The Report Committee for Jackson Lee Salling
certifies that this is the approved version of the following report:

Control flow graph visualization and its application to coverage and fault localization in Python

APPROVED BY

SUPERVISING COMMITTEE:

Sarfraz Khurshid, Supervisor

Christine Julien

Control flow graph visualization and its application to coverage and fault localization in Python

by

Jackson Lee Salling, B.S.E.E.

REPORT

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ENGINEERING

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2015

Dedicated to my parents and to my wife, for their love and support.

Acknowledgments

Thanks to my supervisor Professor Sarfraz Khurshid, who introduced me to the concepts in software testing, including graph coverage, fault localization, and mutation testing. For providing guidance, reading this report, and encouraging this project, I am grateful. Thanks to Professor Christine Julien for reading my report and teaching me about literature reviews. I'd also like to thank several other faculty who influenced me greatly during my time in graduate school. Thanks goes to Vijay Garg, Adnan Aziz, Bill Bard, and Joydeep Ghosh. Several students I met in graduate school were particularly impactful during my time there. They raised the bar for everyone, challenged me, and made my work better; thank you to all of them.

Control flow graph visualization and its application to coverage and fault localization in Python

Jackson Lee Salling, M.S.E.
The University of Texas at Austin, 2015

Supervisor: Sarfraz Khurshid

This report presents a software testing tool that creates visualizations of the Control Flow Graph (CFG) from Python source code. The CFG is a representation of a program that shows execution paths that may be taken by the machine. Similar techniques to the ones here could be applied to many other languages, but the CFGs in this tool are tailored to the Python language. As computers get faster, tools to help programmers be effective at work can become more complex and still give quick feedback, without causing an undue performance burden. This tool explores several approaches to giving feedback to developers through a visualization of the CFG. First, just the viewing of a CFG gives a different perspective on the code. A programmer could choose to juxtapose the CFG with complexity metrics during development, seeing increased complexity as graphs grow larger. Second, the tool implements a mechanism to provide code coverage to Python modules. This feature extends the visualization to show code coverage as a highlighted CFG. Test coverage

requirements are calculated to check node, edge, edge-pair, and prime path coverage. From studying existing testing tools, it appears no existing tool for Python provides all these test coverage levels. Third, the tool provides an interface for adding custom highlighting of the CFG, used here to visualize fault localization. Seeing the most suspicious locations from fault localization techniques could be used to reduce debugging time.

The results of running the tool on several popular Python packages, and on itself, show its performance is competitive with the most popular coverage tool when measuring branch coverage. It is slightly slower on statement coverage alone, but much faster against an unoptimized version and a logic coverage tool. This report also presents ideas for extensions to the tool. Among them is to incorporate program repair using fault localization and mutation operators. Visualizing code as a CFG provides interesting ways to look at many software testing metrics.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software has become a ubiquitous part of our daily lives. Every modern car, home, and the myriad of mobile devices we carry rely on software systems that are complex and costly to develop. Testing tools are an important part of the software engineering lifecycle, as testing regularly consumes more than half of development time. Testing attempts to provide information on the quality of software and the relevance to its intended purpose. Quality improvements may help reduce cost during the life of a software product. The testing tool introduced here aims to give feedback to developers by visualizing the control flow graph (CFG) of software, displaying code coverage on the CFG, and highlighting the CFG to show fault localization information.

Popular tools targeted at professional software developers often aim to make software simpler to test, or the development quicker and easier. One example of a development tool already in use is the modern Integrated Development Environment (IDE). This type of application exists for almost any programming language, and speeds development by packaging together a collection of tools (e.g. a text editor, a compiler, tools for running tests, a debugger). The IDE makes editing, compiling, testing, and debugging a single,

seamless workflow. Another type of tool that is often used is a static analyzer of syntax. Across languages, you will find so-called *lint* tools [8] that help a developer correct typical syntax errors. This class of tools provides quick feedback to a developer to encourage learning and improvement of the code. These tools focus on making software easier to develop and improving quality.

Test coverage or code coverage is an important software testing metric quantifying how well a test suite exercises the code base [4]. Coverage information gives a sense of the overall quality. Coverage can be defined on the source code, e.g. as statement coverage, or equivalently on the CFG as node coverage.

The tool presented in this report produces a visual representation of a CFG, a novel way to display software metrics and provide feedback to developers. The primary role of a CFG is to represent the execution flow of a program. A graphical view of the CFG can help programmers see the possible execution paths of a program and develop an intuition of a program's complexity [18]. This tool was written in Python and works on Python programs, but the concepts here apply to most computer languages. Python is an increasingly popular language used across many industries such as web development, scientific modeling, and data science.

In exploring uses of the CFG, this tool provides a code coverage reporting system that overlays coverage information onto the CFG. The code coverage provided in this tool computes node, edge, edge-pair, and prime path requirements on a graph [4]. It reports on coverage of these test requirements

on a CFG, making it easy to see which part of the graph is not covered. I am aware of no other coverage tool that provides edge-pair or prime path coverage for Python programs. As developers get more familiar with coverage metrics, this tool can provide tougher requirements for testing code.

Coverage is becoming a more common requirement of the software life-cycle, which grows the importance of representing coverage reports. The most widely practiced way to visualize coverage information is to highlight the original source code, line by line. Reports from coverage tools typically provide this information, using green highlighting to represent coverage by a test, and red highlighting to represent lines excluded. This tool adopts the red and green color scheme for the visualizations of Python code. The graph representing this information is drawn using mathematical directed graph notation, where circles and arrows depict nodes and edges; each arrow shows a transition between two nodes. A major advantage of viewing code coverage information on a CFG is the ease of seeing uncovered edges in that graph. Since the edges and nodes can be highlighted orthogonally, two kinds of data can be displayed at once. Edges that were excluded from execution can be highlighted red, while preserving the statement coverage information on the nodes. As long as the density of uncovered elements is low enough to distinguish them from one another, several missing paths can be shown at once, including the uncovered edge-pairs or prime paths.

A highlighted CFG can visually represent other testing information, like fault localization. The tool provides an interface for adding custom high-

lighting to a CFG. Using the tool's code coverage facilities, traces of passing and failing tests provide the input to a fault localization formula, which assigns a suspiciousness value to each line of code.

This report is organized as follows. Chapter 2 covers background information on the Python language, compilation process, abstract syntax trees, control flow graphs, code coverage, and fault localization concepts. Chapter 3, regarding implementation, provides an in-depth look at Python structures, choices to represent CFGs, converting the AST to a CFG, instrumentation of the code, coverage visualization overlays, and fault localization. Chapter 4 showcases the features of the tool, focusing on the central role of CFG visualizations. Chapter 5 outlines the results of running the tool on real world programs and compares those results. Chapters 6 and 7 cover the established literature relevant to the project and future work to be considered. I review related work on control flow, fault localization and program repair, and then present ideas on using these techniques with mutation testing to search the program repair space and suggest repairs to a developer. Last, the conclusion discusses achievements during the development of this project.

Chapter 2

Background

2.1 Python

The Python programming language is a general purpose, dynamically-typed computer language. It uses mandatory whitespace indentation to designate blocks of code. The reference implementation, CPython, is open source and written primarily in C. Other implementations exist including PyPy written in RPython, IronPython written in C#, and Jython for the Java virtual machine. The descriptions of Python in this report refer to the CPython implementation version 2.7 [9].

In Python, functions are objects and may be defined at any point in the code, including inside other functions. Python also has executable statements at the module level mixed in with function definitions. A module is simply a Python source file. It may be imported into another module using the import process. A package is a folder, containing modules, that must have a file called `__init__.py` to be recognized as a package. When a module is imported into a running Python program, the code is compiled and top-level statements are executed.

2.2 Compilation of Python

Figure 2.1: The CPython compilation process

Compiling Python code into bytecode is a four step process [9]. First, the code is parsed into a parse tree. Second, an abstract syntax tree (AST) is generated. The output of the AST step uses Python objects and the standard library module *ast*. Third, the AST is converted to a CPython control flow graph. This control flow graph is represented by C structures, and is CPython specific. Finally, the complete bytecode is generated with offsets calculated and inserted. Figure 2.1 shows this process.

During this compilation process, the easiest point to inspect is the AST form of the code. The AST form has two advantages. One, the syntax has already been parsed to verify the code is valid. Two, the AST can be read and manipulated to productive ends. We convert the AST into a CFG by visiting all the nodes in the AST and turning that into an internal graph format.

2.3 Abstract Syntax Trees in Python

Figure 2.2 shows an example of an AST. The tool presented in this report uses the AST directly in converting to a CFG. It can also modify the AST

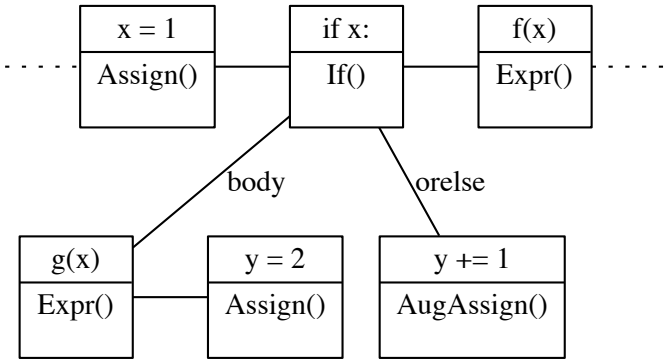


Figure 2.2: Example of a Python AST

to instrument the code for coverage analysis, if that feature is invoked. Any complete implementation of Python should provide an *ast* module interface, therefore this step is implementation agnostic.

The AST is represented as a list of objects, where each object can contain sublists or other object references. Each list, or sublist, contains an arbitrary number of nodes, which represent statements from the code. These nodes are instances of classes from the *ast* module in the standard library. The If class represents *if-else* statements, where the *body* attribute contains a sublist of nodes, and the *orelse* attribute contains another sublist for the else clause. If the *orelse* attribute contains only an empty list, the else statement was excluded from the code. Figure 2.2 shows this recursive tree-list structure. Each node in the AST has attributes for a line number and column offset.

These numbers map back to the original source and give a unique representation to each statement in the code. The tool uses this unique representation for the node names in the CFG.

2.4 Control Flow Graphs and the DOT language

Control flow graphs represent the possible execution paths through a program. They do this by relating each statement or expression in the code with a node in the graph. A transition between statements is represented by an edge in the graph connecting two nodes. Control flow structures introduce choice into the execution, where one of two or more paths may be selected for execution. This choice is shown on a CFG by two or more edges leaving a node. Python does not have a *switch-case* statement, so a maximum of two edges may leave a node. Incoming edges have no such restriction, e.g. a loop with multiple `break` statements will have that many edges going to the same node.

In Thomas McCabe's seminal paper on program complexity, he argues that programs that have ten or more branching points in them are complex [18]. He includes an analysis of FORTRAN programs from the era, where intuition about program complexity seems to follow the number of branching points. One of the valuable things about CFGs is their ability to show the complexity of the functions in the software. McCabe found that high complexity in the CFG was likely to imply difficulty in maintaining code.

The preferred visualization for drawing CFGs uses graph notation. The

DOT language (a human readable, text based format for graph description) is the primary means used to represent directed graphs before they are transformed into graphical visualizations. The *graphviz* package can render DOT files into several file formats. The control flow tool uses DOT as both an internal and external intermediate form to describe the CFG and *graphviz* or another package is required to render it.

2.5 Coverage on graphs

When coverage is defined on a graph directly, the criteria are node coverage, edge coverage, edge-pair coverage, prime path coverage, and complete path coverage [4]. Each are addressed in reverse order. First, if there are any loops in the graph, complete path coverage is impossible as there are an infinite number of paths. Prime path coverage is bounded, and thus solves the infinite set issue. The set of prime paths in a graph contains all the longest simple paths, that is, the set of all the simple paths which aren't subpaths of other simple paths. A simple path is one that contains no loops except if it is a total loop, where the first and last element are equal. The number of prime paths is between linear and exponential with the number of branches. In software, there may be infeasible paths that cannot be executed in practice, which may prevent satisfying the requirements of prime path coverage. Edge-pair coverage contains the set of all the consecutive edges of length two. A CFG produced from Python code cannot have self loops and thus all edge-pairs are simple paths. This means all the edge-pairs are subpaths of the prime

paths. Edge coverage requirements are to cover all edges in the graph. These are a subset of edge-pairs. Node coverage is similar, but must cover all the nodes specifically. Again, all the nodes in the graph are covered by the set of all edges. These criteria form a hierarchy, with progressively stronger criteria subsuming the previous ones. Prime path coverage is the strongest level of coverage included in this tool.

In software, coverage metrics are evaluated on test suites. One goal of the software tester is to get high levels of coverage, ideally attaining full coverage on all criteria. While coverage alone does not guarantee quality, having coverage may provide a chance of catching bugs. When a test suite is run, the execution paths are recorded and compared to the desired coverage criteria. If all the requirements of a criteria are satisfied, the test suite has 100% coverage on that criteria.

2.6 Instrumentation of code for coverage

Instrumentation of code refers to the inclusion of data gathering infrastructure into the executable program [4]. During execution, information on the program is collected and retained for analysis. The canonical way to add instrumentation is to set aside some memory to collect the data, perhaps an array, and place checkpoints into the code which mark the memory. At the end of execution, the memory is dumped to storage for later analysis. The common concrete example of this is to put a boolean array into the program, all values set to false, and place an assignment before each line in the code

that marks the line number in the array to true.

2.7 Fault localization

When code contains a fault, or an error during execution, finding the source of that error is one of the core challenges in debugging. The field of fault localization hopes to aid programmers in finding errors, by using automatic analysis of the code and test suites. For the purposes of this paper, we will restrict the scope of fault localization techniques to spectrum based approaches, those that use the analysis of test coverage information on passing and failing tests [20].

If a test suite has at least one passing and failing test, the coverage information from those tests can help to find the location of faults in the source code. For this report, fault means a defect in the code, error means an incorrect state during execution, and failure refers to an error that propagates to produce an incorrect output. For instance, only lines that were executed by the failing test can contribute to a state of error or eventual failure. Furthermore, lines executed by passing tests would seem to have a lower likelihood of containing the fault than lines only executed by failing tests. Those principles, formulated mathematically, conspire to give a suspiciousness rating to each line of code, encapsulating the likelihood it contains an error. One such formula used is the Tarantula technique, and is given by the equation:

$$suspiciousness(l) = \frac{\frac{\#failing(l)}{totalfailing}}{\frac{\#passing(l)}{totalpassing} + \frac{\#failing(l)}{totalfailing}} \tag{2.1}$$

There are many competing fault localization techniques, and comparison of them is detailed in [1].

Chapter 3

Implementation

The tool presented in this report was written entirely in Python, to inspect Python code. While most of the concepts here could be generally applied, the focus will be on Python specific concerns. Where there could be a universal treatment among languages, I attempted to keep the discussion general enough that others could apply the concepts.

3.1 CFG design choices on Python structures

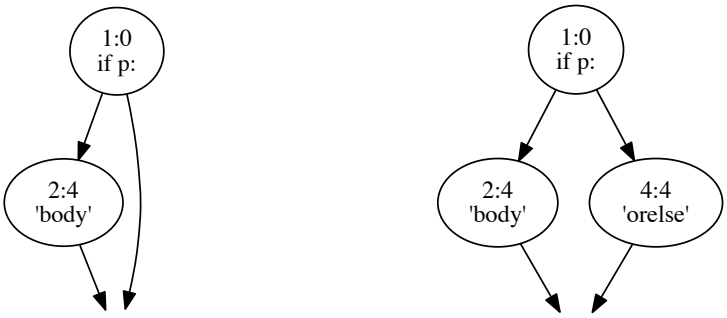
Design decisions made during the process of writing this tool are detailed here. At the most basic level, decisions involve defining the control flow graphs (CFGs) for Python syntax. To avoid any ambiguity I've noticed in the literature, all the control flow structures are discussed and the representative graph is given.

The Python language includes keywords for control flow that are familiar to those who have used other languages, with some notable differences. We start by cataloging the basic control flow structures available in Python. The common structures included are `while`, `for`, `if`, `if-else`, and `try-except-finally`. No *switch-case* statement exists in Python. Instead,

there is the `if-elif`$^{+}$`-else`. This replaces the *else if* pattern used in C and allows consistent indentation of Python statements. There is no *do-while*, the looping construct that always executes the loop body once. The early exits from loops are `break` and `continue`. Exiting from a function happens at the `return` keyword or out of band, by an exception or using `raise`. The Python language adds an `else` block to the `while`, `for`, and `try` structures. This block executes if the structure is left in the standard way, i.e. by the loop predicate becoming false or the try block finishing normally. When a loop exits by a `break` statement, the else block is bypassed. For a `try` statement, the `else` block is executed if no exception is raised. There are also `with` blocks, `yield` statements, ternary operator expressions, anonymous lambda functions, and generators. Each of these are mentioned, even though they need no special treatment.

Each Python statement has a unique position in the code defined by the line number and column of that statement. This information is captured during compilation and each node in the CFG is labeled with a *line:column* marker. The identifying information is visible in the CFG for reference. Also the CFG can be annotated with the relevant source code. For clarity, the code is added by default.

Figure 3.1a shows the CFG for an `if` without the `else` block. Control flow falls through to the next statement, if there is one. Figure 3.1b includes the `else` block. These are the basic conditionals which most languages employ.

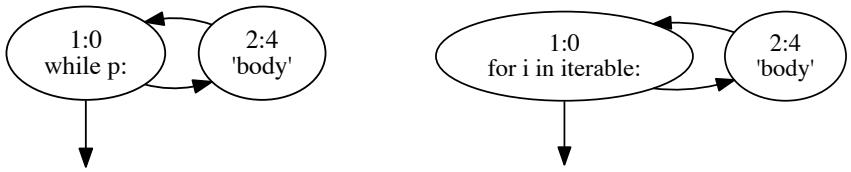


(a) (b)

Figure 3.1: Representations of if and if-else statements from Python

In Figures 3.2a and 3.2b we see the `while` and `for` structures are isomorphic, and identical in this case. Due to the similarity, further examples only use a `while`. The predicate, p, is evaluated at the top of the first loop and the beginning of each subsequent loop. The last statement in the body of the loop flows back to the loop predicate, not to the first statement in the body. This ensures we have no self-loops in a Python CFG and makes explicit that the conditional is executed each time.

The CFG for a `while` with an `else` block is shown in Figure 3.3. When a `break` statement is in the body of a loop, the CFG has an edge to the next statement after the `else` block. Figure 3.4 shows a concrete example, where the body has an `if` and `break`.



(a) (b)

Figure 3.2: Representations of while and for loops from Python

The representation of `try` is without an arrow to the `except` block, as shown in Figure 3.5. This is intended to illustrate that exceptions are special circumstances, and arrival into an `except` block is out of band, not part of normal control flow.

An important point to note is that functions are treated as isolated, disjoint subgraphs. In Python, a function definition can happen anywhere, whether it is inside a module, a class, or another function. These functions are not executed as encountered, of course, but rather they are parsed and saved to be called later. Each function is a separate object that can be covered by tests but is not covered just by executing its definition line. Special control flow considerations are made when a function definition is encountered in the code to keep the subgraph disjoint. Function decorators are designated as part of the function definition code block. Classes are also shown as disjoint subgraphs with member methods, in turn, being separate graphs from the

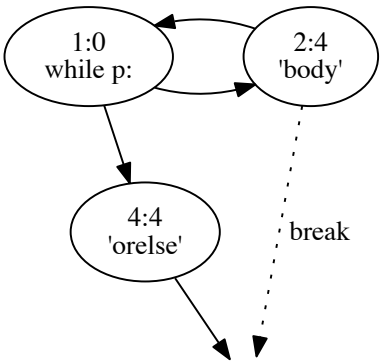


Figure 3.3: Abstract representation of while loop with a break

class. Only class-level data are connected to the class definition in the graph.

Only code contained directly in a module is inspected and translated to a CFG. Concepts like meta-programming or self-generated code, which create Python code dynamically and execute it at run time, make accessing the source code impossible in general.

All the other types of Python statements deserve mention. The `yield` statement is not parsed as a return, but as just another statement. The reason is that control flow will resume at this point if the generator is called to yield another value. The CFG represents this behavior without specifying the exit and reentry explicitly. A `with` block is represented as a simple statement. No special control flow happens, but the language guarantees that an exit method

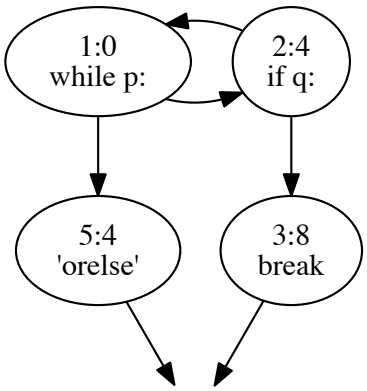


Figure 3.4: Concrete representation of while loop with a break

is called when the block is left. Ternary expressions are also not transformed into special CFG representations, but just treated as expressions. The decision not to show the ternary operator is due to keeping the CFG at the statement level. These sub-statements do provide control flow internally, but as part of one executable statement. In the AST, the ternary operator is allowed only one subexpression per outcome of the predicate. It could be useful to mark each of these as executable during instrumentation, for finer resolution of coverage. In representing control flow, the granularity is kept at the statement level. The *and* and *or* operators of boolean logic also represent lower level control flow. Due to short-circuit evaluation of boolean clauses, these logic operators are practically control flow. Still, boolean predicates are treated as being on

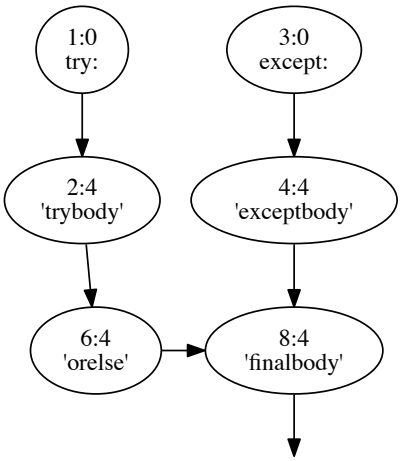


Figure 3.5: Representation of try-except-finally from Python

the enclosing statement. One line generators and their variants, list, set, and dictionary comprehensions, are a shorthand syntax for looping over an iterable collection of items. A generator is effectively a function with looping semantics. Again, we treat generators like calls into functions. They are not represented by small control flow graphs, but only as part of the statement line on which they reside. Creation of nodes and edges for a comprehension would mean the introduction of "dummy nodes" to avoid adding self-loops, since the line and offset is the same for the generator as for the whole statement. The case can be made for any of these syntax elements to have small CFGs generated. The approach taken here is to keep the CFG at the statement level.

3.2 AST to CFG translation

The translation algorithm uses the AST representation of Python code to create a CFG. To convert an AST into a CFG requires domain specific knowledge about control flow statements in that language. The previous section contains all the rules that our algorithm must follow. The next step is to process AST objects to create the intermediate representation of the graph in DOT syntax. The Python objects that represent statements in the AST follow a pattern of having sublists of objects if they are a control flow statement. The AST is effectively a tree of lists. The objects While and If have *body* and *orelse* attributes that contain sublists. The rules of control flow must be captured by the algorithm to correctly connect nodes of the CFG. For example, a While node should contain an edge from the *body* exit back to the While node.

The algorithmic approach is walking the AST node by node as a list, and recursively calling each sublist to be walked. Appropriate information about edges going from predecessors to successors is maintained and passed up or down the stack recursively. Simply put, add a node to the CFG for each statement, and add edges to successor statements or sublists. Special casing is required for *break* and *continue*, as well as looping constructs. The exit of a *body* sublist must return information back up the stack to the appropriate successor object. Edges are added from the tail of *body* and *orelse* attributes, to the next node. The main challenge in writing the AST to CFG translation algorithm is in maintaining control flow information for the edges of the graph. The nodes are easily identified as they map one-to-one with AST objects,

which map to code statements. If the edges follow the control flow rules of the language, the resultant graph will be a valid CFG. A more succinct description of the algorithm is this:

1) Add a node at each AST object by looking at the line and column number
2) Add an edge to each successor node in current list, in sublists by recursive call, and to parent list successor by recursive return
3) Process Return and Raise to have no edges outgoing
4) Pass Break and Continue up the call stack until a loop is found

A valid CFG can be produced from the AST even in the presence of faults or code that is semantically invalid. Dynamic typing can sometimes lead to errors in a program at runtime, like undeclared variables or type mismatch. These errors will not prevent the program from being parsed, converted to an Abstract Syntax Tree (AST), and compiled. The parser requires valid syntax to produce the AST, so any CFG will be valid in terms of syntax even if it fails to run correctly.

Once the CFG is created, the tool can use the graph to compute test requirements. We do this from the dictionary representation of the CFG that is generated in AST to CFG conversion. The module uses the CFG dictionary to compute the test requirements for node coverage, edge coverage, edge-pair coverage, and prime-path coverage. Test requirements are returned as sets of nodes or paths. Paths can be length one, two, or more corresponding to edges, edge-pairs or prime paths. The calculation of coverage metrics given a set of

test paths is a comparison of the requirements for that coverage level to the executed nodes or paths.

One may wonder why keeping the column number is useful for tracking coverage. This is simply because more than one statement may appear on a single line in Python code. Since this code idiom is infrequent, coverage tools have ignored the subsequent statements, and mark the entire line as covered. This tool can discern if each statement was really executed or not.

3.3 Instrumenting the AST for path information

To instrument the code, we use the fact that each statement has a unique identifier from the original syntax, the line number and column number of the statement. The algorithm extracts this information from the node of the AST and inserts an instrumentation statement into the AST which, when executed, records the line and column into a Python list. The inserted statement adds the identifier to the list of visited nodes. This approach follows a framework of instrumenting code for node coverage, but instead of marking a binary variable in an array when the statement is executed, we maintain the entire execution path in a list. The execution path gives deeper data on how the program proceeded than would be given by a binary array. From an execution path, all types of graph coverage can be computed including nodes visited, edges or edge-pairs visited, and comparisons with prime path requirements. Recording the execution path does take more memory, but most test suites run for short periods of time and memory is not a constraint.

The last step of the process is to compile the new instrumented AST using Python's built in `compile` command. Following that, we can execute it in a module namespace so that code is imported into the Python session.

3.4 Importing the instrumented code

After the AST is instrumented and compiled to a code object, the new version is placed into a module namespace. Using the import process and standard library in Python, the tool creates a new module and executes the code object in that namespace. This binds all of the functions into the module object and makes them available by the usual calling convention. The effect is the same as a regular import statement, except we have the instrumented module ready to track execution paths. Modules are placed in the system module dictionary so that other modules can refer to them if needed. One caveat is that forcing module imports like this, one at a time, can occasionally run into import dependency issues. When multiple modules are instrumented from a package, they may import each other and cases may arise where not all run time functions are instrumented. The proposed solution is to use an import hook to load any requested module as an instrumented module on an as-needed basis. This extension is mentioned in future work.

3.5 Graph highlight visualization design choices

Visualizing node and edge coverage comes naturally on the CFG. Showing edge-pair coverage is more difficult, since the concept of edge-pairs is not

displayed independently of edges. Several methods to show edge-pair coverage were considered. In Figure 3.6, one way of showing all the edge-pairs is il-

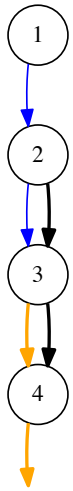


Figure 3.6: Representing edge-pairs independently in a graph

lustrated. The figure represents edge-pairs as independent objects, but shows the difficulty as colors are added to edge-pairs just to tell them apart. Showing edge-pairs on normal single-arrow edges implies that for most edges we represent them being the first edge in a pair *and* the second edge in a pair. Any version of multi-purposing arrows begins to look cluttered. The common case for edge-pairs is that most will be covered along an execution path, with some branched pairs remaining uncovered. By highlighting the uncovered pairs in red, the graph visualization stays simple while some ambiguity may

be introduced. See Chapter 4 for an example.

Visualizing prime path coverage, the next stronger criteria, is similarly difficult. Do we show the covered or uncovered paths, or both? Again the choice is unclear, but to show all the information at once would require each edge have a multiple part visualization that has number of parts equal to the number of prime paths that include that edge. An example on the graph {1->2->3->2->4} is shown in Figure 3.7. This simple graph has five prime paths and is already cluttered.

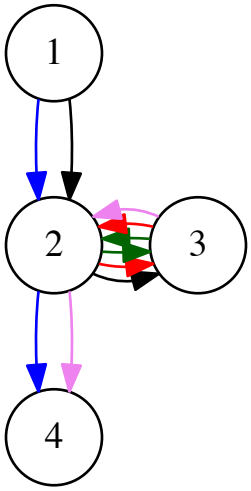


Figure 3.7: Representing prime paths independently in a graph

The number of prime paths in a graph can be very large. Following the

design choices made for edge-pairs, we highlight the uncovered prime paths. Cases where only a few prime paths span the entire edge set are common. In a visualization, virtually no information is added when all the edges appear uncovered. Use of more colors could map some of the information back onto the CFG, but for simplicity let us use only red and green for coverage highlighting.

3.6 Gathering test run data for fault localization

With all the code coverage infrastructure written into this tool, the ability to gather statement hit spectrum data already exists. When a failing test indicates the presence of a bug, locating that fault in the code is the next step. Most jUnit-style test frameworks support a *setUp()* or *tearDown()* method. These methods are called before and after each test, respectively, by the test framework. By adding a command for execution path logging into the setup method, the path for each test is recorded. This path information was gathered for offline analysis to produce statement suspicion ratings. Using the Tarantula technique described in Section 2.7, each statement is assigned a likelihood it contains the bug. From this information, the CFG can be highlighted with the failing test's execution path and the highest suspicion statements. This view of the CFG gives a developer a quick glance at the possible locations of the fault captured by the failing test. Chapter 4 shows an example of this highlighting.

Chapter 4

Motivating Example

For this chapter, all the examples of control flow graphs derive from this code snippet:

```
def sum_up_to(x):
    y = 0
    if x > 0:
        y = x
        while x > 1:
            y += x
            x -= 1
    return y
```

The function returns the sum of positive integers up to x, inclusive. Inputs less than zero return zero. This is also known as the arithmetic sum, and it can easily be computed without a loop using a closed form solution. The implementation has a bug in it – a rather serious bug – since it only gives the correct values for inputs up to 0 and 1. We will investigate fault localization on this bug in Section 4.3.

4.1 Control Flow Graph visualization

The CFG in Figure 4.1 was generated automatically using the control flow tool. The visualization of a CFG easily shows the programmer that there

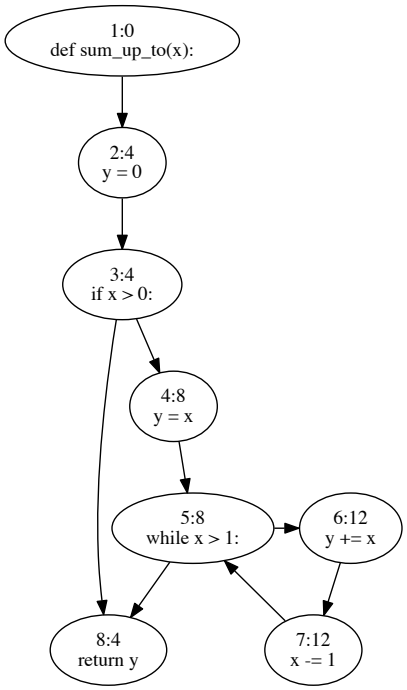


Figure 4.1: Motivating example CFG

is one exit point from this code (the node in the graph that has no subsequent edges). Using this CFG we can generate test requirement criteria for the graph. These criteria include node coverage, edge coverage, edge-pair coverage, and prime path coverage criteria. The next section discusses CFG visualizations with these coverage metrics overlaid.

4.2 Coverage visualizations

For a test execution of input $x = 2$ we show the path and the highlighted CFG illustrating coverage in Figure 4.2. The tool takes coverage information and overlays it on the graph with color highlighting and bold lines. Both node

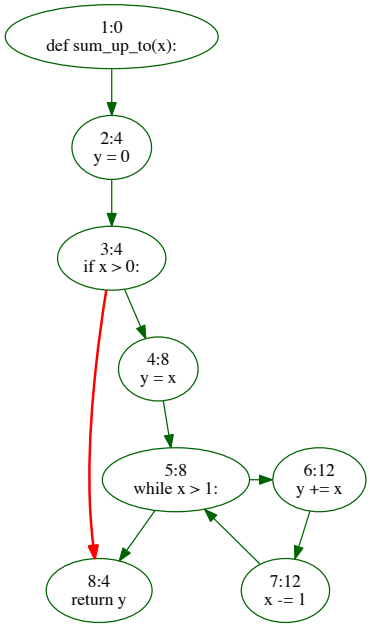


Figure 4.2: CFG with node and edge coverage information

and edge coverage are displayed at the same time and the result is clear from the CFG. We could choose to show only one or the other, but since the nodes and edges are orthogonal sets, these two types of coverage lead to a natural display of information on a visualization. All nodes are covered, resulting in 100% statement coverage. Not all the edges are covered. This example

illustrates the usefulness of easily identifiable edge coverage metrics. The edge from 3:4 to 8:4 remains uncovered. A test engineer could then add one test to get full edge coverage.

The next CFG shows the edge-pair coverage, in Figure 4.3. The test

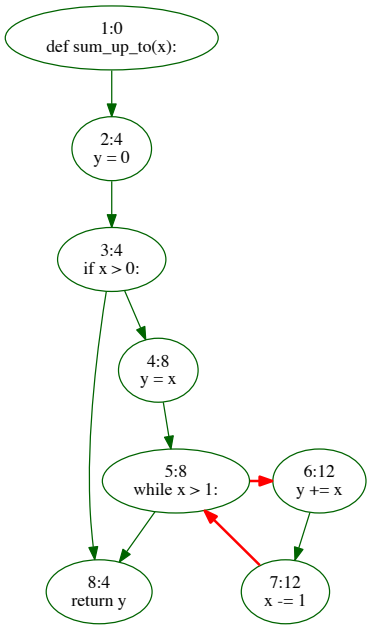


Figure 4.3: CFG with uncovered edge pair information

inputs are $x = 0$, to cover the missed edge from Figure 4.2, and again $x = 2$. To overlay edge-pair information, a decision was made to show uncovered edge-pairs. The discussion in Chapter 3 explains this selection. The uncovered edge-pair is (7:12, 5:8, 6:12). The test cases do not enter the loop twice, and hence they miss this edge-pair.

To show prime path coverage we use the same test inputs, $x = 0$ and $x = 2$. Two prime paths are missed: {(6:12, 7:12, 5:8, 6:12), (7:12, 5:8, 6:12, 7:12)} Both span the same edge subset. Both the missed prime paths imply

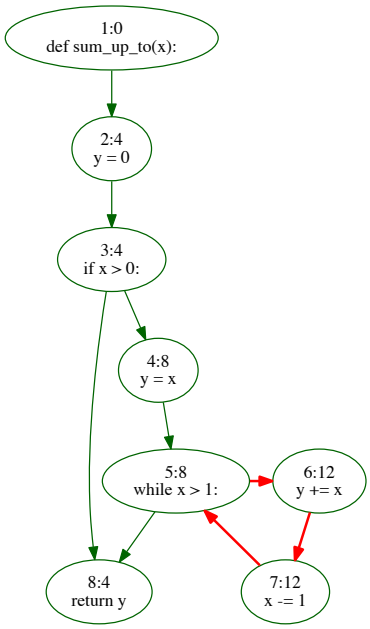


Figure 4.4: CFG with uncovered prime path information

that the loop was not entered a second time. This is the same implication that the edge-pair coverage provided. Therein lies is one of conclusions we can draw about these criteria.

If testing loops with zero, one, and two or more trips through the loop is important to your test team, the edge-pair coverage will do the job. Prime path criteria tends to be verbose, including more paths and covering all

possible combinations of paths that do not have loops. Paths with loops have a prime path for every node in the loop, at least. Sometimes, an edge-pair is semantically infeasible, perhaps due to branch conditions that are dependent with each other. This type of missing test requirement can confound attempts to get full coverage. Picking the right criteria without creating an unnecessary burden on the test team is at the core of software test strategy.

4.3 Fault localization visualization

Up until now, we have ignored the fact that input $x = 2$ produces the wrong output. If this fact is written into an assertion in a test case, that test will surely fail. To find the likely location of the fault in the code, each statement is assigned a suspiciousness rating using the Tarantula technique. The formula used requires at least one passing and failing test. Using test inputs (0, 1, and 2), and expected values (0, 1, and 3), the most suspicious lines are numbers six and seven. The CFG in Figure 4.5 shows the entire failing test case execution path in orange, and the most suspicious statements highlighted in red and bold. These are most suspicious since no passing test executed them. The only edge remaining in green is the edge from 3:4 to 8:4 which was only executed by a passing test. The bug could be fixed by adjusting the assignment to y at line six, but several other options exist. Changing combinations of lines four, five, and six can all be made to fix this bug. This small example serves to show the possibility of using CFGs for fault localization. Caveats include that spectrum based localization does not always

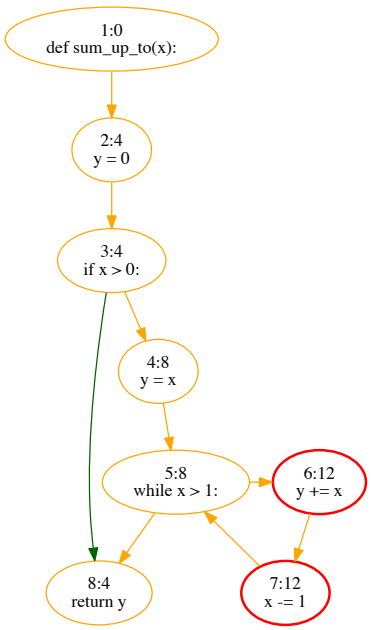


Figure 4.5: CFG with fault localization information

provide the *best* location to fix. Many classes of faults can execute the same path for passing and failing tests, so no information is gained from the traces.

Chapter 5

Results

5.1 Control Flow Graph creation

The graph in Figure 4.1 is an example of the CFG our tool generates from Python source. These CFGs are computationally inexpensive to produce. The CFG for all the source files tested during data gathering took less than 90ms per file. Some of these were 600 lines or more, with complicated looping and exception handling in the control flow. The process to create the CFG includes using the built in Python parser and AST construction, followed by walking the AST and outputting the graph. No optimization passes were made on this code; it is measured here in its original form.

5.2 Comparison with coverage tools

Besides creating a CFG of a program, one of the major features is AST instrumentation for gathering coverage information. Since several coverage tools for Python already exist, benchmarks are an interesting way of comparing them. I ran the coverage tools on four programs with test suites. The first was a simple implementation of the *middle* function, taking in three numbers as input and returning the middle of the three. The second of the benchmarks

shown is the control flow tool calculating the prime path requirements of its own CFG. This function call takes over 6 seconds on the complex CFG input. Third, the *simplejson* package is tested with its included test suite. Simplejson is the library that outputs or parses JavaScript Object Notation (JSON) for data interchange. Last, the benchmarks are run against the *pyramid* web framework, which keeps a high level of test coverage on the project. All times are from a 1.3 GHz Intel Core i5 processor in a mid 2013 laptop.

In Table 5.1, this report's tool is called "controlflow". The control flow tool can be bootstrapped to run on itself. It is invoked to instrument the source files and import them, so that coverage can be tracked. To give comparison, data is shown for the run times using *coverage.py* and *instrumental*, the two best known existing coverage packages. It is compared to *coverage.py* in three modes. One is the basic coverage run tracer, two with the branch coverage option turned on, and three with the C extension removed and running in pure Python. The last comparison is to *instrumental*, which tracks modified condition decision coverage (MCDC) for logic predicates and statement coverage. Instrumental also uses the AST instrumentation approach.

5.3 Discussion of results

The first choice tool for covering Python code is *coverage.py*. Viewing the timing results, it is the fastest on any program of reasonable size. It has a tracer written in C for speed, which works well, delivering the fastest coverage times except on a small example program. The branch coverage option is

	middle	primepaths	simplejson	pyramid
baseline	0.03s	6.4s	1.2s	10.0s
controlflow	0.04s	15.2s	2.6s	15.8s
coverage.py	0.1s	9.7s	2.3s	14.2s
coverage.py branch	0.1s	13.3s	3.6s	14.4s
coverage.py timid (no C)	0.11s	38.9s	22.2s	62.0s
instrumental	0.15s	48.5s	20.4s	41.7s

Table 5.1: Run times of various coverage tools on Python programs

slightly slower, while the "timid" option is much slower. The tool for CFGs compares favorably to *coverage.py* run with the branch option.

The time for running the tool on pyramid is slightly suspect. Forcing import of instrumented modules for the pyramid package appears to break some internal dependencies. All the tests run, and times are probably comparable, but several tests have issues. These issues include reporting the wrong coverage information, due to reloaded modules not recording coverage. A likely fix for this would be to use an import hook mechanism inside Python, instead of forcing import of all modules before the test run. An extension using import hooks is discussed in future work.

The good news is that my tool is faster than both *coverage.py* in the slowest mode and *instrumental*. This execution speed is unoptimized, and as such leaves room for improvement. The fact that the speed is as good as it is bodes well for the coverage tool's use in diverse applications. PyPy might benefit from use of another coverage tool written in pure Python, suitable for its JIT compiler.

An interesting advantage of the approach I took for coverage is that only the modules requested for coverage are measured. *Coverage.py's* use of *sys.settrace* means that all of the execution is tracked. Tracking only one module would lead to performance gains, if coverage was only needed for that module.

5.4 Comparison of test coverage

In most cases all the coverage tools reported the same results, but some notable differences arose. Those differences are discussed here.

A place where the AST instrumentation approach can shine is when multiple statement are on one line. Since each statement has a unique line number and column number from the original source, multiple statements per line can all be treated separately. Neither of the two comparative tools correctly identify uncovered branches from the same line of source. This gives higher coverage totals to these tools, but it is incorrect to do so. Missing statements are correctly handled and included in the report of the control flow tool. This leads to better coverage of code written on one line, and less ambiguity.

Among foremost differences, *coverage.py* and *instrumental* exclude counting expressions containing only a string literal. String literal expressions in Python code are often used for documentation. "Doc strings" are used as part of standard coding idioms and relied upon for extracting documentation for Python modules. Since these strings have no runtime effects, they are ignored

by *coverage.py* and *instrumental*. They are neither counted as coverable code, nor marked as covered. The control flow tool does not ignore them, since they appear as executable expressions in the AST. This leads to lower coverage totals on files with frequent occurrences of documentation string literals, when they are not executed. Lack of special treatment of string literal expressions can be fixed as future work.

Next, the name of the function and its definition are counted as an executed line by *coverage.py*. The line gets counted at load time, before the function is called. As a Python program is executed, like at load time, defined functions get added to the namespace as they are encountered, so it makes sense to mark this line as executed when the function is reachable in the namespace. Still, the same argument could be made about waiting to cover any part of a function until it is called at run time. This choice is part of the instrumentation step of the control flow tool. The default behavior is to wait until a function is called to mark it's definition line as covered. Again, this design leads to lower coverage percentages in some cases.

The *coverage.py* tool supports a special type of comment in Python code, the "pragma: no cover" option. If this comment is on a source line then the associated code is not counted toward coverage. The count of excluded lines is noted in the report, however. Since the tool does not support the "pragma: no cover" comment, this line will count against coverage totals.

Chapter 6

Related Work

6.1 Control Flow Graphs in software historically

In 1970, F. Allen wrote a paper on control flow analysis of programs which appears to be the first publication using directed graph notation to illustrate control flow [3]. During this decade, debate was raging in the software community about structured programming [17]. Structured programming is the concept of relying on readable control flow structures and generally avoiding the use of goto statements. The most basic structures were named D-structures after Edsger W. Dijkstra, who promoted the use of structured programming. The if, while loop, and for loop are all structured programming elements. It was proved that any program could be transformed into a structured program using only D-structures and adding boolean variables. Despite this equivalence, there was concern that programs would be harder to understand without more complex structures. [17] also argues that clarity in programming is important, and that it could be achieved using D-structures or variants like do-while and switch-case. That paper catalogs the control flow graphs (CFGs) for each of the control structures from the literature circa 1975.

Thomas McCabe's cyclomatic complexity paper presents on his well

known complexity metrics based on CFGs [18]. He covers other topics like non-structured flow graphs and the defining traits of all non-structured programming. He also proposes a testing methodology where code should be simplified if the number of feasible independent paths is less than the cyclomatic complexity. This definition of complexity on a flow graph, one plus the number of branch points, is equal to the number of linearly independent paths needed to test the code.

6.2 Coverage tools for Python

Of all the tools in this area, *coverage.py* is the best known package for Python. It is a mature, stable, and fast package, and the tracer that collects data is written in C for speed. If you are testing Python code, this is likely the package you use when measuring test coverage. It is recommended that *coverage.py* be the default coverage tool when you need the information it provides. It can track line coverage, as well as branch coverage, albeit at a slower speed. In benchmarks for this report, branch coverage took an average of about 25% more time to run.

Another Python tool is called *instrumental*. It uses the same architectural approach, modifying the AST, as the tool presented here. The major feature which instrumental adds is condition and decision coverage of predicates. This is called MCDC, modified condition decision coverage. Each clause in a predicate must be tested to both true and false when it determines the predicate. In some industries, like aviation, testing of each clause in a pred-

icate is a requirement. The performance of *instrumental* is comparable to turning off the C extension speedups in *coverage.py*. Against the fast version of coverage, it takes about 5 times longer.

6.3 Fault localization

During software development or debugging, finding the location of faults is one of the most time consuming processes. Automatic fault localization refers to techniques used to help find the location of a software fault in the code. Many techniques have been proposed and studied, and a good summary of the topic is in the survey paper [20]. Of interest to this report is the effectiveness of fault localization techniques. To show fault localization I chose the popular Tarantula formula for suspiciousness. This approach falls under the category of executable statement hit spectrum analysis. These spectrum-based techniques are compared and rated on effectiveness in [1]. The paper concludes that for localizing faults, the Tarantula technique is not optimal and the Ochiai coefficient performs consistently better. One application of these localization methods is to evaluate the quality of automatic test generation. [7] shows the relative effectiveness of several test generation techniques measured with fault localization. The better fault localization becomes, the more likely it is to become a mainstream software testing component. Qi et al. [19] compare localization techniques in the context of providing faster program repair.

6.4 Automatic program repair

What if it was possible to automatically find ways of fixing faults in a program? That is goal of research into automatic program repair. This section reviews the existing literature on repair techniques, and sets the stage for the ideas in the following chapter on program repair.

Much of the relevant work in this field uses a broad, but powerful, approach of genetic programming. This can be viewed as a randomized algorithm that tries many combinations of actions, hopefully working toward the goal. Progress metrics are evaluated along the course of an algorithm, allowing bad candidates to expire and more fit candidates to continue. The name, and approach, comes from biology, and mimics the gene mutation and mixing process and survival of the fittest selection process. The literature includes papers on test generation and co-evolution of repairs [6], genetic program repair for C programs, [13–16], and using program repair to evaluate fault localization effectiveness [19]. The genetic programming approach is used on Python, with a limited number of mutations, in [2].

Much of the literature uses a system called GenProg for C programs, which is aptly named since it's based on genetic programming. The approach the authors of GenProg take is to use only three basic operators in the code. Delete a line, add a line, or replace a line. Adds and replaces are done by randomly selecting amongst the existing code base. This approach works well and details of costs and effectiveness are in [14].

Using mutation testing for repair, combined with fault localization, is argued to be an effective and efficient approach [10]. Debroy et al. uses Tarantula and a significant number of operator mutations, and finds that about 1 in 5 bugs can be fixed with the system. A system for Python called MutPy brings mutation testing to version 3 of Python [11].

Chapter 7

Future Work

7.1 Using mutation testing for program repair

I propose an extension to this work that provides suggestions for fixing faults in the code. Since there are techniques for fault localization, the next logical step is to automatically repair those faults. Providing automatically generated corrections to the code would be a boon to programmers tasked with debugging. Of course, providing a robust repair that passes all tests is the best outcome of such a system. Existing literature has worked to provide such repairs [14]. However, providing *suggestions* to the developer about candidate fixes could be just as valuable in practice. A human generally has to review bug fixes. If candidate repairs spur on inspiration for a robust fix, the end result is the same as a programmatically generated one. Whereas automatic program repair would seek to fix bugs independently, suggesting program repairs implies a programmer that is aided by an algorithm.

The problem of program repair is considered a search problem. The search space is all possible programs, which is an infinite set. Even the set of all possible valid Python programs is still infinite. It makes intuitive sense to start with the current faulty program and make changes to it. The set of

nearby Python programs is significantly smaller and is bounded depending on how you define "nearby". The search proposed here is based on the idea that mutation operators are close to typical programmer errors. By performing a bounded exhaustive search of possible mutant programs, we might find one which survives the test suite. Even if we don't find a candidate that passes all tests, we can provide a list of best fitting candidates. If the usual goal of mutation testing is to kill mutants, the goal here would be to survive. A surviving mutant is defined as one that passes a failing test and the previously passing tests, as well. The groundwork for this approach is in [10], which uses mutation operators to find repairs on C and Java programs.

The task is a search problem amongst the set of candidate programs, which is exponential with program length. The search space grows as $O(m^r)$ with the number of mutations, m, and number of repairs chosen, r. To see this, realize that there are $\binom{m}{r}$ possible candidates at order r,

$$\binom{m}{r} = \frac{m!}{(m-r)!r!} \tag{7.1}$$

so while $r << m$,

$$\binom{m}{r} = \frac{m * (m-1) * ... * (m-r+1)}{r!} \approx m^r/r! \tag{7.2}$$

Let's take an example program and say it has a fault on more than one line, then allow 10 mutations to the original program. This means there are $2^{10} = 1024$ possible programs since each mutation can be on or off independently. The *correct program* is the one candidate that is most preferred and

passes all tests. There are likely to be equivalent mutants to both the original program and to the correct program. Killable mutants pervade the space, by definition, otherwise software would be less error prone to write. If it takes three mutations to produce the correct program, there are $\binom{10}{3} = 120$ possibilities; for one and two mutations there are 10 and 45 candidates respectively. To reduce the search time, keeping m and r small is required. Using fault localization to selectively place the mutants targets the search, and restricting the number of repairs limits the search. Finding a surviving mutant does not guarantee the correct program, but might be a good place to start for human intervention.

Comparing to a genetic algorithm, a bounded exhaustive search provides different qualities. GenProg samples each possible repair independently, but aggregates these repairs into a candidate [14]. By the Central Limit Theorem, we know the number of repairs per candidate will cluster around the average in a normal distribution. This favors the candidates with close to the average number of repairs. Weighted sampling biases the results toward lines with a high suspiciousness weighting. Crossover of mutants, by mixing some mutations of one candidate with another candidate, increases the number of repairs per candidate and probably serves to increase the diversity of candidates. There is some finite probability of repeating tests against the same candidate during this search. Whereas a bounded exhaustive search does away with any random sampling, no candidates are repeated. There is no chance for exploring deep into the search space, but on the other hand all the lower

order mutants can be tested. Number of candidate patches (NCP) is a metric introduced in [19] to count candidates before a successful patch. An interesting way of evaluating techniques against each other is to use this metric.

A development environment that automatically suggested repairs would be highly valuable to most programmers. If some surviving mutants were found, they might be close to the correct program. Even if no surviving mutants are found, mutants that are closest to surviving could provide faster insight into fixing faults manually. The repairs could be annotated onto the CFG, starting from the fault localization highlighted version.

7.2 Import hook for instrumentation

Adding an import hook mechanism is a top priority for extending the code. The Python import system supports registering custom import of modules. The existing code forces import of user specified modules for instrumentation. This can lead to problems, as in the case of measuring coverage on the pyramid package. If an import hook inserted into the process could catch all specified module imports, the result would be cleaner and less likely to encounter module override issues and dependencies.

7.3 Other general extensions

Adding "pragma: no cover" support would give this tool comparable semantics to existing tools. Adding special case handling of string literals

used for documentation serves the same goal of producing familiar coverage results. The coverage instrumentation part of this control flow tool is interesting enough that it could find use as a stand alone tool. Splitting the project into separate modules has not been considered up until now.

A general extension that may be useful is to visualize the CFG as a text-based graph, thus allowing IDEs and editors to include CFGs alongside the source code. This might bring the idea of visualizing CFGs more into the mainstream of software testing.

Chapter 8

Conclusion

The software testing tool presented here explores visualization of code, coverage, and fault localization. It produces a control flow graph (CFG) from a Python source code file. It also includes an entire system for instrumenting Python code to track execution paths. The visualization features provide an interface to the intermediate representation of the CFG to provide highlighting for coverage or fault localization. Test coverage of the code is reported from the execution paths and can be compared to four graph coverage criteria. The criteria are node, edge, edge-pair, and prime path coverage. Currently, no other Python tool provides edge-pair or prime path criteria for coverage. The entire Python syntax is faithfully translated into a CFG, following the structures discussed in the implementation chapter. Even function decorators are included in the CFG; however, no meta-programming or self-generated code can be inspected.

Visualizations of the CFG provide an interesting perspective on the analyzed program. The CFG was the canonical representation for exploring ways of visualizing coverage and fault localization. I explored three categories of visualization, from the basic CFG, to a highlighted version illustrating coverage,

and finally a view of fault localization overlaid on the CFG. Special versions of the coverage highlighted CFG display edge-pair or prime path coverage. The hope is that this work will provide useful visual feedback to developers of Python computer programs.

Testing speeds on a sample of popular packages show the system is competitive with branch coverage measurement using the state of the art, *coverage.py*. The system could be optimized further to increase speed. Future extensions include a full system for fault localization and a repair suggestion tool based on mutations. Integrating into the import hook process available for Python would make this testing tool easier to use on large packages. Adding a text based representation of the CFG might extend its usefulness by integrating into the IDE.

I have shown that visualization of CFGs could provide interesting ways of looking at commonly used software testing metrics. The tool described by this report gives Python developers access to some of the formal coverage criteria largely used in academic models. Python is a popular language, and Python developers must have useful ways of testing and visualizing code.

Bibliography

[1] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. An evaluation of similarity coefficients for software fault localization. In *Dependable Computing, 2006. PRDC'06. 12th Pacific Rim International Symposium on*, pages 39–46. IEEE, 2006.

[2] Thomas Ackling, Bradley Alexander, and Ian Grunert. Evolving patches for software repair. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 1427–1434. ACM, 2011.

[3] Frances E Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM, 1970.

[4] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2008.

[5] Andrea Arcuri. Evolutionary repair of faulty software. *Applied Soft Computing*, 11(4):3494–3514, 2011.

[6] Andrea Arcuri and Xin Yao. A novel co-evolutionary approach to automatic software bug fixing. In *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on*, pages 162–168. IEEE, 2008.

[7] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. Directed test generation for effective fault localization. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 49–60. ACM, 2010.

[8] Pylint contributors. Pylint - code analysis for python. http://www.pylint.org, May 2015.

[9] Python contributors. Welcome to python.org. http://www.python.org, May 2015.

[10] Vidroha Debroy and W Eric Wong. Using mutation to automatically suggest fixes for faulty programs. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 65–74. IEEE, 2010.

[11] Anna Derezińska and Konrad Hałas. Analysis of mutation operators for the python language. In *Proceedings of the Ninth International Conference on Dependability and Complex Systems DepCoS-RELCOMEX. June 30–July 4, 2014, Brunów, Poland*, pages 155–164. Springer, 2014.

[12] Ethan Fast, Claire Le Goues, Stephanie Forrest, and Westley Weimer. Designing better fitness functions for automated program repair. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 965–972. ACM, 2010.

[13] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 947–954. ACM, 2009.

[14] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 3–13. IEEE, 2012.

[15] Claire Le Goues, Stephanie Forrest, and Westley Weimer. Current challenges in automatic software repair. *Software Quality Journal*, 21(3):421–443, 2013.

[16] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *Software Engineering, IEEE Transactions on*, 38(1):54–72, 2012.

[17] Henry F Ledgard and Michael Marcotty. A genealogy of control structures. *Communications of the ACM*, 18(11):629–639, 1975.

[18] Thomas J McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, (4):308–320, 1976.

[19] Yuhua Qi, Xiaoguang Mao, Yan Lei, and Chengsong Wang. Using automated program repair for evaluating the effectiveness of fault localization

techniques. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 191–201. ACM, 2013.

[20] W Eric Wong and Vidroha Debroy. A survey of software fault localization. *Department of Computer Science, University of Texas at Dallas, Tech. Rep. UTDCS-45*, 9, 2009.

Vita

Jackson Salling was born and lives in Austin, Texas. He attended the University of Texas at Austin for undergraduate studies where he received the degree of Bachelor of Science in Electrical Engineering in 2003. He works for Fluke Networks as a hardware and software engineer, designing test equipment for fiber optics. In 2013, he started graduate school in software engineering.

Address: salling@utexas.edu

This report was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.