil

ACM ISBN 979-8-4007-1629-4/23/09…$15.00
https://doi.org/10.1145/3624032.3624035

ACM Reference Format:
Vitor H. Guilherme and Auri M. R. Vincenzi. 2023. An initial investigation of ChatGPT unit test generation capability. In *8th Brazilian Symposium on Systematic and Automated Software Testing (SAST 2023), September 25–29, 2023, Campo Grande, MS, Brazil*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3624032.3624035

1 INTRODUCTION

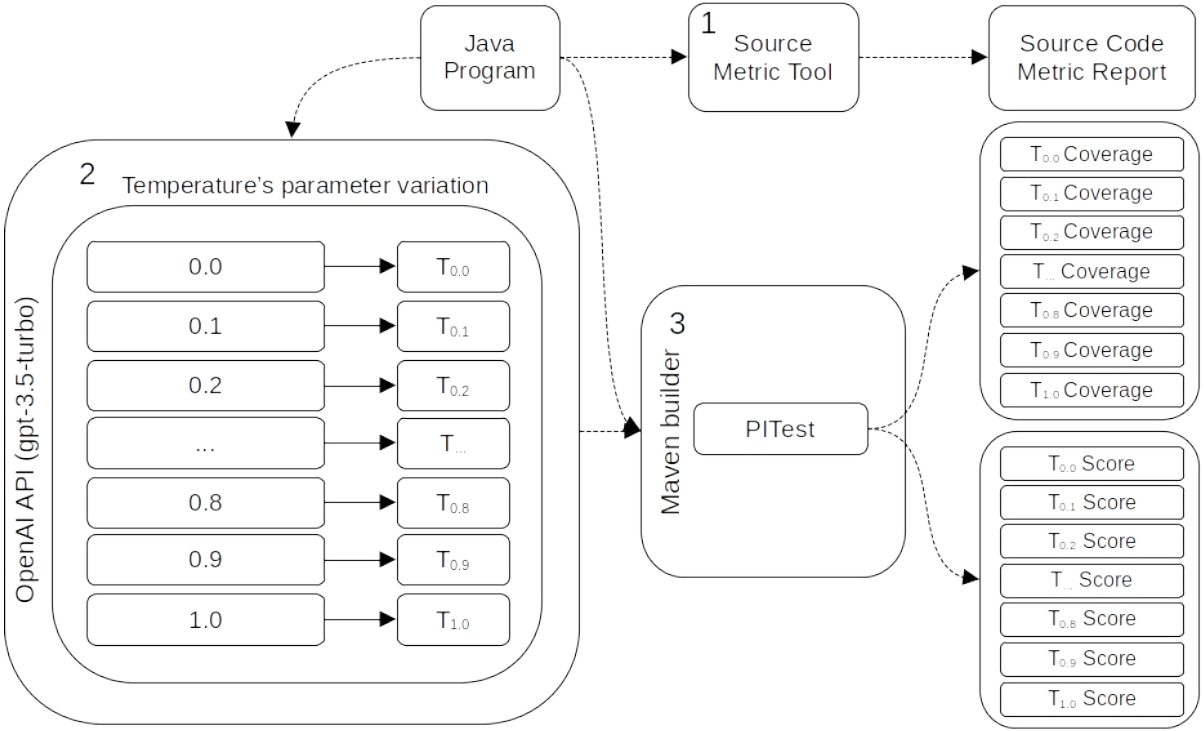
Unit testing is an essential practice in software development to ensure the correctness and robustness of individual code units. These tests, typically written by developers, play a crucial role in identifying defects and validating the expected behavior of software components. DevOps pipelines are strongly based on the quality of the unit tests. However, manually generating comprehensive unit tests can be challenging and time-consuming, often requiring significant effort and expertise. To address these challenges, researchers have explored automated approaches for test generation [1, 8], leveraging advanced techniques and tools.

In this work, we focus on evaluating the quality of Java unit tests generated by an OpenAI Large Language Model (LLM) that has demonstrated remarkable capabilities in generating tests across various domains [23, 28, 29]. Our evaluation will utilize three key quality parameters: code coverage, mutation score, and build and execute success rate of test sets. Code coverage quantifies the extent to which the tests exercise different parts of the code, indicating the thoroughness of the test suite. On the other hand, the mutation score measures the ability of the tests to detect and kill mutated versions of the code, providing insights into the fault-detection capability [2]. Finally, the build and success execution rate measures the reliability of the generated tests.

In order to conduct a thorough and comprehensive analysis, this study will compare the quality of the unit tests generated by the selected LLM with those produced by other prominent Java test generation tools, such as EvoSuite[1]. This comparative evaluation aims to determine if the LLMs can outperform state-of-the-art Java test generation tools and will leverage relevant data from Araujo and Vincenzi [3] research to provide a meaningful benchmark for comparison. By assessing the effectiveness and performance of the LLMs against established tools, we can gain valuable insights into their capabilities and potential advantages in generating high-quality unit tests for Java programs.

This study is part of an ongoing project which aims to support mutation testing in a full automated way[2]. In this sense, we are

[1] https://www.evosuite.org/
[2] Mutation-Based Software Testing with High Efficiency and Low Technical Debt: Automated Process and Free Support Environment Prototype (FAPESP Grant N° 2019/23160-0)



Recent Publications

```
Generate test cases just for the {cut}
Java class in one Java class file with
imports using JUnit 4 and Java 8:

{code}
```

Figure 2: Prompt version 1 for test set generation

Table 3: Average Data for Each Temperature Parameter – Prompt version 1

Temp.	# of Suc. Test	% of Suc.	AVG Cov.	AVG Score
0.0	37	37.4	83.0	51.3
0.1	37	37.4	85.7	51.6
0.2	37	37.4	84.6	52.3
0.3	38	38.4	86.1	53.9
0.4	39	39.4	86.9	53.4
0.5	35	35.4	88.3	53.6
0.6	52	52.5	83.9	54.4
0.7	36	36.4	88.9	54.8
0.8	45	45.5	87.6	54.2
0.9	42	42.4	81.5	49.2
1.0	41	41.4	81.8	52.9

Recent Publications

Table 4: Average Data for Each Temperature Parameter – Prompt version 2

Temp.	# of Suc. Test	% of Suc.	AVG Cov.	AVG Score
0.0	61	61.6	93.5	58.8
0.1	59	59.6	93.4	57.4
0.2	64	64.6	90.7	57.4
0.3	63	63.6	91.2	57.7
0.4	59	59.6	92.0	57.8
0.5	55	55.6	93.3	57.7
0.6	63	63.6	88.0	55.9
0.7	54	54.5	89.9	55.4
0.8	55	55.6	88.6	55.3
0.9	54	54.5	85.8	54.1
1.0	61	61.6	87.7	54.1

```
I need functional test cases to cover all
decisions in the methods of the class
under testing.
All conditional expressions must assume
true and false values.
Tests with Boundary Values are also
mandatory. For numeric data, always use
positive and negative values.
All tests must be in one Java class file.
Include all necessary imports.
It is mandatory to throws Exception
in all test method declarations.
It is mandatory to include timeout=1000
in all @Test annotations.
It is mandatory to test for the default
constructor.
Each method in the class under test must
have at least one test case.
Even simple or void methods must have a
test calling it with valid inputs.
@Test(expected= must be used only if the
method under testing explicitly throws
an exception.
Test must be in JUnit 4 framework format.
Test set heather package and import
dependencies:
package ds;
import org.junit.Test;
import org.junit.Before;
import static org.junit.Assert.*;
import ds.*;
The class under testing is {clazz}.
The test class must be {cut}Test

Class under testing
******************

{code}
```

Figure 3: Prompt version 2 for test set generation

Recent Publications

Table 5: Number of successful tests per temperature per project

ID	Program	Temperature											
		0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	All
1	Max	0	1	0	1	2	0	1	1	1	1	2	10
2	MaxMin1	2	1	2	3	3	2	2	2	0	3	2	22
3	MaxMin2	3	3	3	2	2	2	3	3	2	2	2	27
4	MaxMin3	0	0	0	0	0	0	0	0	0	0	1	1
5	Sort1	3	3	3	3	3	3	3	3	3	3	3	33
6	FibRec	0	0	1	0	1	2	2	2	2	3	2	15
7	FibIte	0	0	0	1	2	1	2	0	3	1	2	12
8	MaxMinRec	3	3	3	3	2	2	3	2	3	2	3	29
9	Mergesort	3	3	3	3	3	3	3	3	3	3	3	33
10	MultMatrixCost	0	0	0	0	0	0	0	0	0	0	0	0
11	ListArray	3	3	3	3	3	2	3	3	2	2	2	29
12	ListAutoRef	3	3	3	2	3	3	3	3	3	2	1	29
13	StackArray	3	3	3	3	3	3	3	3	3	3	2	32
14	StackAutoRef	3	3	3	3	2	3	3	2	3	2	3	30
15	QueueArray	0	0	1	2	0	2	3	3	3	2	2	18
16	QueueAutoRef	3	3	3	3	3	2	3	2	2	3	1	28
17	Sort2	0	0	1	0	0	0	0	0	0	1	1	3
18	HeapSort	0	0	0	0	0	0	0	0	0	0	0	0
19	PartialSorting	0	0	0	0	0	0	0	0	0	0	0	0
20	BinarySearch	3	2	3	2	3	2	2	3	0	0	2	22
21	BinaryTree	3	3	3	3	3	2	3	2	2	3	3	30
22	Hashing1	3	3	3	3	1	2	1	2	2	2	2	24
23	Hashing2	0	1	1	1	0	1	0	0	2	1	1	8
24	GraphMatAdj	3	3	3	3	3	3	3	0	2	2	3	28
25	GraphListAdj1	3	3	2	3	3	1	2	1	1	3	2	24
26	GraphListAdj2	3	3	3	3	3	3	3	2	3	3	2	31
27	DepthFirstSearch	3	2	3	3	2	2	3	2	2	0	2	24
28	BreadthFirstSearch	3	2	3	1	1	1	0	1	1	0	3	16
29	Graph	2	2	2	3	2	2	2	2	1	1	1	20
30	PrimAlg	0	0	0	0	0	0	1	1	0	0	1	3
31	ExactMatch	3	3	3	3	3	3	3	3	3	3	3	33
32	AproximateMatch	3	3	3	3	3	3	3	2	3	2	3	31
33	Identifier	0	0	0	0	0	0	0	1	0	1	1	3
# Successful Test		61	59	64	63	59	55	63	54	55	54	61	648
% Successful Test		61.6	59.6	64.6	63.6	59.6	55.6	63.6	54.5	55.6	54.5	61.6	59.5
# of Programs without test		12	10	8	8	9	8	8	8	9	8	3	3
% of Programs without test		36.4	30.3	24.2	24.2	27.3	24.2	24.2	24.2	27.3	24.2	9.1	9.1

Table 6: All LLM test sets versus baseline test sets

ID	LLM Suite		Baseline Suite[11]	
	Coverage	Score	Coverage	Score
1	100.0	85.7	100.0	64.3
2	100.0	85.7	100.0	83.8
3	100.0	85.7	100.0	84.3
4	100.0	64.5	100.0	79.8
5	100.0	80.0	100.0	78.5
6	100.0	100.0	100.0	100.0
7	100.0	100.0	100.0	100.0
8	100.0	83.3	100.0	83.1
9	100.0	96.4	100.0	95.5
10	0.0	0.0	100.0	45.6
11	100.0	93.5	100.0	78.1
12	100.0	87.0	100.0	83.9
13	100.0	96.8	100.0	81.3
14	100.0	93.5	100.0	85.5
15	100.0	93.0	100.0	90.5
16	100.0	67.6	100.0	82.4
17	100.0	57.6	100.0	68.8
18	0.0	0.0	100.0	73.9
19	0.0	0.0	100.0	84.4
20	100.0	94.7	100.0	70.4
21	81.3	62.5	88.8	93.8
22	100.0	57.3	100.0	93.9
23	98.1	63.3	100.0	86.6
24	100.0	73.4	96.2	69.0
25	100.0	78.9	100.0	81.9
26	98.0	77.2	99.2	78.1
27	100.0	78.7	100.0	81.0
28	100.0	78.7	100.0	82.1
29	100.0	78.7	100.0	81.4
30	100.0	71.1	100.0	42.4
31	100.0	39.5	100.0	58.6
32	100.0	38.4	100.0	51.6
33	100.0	64.0	100.0	75.8
AVG	90.2	70.5	99.5	78.5
SD	29.2	27.5	2.0	13.9
AVG**	99.2	77.6	99.5	79.5
SD**	3.4	16.5	2.1	13.1

** - AVG and SD removing zeros.

Recent Publications

An Experimental Study Evaluating Cost, Adequacy, and Effectiveness of Pynguin's Test Sets[†]

Lucca R. Guerino[*]
Federal University of São Carlos
São Carlos, SP, Brazil
luccaguerino@estudante.ufscar.br

Auri M. R. Vincenzi[*]
Federal University of São Carlos
São Carlos, SP, Brazil
auri@ufscar.br

ABSTRACT

Context: Software testing is a very relevant step in quality assurance, but developers frequently overlook it. We pursued testing automation to minimize the impact of missing test cases in a software project. **Problem**: However, for Python programs, there are not many tools able to fully automate the generation of unit test sets, and the one available demands studies to provide evidence of the quality of the generated test set. **Solution**: This work aims to evaluate the quality of different unit test generation algorithms for Python, implemented in a tool named Pynguin. **Method**: In the analysis of the selected programs, the Pynguin test generation tool is executed with each of its algorithms, including random, as a way to generate complete unit test sets. Then, we evaluate each generated test set's efficacy, efficiency, and cost. We use four different fault models, implemented by four mutation testing tools, to measure efficacy. We use line and branch coverage to measure efficiency, the number of test cases, and test set execution time to measure cost. **Summary of Results**: We identified that RANDOM test set performed worst concerning all evaluated aspects, DYNAMOSA and MOSA, the two algorithms that generate the best test sets regarding efficacy, efficiency, and cost. By combining all Pynguin smart algorithms (DYNAMOSA, MIO, MOSA, WHOLE-SUITE), the resultant test set overcomes the individual test sets efficiency by around 1%, for coverage and efficacy by 4.5% on average, concerning previous mutation score, at a reasonable cost, without a test set minimization.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**; **Empirical software validation**; **Software defect analysis**.

KEYWORDS

software testing, experimental software engineering, automated test generation, coverage testing, mutation testing, testing tools

[*] All authors contributed equally to the paper content.

[†] This work is partially supported by Brazilian Funding Agencies CAPES - Grant 001, FAPESP - Grant n° 2019/23160-0, and CNPq.

SAST 2023, September 25–29, 2023, Campo Grande, MS, Brazil

ACM ISBN 979-8-4007-1629-4/23/09…$15.00
https://doi.org/10.1145/3624032.3624034

ACM Reference Format:
Lucca R. Guerino and Auri M. R. Vincenzi. 2023. An Experimental Study Evaluating Cost, Adequacy, and Effectiveness of Pynguin's Test Sets[†]. In *8th Brazilian Symposium on Systematic and Automated Software Testing (SAST 2023), September 25–29, 2023, Campo Grande, MS, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3624032.3624034

1 INTRODUCTION

Software testing is one of the most crucial steps in software quality assurance, yet it often escapes the attention of most developers. The potential to employ tools to automate the testing process could alleviate the burden of a frequently neglected and resource-intensive task: testing. Recent years have witnessed numerous advancements in software testing and process automation, although significant room for improvement remains. There are initiatives of a complete automated testing process [1, 8], and the automatic test generators are a fundamental piece of this process [3, 14].

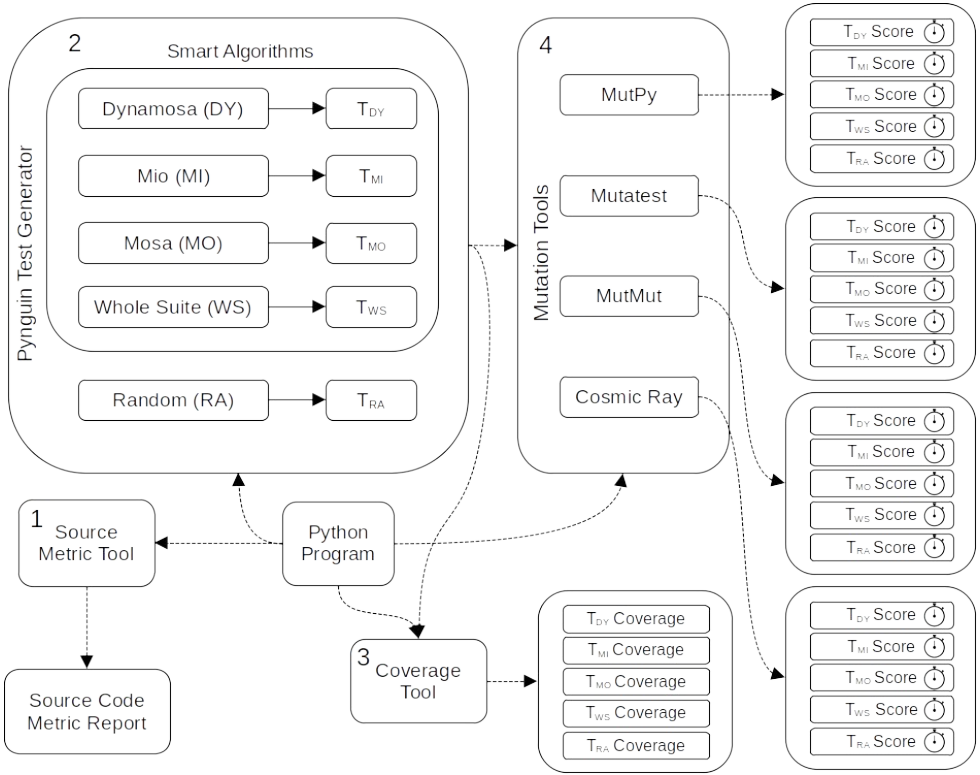
In 2023, the Tiobe Index [23] highlights Python as the most popular programming language. A few choices are optimal when considering unit test generators for Python, partly because Python is a dynamically typed language. This feature, allowing variables to have undefined data types, presents challenges for developing automated test generators [7, 13].

Some automated test-data generators exist for Python language, but not all of them work in a fully automated way. One of them that works is a tool called Pynguin [14]. This tool supports the generation of automatic test cases based on different generation algorithms: DYNAMOSA [19], MIO [4], MOSA [18], WHOLE-SUITE [20] and also a feedback oriented random test generation [17].

Due to Python's popularity and the existence of different algorithms that generate unit test sets automatically, it is natural to question the quality of the generated test sets. This paper investigates the quality of different Pynguin generated test sets concerning cost, adequacy, and effectiveness. We will use the test suite's size and the tests' execution time against mutation tools as cost metrics. We will measure test adequacy using line and branch coverage. Lastly, we will evaluate the effectiveness of the automatically generated test sets using four different fault models on four distinct mutation testing tools: MutPy [10], MutMut [11], Mutatest [12], and Cosmic-Ray [6].

We use these four mutation testing tools to expand the range of potential flaws that Pynguin test sets can detect. Moreover, no concrete evidence suggests that one mutation tool for Python excels others concerning test set generators. Apart from helping choose the best Pynguin's algorithm or unit test set generation, the collected data will also enable a further comparative analysis of Python's mutation testing tools [2].

Therefore, we can summarize these paper's contributions:



Recent Publications

Table 6: Branch Coverage of Each Test Set

ID	RA	DY	MI	MO	WS
P01	37.10	80.65	64.52	80.65	70.97
P02	50.00	77.78	59.26	72.22	68.52
P03	44.83	76.72	66.38	75.00	73.28
P04	66.67	100.00	100.00	100.00	100.00
P05	85.71	100.00	100.00	100.00	100.00
P06	71.05	97.37	92.11	94.74	94.74
P07	77.27	90.91	90.91	90.91	90.91
P08	83.33	83.33	83.33	83.33	83.33
P09	94.44	100.00	100.00	100.00	100.00
P10	80.00	100.00	100.00	100.00	100.00
P11	100.00	100.00	100.00	100.00	100.00
P12	75.00	88.89	91.67	91.67	83.33
P13	88.89	94.44	94.44	94.44	94.44
P14	80.56	97.22	94.44	97.22	94.44
P15	100.00	100.00	100.00	100.00	100.00
P16	73.08	80.77	61.54	88.46	61.54
P17	70.83	91.67	91.67	91.67	91.67
P18	100.00	100.00	100.00	100.00	100.00
P19	83.33	83.33	83.33	83.33	83.33
P20	94.44	94.44	94.44	94.44	100.00
P21	100.00	100.00	50.00	100.00	100.00
AVG	78.88	92.26	86.57	92.29	90.02
SD	17.63	8.25	15.70	8.54	11.98

Table 7: Average mutation score for each mutation tool

Tool	RA	DY	MI	MO	WS
MutPy - AVG	73.99	79.13	74.76	77.83	77.08
MutMut - AVG	62.34	69.43	62.84	68.08	66.77
Mutatest - AVG	64.92	70.18	65.72	68.75	67.69
Cosmic-Ray - AVG	67.40	75.51	71.55	74.47	72.80
MutPy - SD	15.64	11.33	13.73	12.33	14.63
MutMut - SD	13.01	11.18	16.18	12.41	13.73
Mutatest - SD	18.55	17.07	19.05	18.98	18.03
Cosmic-Ray - SD	19.68	16.92	19.97	18.26	18.72

Recent Publications

Table 12: All-Smart test set mutation score

ID	Mutpy	MutMut	Mutatest	CosmicRay
P01	81.17	86.11	92.68	80.09
P02	66.13	60.61	48.15	72.90
P03	74.56	72.63	60.00	75.95
P04	92.45	88.24	95.45	100.00
P05	84.00	73.77	85.71	87.14
P06	79.64	78.95	91.67	72.60
P07	88.68	84.09	81.48	88.64
P08	79.34	64.00	70.00	70.37
P09	91.82	76.09	76.67	90.28
P10	91.18	87.04	70.59	93.24
P11	88.06	77.78	80.95	100.00
P12	86.02	83.56	96.00	85.48
P13	91.44	83.16	74.29	80.22
P14	83.84	81.08	80.00	85.71
P15	91.67	85.00	90.48	81.25
P16	46.96	62.30	57.69	45.41
P17	71.32	60.29	45.45	35.09
P18	89.66	83.33	84.62	94.74
P19	83.23	68.42	62.5	70.00
P20	87.70	65.96	68.00	84.72
P21	91.43	55.56	58.82	80.00
AVG	82.87	75.14	74.82	79.71
SD	10.70	10.03	14.85	15.49

Table 14: All-Smart test set surviving mutants

ID	MutPy			MutMut			Mutatest			Cosmic-Ray		
ID	Live	Total	%	Live	Total	%	Live	Total	%	Live	Total	%
P01	45	239	18.8	10	72	13.9	3	41	7.3	42	211	19.9
P02	84	248	33.9	39	99	39.4	14	27	51.9	58	214	27.1
P03	130	511	25.4	49	179	27.4	8	20	40.0	70	291	24.1
P04	4	53	7.6	2	17	11.8	1	22	4.6	0	18	0.0
P05	16	100	16.0	16	61	26.2	5	35	14.3	18	140	12.9
P06	45	221	20.4	20	95	21.1	2	24	8.3	40	146	27.4
P07	12	106	11.3	7	44	15.9	5	27	18.5	5	44	11.4
P08	25	121	20.7	18	50	36.0	9	30	30.0	16	54	29.6
P09	9	110	8.2	11	46	23.9	7	30	23.3	7	72	9.7
P10	12	136	8.8	7	54	13.0	5	17	29.4	5	74	6.8
P11	8	67	11.9	6	27	22.2	4	21	19.1	0	14	0.0
P12	26	186	14.0	12	73	16.4	1	25	4.0	9	62	14.5
P13	19	222	8.6	16	95	16.8	9	35	25.7	18	91	19.8
P14	32	198	16.2	14	74	18.9	5	25	20.0	16	112	14.3
P15	4	48	8.3	3	20	15.0	2	21	9.5	3	16	18.8
P16	61	115	53.0	23	61	37.7	11	26	42.3	101	185	54.6
P17	37	129	28.7	27	68	39.7	18	33	54.6	74	114	64.9
P18	3	29	10.3	1	6	16.7	2	13	15.4	1	19	5.3
P19	26	155	16.8	18	57	31.6	9	24	37.5	21	70	30.0
P20	15	122	12.3	16	47	34.0	8	25	32.0	11	72	15.3
P21	3	35	8.6	4	9	44.4	7	17	41.2	1	5	20.0
SUM	616	3151	19.6	319	1254	25.4	135	538	25.1	516	2024	25.5
AVG	29.3	150.0	17.1	15.2	59.7	24.9	6.4	25.6	25.2	24.6	96.4	20.3
SD	31.1	106.3	11.0	12.0	38.4	10.3	4.4	6.7	15.2	28.8	77.3	15.9

Recent Publications

Static and Dynamic Comparison of Mutation Testing Tools for Python

Lucca Renato Guerino
luccaguerino@estudante.ufscar.br
Department of Computing
Federal University of São Carlos
São Carlos, SP, Brazil

Pedro Henrique Kuroishi
phk@ufscar.br
Department of Computing
Federal University of São Carlos
São Carlos, SP, Brazil

Ana C. R. Paiva
apaiva@fe.up.pt
INESC TEC, Faculty of Engineering University of Porto
Porto, Portugal

Auri Marcelo Rizzo Vincenzi*
auri@ufscar.br
Department of Computing
Federal University of São Carlos
São Carlos, SP, Brazil

ABSTRACT

Context: Mutation testing is a rigorous approach for assessing the quality of test suites by injecting faults (i.e., mutants) into software under test. Tools, such as CosmicRay and Mutpy, are examples of Mutation Testing tools for Python software programs. **Problem**: With different Python mutation testing tools, comparative analysis is lacking to evaluate their effectiveness in different usage scenarios. Furthermore, the evolution of these tools makes continuous evaluation of their functionalities and characteristics necessary. **Method**: In this work, we evaluate (statically and dynamically) four Python mutation testing tools, namely CosmicRay, MutPy, MutMut, and Mutatest. In static evaluation, we introduce a comparison framework, adapted from one previously applied to Java tools, and collected information from tool documentation and developer surveys. For dynamic evaluation, we use tests built based on those produced by Pynguin, which are improved through the application of Large Language Models (LLMs) and manual analyses. Then, the adequate test suites were cross-tested among different tools to evaluate their effectiveness in killing mutants each other. **Results**: Our findings reveal that CosmicRay offers superior functionalities and customization options for mutant generation compared to its counterparts. Although CosmicRay's performance was slightly lower than MutPy in the dynamic tests, its recent updates and active community support highlight its potential for future enhancements. Cross-examination of the test suites further shows that mutation scores varied narrowly among tools, with a slight emphasis on MutPy as the most effective mutant fault model.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**; **Empirical software validation**.

SBQS 2024, November 5–8, 2024, Salvador, Brazil

ACM ISBN 979-8-4007-1777-2/24/11…$15.00
https://doi.org/10.1145/3701625.3701659

KEYWORDS

Software Testing, Experimental Software Engineering, Automated Test Generation, Coverage Testing, Mutation Testing, Testing Tools, Python Mutation Tools

ACM Reference Format:

Lucca Renato Guerino, Pedro Henrique Kuroishi, Ana C. R. Paiva, and Auri Marcelo Rizzo Vincenzi. 2024. Static and Dynamic Comparison of Mutation Testing Tools for Python. In *XXIII Brazilian Symposium on Software Quality (SBQS 2024), November 5–8, 2024, Salvador, Brazil.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3701625.3701659

1 INTRODUCTION

Python has emerged as a prominent programming language over the past years. Its ranking in the Tiobe Index [18] is currently recognized as the most widely used language for applications worldwide. Ensuring the quality of Python applications is paramount, and adequate testing is relevant. One way to assess the quality of tests is through mutation testing, a rigorous approach to evaluate a set of tests by injecting known faults into the code.

There are several mutation testing tools for Python programs. However, determining the best tool depends on several factors. This paper aims to evaluate and compare four tools from two distinct perspectives. The first perspective utilizes a static framework previously employed in the assessment of Java mutation testing tools [1], adapted for this study. Building on the work by Amalfitano et al. [1], we will also undertake a dynamic evaluation of these tools. This involves building test suites suitable for each tool, that is, capable of killing mutants generated by that tool, and then using that adequate test suite to evaluate whether they are also capable of killing mutants generated by the other tools. This paper aims to compare four mutation testing tools for Python: MutPy [6], MutMut [7], Mutatest [8], and Cosmic-Ray [3].

Inspired by the work of Guerino and Vincenzi [5], we decided to use their publicly available benchmark. Complementing Pynguin's generated test sets with manual and LLM support, we generate an adequate test set for each mutation testing tool and each subject program, ensuring an even higher degree of software reliability and robustness. We also perform a static assessment of the current state of mutation testing tools and their capabilities by using an adapted version of the framework proposed by [1].

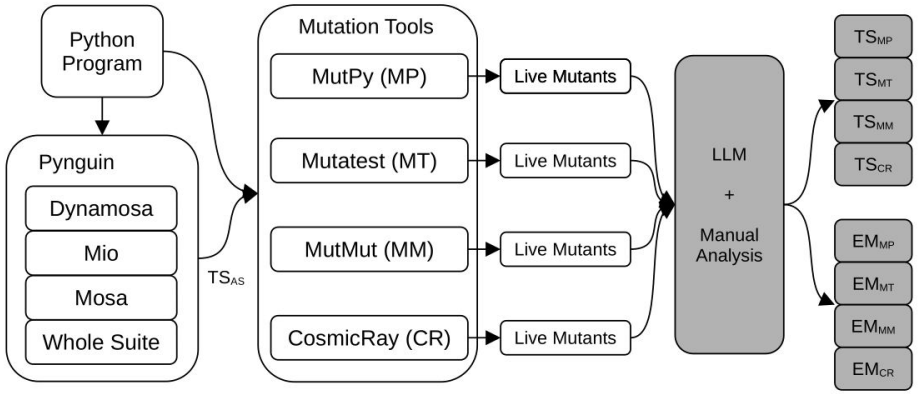


Table 2: Static Framework

Attribute	Mutation Tool: MutPy	Mutatest	MutMut	CosmicRay
Version				
Licence	Apache 2	MIT	BSD-3	MIT
Release version	"0.6.1"	"3.1.0"	"2.4.4"	"8.3.7"
Release year	2019	2022	2023	2023
Last resolved issue	2021	2023	2024	2023
Deployment				
Python version	Python 3.3+	Python 3.7+	Python 3.7+	Python 3.5+
Build-tool integration	-	-	-	✓
Testing framework	Unittest & Pytest	Unittest & Pytest	Unittest & Pytest	Unittest & Pytest
Mutation process				
Mutation level				
Byte code	-	-	-	-
Source code	✓	✓	✓	✓
Test selection				
Automated	-	-	-	-
Manual	✓	✓	✓	✓
Manual (GUI)	-	-	-	-
Mutation operator selection				
Operators	✓	✓	✓	✓
Operator classes	-	-	-	✓
Mutant inspection				
Over LOC	✓	✓	-	✓
Over Live Mutants LOC	-	-	✓	-
Side by side	-	-	-	-
Kill matrix				
Test method	-	-	-	-
Test class	-	-	-	✓
Analysis runtime reduction	-	-	-	-

Table 5: Cross-Examination Score

Tool	TS_{MP}	TS_{MT}	TS_{MM}	TS_{CR}	AVG	SD
MP	100	97.04	96.97	96.65	97.67	1.57
MT	97.92	100	97.26	96.46	97.91	1.52
MM	98.38	96.44	100	95.23	97.51	2.11
CR	99.38	99.08	98.72	100	99.3	0.54
AVG	98.92	98.14	98.24	97.09	-	-
SD	0.94	1.68	1.4	2.04	-	-

Guerino, Lucca Renato, Pedro Henrique Kuroishi, Ana Cristina Ramada Paiva, e Auri Marcelo Rizzo Vincenzi. "Static and Dynamic Comparison of Mutation Testing Tools for Python". Em XXIII Brazilian Symposium on Software Quality. Salvador, BA: SBC, 2024. https://doi.org/10.1145/3701625.3701659.

Agenda

- Software Testing Basic Definitions
- Mutation Testing
- Research Project on Mutation Testing
- Some Recent Publications
- Other Research Topics Under Development
- Final Remarks

Research Project on Mobile Testing (1)

Conselho Nacional de Desenvolvimento Científico e Tecnológico – CNPq (Grant nº 403603/2020-0)

- **Software Testing Platform for Mobile Devices**
  - The proposal outlines the development of a software testing platform for mobile devices, addressing the challenges of testing software on diverse and rapidly evolving mobile devices. It discusses the limitations of traditional testing methods, such as emulators and device farms, and introduces the concept of the Distributed Bug Buster (DBB) platform, which leverages collaborative economy principles to facilitate testing on idle Android devices. The project's specific focus is on adapting DBB to enable testing not only on external devices but also on internal device farms within an organization, addressing issues related to equipment proximity and security. This initiative is part of the MAI/DAI 2020 program, in collaboration with VonBraun Labs.
- Pedro Henrique Kuroishi (Ph.D student - sandwich at FEUP - til March/2024)
- In collaboration with Profa. Dra. Ana Paiva (FEUP/UPorto) and Prof. Dr. João Bispo (FEUP/UPorto)

Research Project on Mobile Testing (2)

- Mobile Testing
  - Kuroishi, Pedro Henrique, Ana Cristina Ramada Paiva, José Carlos Maldonado, e Auri Marcelo Rizzo Vincenzi. “Testing infrastructures to support mobile application testing: A systematic mapping study”. Information and Software Technology 177 (2025): 107573. https://doi.org/10.1016/j.infsof.2024.107573.
  - Kuroishi, Pedro Henrique, José Carlos Maldonado, e Auri Marcelo Rizzo Vincenzi. “Towards the Implementation of a Mobile Application Testing Infrastructure at Von Braun Labs”. Em 2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE), 91–101, 2023. https://doi.org/10.1109/ISSRE59848.2023.00078.
  - Kuroishi, Pedro Henrique, José Carlos Maldonado, e Auri Marcelo Rizzo Vincenzi. “Towards the definition of a research agenda on mobile application testing based on a tertiary study”. Information and Software Technology 167 (2024): 107363. https://doi.org/10.1016/j.infsof.2023.107363.
  - Vincenzi, A. M. R., João Bispo, Pedro Henrique Kuroishi, Ana Rita Veiga, David Roberto Cravo da Mata, Francisco Bernardo Azevedo, e Ana C. R. Paiva. “{METFORD} – {Mutation tEsTing Framework fOR anDroid}”. Journal of Systems and Software, maio de 2024. https://ssrn.com/abstract=4914846.

Research Project on FinOps

- **Automation of cloud application deployment using TOSCA to assist the adoption of DevOps and FinOps**
  - The use of public clouds for application deployment has grown exponentially in recent years. However, some of the challenges in this practice include the lack of automation in the deployment process, managing infrastructure-related costs, and choosing a cloud service provider. In this context, DevOps and FinOps enable greater automation of the application's build, deployment, and testing processes, as well as cost management for the necessary infrastructure. This project explores the idea of generating code automatically to support these two cultures from the application's code stored in a Git repository, using the Topology and Orchestration Specification for Cloud Applications (TOSCA) to manage infrastructure modeling and deployment. The goal is to streamline cloud application development by automating essential activities for DevOps and FinOps, resulting in reduced development and deployment times and infrastructure costs.
- Bruno Lourenço Lopes (Ph.D student)

Research Project on What is a Good Test Case

- **Definition of Guidelines for Creating Good Test Cases based on Practitioners' Experience**
  - Building upon the insights from the paper "Practitioners' Views on Good Software Testing Practices" (Kochhar et al., 2019) and our prior work on "Test case quality: an empirical study on belief and evidence," this research endeavor is driven by the objective of formulating comprehensive quality guidelines for test case creation, primarily founded on the practical wisdom of industry professionals. This multifaceted project encompasses the systematic exploration and analysis of current testing practices, the extraction of valuable insights from practitioners, the systematic characterization of best practices, the proposition of a cohesive framework for creating high-quality test cases, and rigorous validation through a combination of experimental investigations and in-depth interviews with industry experts.
- Camilo Vilotta Ibara (Ph.D student)

Research Project on Using LLM in Integration Testing

- **Exploring the Effectiveness of LLMs in Creating and Improving Integration Test Sets**
  - This research investigates the effectiveness of LLMs in enhancing or creating integration test suites, primarily focusing on RESTful APIs. The research seeks to explore the potential of LLMs for automated test generation and to develop customizations to maximize their support in automatically generating integration test suites. After preliminary studies, the research is specifically directed towards testing RESTful APIs, given their extensive use in distributed systems and data exchange between systems. The research comprises a systematic review, the definition of LLMs for this context, and their customization for generating and analyzing test suites for RESTful APIs. The results will be compared with tests generated by other automated tools and by developers. This research aims to contribute to the advancement of the RESTful API testing framework by harnessing the potential of LLMs as a tool for automating and enhancing test suite generation, exploring test case generation not only from product source code but also from API specification.
- André Mesquita Rincon (Ph.D student)

Agenda

- Software Testing Basic Definitions
- Mutation Testing
- Research Project on Mutation Testing
- Some Recent Publications
- Other Research Topics Under Development
- Final Remarks

Final Remarks

- Testing will always be a difficult task
- Automation is possible, but we need to understand its limitations
- Even with technological advances, some undecidable problems remain (equivalent mutant detection)
- Especially with our project on full mutation testing automation, I think we can do it, but I also need to calculate the associated risk of this full automation
- We cannot do everything automatically. Missing decisions became what we are calling a “Mutation Testing Technical Debt”

IEEE Software, Volume 40, Issue 5, pp. 76-84, DOI: https://doi.org/10.1109/MS.2023.3267296, September/October 2023

Myths and Facts about a Career in Software Testing: A Comparison between Students' Beliefs and Professionals' Experience

Ronnie de Souza Santos[1,3], Luiz Fernando Capretz[2], Cleyton Magalhães[3], Rodrigo Souza[3]
[1] Shannon School of Business, Cape Breton University – Canada
[2] Department of Electrical & Computer Engineering, Western University – Canada
[3] Post-Baccalaureate program on Agile Testing, CESAR School – Brazil

Email: ronnie_desouza@cbu.ca, lcapretz@uwo.ca, cvcm@cesar.school, recs@cesar.school

https://arxiv.org/pdf/2311.06201.pdf

ABSTRACT. Testing is an indispensable part of software development. However, a career in software testing is reported to be unpopular among students in computer science and related areas. This can potentially create a shortage of testers in the software industry in the future. The question is, whether the perception that undergraduate students have about software testing is accurate and whether it differs from the experience reported by those who work in testing activities in the software development industry. This investigation demonstrates that a career in software testing is more exciting and rewarding, as reported by professionals working in the field, than students may believe. Therefore, in order to guarantee a workforce focused on software quality, the academy and the software industry need to work together to better inform students about software testing and its essential role in software development.

INDEX TERMS software testing, software quality, career, software engineering education.

Thank You All

- Additional questions, doubts, discussions…
- You may contact me at any time: auri@ufscar.br
- https://linkedin.com/in/aurimrv
- UPorto - Software Engineering Laboratory - room I122

Cited References

- Dijkstra, E. J.N. Buxton and B. Randell, eds, Software Engineering Techniques, April 1970. Report on a conference sponsored by the NATO Science Committee, Rome, Italy, 27–31 October 1969. p. 16. Possibly the earliest documented use of the famous quote.
- Myers, G. J. The Art of Software Testing. Wiley, New York, 1979.
- ISO/IEC/IEEE Systems and software engineering – vocabulary. ISO/IEC/IEEE 24765:2010(E), v. 1, n. 24765:2010(E), p. 1–418, 2010.
- Pezzè, M.; Young, M. Software Testing and Analysis: Process, Principles and Techniques. John Wiley & Sons, 2007.
- Roper, M. Software Testing. McGrall Hill, 1994.
- Repositório GitHub com os slides e exemplo: https://github.com/aurimrv/UnixCal

Previous Research History

- Software Testing Automation
  - JaBUTi (prototype control- and data-flow testing tool for Java Bytecode)
    - Vincenzi, A. M. R., J. C. Maldonado, W. E. Wong, e M. E. Delamaro. “Coverage Testing of Java Programs and Components”. Journal of Science of Computer Programming 56, nº 1–2 (abril de 2005): 211–30. http://dx.doi.org/10.1016/j.scico.2004.11.013.
  - AMT - Android Mirroring Tool
    - Andrade Freitas, Eduardo Noronha de, Celso G. Camilo-Junior, Kenyo Abadio Crosara Faria, e Auri Marcelo Rizzo Vincenzi. “AMT: An Android Mirror Tool for Instant Feedback Across Platform”. Em VII Congresso Brasileiro de Software: Teoria e Prática - CBSoft 2016 - Sessão de Ferramentas, 97–104. Maringá, PR: SBC, 2016.
  - DBB (collaborative economy concept for mobile testing)
    - Faria, Kenyo Abadio Crosara, Raphael de Aquino Gomes, Eduardo Noronha de Andrade Freitas, e Auri Marcelo Rizzo Vincenzi. “On using collaborative economy for test cost reduction in high fragmented environments”. Future Generation Computer Systems 95 (2019): 502–10. https://doi.org/10.1016/j.future.2019.01.023.

Previous Research History

- Technology Evaluation
  - Guerino, Lucca Renato, Pedro Henrique Kuroishi, Ana Cristina Ramada Paiva, e Auri Marcelo Rizzo Vincenzi. “Static and Dynamic Comparison of Mutation Testing Tools for Python”. Em XXIII Brazilian Symposium on Software Quality. Salvador, BA: SBC, 2024. https://doi.org/10.1145/3701625.3701659.
  - Guilherme, Vitor Hugo, e Auri M. R. Vincenzi. “An initial investigation of ChatGPT unit test generation capability”. In 8th Brazilian Symposium on Systematic and Automated Software Testing -- SAST’2023, 15--24. Campo Grande, MS: ACM Press, 2023. https://doi.org/10.1145/3624032.3624035.
  - Guerino, Lucca Renato, e Auri M. R. Vincenzi. “An Experimental Study Evaluating Cost, Adequacy Effectiveness of Pynguin’s Test Sets”. In 8th Brazilian Symposium on Systematic and Automated Software Testing -- SAST’2023, 5--14. Campo Grande, MS: ACM Press, 2023. https://doi.org/10.1145/3624032.3624034.
  - Araujo, Filipe Santos, e Auri Vincenzi. “How Far Are We from Testing a Program in a Completely Automated Way, Considering the Mutation Testing Criterion at Unit Level?” In Proceedings of Brazilian Symposium on Software Quality (SBQS), 151–59. SBC, 2020. https://doi.org/10.1145/3439961.3439977.
  - Vincenzi, Auri M. R., Tiago Bachiega, Daniel G. de Oliveira, Simone R. S. de Souza, e José C. Maldonado. “The Complementary Aspect of Automatically and Manually Generated Test Case Sets”. In Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation, 1:23–30. A-TEST 2016. ACM, 2016. https://doi.org/10.1145/2994291.2994295.