IMPERIAL COLLEGE LONDON
DEPARTMENT OF COMPUTING

SPLAT

Simple Python Lazy Automated Tester

Final Year Individual Project

MEng Project Report

Lee Wei Yeong

lwy08@doc.ic.ac.uk

Supervisor: Prof. Susan Eisenbach

Second Marker: Dr. Tristan Allwood

June 2012

Abstract

Writing unit test suites has become an indispensable part of the software engineering process. These test cases, collectively, aim to provide confidence to the client in the final delivery of the shipped product.

Unfortunately, it can be rather tedious to write comprehensive unit tests by hand, especially if the work contributing to this cost is hidden from the customer. Therefore, this project attempts to automate this software testing process, in order to make software development more efficient overall.

Traditionally, the core of software development often revolves around programming in statically typed languages like C/C++, Objective-C or Java/Scala. However, in recent times, it is increasingly common to find their dynamic counterparts, such as Javascript, Python or Ruby, employed in any software stack, and not only just for quick scripting tasks.

This then necessitates the exploration of automated testing tools for dynamic languages, which is the motivation for this project, that is, to automatically generate unit tests using Python, and also address the present inadequate availability of such tools.

In addition, several techniques for accomplishing this have been suggested in papers, although many of which, including *lazy instantiation*, were applied extensively to static languages, like Haskell. Hence, this project intends to research into how a hybrid of these existing ideas could possibly inspire and be adapted to solve the automated unit test generation problem for this domain, in such a way that consistently high code coverage can be achieved across a variety of test programs.

Acknowledgements

First of all, I would like to thank Professor Susan Eisenbach for supervising my project, and giving me invaluable advice and guidance throughout my work.

Secondly, I thank Dr. Tristan Allwood for his support, ideas, encouragement, and the useful discussions that we had regarding the direction of my project.

Also, many thanks to Chong-U Lim for his insight during our conversations.

Finally, I would like to thank my family for their continued full support during the course of my university studies.

Contents

Chapter 1

Introduction

This is the Report for my Final Year Project, *Simple Python Lazy Automated Tester*, hence the acronym 'SPLAT' appearing on the cover page, which represents the name of the tool created from this research.

1.1 Motivation

Professional software engineers often write tests while developing code, especially for large complex codebases. These tests are highly beneficial for generating confidence in a bug-free solution delivery.

However, writing tests is not always easy to get right, and can be quite costly. It is reported that testing code is responsible for *approximately half* the total cost of software development [Edv99][HK08][KHC+05].

Furthermore, this task becomes gradually more time-consuming as software grows in terms of complexity. Given similar resource constraints, it can become increasingly difficult to consistently achieve high test code coverage.

Moreover, a significant proportion of overall development time is spent writing test code not eventually included in production. Hence this work, though critical to assuring the quality of software[Har00], is ultimately invisible to the client, as far as billing and accountability is concerned.

This has led to a large body of work on automatically generating test suites, particularly notable within the imperative programming community [ACE11].

Even then, the present need for manual testing indicates that there still remains much scope for improvement in this area. A recent example supporting this claim is Google handing out a record $26k in bug bounties for security researchers reporting Chrome vulnerabilities [Kei11].

Therefore, this raises the question of whether full automatic discovery [Ber07] for all these bugs could be possible, in order to eliminate this cost, let alone any bug exploits.

1.2 Automated software testing for dynamic languages

Whilst research in this field is typically devoted to statically typed programming languages such as C/C++, Objective-C or Java/Scala, relatively less emphasis is placed on their dynamic counterparts like Javascript, Python or Ruby.

One such paper implements the search-based software testing (SBST) technique, to automatically generate test scenarios for Ruby code, using genetic algorithms [MFT11]. There is neither any equivalent tool targeting Python instead, nor any chance of porting these tests for Python programs.

This observation is made in contrast to the rapid growth in popularity of dynamic languages in recent years, especially Python. Python was named the 'The Importance Of Being Earnest' (TIOBE) Programming Language of the Year, both in 2007 and 2010 [BV11].

In this paper [MFT11], the authors claimed success in achieving consistent and significantly high code coverage over a preselected set of test inputs with their tool, when compared against the naïve random test case generator. Would it be possible to better this using a suitable adaptation of existing techniques, and/or to maintain this coverage across a more extensive range of programs?

As Python, like Ruby, is a reflective, dynamically typed language, it would seem logical to adopt a similar approach in solving this problem, specifically by generating test scenarios via *runtime code analysis* [MFT11].

1.3 Python

The Python programming language contains a variety of interesting features which encourage rapid experimentation with automatic testing techniques.

This is primarily because Python is an open source, general purpose, multi-paradigm, cross-platform compatible, dynamically typed language, offering duck typing, and in active development and support. It also provides *excellent builtin introspection and reflection capabilities*, to inspect and manipulate code at runtime.

At the heart of the language design philosophy [Pet04], there should be one – and preferably only one – obvious way to do things. The importance of readability promotes a *clean, concise and elegant syntax*, which makes demonstrating 'proof of concept' code easy.

For instance, the following Python code snippet certainly reads more fluently than its C# counterpart:

Sample C# code

```
if ("hello".indexOf("e") >= 0)
{
   return true;
}
```

Equivalent Python code

```
if 'e' in 'hello':
   return True
```

Python features a fundamental testing infrastructure toolset based on unittest, doctest and py.test. However, there is limited availability of testing support tools built on top of those. Many of these either target outdated versions of Python, or are discontinued. There are a few candidate tools for automated testing, for instance, pythoscope and pytestsgenerator, which generate tests by performing static code analysis. However there are no tools which perform dynamic test case generation.

1.4 Project contributions

Within the context given above, this project makes the following key contributions:

- A discussion of the possible ways considered for automated software testing, focusing on test data generation by using information gathered at runtime
- A motivating example describing automated lazy testing in Python
- Implementation as a Python module, to automatically generate consistently high coverage test suites, primarily evaluated against Python libraries like python-graph, and other Python module implementations of famous algorithms
- Investigating effectiveness of the *lazy instantiation* testing technique, as illustrated by IRULAN in Haskell [ACE11], for Python
- Further advance the work in the field of automated software testing, especially for dynamic languages

To this end, we take advantage of the main features of the core Python language, ie. strong introspection and reflective capabilities, combined together with its extensive tool support from the Python Package Index (PyPI) repository.

The concepts discussed in this paper are concretely demonstrated in a tool called SPLAT, a high coverage test suite generator for Python modules, written in Python, but portable to target other languages. This tool has been successfully applied to some of the most popular frameworks, achieving the initial objective of consistently high test code coverage, comparable to those manual unit tests written by hand, and even potentially discovering

several bugs in the process as well. The tool has also been extended to support regression testing, where reports on a sample of case studies are included in this report.

1.5 Report organisation

The rest of this paper is structured as follows. Firstly, relevant background material is reviewed in Chapter 2. Thereafter, the various algorithms and techniques used to automatically generate tests are formally introduced in Chapter 3. These ideas presented here are then implemented in the tool SPLAT, constituting the subject of Chapter 4. This is accompanied by a detailed description of its software design architecture, together with several worked examples, for clarification purposes. A summary of the extent of success of the project is discussed in Chapter 5, before some final conclusions are drawn, and suggestions are given to possible future work in Chapter 6.

Chapter 2

Background

2.1 Introduction

This part of the paper is intended to provide an overview and discussion of the relevant literature to this project, forming the basis for the reader to follow on later content. Firstly, Sections 2.2 to 2.4 review the general field of automated software testing. Section 2.5 deals with the papers that inspired and influenced this project. Relevant characteristics of dynamically typed programming languages, ie. those related to Python, are then discussed in Section 2.6. Finally, the associated technical difficulties are highlighted in Section 2.7.

2.2 Basic concepts

Software testing delivers quality assurance in the product to the customer. It verifies the absence of software bugs, as far as the verification that implementation complies with original client specification goes. The following terms commonly found in *automated test data generation research* are defined below.

2.2.1 Definition of terms

General testing

- *Test data*: data specifically identified for use in testing the software
- *Test case*: set of conditions under which the correct behaviour of an application is determined
- *Test suite*: a collection of test cases
- *Test automation*: use of software to control test execution, comparison of actual and expected results, setting up of test preconditions, and other test control and reporting functions
- *Test coverage*: measurement of extent to which software has been exercised by tests

Graph theory

- *Path*: sequence of nodes and edges. If a path begins from the entry node, and terminates at the exit node, then it is a *complete* path.

- *Branch predicate*: condition in a node leading to either a true or false path
- *Path predicate*: collection of branch predicates which are required to be true, in order to traverse the path
- *Feasible path*: path with valid input for execution
- *Infeasible path*: path with no valid input for execution
- *Constraint*: an expression of conditions imposed on variables to satisfy

2.2.2 Validation criteria

Software is usually checked for product quality, in terms of constraints placed on attributes like speed, efficiency, reliability, safety and scalability. This is an area in which automation excels over manual testing, thus it becomes a key focus of this paper.

2.3 Implementations

This section provides a general outline of the various types of existing ways to automatically generate unit tests.

2.3.1 Static method

This method generate tests without executing the software, generally via symbolic execution to solve for constraints on input variables. The static approach to test case generation derives a test case that will traverse a chosen path, as it satisfies that path predicate.

This paper [TK] mentions the problem of infeasible path detection in case of loops with a variable number of iterations, and claims that it is weaker than the dynamic method at gathering type information, hence it is useful only for straight forward code. The main difficulty in this technique is solving non-linear constraints.

2.3.2 Dynamic method

Instead of using variable substitution, the software under test is executed (frequently using more than one pass) with some selected input. Code instrumentation will monitor and report if program execution follows an intended path. Search methods can be varied to pursue more "interesting" paths. Variables are then updated each time before the next execution, until this goal is achieved, at which point, the associated test case is generated.

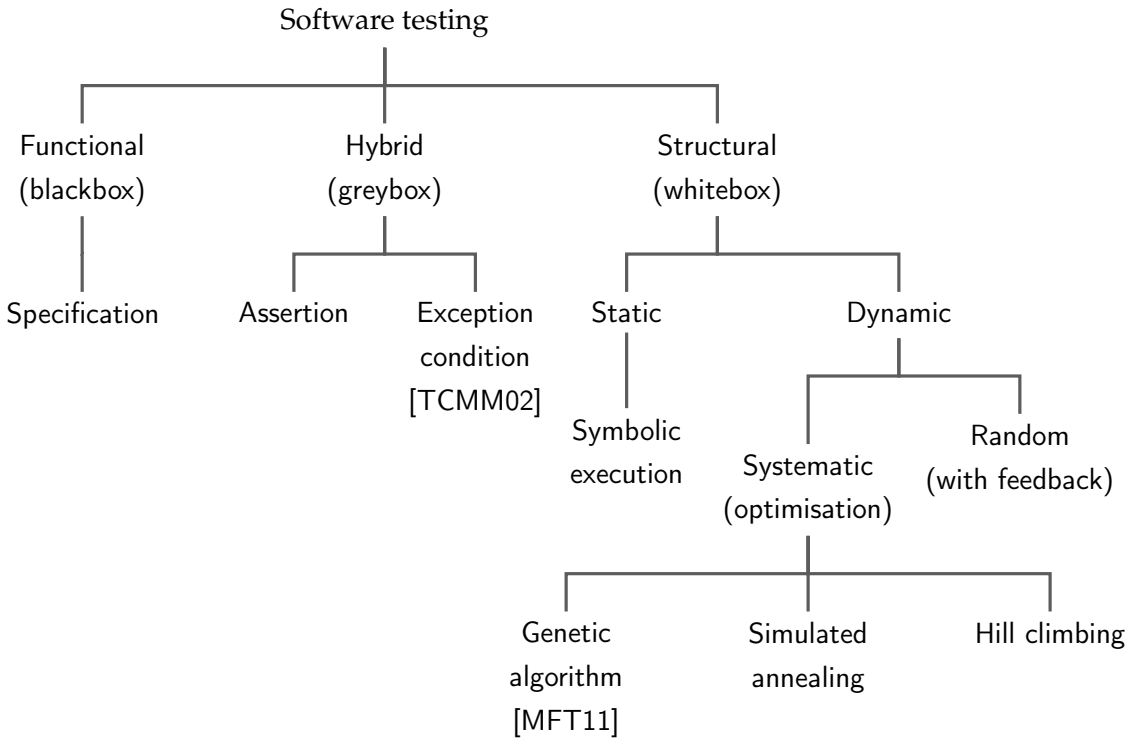
According to the paper [TK], the authors note that research has attempted to combine symbolic reasoning with dynamic execution, or modifying inputs by heuristic function minimisation techniques. However, problems such as scalability and non-termination of infeasible paths arise from this approach.

2.3.3 Hybrid

Recent research on test data generation combines both these methods to try to mitigate these disadvantages, in order to obtain high coverage of feasible execution paths. It does not traditionally enumerate through the entire program input space, but instead solves partial path predicates to generate test cases. Therefore, this paper intends to further explore this area, to improve the efficiency in automated test data generation.

2.4 Overview

The following diagram [McM04] outlines the different kinds of software testing:



Are you going to talk about these different types of testing? Where do you fit into this diagram? Another interesting paper is the idea of testability transformations [KHC+05], where source code is refactored to facilitate software testing, like unrolling loops for example.

2.5 Current state of the art

These survey papers [McM04] [HK08] [TK] provide a high level overview of software testing techniques.

The closest automated unit test generator in a dynamic language is RuTeG [MFT11], which uses evolutionary algorithms on Ruby source code to automatically create unit tests in Ruby. Apart from the description in the paper, there is no source code available

online to try it out.

With respect to automated test generation in Python, there is only minimal work done. The most recent tools, such as pythoscope v0.4.3 (Feb 2010) and pytestsgenerator v0.2 (Feb 2009), perform static analysis on Python source code, as opposed to dynamic testing on Python bytecode.

2.5.1 Pythoscope

Pythoscope is an open source unit test generator for Python code. It is able to create additional test cases for user-specified points of entry, and also appends new test cases, thus allowing the user to safely modify and extend generated test cases. An example usage is provided below to illustrate this application in action:

Not test, but stub! How well does it work? How is it evaluated?

2.5.2 Python Tests Generator

This automated test case generator creates unit tests for Python modules. The authors Vijakumar and Karthikeyan developed this tool on 32-bit Linux. Its purpose is to simplify usage of the existing PyUnit framework, and generate logical test cases for classes and methods. It also consists of a WxPython GUI, in addition to the CLI. The application is packaged for distribution using the distutils module.

According to accompanying documentation, this tool is intended to accomplish the following objectives:

- Read a specified python module
- List the Classes, Functions and Properties of that module (for the user's selection)
- Drill down the Classes for methods and properties
- Generate basic set of test cases for each class or method selected

The predetermined logic for the test cases to be generated include:

- Number of arguments
- Valid arguments
- Invalid arguments
- Custom logic

An example demonstration of the software follows:

Listing 2.1: Sample input Python program

```
class TestClass:
        def __init__(self):
                """
                Test Class without arguments
                """

        def TestClassMethod(self):
                """
                TestClass Method without args
                """
                print "TestClassMethod"
```

Listing 2.2: Generated unit test suite

```
import unittest
import sys
import sample

class PyUnitframework(unittest.TestCase):
        '''Test Cases generated for sample module'''
# format:
# test_<Test_Number>_<Entity_Name>[<Arg_Status>]<Predicted_Status>_<Comment>
        def test_1_TestClassMethod_WithoutArgs_Pass_With_Only_Valid_Arguments(
            self):
                sample.TestClass().TestClassMethod()
        def test_2_TestClassMethod_WithArgs_Fail_With_Arg(self):
                self.assertRaises(Exception,sample.TestClass().TestClassMethod,
                    'TestStr0')

if __name__=="__main__":
        testlist=unittest.TestSuite()
        testlist.addTest(unittest.makeSuite(PyUnitframework))
        result = unittest.TextTestRunner(verbosity=2).run(testlist)
        if not result.wasSuccessful():
                sys.exit(1)
        sys.exit(0)
```

Listing 2.3: Results of test execution

```
test_1_TestClassMethod_WithoutArgs_Pass_With_Only_Valid_Arguments (__main__.
    PyUnitframework) ... TestClassMethod
ok
test_2_TestClassMethod_WithArgs_Fail_With_Arg (__main__.PyUnitframework) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.003s

OK
```

2.5.3 IRULAN

As for implementing the idea of lazy instantiation: No previous explanation of term, IRULAN [ACE11] is the canonical tool written to demonstrate this concept in Haskell.

IRULAN has four key objectives to achieve, which this project intends to do similarly, where possible:

1. Automatic inference of constructors and functions to generate test data
2. Needed narrowing / lazy instantiation
3. Inspection of elements inside returned data structures
4. Efficient handling polymorphism by (lazily) instantiating all possible instances

How does your tool relate?

A sample execution to discover errors in the following code snippet:

```
module IntTreeExample where
data IntTree
        = Leaf
        | Branch IntTree Int IntTree
insert :: Int -> IntTree -> IntTree
insert n Leaf = Branch Leaf n Leaf
insert n (Branch left x right)
        | n < x = Branch (insert n left) x right
        | n > x = Branch left x (insert n right)
```

produces the following output:

```
$ irulan --ints='[0,1]' --enable-case-statements -a --
   maximumRuntime=1 source IntTreeExample
...
insert 1 (Branch ? 1 ?1) ==> !
```

```
IntTreeExample.hs:(8,0)-(11,41): Non-exhaustive patterns in
   function insert
insert 0 (Branch ? 0 ?1) ==> !
IntTreeExample.hs:(8,0)-(11,41): Non-exhaustive patterns in
   function insert
case insert 0 (Branch (Branch ? 0 ?1) 1 ?2) of
Branch x _ _ -> x ==> !
IntTreeExample.hs:(8,0)-(11,41): Non-exhaustive patterns in
   function insert
case case insert 0 (Branch (Branch (Branch ? 0 ?1) 1 ?2) 1 ?3)
   of
Branch x _ _ -> x of
Branch x _ _ -> x ==> !
IntTreeExample.hs:(8,0)-(11,41): Non-exhaustive patterns in
   function insert
...
```

2.6 Features of Python

to be completed

2.7 Challenges

There are several facets of complexity to this problem, which this work hopes to tackle.

2.7.1 Function argument instantiation

Function arguments include primitives, data structures like lists, maps or trees, and objectsHow are these (treated) different?. Depending on available resources, scope might be restricted to only certain numeric types or specific functions, eg. methods invoking `strcmp()` - ambiguous.

This is especially applicable not only to class constructors when creating appropriate objects, but also automatically generating initial user input.

It is vital to ensure that an exhaustive search for what? is not performed, because there would quickly be an exponential blow up, especially in functions with multiple input arguments, as well as being inefficient, due to many meaningless test cases.

There are several possible ways to conduct the search for such corner cases. Previous algorithms range from naïve systematic enumeration of all possible values to variants of random testing.

Therefore, the task here is to come up with a more efficient way of prioritising pathological boundary parameter value generation, under real time and space constraints. Some leading intuition follows.

Lazy instantiation

It might be reasonable to begin with “lazy instantiation"[ACE11], where dummy nullified objects are passed in initially, and test data are only generated for return values when the methods on them are actually invoked. This supposes multiple runs through the same code block, and using feedback from previous iteration to direct future execution. A prototype of this is available together with the original project proposal.

Runtime in-memory manipulation

It is also envisioned that the dynamic language features of Python be exploited in order to rapidly generate useful test data. One idea is to manipulate and observe the behaviour of code blocks in memory at runtime, by monkeypatching or hotswapping code under test (CUT) for stubs, but with a hook to log incoming parameters during a sample execution, in order to determine their initial starting range & types.

On a related note, a cross-cutting concern such as logging may be implemented using the concept of Aspect Oriented Programming (AOP), with tools like pytilities, Aspyct, aspects.py or PythonDecoratorLibrary.

Random testing

Apart from random testing with feedback RANDOOP [PLEB07], and preferring configuration diversity over a single optimal test configuration in Swarm Testing [Reg11], another suggestion is to inspect stack frames of previous executions to grasp a better initial starting point for generating parameters.

2.7.2 Optimising search space coverage

The suggestion to parallelise the search space for interesting values over the entire range of integers for example, is to use the General Purpose Graphics Processing Unit (GPGPU) toolkit like Nvidia’s CUDA, HADOOP, or Node.js - not parallel, of which its feasibility still needs to be determined.

2.7.3 Testing a dynamically typed language

Much of the body of work in the software testing community concerns testing against static languages, rather than dynamic languages, or even Python in particular.

Dynamically typed languages are characterised by values having types, but not variables, hence a variable can refer to a value of any type, which can possibly cause test data

generation to become more complicated. Python therefore heavily employs duck typing, to determine an object's type by inspection of its method or attribute signatures.

Tools arising from research efforts into testing for static languages lacks adequate support for code written in dynamic languages, including typical features such as `eval()`, closure, continuations, functional programming constructs, and macros, thus this paper aims to look into this further, in the context of Python.

2.7.4 Non-terminating program executions

Another difficulty associated with this problem domain is detecting infinite executions when generating test code. This can be most commonly attributed to (the error of) infinite loops present, which may even be nested. It is impossible to detect all kinds of loops fully automatically, but many such can [TK]. An immediate solution is to implement timeouts, with custom duration according to CUT. Early detection so as to improve efficiency is difficult.

2.7.5 Early detection of path infeasibility

The paper [TK] claims one of the most time consuming task of automatic test data generation is the detection of infeasible path after execution of many statements. Hence, backtracking on path predicates [Kor90], satisfiability of a set of symbolic constraints [ZW01], selectively exploring a subset of "best" paths [PM87] are some of the past attempts at solving this issue. This is a major problem of test data generation based on actual value, incurring both costly and unnecessary computation.

2.7.6 Improving code coverage

Achieving consistently high code coverage over a wide range of programs (not to mention running within reasonable time and space) via generated test cases ultimately defines the extent of success of this project. This allows for effective fault detection, which may be of different types. An alternative measurement of code coverage improvement involves identifying error prone regions of code where more rigorous testing would prove beneficial [Nta88] [Inc87]. There already exists other empirical studies for code coverage in different test data generation algorithms documented, providing some competitive standards to match up to [HK08] [RU99] [LMH09].

2.8 Summary

In this chapter, the relevant background literature and theory to understand this project has been explained in sufficient detail. The next chapter presents the theoretical contribution of this project to the field, including the problem specification, approach taken,

and algorithms and techniques used.

Chapter 3

Contributions

<Missing introductory paragraph>

3.1 Specification

The strategy in this project emphasises dynamic test data generation, where intermediate runtime data is gathered, represented in some suitable form, and used to guide subsequent testing iterations.

Our strategy assumes that the CUT is not obsfucated, so reverse engineering and code reconstruction lies out of the scope of this investigation.

Moreover, we are dealing only with Object Oriented (OO) style Python programs, ie. those involving classes and objects.

As an simplifying assumption, the CUT here is limited to contain at most one program entry point. If the CUT is found not to contain a main entry point, then tests are generated for the individual classes and functions separately, as discovered by the runtime engine.

In addition, test cases should be generated without requiring any user input.

Example

Given a basic standard complete implementation of class LinkedList, with a sample of prototype of method signatures detailed below:

```
def add(self, isAdd):
def size(self):
...
```

This project aims to then create the following test suite to validate its behaviour:

```
Test #1:
        l = LinkedList()
        assert (l.size == 0)
Test #2:
        l = LinkedList()
        l.add(true)
        assert (l.size == 1)
Test #3:
        l = LinkedList()
```

```
        l.add(true)
        i = l.iterator
        assert (i.hasNext)
```

These generated test cases within the suite should ideally be as close to natural language as possible, as a project extension.

3.2 Approach

Some notable aspects of the proposed solution:

i. developed incrementally

ii. bytecode inspection

iii. runtime construction of control flow graph (CFG)

iv. runtime code manipulation

v. introspection & reflection

vi. Python 2.7.x module

vii. target Mac OS X / Ubuntu Linux

A preliminary sample of the bytecode investigation for simple language constructs can be found in Appendix **??**.

The key deliverable from this project will be unit test suites, in terms of a language-neutral Domain Specific Language (DSL), or JSON, consisting of various assertions, capture expressions, and value assignments. This affords flexibility in later system extensions to target other dynamic programming languages.

An API may be exposed if there are reusable components, eg. algorithms, developed in this tool. It is also planned to provide visualisation of this process, in the form of a GUI frontend, powered by wxPython/GTK.

The resulting end product can be applied to regression testing as well, to report changes in behaviour across different versions, as software evolves over time.

Available tools

Detailed below as follows are the selection of Python resources for various purposes:

i. *parsing modules* - ANTLR, PyParsing, Ply (Python Lex-Yacc), Spark, parcon, RP, LEPL,

ii. *measuring code coverage* - coverage.py, figleaf, trace2html

iii. *unit testing* - (X)PyUnit, TestOOB, unittest, nose, py.test, peckcheck

iv. *mutation testing* - Pester

v. *bytecode inspection & manipulation* - Decompyle (2.3), UnPyc (2.5,2.6), pyREtic (in memory RE)

vi. *Python DSL* - Konira

vii. *syntax highlighting* - Pygments

viii. *CUDA Python bindings*

ix. *Python language reference* - Grammar

x. *documentation* - epydoc

xi. *Alternative implementations* - PyPy, Unladen Swallow

xii. *fuzzing tools?*

xiii. *supporting tools* - virtualenv/pip

3.3 Algorithm

3.4 Summary

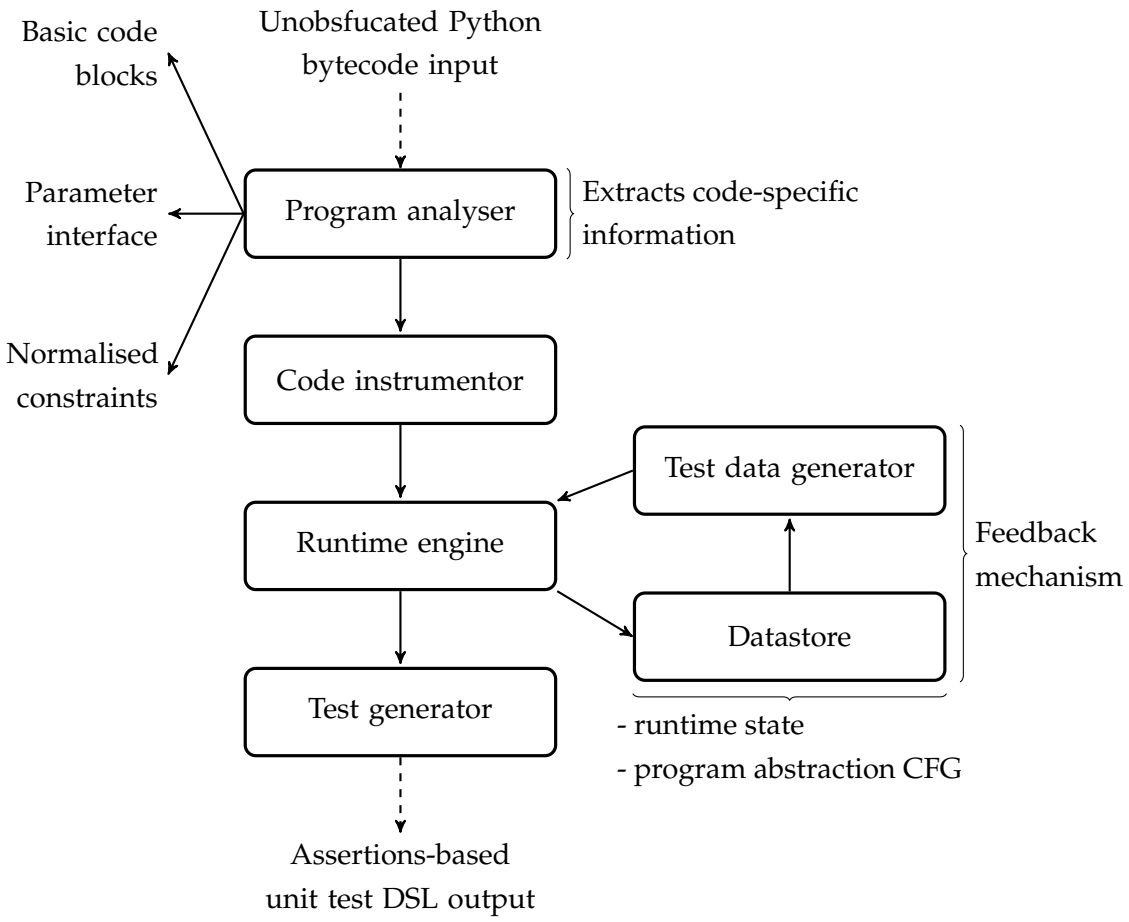
In this chapter, the current research work has been highlighted. In order to demonstrate the idea of automated lazy testing in Python, an example usage of the tool is provided, using the ideas found in this chapter. The structure of this tool, and the examples, is the subject of the next chapter.

Chapter 4

SPLAT

4.1 Architecture

The software design architecture is outlined below:



4.2 Examples

...

4.3 Summary

In this chapter, the working automated testing tool has been shown, but exactly how well does it actually perform against current standards? This notion will be made more precise, both quantitatively and qualitatively, in the next chapter focusing on the evaluation of this work.

Chapter 5

Evaluation

Experimental evaluation entails the following:

i. comparison with existing work, eg. Pythoscope (2010), PyTestsGenerator (2009)

ii. benchmark against popular Python libraries - python-graph, and Python module implementations of famous algorithms

iii. measure quality of test cases generated using metrics - code coverage (statement, path, branch), Linear Code Sequence And Jump (LCSAJ), bugs, crash discovery (pathological inputs)

iv. performance and efficiency - runtime and space complexity

v. generality of output - extensions to generating unit tests for programs in other languages

5.1 Summary

The results achieved and comparison of desired outcomes with original expectations are examined here in this chapter. To finish off, the next chapter summarises this entire paper and hints at what might be possible future research directions in this area of automated software testing.

Chapter 6

Conclusion & Future Work

The following is a simple non-exhaustive enumeration of 'could-haves', if time permits:

- An API to allow for other developers to contribute to the development of this tool, and make use of the algorithms contained therein in isolation
- Comprehensive documentation on the internal workings and usage of the tool, with examples provided as well
- Visualisation for the tool via a user-friendly GUI frontend, powered by wxPython /GTK
- Improve sophistication of the tool to generate more robust and thorough tests, test code containing more complex interaction of language constructs, or deal with new language features in Python 3000
- Optimise efficiency of tool in test generation, by compiling on a faster Python implementation for instance, or scaling parallel search
- Benchmark tool across a wider range of different Python frameworks and libraries
- Explore other techniques and algorithms to attempt to improve overall test code coverage

Bibliography

[ACE11] Tristan O. R. Allwood, Cristian Cadar, and Susan Eisenbach. High coverage testing of haskell programs. In Matthew B. Dwyer and Frank Tip, editors, *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, pages 375–385. ACM, 2011.

[AES07] Andreas S. Andreou, Kypros A. Economides, and Anastasis A. Sofokleous. An automatic software test-data generation scheme based on data flow criteria and genetic algorithms. *Computer and Information Technology, International Conference on*, 0:867–872, 2007.

[BAR09] Yosi Ben Asher and Nadav Rotem. The effect of unrolling and inlining for python bytecode optimizations. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, pages 14:1–14:14, New York, NY, USA, 2009. ACM.

[Ber07] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*, FOSE '07, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society.

[Bot01] Leonardo Bottaci. A genetic algorithm fitness function for mutation testing, April 2001.

[BV11] TIOBE Software BV. TIOBE Programming Community Index for December 2011. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`, 2011. [Online; accessed 7-January-2012].

[CKK+05] Yoonsik Cheon, Myoung Yee Kim, Myoung Yee Kim, Ashaveena Perum, and Ashaveena Perum. A complete automation of unit testing for java programs. In *In Proceedings of the 2005 International Conference on Software Engineering Research and Practice*, pages 290–295. CSREA Press, 2005.

[DLL+09] Rozita Dara, Shimin Li, Weining Liu, Angi Smith-Ghorbani, and Ladan Tahvildari. Using dynamic execution data to generate test cases. *Software Maintenance, IEEE International Conference on*, 0:433–436, 2009.

[Edv99] Jon Edvardsson. A survey on automatic test data generation, 1999.

[GR08] Nirmal Kumar Gupta and Mukesh Kumar Rohil. Using genetic algorithm for unit testing of object oriented software. In *Proceedings of the 1st International Conference on Emerging Trends in Engineering and Technology (ICETET '08)*, pages 308–313. IEEE, July 2008.

[Har00] Mary Jean Harrold. Testing: A roadmap. In *In The Future of Software Engineering*, pages 61–72. ACM Press, 2000.

[HK08] Seung-Hee Han and Yong-Rae Kwon. An empirical evaluation of test data generation techniques. *J. Computing Science and Engineering*, Sep 2008.

[Inc87] D. C. Ince. The Automatic Generation of Test Data. *The Computer Journal*, 30(1):63–69, 1987.

[Jac10] Jonathan Jacky. Pymodel: Model-based testing in python. `http://staff.washington.edu/jon/pymodel/www/`, March 2010.

[Kei11] Gregg Keizer. Computerworld News Article: Security - Google pays record $26K in Chrome bug bounties. `http://www.computerworld.com/s/article/9221186/Google_pays_record_26K_in_Chrome_bug_bounties`, 2011. [Online; accessed 6-January-2012].

[KHC+05] Bogdan Korel, Mark Harman, S. Chung, P. Apirukvorapinit, R. Gupta, and Q. Zhang. Data dependence based testability transformation in automated test generation. In *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering*, pages 245–254, Washington, DC, USA, 2005. IEEE Computer Society.

[Kor90] B. Korel. Automated software test data generation. *IEEE Trans. Softw. Eng.*, 16:870–879, August 1990.

[LMH09] Kiran Lakhotia, Phil McMinn, and Mark Harman. Automated test data generation for coverage: Haven't we solved this problem yet? In *Proceedings of the 2009 Testing: Academic and Industrial Conference - Practice and Research Techniques*, TAIC-PART '09, pages 95–104, Washington, DC, USA, 2009. IEEE Computer Society.

[McM04] Phil McMinn. Search-based software test data generation: a survey: Research articles. *Softw. Test. Verif. Reliab.*, 14:105–156, June 2004.

[MFT11] Stefan Mairhofer, Robert Feldt, and Richard Torkar. Search-based software testing and test data generation for a dynamic programming language. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 1859–1866. ACM, 2011.

[MMSW97] C. C. Michael, G. E. McGraw, M. A. Schatz, and C. C. Walton. Genetic algorithms for dynamic test data generation. In *Proceedings of the 12th international conference on Automated software engineering (formerly: KBSE)*, ASE '97, pages 307–, Washington, DC, USA, 1997. IEEE Computer Society.

[Mur07] Branson W. Murrill. Automated test data generation and reliability assess-

ment for software in high assurance systems. *High-Assurance Systems Engineering, IEEE International Symposium on*, 0:409–410, 2007.

[Nta88] S. C. Ntafos. A comparison of some structural testing strategies. *IEEE Trans. Softw. Eng.*, 14:868–874, June 1988.

[Pet04] Tim Peters. PEP 20 – The Zen of Python. `http://www.python.org/dev/peps/pep-0020/`, 2004. [Online; accessed 7-January-2012].

[PLEB07] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *In ICSE*. IEEE Computer Society, 2007.

[PM87] R. E. Prather and J. P. Myers, Jr. The path prefix software testing strategy. *IEEE Trans. Softw. Eng.*, 13:761–766, July 1987.

[Reg11] Alex Groce & Chaoqiang Zhang & Eric Eide & Yang Chen & John Regehr. Swarm testing. In *Swarm Testing*. Oregon State University, Corvallis, OR; University of Utah, Sep 2011.

[RU99] Gregg Rothermel and Roland H. Untch. Test case prioritization: An empirical study. In *In Proceedings of the International Conference on Software Maintenance*, pages 179–188, 1999.

[SD01] Nguyen Tran Sy and Yves Deville. Automatic test data generation for programs with integer and float variables. In *In 16th IEEE International Conference on Automated Software Engineering(ASE01*, pages 3–21, 2001.

[SG06] Arjan Seesing and Hans-Gerhard Gross. A genetic programming approach to automated test generation for object-oriented software. *International Transactions on Systems Science and Applications*, 1(2):127–134, September 2006. Special Issue Section on Evaluation of Novel Approaches to Software Engineering Guest Editors: Pericles Loucopoulos and Kalle Lyytinen.

[SN] Selvakumar Subramanian and Ramaraj Natarajan. A tool for generation and minimization of test suite by mutant gene algorithm.

[TCM+98] Nigel Tracey, John Clark, Keith Mander, John Mcdermid, and Heslington York. An automated framework for structural test-data generation. In *Proceedings of the International Conference on Automated Software Engineering; IEEE*, 1998.

[TCMM02] Nigel Tracey, John Clark, John McDermid, and Keith Mander. *A search-based automated test-data generation framework for safety-critical systems*, pages 174–213. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[TK] Hitesh Tahbildar and Bichitra Kalita. Automated software test data generation: Direction of research.

[ZLM03] R. Zhao, M.R. Lyu, and Yinghua Min. Domain testing based on character string predicate [software testing]. In *Test Symposium, 2003. ATS 2003. 12th Asian*, pages 96 – 101, nov. 2003.

[ZW01] Jian Zhang and Xiaoxu Wang. A constraint solver and its application to path feasibility analysis. *International Journal of Software Engineering and Knowledge Engineering*, 11(2):139–156, 2001.