muupi: An Abstract Syntax Tree Based Mutation Testing Tool for Python 2.x Programs

Author: Xin Liu
Supervisor: Dr. Alex Groce

A masters project presented to the FACULTY OF THE OSU GRADUATE SCHOOL OREGON STATE UNIVERSITY in partial fulfillment of the requirements for the degree of Master of Science in the School of Electrical Engineering and Computer Science

March 21, 2017
Declaration of Authorship

I, Xin Liu, declare that this research paper titled, “muupi: An Abstract Syntax Tree Based Mutation Testing Tool for Python 2.x Programs” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.

- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.

- Where I have consulted the published work of others, this is always clearly attributed.

- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.

- I have acknowledged all main sources of help.

- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

______________________________________________

Date:

______________________________________________
Abstract

Faculty Name

School of Electrical Engineering and Computer Science

Master of Science

*muupi*: An Abstract Syntax Tree Based Mutation Testing Tool for Python 2.x Programs

by Xin Liu
Mutation testing is one of the effective approaches measuring test adequacy of test suites. It is widely used in both academia and industry. Unfortunately, the adoption and practical use of mutation testing for Python 2.x programs face three obstacles. First, limited useful mutation operators. Existing mutation testing tools support very limited amount of mutation operators, which also limits the use of mutation testing strategy in research or practical projects. Second, no consideration for Python specific language features. Current mutation operators are only designed based on traditionally structural mutation testing. Third, coarse-grained mutation design. To measure test adequacy of a set of test suite more precisely, finer-grained mutation operator design is required.

In this paper, muupi is introduced as a new mutation testing tool for Python 2.x programs. It integrates a richer set of newly designed mutation operators, which make it possible to obtain both the capability of fine-grained mutation and the flexibility of mutation operator extension.
Acknowledgements

Firstly, I would like to express my sincere gratitude to my advisor Dr. Alex Groce for the continuous support of my Master study and related research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this paper.

Besides my advisor, I would like to thank the rest of my committee: Dr. Prasad Tadepalli and Dr. Mike Bailey, for their insightful comments and encouragement, but also for the hard questions which incent me to widen my research from various perspectives.

My sincere thanks also goes to my fellow doctoral and master students for their feedback, cooperation and of course friendship. A very special gratitude goes out to Kazuki Kaneoka. We had many wonderful discussions on our projects and shared ideas with each other.

Last but not the least, I would like to thank my parents, who have provided me through moral and emotional support in my life.

Thanks for all your encouragement.
# Contents

<table>
<thead>
<tr>
<th>Section Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration of Authorship</td>
<td>iii</td>
</tr>
<tr>
<td>Abstract</td>
<td>vi</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>vii</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Basic Concepts</td>
<td>1</td>
</tr>
<tr>
<td>1.1.1 Mutation Testing</td>
<td>1</td>
</tr>
<tr>
<td>1.1.2 Dynamic Programming Language</td>
<td>2</td>
</tr>
<tr>
<td>1.1.3 Abstract Syntax Tree</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Related Works</td>
<td>3</td>
</tr>
<tr>
<td><strong>2 Mutation Operators in Muupi</strong></td>
<td>5</td>
</tr>
<tr>
<td>2.1 Value Mutation Operators</td>
<td>5</td>
</tr>
<tr>
<td>2.1.1 Constant Replacement - CRP</td>
<td>6</td>
</tr>
<tr>
<td>2.1.2 Constant Deletion - CDL</td>
<td>6</td>
</tr>
<tr>
<td>2.2 Operator Mutation Operators</td>
<td>7</td>
</tr>
<tr>
<td>2.2.1 Arithmetic Operator Deletion - AOD</td>
<td>7</td>
</tr>
<tr>
<td>2.2.2 Arithmetic Operator Replacement - AOR</td>
<td>7</td>
</tr>
<tr>
<td>2.2.3 Assignment Operator Replacement - ASR</td>
<td>8</td>
</tr>
<tr>
<td>2.2.4 Bitwise Operator Deletion - BOD</td>
<td>9</td>
</tr>
<tr>
<td>2.2.5 Bitwise Operator Replacement - BOR</td>
<td>9</td>
</tr>
<tr>
<td>2.3 Decision/Condition Mutation Operations</td>
<td>9</td>
</tr>
</tbody>
</table>
# List of Figures

1.1 Mutation-Testing ........................................... 2  
1.2 AST-DEMO .................................................. 3  
2.1 CRP-NUM ...................................................... 6  
2.2 CRP-STR ...................................................... 6  
2.3 CDL ........................................................ 7  
2.4 AOD-UAdd ................................................... 8  
2.5 BOD ........................................................ 9  
2.6 LOD ........................................................ 10  
2.7 LOI ........................................................ 10  
2.8 LCR-1 ......................................................... 10  
2.9 LCR-2 ......................................................... 11  
2.10 BCR ........................................................ 12  
2.11 SMD-Expr ................................................... 12  
2.12 FHD ........................................................ 12  
2.13 EXD ........................................................ 13  
2.14 OIL ........................................................ 13  
2.15 OIL ........................................................ 13  
2.16 RIL ........................................................ 14  
2.17 ZIL ........................................................ 14  
2.18 ZIL ........................................................ 15  
2.19 SSID ......................................................... 15  
2.20 SEID ......................................................... 15
2.21 STID .................................................. 16
2.22 SVD .................................................. 16
2.23 SVD .................................................. 16
3.1 Muupi Flowchart ................................. 18
3.2 Muupi Flowchart ................................. 21
4.1 Piechart ............................................. 24
List of Tables

2.1  AOR replacement rules ........................................ 8
2.2  ASR replacement rules ........................................ 8
2.3  BOR replacement rules ........................................ 9
2.4  ROR replacement rules ........................................ 11
4.1  137 mutants generated from 19 mutation operators for calculator.py ........................................ 25
Chapter 1

Introduction

In this chapter, we introduce several basic concepts closely related to the topic of the paper, and present related works in both academia and open source community. Also, we propose three research questions that we want to answer to answer in our research.

1.1 Basic Concepts

1.1.1 Mutation Testing

In software development life cycle, the purpose of software testing is to guarantee the qualify of target system. Test suites with high test adequacy commonly ensure a system with high quality. Mutation testing is one of the effective approaches measuring test adequacy of test suites.

In mutation testing [1] [2], a small change is introduced into the source code under test each time, which generates a modified version of the original source code, namely a mutant. A mutant (usually) behaves differently than the original source code. A test suite is then run against both the original source code and the mutant (such as in Figure 1.1. If some test passes for the original source code, but fails for the mutant, the mutant is killed by the test
Chapter 1. Introduction

suite. The ratio of the number of killed mutants to the total number of mutants is an effective measure of test adequacy of test suite\cite{3}. The larger the ratio, the better the effectiveness of test suite in detecting real faults. Moreover, mutation testing is also widely used in academic research to evaluate the effectiveness of novel testing techniques\cite{4}.

![Figure 1.1: The basic idea of mutation testing.](image)

1.1.2 Dynamic Programming Language

As a mainstream programming language, Python is widely used in web development, data analysis, and etc by different organizations. Different from such programming languages with static typing system as Java and C#, Python is a dynamic programming language\cite{5}. With dynamic typing system, Python presents better flexibility, easier to use, and more advantages than traditional static programming languages. However, it also puts new challenges in front of traditional software testing techniques. Mutation testing is one of them. While applying mutation operators, which are designed for static programming languages, in dynamically typed program, type consistency rules cannot be statically verified. That would cause an incompetent
1.2. Related Works

According to our investigation, four mutation testing tools are available in Python 2.x language community: Pester [7], PyMuTester [8], Mutant [9], and Elcap [10]. Pester is Jester for Python programs, which defines a pattern matching expression in a configuration file to achieve program mutation. Therefore, its power is very limited. PyMuTester and Mutant are a proof of concept. PyMuTester defines two mutation operators, IfCondition...
Negation and Loop Skipping, while Mutant supports three different operators, ComparisonMutation, ModifyConstantMutation and JumpMutation. Elcap supports eight mutation operators, which are called StringMutator, NumberMutator, ArithmeticMutator, ComparisonMutator, LogicalMutator, FlowMutator, YieldMutator and BooleanMutator.

These four tools made attempts on mutation testing. However, three major issues are exposed in these tools:

- **RQ1** Existing mutation testing tools support very limited amount of mutation operators, which are insufficient to verify the effectiveness of test suites;

- **RQ2** The mutation operators used in these tools are simply designed for traditional structural mutation testing, but not get the Python specific language features involved;

- **RQ3** These mutation operators are coarse-grained. To verify the effectiveness of test suites, finer-grained mutation operators are required.

To solve these issues mentioned above and provide a much more powerful support on mutation testing for Python 2.x community, we presents twenty-two mutation operators designed for Python 2.x programs in Chapter 2. Based on some selected mutation operators used in static programming languages, we adapt them for Python 2.x programs. In addition, we discuss some special mutation operators, which are designed for some special syntax features of Python 2.x. All operators introduced in this paper can avoid generating incompetent mutants. We introduce a new tool in Chapter 3, which implements the mutation operators designed in Chapter 2. In Chapter 4 we take a simple Python 2.x program as an example to illustrate the capability of our tool. Finally, We make conclusions on our work and discuss our future plan on muupi in Chapter 5.
Chapter 2

Mutation Operators in Muupi

In this chapter, we describe the definitions of twenty-two mutation operators for Python 2.x programs in detail. These mutation operators are classified into five groups based on the types of points to be mutated. These groups of mutation operators cover value types, various operators, statements, and Python specific syntax.

To illustrate these mutation operators more clearly, we take as an example a calculator program [11], which is implemented in Python 2.x. We explain these mutation operators and their effects by comparing the program and its mutants.

2.1 Value Mutation Operators

This group of mutation operators is an attempt to change the values to detect errors in the programs. The most common strategy is to change the constants. Based on the principle of value-domain partitioning, we usually change one value to a larger one and a smaller one. Two mutation operators are defined in this group.
2.1.1 Constant Replacement - CRP

CRP operator generates mutants by replacing each numeric or string constant with a new constant of same type. For a numeric constant $c$, CRP operator adopts five replacement rules: $-1$, 0, $+1$, $c+1$, $c-1$ (such as in Figure 2.1). For a string constant $s$, CRP operator substitutes an empty string for it (such as in Figure 2.2).

![Figure 2.1: The numeric constant 5 mutates to 4 guided by 'c-1' rule](image)

```python
20 if (__name__ == '__main__');
21 Calculator.floorDiv(5, 2)
22 Calculator.add(5, 5)
```

![Figure 2.2: The string constant 'divided by zero' mutates to an empty string](image)

```python
14 except ZeroDivisionError: 14 except ZeroDivisionError:
15 print 'divided by zero' 15 print '?'
16 except: 16 except:
17 print 'unknown exception' 17 print 'unknown exception'
```

2.1.2 Constant Deletion - CDL

CDL operator handles the constants in any built-in collection data structures of Python language, such as List, Set and Tuple. Different from the semantics of CRP operator, CDL changes the number of elements in a collection data structure by removing one element per mutation. In particularly, for both List and Tuple data structures, the semantics of ordering change is introduced in CDL. When we implement CDL, in DELETION step, always the first element in the collection is removed, while in RECOVERY step, the most recently deleted element is appended to the tail of the collection. Such a design is helpful to detect the faults related to the element ordering (such as in Figure 2.3).
2.2. Operator Mutation Operators

This group of mutation operators is an attempt to change a set of built-in Python operators to detect errors in Python programs. The built-in Python operators involved are arithmetic operators, augmented assignment operators, and bitwise operators.

2.2.1 Arithmetic Operator Deletion - AOD

AOD operator changes the semantics of a program by removing unary arithmetic operators + or −. It is apparent that a numeric value or object prefixed with or without arithmetic operator − has different semantics. For operator +, it seems the semantics does not change before and after removing +. According to Python Standard Library Documentation, however, the appearance of + might introduce new semantics in some cases. As an example shown in Figure 2.4, +obj and obj show their values with different precisions. The reason is that the appearance of the operator + invokes a special function __pos__ on an operand, which can overloads the unary operator + of class instances.

2.2.2 Arithmetic Operator Replacement - AOR

AOR operator replaces unary and binary arithmetic operators guided by the replacement rules defined in Table 2.1.
>>> from decimal import Decimal
>>> obj = Decimal('3.1415926535897932384626433832795028841971')
>>> +obj != obj
True
Decimal('3.1415926535897932384626433832795028841971')

Figure 2.4: +obj and obj have different precision. Operator + rounds decimal precision back to normal precision.

Table 2.1: AOR replacement rules

<table>
<thead>
<tr>
<th>Python AST Type</th>
<th>Before Mutation</th>
<th>After Mutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>UAdd</td>
<td>+</td>
<td>−</td>
</tr>
<tr>
<td>USub</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td>Add</td>
<td>+</td>
<td>[−, *, /]</td>
</tr>
<tr>
<td>Sub</td>
<td>−</td>
<td>[+ , * , /]</td>
</tr>
<tr>
<td>Mult</td>
<td>*</td>
<td>[+ , − , /]</td>
</tr>
<tr>
<td>Div</td>
<td>/</td>
<td>[+ , − , *]</td>
</tr>
</tbody>
</table>

2.2.3 Assignment Operator Replacement - ASR

ASR operator handles five kinds of augmented assignment operators, which are + = , − = , * = , / = , // = . The replacement rules are defined in Table 2.2.

Table 2.2: ASR replacement rules

<table>
<thead>
<tr>
<th>Python AST Type</th>
<th>Before Mutation</th>
<th>After Mutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>+</td>
<td>[− , *, / , //]</td>
</tr>
<tr>
<td>Sub</td>
<td>−</td>
<td>[+ , * , / , //]</td>
</tr>
<tr>
<td>Mult</td>
<td>*</td>
<td>[+ , − , / , //]</td>
</tr>
<tr>
<td>Div</td>
<td>/</td>
<td>[+ , − , * , //]</td>
</tr>
<tr>
<td>FloorDiv</td>
<td>//</td>
<td>[+ , − , * , //]</td>
</tr>
</tbody>
</table>
2.3. Decision/Condition Mutation Operations

2.2.4 Bitwise Operator Deletion - BOD

Similar with AOD operator, BOD operator mutates a target program by removing bitwise invert operator (such as in Figure 2.5).

```python
94    def bit_negate(cls, x):
95        return ~(x)
96    def bit_negate(cls, x):
97        return x
```

**Figure 2.5:** \(\sim x\) is mutated into \(x\) by removing the invert operator.

2.2.5 Bitwise Operator Replacement - BOR

Similar with AOR operator, BOR operator mutates five kinds of binary bitwise operators: AND, OR, XOR, \(<<\) (left shift), and \(>><\) (right shift). The replacement rules are shown in Table 2.3.

<table>
<thead>
<tr>
<th>Python AST Type</th>
<th>Before Mutation</th>
<th>After Mutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>BitAnd</td>
<td>&amp;</td>
<td>[&amp;, \oplus]</td>
</tr>
<tr>
<td>BitOr</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BitXor</td>
<td>\oplus</td>
<td>[&amp;, \oplus]</td>
</tr>
<tr>
<td>LShift</td>
<td>(&lt;&lt;)</td>
<td>(\rangle\rangle)</td>
</tr>
<tr>
<td>RShift</td>
<td>(\rangle\rangle)</td>
<td>(&lt;&lt;)</td>
</tr>
</tbody>
</table>

2.3 Decision/Condition Mutation Operations

This group of mutation operators is an attempt to change decision/condition expressions to detect logical errors in Python programs. All built-in Python relational operators and logical operators (AND, OR, NOT) are get involved in the definitions of decision/condition mutation operators.
2.3.1 Logical Operator Deletion - LOD

LOD operator negates the semantics of a decision expression by removing the prefixed logical reverse operator (such as in Figure 2.6)

```
59 n = len(nums)  59 n = len(nums)
60 if [p] n > 0); ~ 60 if [p] n > 0); ~
61 return 0 61 return 0
```

**Figure 2.6:** The prefixed logical reverse operator is removed in mutant.

2.3.2 Logical Operator Insertion - LOI

Similar with LOD operator, LOI operator also negates the semantics of a decision expression. But different from LOD operator, LOI operator prefixes a decision expression with a logical reverse operator. The effect of LOI operator is shown in Figure 2.7.

```
19 def divide(cis, x, y):
20   if [p] (y == 0) and True) or True): ~
21   return None 21   return None
```

**Figure 2.7:** Reverse the decision expression of if-statement by inserting a prefixed logical reverse operator.

2.3.3 Logical Connector Replacement - LCR

LCR operator replaces logical operators and (and or) with or (and and) shown as Figure 2.8 (and Figure 2.9).

```
18 @classmethod
19 def divide(cis, x, y):
20   if [p] (y == 0) or True) or True) or True): ~
21   return None 21   return None
22   quotient = x / y 22   quotient = x / y
```

**Figure 2.8:** Boolean operator and mutates into or.
2.4. Statement Mutation Operators

2.4.1 Break Continue Replacement - BCR

BCR operator introduces new semantics into the original program by replacing `break`- (or `continue`-) statement with `continue`- (or `break`-) statement. The effect of substituting a `continue` statement for `break` statement is shown in Figure 2.10.
Chapter 2. Mutation Operators in Muupi

2.4.2 Statement Deletion - SMD

SMD operator changes the semantics of a program by removing a statement. The statement deletion is achieved by replacing a statement with `pass` statement in Python program (such as in Figure 2.11). SMD operator deals with five type Python statements: raise, assignment, augmented assignment, function calls, and expressions.

2.4.3 Finally Handler Deletion - FHD

FHD operator is designed to disable the effects of the statements in the body of a Python Try-Finally handler. To implement the purpose, a `pass` statement takes the place of all statements of the body of the Try-Finally handler (such as in Figure 2.12).
2.4. Statement Mutation Operators

2.4.4 Exception Disabling - EXD

Similar to FHD operator, EXD operator works on Python Try-Except handler. It substitutes a `pass` statement for the statements in the body of an Try-Except handler (such as in Figure 2.13).

![Figure 2.13: The print statement is replaced by pass statement after running EXD operator.](image)

2.4.5 One Iteration Loop - OIL

OIL operator works on both `for`- or `while`-loop statements and changes the semantics of a program by changing the iteration number of a loop statement. To achieve the goal, OIL operator borrows `break` statement and put it as the last statement into the body of a `for`- or `while`-loop (such as in Figure 2.14 and 2.15) so that the loop is simply executed only once.

![Figure 2.14: For-loop runs a single round.](image)

![Figure 2.15: While-loop runs a single round.](image)
2.4.6 Reverse Iteration Loop - RIL

RIL operator introduces new semantics by reversing the original access ordering of the elements in an iterator. In Python, `for`-loop statement is defined as: `for item in iterator`. Therefore, by wrapping `iterator` with `reversed()` function, the goal of visiting elements in reversed ordering is achieved. Figure 2.16 shows the result of the replacement.

<table>
<thead>
<tr>
<th>64</th>
<th>for i in <code>range(n)</code>:</th>
<th>64</th>
<th>for i in <code>reversed(range(n))</code>:</th>
</tr>
</thead>
<tbody>
<tr>
<td>65</td>
<td>flag = (num[i] == 0)</td>
<td>65</td>
<td>flag = (num[i] == 0)</td>
</tr>
<tr>
<td>66</td>
<td>if flag:</td>
<td>66</td>
<td>if flag:</td>
</tr>
<tr>
<td>67</td>
<td>continue</td>
<td>67</td>
<td>continue</td>
</tr>
<tr>
<td>68</td>
<td>res = cls.add(res, num[i])</td>
<td>68</td>
<td>res = cls.add(res, num[i])</td>
</tr>
</tbody>
</table>

**Figure 2.16:** Reverse a `for`-loop.

2.4.7 Zero Iteration Loop - ZIL

Similar with OIL, ZIL operator also manipulates `for-` and `while-` loop. Different from OIL, the semantics of ZIL operator is bypassing all statements in the body of a loop structure. To achieve the effects, a `break` statement is inserted into the head of the loop body. When the running logic goes into the loop body, `break` statement forces the running logic to jump out of the loop body. Figure 2.17 and Figure 2.18 show the effects of ZIL operator on `for-` and `while-` loops, respectively.

<table>
<thead>
<tr>
<th>63</th>
<th>for i in <code>range(n)</code>:</th>
<th>63</th>
<th>for i in <code>range(n)</code>:</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>flag = (num[i] == 0)</td>
<td>64</td>
<td>flag = (num[i] == 0)</td>
</tr>
<tr>
<td>65</td>
<td>if flag:</td>
<td>65</td>
<td>if flag:</td>
</tr>
<tr>
<td>66</td>
<td>continue</td>
<td>66</td>
<td>continue</td>
</tr>
<tr>
<td>67</td>
<td>res = cls.add(res, num[i])</td>
<td>67</td>
<td>res = cls.add(res, num[i])</td>
</tr>
</tbody>
</table>

**Figure 2.17:** Ignore a `for`-loop.
2.5 Python Specific Mutation Operators

For some Python specific features, we design four kinds of mutation operators: **Slice Start Index Deletion**, **Slice End Index Deletion**, **Slice Step Index Deletion**, and **Self Variable Deletion**. As their names show, the first three operators are designed for Python Slice syntax, while the last one is for `self` variable.

2.5.1 Slice Start Index Deletion - SSID

In Python Slice syntax there are three arguments, `startIndex`, `stopIndex`, and `step`. SSID operator is designed to remove `startIndex` argument of a Python Slice (such as in Figure 2.19).

2.5.2 Slice End Index Deletion - SEID

Similar with SSID operator, SEID operator removes `stopIndex` argument of a Python Slice as shown in Figure 2.20.
2.5.3 Slice Step Index Deletion - STID

STID operator removes \textit{step} argument of a Python \textit{Slice} as shown in Figure 2.21.

```python
if len(nums) > 0:
    for num in nums[start:][start:end]:
        res += num
```

\textbf{Figure 2.21:} Remove \textit{step} argument of Python \textit{Slice}.

2.5.4 Self Variable Deletion - SVD

In Python, \textit{self} variable is commonly used in a class definition to reference instance variables or functions defined in the class. SVD operator is designed to remove \textit{self} variable. Figure 2.22 and 2.23 show different cases in which SVD are put into use.

```python
def __init__(self):
    name = 'name_value'
    text = self.name
    self.name = 'name_value'
    text = name
```

\textbf{Figure 2.22:} Remove \textit{self} variable.

```python
def __init__(self):
    self.name = 'name_value'
    text = self.name
```

\textbf{Figure 2.23:} Remove \textit{self} variable.
Chapter 3

Muupi Architecture and Workflow

In this chapter, we present a mutation testing tool \textit{muupi}, which implements all mutation operators discussed in Chapter 2. First, we describe the architecture of muupi in details, which consists of five major components. And then, we illustrate the standard workflow of muupi.

3.1 Muupi Architecture

Muupi consists of five major components, which are \textit{Command Parser}, \textit{Operator Generator}, \textit{Mutant Generator}, \textit{Test Runner}, and \textit{Utilities}. Among them, \textit{Operator Generator} and \textit{Mutant Generator} form the core of muupi. Figure 3.1 shows the architecture of muupi.

3.1.1 Command Parser

As the entry point of muupi, Command Parser takes as input a command line. The format of a simple command line for mutant generation is shown
Figure 3.1: Muupi architecture.

as below.

```
muupi -m <full name of target module> -p <target-file-path> -o <operator-list>
```

Command Parser parses the command line and extracts different information based on different options, such as the full name and file path of a program to mutate. Such information is required for consecutive steps, and acts as inputs of other components.

### 3.1.2 Operator Generator

The core of muupi consists of two major components. Operator Generator is one of them. Taking as input the mutation operator information from Command Parser, Operator Generator is responsible for assembling mutation operators. Mutation operators discussed in Chapter 2 are defined in this component. By iterating the operator list, Operator Generator selects
corresponding pre-defined mutation operators, assembles the type of possibly interested abstract syntax tree nodes and mutation operator into a tuple, then put all tuples into a collection which will be consumed by another muupi component, Mutant Generator, later.

3.1.3 Utilities

Utilities component works as a helper component. The tasks of this component cover loading modules of both program under test and unit test suites, printing abstract syntax tree, comparing mutants with original program, and outputting mutant files. Both program-under-test module and test-suite module returned from Utilities component will be used by Mutant Generator and Test Runner, respectively.

3.1.4 Mutant Generator

Mutant Generator is another core component in muupi. It is responsible for generating mutants. Mutant Generator takes as inputs both a collection of mutation operators, which is from Operator Generator, and the module of program under test, which is from Utilities component. The process of mutant generation contains four major steps:

1. Traverse the abstract syntax tree of the program-under-test module to search a node, of which the type matches with some element in the collection of mutation operators;

2. Mutate the node based on the correlated mutation operator’s definition;

3. Put the mutant module generated in last step into a mutant queue;
4. Repeat first steps until there is no node matching any element in the mutation operator collection.

The final result returned by Mutant Generator component is the queue of mutant modules. These mutants will be used to verify the effectiveness of unit test suite module in Test Runner.

3.1.5 Test Runner

When both mutants and test suite module are well prepared, Test Runner takes them as inputs and run test suite module against each mutant. In current version of muupi, Test Runner only supports PyUnit framework. Therefore, it is required that test suite module should be created based on this unit test framework. In the future, muupi will support more popular Python unit test frameworks.

3.2 Mutation Workflow

In this section, we illustrate the standard workflow of muupi. The whole process consists of four phases: parsing commands, loading necessary modules, generating mutants, and running mutation testing. Figure 3.2 visualizes an entire process.

- **Phase I**: Muupi parses the command arguments for three kinds of information: mutation operators to run, the file paths of both program under test and test suites.

- **Phase II**: Based on the information from the first phase, then, the mutation operator generator of muupi generates a set of mutation operators,
3.2. Mutation Workflow

and the utilities component of muupi loads the modules of program under test and test suites.

- **Phase III**: Mutant generator takes as argument mutation operators and program under test to parse and manipulate the abstract syntax tree of the program under test using ast library [13]. The output of the third phase is the mutants of the program under test.

- **Phase IV**: Test runner component runs test suites from the second phase against the mutants generated in third phase. As a result, the ratio of the number of killed mutants to all mutants will be computed, also named mutation factor, implies the effectiveness of muupi to detect faults.
Chapter 4

Example: Calculator

In this chapter, we use a simple calculator program [11] to demonstrate how to use muupi to generate mutants. This Python program consists of two classes: Calculator and ScientificCalculator. In Calculator class, eighteen class functions are defined, while four instance functions in ScientificCalculator class.

We use following command to generate mutants for the example program:

```
muupi -m sample.calculator -p ./sample/ -o ALL
```

- `-m` option specifies the module’s full name of program under test. In the command above, the full name is sample.calculator.

- `-p` option specifies the file path of program under test.

- `-o` option specifies mutation operators to use. In this example, the argument, `ALL`, is default value, which means all pre-defined mutation operators to be used in mutation testing.

In current version of muupi there are additional five options defined as below.

- `-t` option specifies the full name of test suite module.

- `-T` option specifies the path of source code of test suite module.
• -g option specifies a generator. This option is designed for integration with TSTL project. The argument could be one of generators designed in TSTL project.

• -P option specifies the path of a generator.

• -l option lists all available generators.

With the command mentioned above, muupi produces 137 mutants of program under test based on 19 pre-defined mutation operators. Figure 4.1 illustrates the distribution of the mutants from each mutation operator.

![Figure 4.1: 137 mutants for calculator.py](image)

Table 4.1 shows more details of the results, including the ast type of nodes mutated under each operator, the number of mutants generated by each operator, and the percentage of the number of mutants from each operator over the total amount.
### Table 4.1: 137 mutants generated from 19 mutation operators for calculator.py

<table>
<thead>
<tr>
<th>Mutation Operator</th>
<th>AOD</th>
<th>AOR</th>
<th>ASR</th>
<th>BCR</th>
</tr>
</thead>
<tbody>
<tr>
<td>AST Node Type</td>
<td>UnaryOp</td>
<td>BinOp</td>
<td>AugAssign</td>
<td>Continue</td>
</tr>
<tr>
<td>Mutants</td>
<td>1</td>
<td>66</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>Percentage (%)</td>
<td>1</td>
<td>48</td>
<td>12</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mutation Operator</th>
<th>BOD</th>
<th>BOR</th>
<th>COR</th>
<th>LCR</th>
<th>LOD</th>
</tr>
</thead>
<tbody>
<tr>
<td>AST Node Type</td>
<td>UnaryOp</td>
<td>BitAnd</td>
<td>Eq</td>
<td>BoolOp</td>
<td>UnaryOp</td>
</tr>
<tr>
<td></td>
<td>BitOr</td>
<td>Bitxor</td>
<td>LShift</td>
<td>RShift</td>
<td></td>
</tr>
<tr>
<td>Mutants</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Percentage (%)</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mutation Operator</th>
<th>FHD</th>
<th>EXS</th>
<th>LOI</th>
<th>ZIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>AST Node Type</td>
<td>TryFinally</td>
<td>ExceptHandler</td>
<td>BoolOp</td>
<td>FOR WHILE</td>
</tr>
<tr>
<td></td>
<td>UnaryOp</td>
<td></td>
<td>BoolOp</td>
<td></td>
</tr>
<tr>
<td>Mutants</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Percentage (%)</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mutation Operator</th>
<th>OIL</th>
<th>RIL</th>
<th>SEID</th>
<th>SMD</th>
<th>SSID</th>
<th>STID</th>
</tr>
</thead>
<tbody>
<tr>
<td>AST Node Type</td>
<td>For</td>
<td>For</td>
<td>Slice</td>
<td>Assign</td>
<td>AugAssign</td>
<td>Slice</td>
</tr>
<tr>
<td></td>
<td>While</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mutants</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>18</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Percentage (%)</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>13</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Chapter 5

Conclusion and Future Work

In this paper, we presented twenty-two mutation operators and a new mutation testing tool muupi for Python 2.x programs. We explained the architecture and demonstrated the standard workflow of muupi.

With the rapid development of software industry, Python language also evolves continuously. To answer the requests and demands from independent developers and organizations, more novel and powerful features are supported in Python, such as functional programming. Such changes also drive muupi to enrich our current mutation operators. In addition, according to official Python 2.x documentation, there are still a plenty of Python specific language features, such as decorator, super calling, and etc. We will gradually design new mutation operators to cover all these features.

Muupi is a proof-of-proof tool. In current stage, we focus on the correctness and effectiveness of mutation operators we designed. According to practical use, one issue we found in current version of muupi is the running performance. To simplify the development, we adopted single process/thread design on the mutant generation engine, which means that muupi generates mutant sequentially, namely a mutant one time. Such a running logic causes the performance issue. In the future, we plan to improve current design of mutant generation engine by (1) detecting the interest points to mutate, (2)
making multiple copies of the abstract syntax tree of the original program according to the result of first step, (3) creating one thread for each copy of the abstract syntax tree and generating mutants concurrently.

In current implementation of muupi, mutation operators are tightly coupled with other components. It limits third-party developers or researchers to customize their own mutation operators. In our plan, we will design a configuration file to maintain a mutation operator list, and make use of reflection mechanism of Python language to achieve dynamic load of mutation operators. Such an improvement in current design will decouple the component of mutation operators from muupi and bring about better flexibility.
Bibliography


