Imperial College London
Department of Computing

Pyrulan

Automated lazy testing for Python

Final Year Individual Project

Interim Report (draft)

Lee Wei Yeong

Supervisor: Prof. Susan Eisenbach

January 2012

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. In risus. Ut vitae velit. Aenean malesuada adipiscing sem. Quisque ligula mauris, posuere sed, elementum id, suscipit et, urna. Phasellus suscipit tristique nisl. Vivamus ac urna. Pellentesque egestas facilisis velit. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Morbi eu augue. Vestibulum congue placerat sem. Integer eget urna id risus suscipit eleifend. Etiam at est. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Praesent orci. Suspendisse potenti. Morbi egestas, justo vel cursus eleifend, dui leo dapibus ante, vitae pretium purus massa sed enim. Vivamus eu libero in nulla sagittis aliquam. Quisque semper quam id dui. Nulla facilisi.

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 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

1. Introduction **6**
1.1. Motivation 6
1.2. Automated software testing for dynamic languages 6
1.3. Python 7
1.4. Project contributions 8
1.5. Report organisation 8

2. Background **9**
2.1. Introduction 9
2.2. Basic concepts 9
2.2.1. Definition of terms 9
2.2.2. Validation criteria 10
2.3. Implementations 10
2.3.1. Static method 10
2.3.2. Dynamic method 10
2.3.3. Hybrid 11
2.4. Overview 11
2.5. Current state of the art 12
2.6. Challenges 12
2.6.1. Parameter instantiation 12
2.6.2. Optimising search space coverage 13
2.6.3. Testing a dynamically typed language 13
2.6.4. Non-terminating program executions 14
2.6.5. Early detection of path infeasibility 14
2.6.6. Improving code coverage 14
2.7. Summary 14

3. PYRULAN **15**
3.1. Specification 15
3.2. Approach 16
3.3. Architecture 17
3.4. Algorithm outline 17

4. Evaluation / Discussion **18**

5. Conclusion & Future Work **19**

6. Project Plan **20**
6.1. Estimated timetable 20
6.2. Possible extensions 20

Bibliography **24**

A. Figures **25**

B. Code listings **26**

Chapter 1

Introduction

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 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, which is eventually not included in production. Hence this work, though critical to software quality assurance [Har00], is ultimately hidden from the client, as far as billing and accountability is concerned.

This has led to a large body of work on automatically generating test suites in 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 static programming languages such as Java or C/C++, relatively less emphasis is placed on their dynamic counterparts like Javascript, Ruby or Python.

One such paper implements search-based software testing (SBST) technique, to automatically generate test scenarios for Ruby code, using genetic algorithms [MFT11]. There is no equivalent tool targeting Python instead.

This observation is made in contrast to the rapid growth trend in popularity of dynamic languages, especially in recent years, and in particular, Python. Python was awarded the TIOBE Programming Language of the Year 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 both Python and Ruby are reflective and dynamic languages, it would be 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 it. The importance of readability promotes a *clean, concise and elegant syntax*, advantageous to easily demonstrate proof of concepts.

For instance, Python code is reads more fluently than C#, as depicted below:

Sample C# code

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

Equivalent Python code

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

Apart from the fundamental testing infrastructure toolset of unittest, doctest and py.test offered in Python, there is limited availability of testing support tools built on top of this. Many of these either target outdated versions of Python, or are discontinued. There are even fewer of such tools for automated testing, for instance, pythoscope and pytestsgenerator, which generate tests by performing static code analysis, and a lack of any automated dynamic test case generators at all.

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 testing, focusing on test data generation only using information at runtime
- A motivating example describing automated lazy testing in Python
- Implementation as a Python module to automatically generate high coverage test suites, primarily for Python libraries like Django web framework, NumPy/SciPy, wxPython
- Investigating feasibility of the lazy testing technique, as illustrated by Irulan for Haskell [ACE11]
- Further advance the work in the field of automated software testing, especially for dynamic languages

In order to accomplish these goals, we take advantage of the main features of the Python language, ie. ..., together with its extensive repository, the Python Package Index (PyPI). To this end, ...

The concepts discussed in this paper are concretely demonstrated in a tool called PYRULAN, a high coverage test suite generator for Python library code, written in Python. This tool has successfully been applied to some of the most popular frameworks, and achieved the initial objective of consistently high test code coverage, discovering several bugs in the process as well. The tool has also been extended to conducted property and regression testing, where reports on a sample of case studies are included.

1.5. Report organisation

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

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 later content. Firstly, Section **??** reviews the general field of automated software testing. Section 2.5 deals with the papers that inspired and influenced the project. Relevant characteristics of dynamically typed programming languages, with special focus on Python, are then discussed in Section **??**. Finally, the associated technical difficulties are highlighted in Section 2.6.

2.2. Basic concepts

Software testing delivers quality assurance in the product to the customer. It verifies that software bugs are absent, as far as 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 whether it meets certain functional, non-functional and business related requirements.

Functional requirements

Functional requirements are associated with specific product features and functionality.

Non-functional requirements

Non-functional requirements refer to product quality, in terms of constraints placed on attributes like speed, efficiency, reliability, safety and scalability. This is an area where automation excels in, over manual testing, and thus also becomes the key focus of this paper.

Business requirements

Business requirements reflect customer concerns, with respect to fulfilling the demands of daily work processes.

2.3. Implementations

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, software under test is executed (frequently 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.

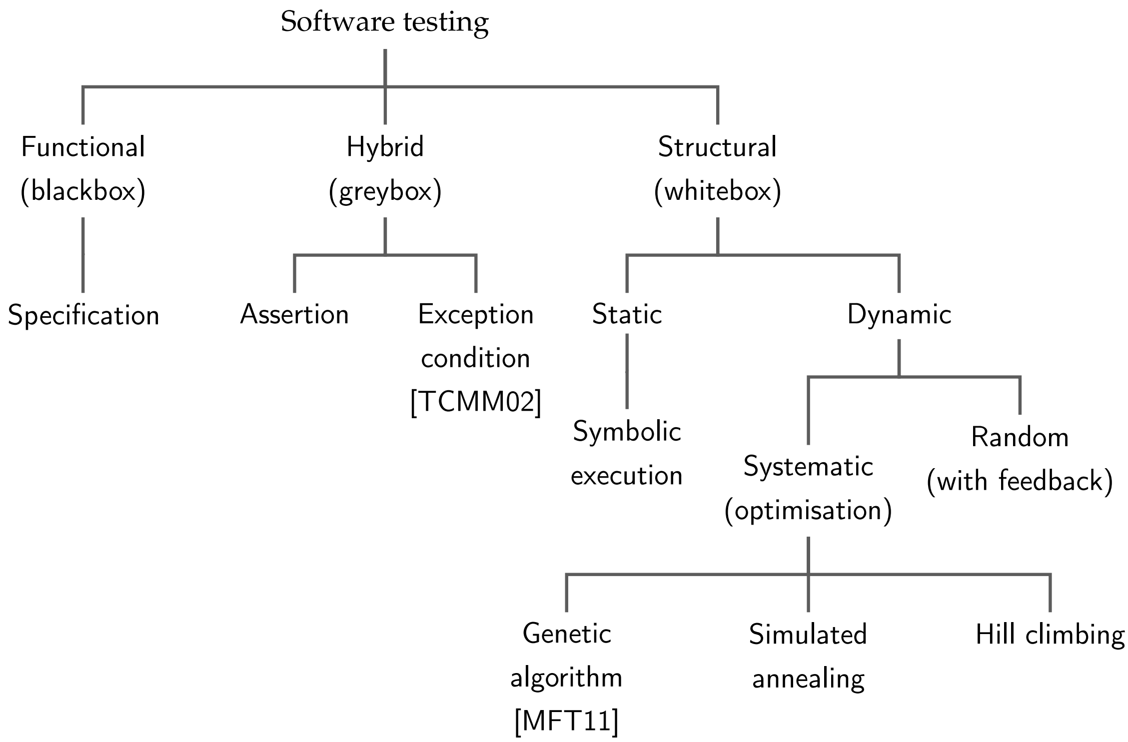
According to the paper [TK], the authors note that research has attempted to combine symbolic reasoning with dynamic execution, or modifying inputs by heuristic function minimisation techniques. However, problems such as scalability and non-termination of infeasible paths arise from this approach.

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:



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

There are three relevant reference survey papers [McM04] [HK08] [TK] describing a high level overview of software testing techniques.

At present, there is only minimal work done surrounding unit test generation in Python, namely, the most recent static analysis tools such as pythoscope v0.4.3 (Feb 2010) and pytestsgenerator v0.2 (Feb 2009) - only accepts Python source code. The closest unit test generator in a dynamic language is RuTeG [MFT11], which uses evolutionary algorithms to automatically create unit tests in Ruby. With respect to the idea of lazy instantiation, Irulan [ACE11] is the 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 constructors and functions to generate test data
2. Needed narrowing / lazy instantiation
3. Inspection of elements inside returned data structures
4. Efficiently handle polymorphism by (lazily) instantiating all possible instances

2.6. Challenges

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

2.6.1. Parameter instantiation

Parameters refer to primitives, data structures like lists, maps or trees, and objects. Depending on available resources, scope might be restricted to only certain numeric types or specific functions, eg. methods invoking `strcmp()`.

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 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 conducting 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.

2.6.2. 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, of which its feasibility still needs to be determined.

2.6.3. 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.

2.6.4. 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.

2.6.5. 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.

2.6.6. 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].

2.7. Summary

In this section, the relevant background literature and theory for this project has been discussed. In order to demonstrate the idea of automated lazy testing, an example demonstration has been developed, using some of the features that were mentioned in this chapter. This example is the subject of the next chapter.

Chapter 3

PYRULAN

3.1. Specification

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

It assumes that the CUT is unobsfucated, so reverse engineering and code reconstruction lies out of the scope of this investigation.

Moreover, we are dealing only with Object Oriented Programming (OOP) style Python programs, ie. 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 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 B.1.

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

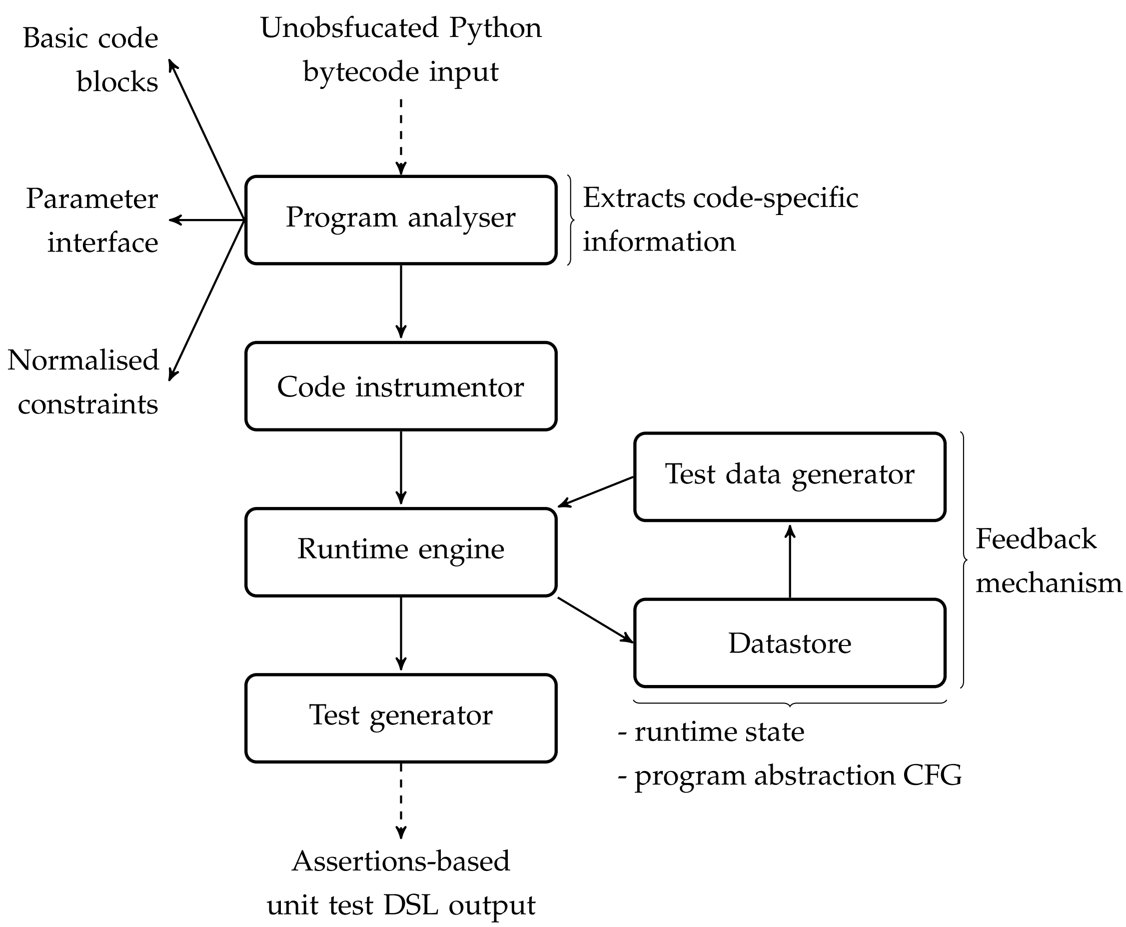
x. *documentation* - epydoc

xi. *Alternative implementations* - PyPy, Unladen Swallow

xii. *fuzzing tools?*

xiii. *supporting tools* - virtualenv/pip

3.3. Architecture



3.4. Algorithm outline

Chapter 4

Evaluation / Discussion

Experimental evaluation entails the following:

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

ii. benchmark against popular Python libraries - Google App Engine (GAE), Django web framework, NumPy/SciPy (Numeric / Scientific Python utilities), Twisted event-driven networking engine, PyPi package index

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

Chapter 5

Conclusion & Future Work

Chapter 6

Project Plan

6.1. Estimated timetable

Key milestones, and fallback positions

6.2. Possible extensions

Relative priority? Rationale?

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 assessment 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.

Appendix A

Figures

Appendix B

Code listings

Listing B.1: Sample Python Bytecode

```
def is_prime(n):
    if n < 2:
        return False
    for i in xrange(2, n):
        if n%i == 0:
            return False
    return True
print sum(is_prime(n) for n in xrange(100))

# the Python Virtual Machine Instructions (bytecode)
# can be disassembled to mimic assembly code
# for instance:
# LOAD_FAST   var_num
#   --> pushes a reference to local co_varnames[var_num] onto the stack
# STORE_FAST  var_num
#   --> stores top of stack into the local co_varnames[var_num]
# LOAD_CONST  consti
#    --> pushes co_consts[consti] onto the stack

import dis

def myfunc():
    a = 2
    b = 3
    print "adding␣a␣and␣b␣and␣3"
    c = a + b + 3
    if c > 7:
        return c
    else:
        return None

# this disassembles the above function's code
dis.dis(myfunc)

"""
  7           0 LOAD_CONST               1 (2)
              3 STORE_FAST               0 (a)

  8           6 LOAD_CONST               2 (3)
```

```
              9 STORE_FAST               2 (b)

  9          12 LOAD_CONST               3 ('adding a and b and 3')
             15 PRINT_ITEM
             16 PRINT_NEWLINE

 10          17 LOAD_FAST                0 (a)
             20 LOAD_FAST                2 (b)
             23 BINARY_ADD
             24 LOAD_CONST               2 (3)
             27 BINARY_ADD
             28 STORE_FAST               1 (c)

 11          31 LOAD_FAST                1 (c)
             34 LOAD_CONST               4 (7)
             37 COMPARE_OP               4 (>)
             40 JUMP_IF_FALSE            8 (to 51)
             43 POP_TOP

 12          44 LOAD_FAST                1 (c)
             47 RETURN_VALUE
             48 JUMP_FORWARD             5 (to 56)
        >>   51 POP_TOP

 14          52 LOAD_CONST               0 (None)
             55 RETURN_VALUE
        >>   56 LOAD_CONST               0 (None)
             59 RETURN_VALUE

"""
```