Supported Programming for Beginning Developers

A Thesis
presented to
the Faculty of California Polytechnic State University,
San Luis Obispo

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in Computer Science

by
Andrew Gilbert
March 2019

© 2019
Andrew Gilbert
ALL RIGHTS RESERVED

COMMITTEE MEMBERSHIP

TITLE:	Supported Programming for Beginning Developers
AUTHOR:	Andrew Gilbert
DATE SUBMITTED:	March 2019
COMMITTEE CHAIR:	John Clements, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Aaron Keen, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Hasmik Gharibyan, Ph.D. Professor of Computer Science

ABSTRACT

Supported Programming for Beginning Developers

Andrew Gilbert

Testing code is important, but writing test cases can be time consuming, particularly for beginning programmers who are already struggling to write an implementation. We present TestBuilder, a system for test case generation which uses an SMT solver to generate inputs to reach specified lines in a function, and asks the user what the expected outputs would be for those inputs. The resulting test cases check the correctness of the output, rather than merely ensuring the code does not crash. Further, by querying the user for expectations, TestBuilder encourages the programmer to think about what their code ought to do, rather than assuming that whatever it does is correct. We demonstrate, using mutation testing of student projects, that tests generated by TestBuilder perform better than merely compiling the code using Python's built-in `compile` function, although they underperform the tests students write when required to achieve 100% test coverage.

ACKNOWLEDGMENTS

- Thanks to Andrew Guenther for uploading this template.
- GNU Parallel [49] was helpful in running many tests quickly.
- The 2018 paper "Soft Contract Verification for Higher-order Stateful Programs" by Nguyễn et al. [31] was helpful in my analysis of TestBuilder. I only wish I had time to implement some of the ideas presented there for TestBuilder.

TABLE OF CONTENTS

Page
LIST OF TABLES . . . viii
LIST OF FIGURES . . . ix
CHAPTER
Glossary . . . xi
1 INTRODUCTION . . . 1
1.1 Operation . . . 2
1.2 Thesis Layout . . . 4
2 BACKGROUND . . . 6
2.1 Concolic Testing . . . 6
2.2 Directed Test Case Generation . . . 7
2.3 SMT Solvers . . . 8
3 METHODS . . . 16
3.1 Python . . . 16
3.2 Types . . . 17
3.2.1 Z3 Types . . . 17
3.2.2 Python Types . . . 18
3.2.3 Type Mapping . . . 19
3.3 Cartesian Product Algorithm . . . 20
3.4 Type System . . . 21
3.4.1 No Type System . . . 22
3.4.2 Statically Checking `Any` Variants . . . 22
3.4.3 Static Expression Extraction . . . 23
3.4.4 Type Checking . . . 24
3.5 Standard of Truth . . . 26
3.6 Function Substitution . . . 26
3.7 Preprocessor . . . 27
4 MODEL . . . 30
4.1 Overview . . . 30
4.1.1 SSA IR Reduction . . . 30
4.1.2 Compilation of SSA IR to Z3 Expressions . . . 31
4.2 Examples . . . 37
4.2.1 Identity Function . . . 37
4.2.2 Basic Expression in Function . . . 38
4.2.2.1 What about passing type sets down? . . . 39
4.2.3 Conditional . . . 40
4.2.4 Pairs and Mutation . . . 41
5 EVALUATION . . . 44
5.1 Grade Boosting . . . 45
5.2 Student Tests vs. TestBuilder Tests . . . 46
5.3 Refactoring Legacy Code . . . 47

6 RESULTS . . . 48
6.1 Grade Boosting . . . 48
6.2 Student Tests vs. TestBuilder Tests . . . 48
6.3 Refactoring Legacy Code . . . 50
7 CAPABILITIES AND LIMITATIONS . . . 52
7.1 Capabilities . . . 52
7.2 Limitations . . . 54
8 TOOLS . . . 55
8.1 Libraries . . . 55
8.1.1 Dataclasses . . . 55
8.2 Testing . . . 56
8.2.1 pytest . . . 56
8.2.2 Hypothesis . . . 57
8.2.3 unittest.mock . . . 57
8.3 Typing . . . 62
8.3.1 mypy . . . 62
8.3.2 MonkeyType . . . 64
8.4 Debugging . . . 65
8.4.1 `breakpoint()` . . . 65
8.4.2 pdb . . . 65
8.4.3 Pdb++ . . . 65
8.4.4 Embedded IPython . . . 65
8.5 Formatting and Style . . . 66
8.5.1 Black . . . 66
8.5.2 isort . . . 67
8.5.3 flake8 . . . 67
8.5.4 Vulture . . . 67
8.6 CLI . . . 68
8.6.1 Docopt . . . 68
9 RELATED WORK . . . 69
9.1 Automatic Test Generation . . . 69
9.2 Concolic Testing . . . 74
10 FUTURE WORK . . . 76
10.1 Obvious Deficiencies . . . 76
10.2 Alternative Applications . . . 77
10.2.1 Property Proving . . . 77
10.2.2 Test Case Minimization . . . 77
10.3 Additional Capabilities . . . 77
10.3.1 Boundary Test Case Generation . . . 77
10.3.2 Concolic-style Testing . . . 78
10.3.3 Mutation-Driven Test Generation . . . 78
11 CONCLUSION . . . 79
REFERENCES . . . 80

LIST OF TABLES

Table Page
3.1 Mapping from Python to Z3 types. 20

5.1 Top three most common final lines of output from failed TestBuilder runs. . 45

6.1 Results of mutation testing. 51

LIST OF FIGURES

Figure Page
1.1 Code under test. 3
1.2 TestBuilder interface.. 3
1.3 Generated test cases.. 4
2.1 Simplified grammar of Z3 terms 15
3.1 Rules without typing information 22
3.2 Rules for `BinOp Add` with static type checking. 23
3.3 Rule for addition with static extraction of values from wrappers. 23
3.4 Typechecking rule for addition 24
3.5 Additional typechecking rules for addition with Any typed variables. 24
3.6 Using a variable as both of its parent types.. 29
4.1 Grammar of SSA IR 31
4.2 Reduction of SSA IR 32
4.3 Reduction of SSA IR expressions 33
4.4 Definition of δ for SSA IR 33
4.5 Compilation of binary operator. 37
6.1 Project breakdown 49
6.2 Mutation killing performance of student and TestBuilder tests. 50
7.1 Not-quite type error 54
8.1 Example pytest run on a module with inline tests 58
8.2 Example Hypothesis test 59

8.3 Example Hypothesis inputs 60

8.4 Example test using `unittest.mock` 61

8.5 Example of Python class with PEP 484 type annotations 63

8.6 Stubs for golden master `LinkedList` implementation generated by MonkeyType 64

8.7 The pdb++ user interface 66

8.8 Example of Vulture catching an uncalled function 67

Glossary

Duck typing See definition in [38].

EAFP it is “easier to ask for forgiveness than permission [39]”.

MOOC Massive Open Online Course.

Chapter 1

Introduction

"Why write comments?" "Why write unit tests?" "It works — if it ain't broke, don't fix it." While most programmers would agree on the importance of commenting or unit testing or refactoring, students may not understand the importance of these practices. Unit testing, in particular, seems like a waste of time, since running the program often suffices to remove the obvious bugs. But obvious bugs are obvious. The harder bugs to find are those caused by edge cases[1] and unexpected inputs. These require creativity and care on the part of the tester to catch. Computers are supremely uncreative, but they are very good at carefully trying everything. By applying computers to the problem of testing code, we can help students discover the unexpected edge cases which cause their programs to fail.

What does it mean to say that software "fails" or "has a bug?" One meaning of software "having a bug" is that the software crashes: it does an operation which is illegal, such as division by zero, or it throws an exception because some part of the code produces improper values. On the other hand, sometimes when we say that software is incorrect we mean that it does not perform as the user desires: perhaps a function which, instead of adding its inputs, multiplies them. How do we convert these concepts into rules which a computer can apply? The simplest rule is

[1] Recall that there are two hard problems in computer science: cache invalidation, naming things, and off-by-one errors [6].

to find valid inputs on which the software terminates unexpectedly. But software can terminate correctly for every input and still never produce the desired result.

Automatically determining the desired result is challenging. It might seem that if programmers wrote better specifications, we could compare the software with them automatically. But Jack Reeves argues that software is itself the design which results from the software engineering process [41], much like how a blueprint is the result of a civil engineering process. The best documentation of how the software is to work is in fact its own source code. Nevertheless, we still want to find the logical inconsistencies or the incorrect behaviors that may be present in it.

This is the basis of software testing: providing a necessarily-incomplete specification of the intended behavior of the program, with the goal of discovering both logical inconsistencies and incorrect results. The programmer specifies that, given these inputs, the program should produce this output. Much effort has gone into finding ways to write more effective tests. Yet beginners struggle to do enough testing [13].

By automatically generating test cases which exercise every line in the program, we can assist beginning programmers in confirming that their code works, while reducing the burden of manually finding inputs which result in execution following a particular path. The program presented in this thesis, TestBuilder, is a small step toward that goal. It is able to generate test cases for a limited subset of Python which includes mutation but which does not include unbounded loops. The ideas in this work should be extensible to a subset of Python suitable for beginning programmers.

1.1 Operation

As an introductory example, consider the code in fig. 1.1 on the following page. TestBuilder will run on this code and attempt to create tests for lines 3, 4, 6, 7, and 10. Note that there is no way to reach line 10, as any time `d > 0`, `c + 2` will be a type

```
1 def func(a):
2     if a < 2:
3         c = "abc"
4         d = 3
5     else:
6         c = 4
7         d = -2
8     if d > 0:
9         if c + 2 > 4:
10            return True
```

Figure 1.1: Code under test.

```
➡  testbuilder clements_example.py
Generating unit tests for func
Working on test 0 at 3
==================================================
def func(a):
    if a < 2:
        c = 'abc'
        d = 3
    else:
        c = 4
        d = -2
    if d > 0:
        if c + 2 > 4:
            return True

Suppose we pass the following arguments:
{'a': 2}

What is the expected output of func from these arguments? None
```

Figure 1.2: TestBuilder interface.

error. Thus TestBuilder will fail to generate a test case for line 10, and will print out a note to the user. For lines 3, 4, 5, and 6, however, TestBuilder will generate test cases. The test cases for lines 3 and 4 and for lines 5 and 6 are expected to be duplicates of one another, as each pair of lines has the same set of conditionals needed to reach it.

Running TestBuilder on the code in fig. 1.1, it produces a series of questions like the one shown in fig. 1.2. In this case, note that I, as the user, have typed `None` at the prompt, indicating that the expected output of the function is `None` when given the input $a = 2$.

```python
func = import_module("clements_example").func
def test_func():
    a = 2
    actual = func(a)
    expected = None
    assert convert_result(actual) == expected

func = import_module("clements_example").func
def test_func_3():
    a = 0
    with pytest.raises(TypeError):
        func(a)
```

Figure 1.3: Generated test cases.

The resulting test cases are shown in fig. 1.3.[2] Note that the duplicate test cases are omitted; there is an additional copy of each of the test cases shown for the other line in the pair, as discussed above. Also note that the `expected` value in the first test is exactly what we specified at the prompt in fig. 1.2 on the previous page. The second test case confirms that a `TypeError` occurs when we try to run the code with $a = 0$. Both test cases are in pytest style. The unusual import technique involving `import_module` allows TestBuilder to handle files with names which are not valid Python identifiers. This could be easily changed to use the `from file import name` form, should that be considered more desirable.

1.2 Thesis Layout

Chapter 2 provides background information on the techniques we are using. Chapter 3 describes our approach to test case generation, which draws heavily on concolic testing. Chapter 4 presents a formal model of the compilation process we use. Chapter 5 presents the evaluations conducted on TestBuilder, while chapter 6 describes the results. Chapter 7 describes the specific abilities of TestBuilder. Chapter 8 presents

[2]The file-wide imports have been omitted for space.

a list of some additional tools which could be of use in helping beginning programmers write better code, while chapter 9 discusses the existing work in concolic testing, test case generation, and similar techniques. Chapter 10 describes future work which could potentially make TestBuilder more capable. Finally, chapter 11 concludes the thesis.

Chapter 2

Background

2.1 Concolic Testing

In their 2005 paper [23], Godefroid, Klarlund, and Sen present an early concolic testing tool named DART. Their tool integrates three factors: "[automated] extraction of the interface of a program," "[random] testing to simulate the most general environment the program can operate in," and "dynamic analysis of how the program behaves under random testing and automatic generation of new test inputs to [direct] systematically the execution along alternative program paths."

A concolic testing tool initially runs the system under test with a random input. It tracks symbolic and concrete values of every variable, and notes the symbolic value of the conditionals which are reached until the system exits. The list of conditionals is converted into inputs to a solver and the final conditional reached before termination is inverted. The solver is used to find a result which satisfies all the conditionals, if possible, including the new inverted final conditional. This finds inputs that are expected to take a new path through the program. The program is then executed once again using the new inputs. Previously unseen conditionals are tracked and added to the list of conditionals. This repetition is continued until all the paths

found are exhausted. At this point, complete path coverage has been achieved on the program, under optimal conditions.[1]

Notably lacking from the description above is any discussion of how a concolic tester determines whether the system under test is behaving as expected. DART, for example, looks for "program crashes, assertion violations, and non-termination."

TestBuilder uses the concept of solving a list of conditionals with an SMT solver in order to choose inputs to reach a point in a program. However, rather than generating that list of conditionals dynamically during program execution, TestBuilder statically analyzes the program and converts the required conditionals into constraints for the SMT solver.

Due to the nature of the basic concolic algorithm, where all conditionals taken to reach a point are solved, concolic testing seeks full path coverage. TestBuilder, on the other hand, aims for full statement coverage. Thus, when constructing the expression to pass to the SMT solver, it includes the conditionals from every path which could lead to the desired statement. This allows the SMT solver to choose any path it desires when constructing a solution. By being careful about how the conversion is done, it avoids the exponential explosion of paths which might be expected.

2.2 Directed Test Case Generation

In order to support the generation of tests on code with existing test coverage, we designed TestBuilder to allow the user to choose specific lines to generate test cases for. Requesting a test case for a specific line causes TestBuilder to find all code paths that may reach that line and solve for inputs which follow one of them. The desire to

[1] One of the problems which can befall a concolic tester is if the code uses operations it cannot model. In that case, it can approximate them by using the concrete value at that point in the program [23]. This results in a loss of symbolic information and removes the guarantee that the tester will cover every path, as it is possible it will be unsuccessful in attempting to trigger a run down a desired direction.

support this use case makes concolic testing an extension of the current TestBuilder system, as directing concolic testing to a specific part of the code would require initial code analysis to determine the possible paths which would lead execution to the desired line. As a potential future adaptation, TestBuilder could be extended to support concolically executing one of the available paths. This would likely provide many of the benefits of concolic testing, while still allowing the user to only generate test cases for specific lines.

2.3 SMT Solvers

An SMT solver is used to solve for inputs which meet the required conditions to navigate to the desired line. SMT solvers use a system of theories combined with a SAT solver to find satisfying assignments of variables. Typically, SMT solvers support theories such as "integers," allowing them to reason about expressions on integer values, rather than only the boolean logic SAT solvers are natively able to work with. In TestBuilder, we use the Z3 SMT solver, initially described by Moura and Bjørner [29].

One note on terminology: Z3 refers to the type of a variable as its sort. For simplicity, I will use type to refer to Z3 sorts below.

As an example of what an SMT solver can do, suppose we wish to find an integer that is less than fifty using Z3. We will need to do two things: declare a variable to represent some integer, and then constrain that variable to be less than fifty. Z3 can then be instructed to find a value for the variable which satisfies the constraints upon it. Using the Z3 bindings for Python, we would write[2]

```
import z3

x = z3.Int('x') # Create a new Z3 variable
```

[2]This merely prints out the solver results. In practical applications, we would use a `Solver` instance, in order to have access to the results from within Python.

```
z3.solve(x < 50) # Find a value for x which meets this constraint
```

This will print a solution to the expression provided to `solve`, such as

$$x = 49$$

Notice that Z3 variables are declared with static types. Constructors like `Int` are provided by Z3 for all of the built-in types. In Python, they will return special objects that have some Python operators redefined to produce Z3 values representing the symbolic result of the operations.

For example, the expression `x < 50` is legal due to redefinition of the less-than operator on instances of Z3 integers in Python. Thus, the Python expression `x < 50` turns into an instance of a Z3 class representing the expression $x < 50$.

If we have multiple constraints, we can create logical expressions using provided Z3 logical operators:

```
z3.solve(z3.And(x > 38, x < 50))
```

This will require that $x > 38$ and $x < 50$. A satisfying value for x will be printed, such as

$$x = 49$$

Now suppose we also had some boolean constraints we wished to impose. Suppose that we want to find a boolean b and an integer x such that either $b \wedge (x > 50)$ or $\neg b$. We could write something like this:

```
import z3
x = z3.Int('x')
b = z3.Bool('b')
z3.solve(z3.Or(z3.And(b, x > 50), z3.Not(b)))
```

Z3 might then produce a result such as

$$b = True \qquad x = 51$$

which satisfies this set of constraints.

As mentioned above, Z3 also supports custom datatypes, which are tagged unions. If we wish to describe a binary tree with integer leaves and no values on the interior nodes, we might write the following:

```
import z3
mktree = z3.Datatype('tree')
# The first argument is the name of the constructor, the other
# arguments define the fields and their types. In this case, both
# fields contain trees themselves.
mktree.declare('Node', ('left', mktree), ('right', mktree))
# z3.IntSort() defines the val field to store Z3 integers.
mktree.declare('Leaf', ('val', z3.IntSort())


tree = mktree.create()
```

Functions like `z3.IntSort()`, used in the declaration of `Leaf`, are provided by Z3 for all the built-in types it supports, allowing for the definition of fields that store values of those types. When defining a datatype, the `Datatype` object used to define it can also be used as the type of recursively defined fields, as seen in the definition of `Node`.

The resulting type — `tree` — can be used to create variables. The Python class for it also has methods which allow access to its various constructors. So, for example, we can write

```
tree.Leaf(34)
```

which creates a new Leaf tree with 34 as its value.

The custom type also has methods to constrain variables of its type to only use certain constructors. For example, suppose we want a tree which has a `Node` at the root, rather than being a single `Leaf`. We would write

```
# This declares a variable of type Tree, just like z3.Int declares a
# variable of type Int
t = z3.Const('t', tree)


z3.solve(tree.is_Node(t))
```

and Z3 might return something like

$$t = \text{Node}(\text{Leaf}(0), \text{Leaf}(1))$$

It chooses values to satisfy the constraint presented — in this case, requiring that the resulting value use the `Node` constructor of `tree`.

As mentioned above, Z3 is statically typed. However, custom datatypes offer a way to provide a sort of dynamic typing for Z3. We can define a type which can wrap values of various types, and then all the values can be treated as having one type. This is rather reminiscent of the Any types used in typecheckers for dynamic languages, such as the mypy typechecker for Python [30]:

```
import z3
mkany = z3.Datatype('Any')
# Make a constructor named Int which has one field, i, which holds
# an integer
mkany.declare('Int', ('i', z3.IntSort()))
# Make another constructor
```

```
mkany.declare('Bool', ('b', z3.BoolSort()))
Any = mkany.create()
```

Using this, we can declare a variable[3] of Any type:

```
a = z3.Const('a', Any)
```

and we can now write a Z3 query such as

```
# define a new variable of Any type
a = z3.Const('a', Any)
z3.solve(Any.is_Int(a), Any.i(a) > 50)
```

to ask Z3 to choose a value for `a` which is an instance of the `Int` constructor containing a value for the `i` field which is greater than 50. This might produce a result like

$$a = \text{Int}(51)$$

which would mean that the expressions provided are satisfied if `a` is constructed with the `Int` constructor and the `i` field of that constructor contains 51.

Notice that we used two constraints here: both that the value of the `i` field is greater than 50, and that `a` itself is constructed with the `Int` constructor. If we omit this latter constraint, as here:

```
z3.solve(Any.i(a) > 50)
```

then Z3 will only apply the constraint if it happens to choose the `Int` constructor for `a`. Running this, we get the result

[3] Z3 refers to variables as "constants;" for Z3, variables are the things quantified in first-order logic.

$$a = \text{Bool}(\mathit{False})$$

which clearly is not the integer we would like to get. Thus, when using this technique to define dynamically-typed values, we need to add constraints on the constructors used for the result, as well as the constraints on the fields of those constructors.[4]

We use this form of definition for dynamic values extensively throughout TestBuilder. Thus, for the rest of this thesis, we will use the grammar defined in fig. 2.1 on page 15 to write the Z3 expressions involved.

Note that, in Z3, equality is defined between all types. This has been omitted from the grammar for clarity.

A few examples should clarify the use of this grammar.

First, consider our original example of searching for an integer less than 50. Using Python and our Any type, along with the a variable from before, we could express this as

```
z3.And(Any.is_Int(a), Any.i(a) < 50)
```

Running a solve on this in Z3 produces the result

$$a = \text{Int}(0)$$

[4]This appears to arise from the SMT-LIB specification, which Z3 supports as one of its input formats. The specification reads

> [Remark 11] (Partiality of selectors). As in classical first-order logic, all function symbols in a signature Σ are interpreted as total functions in a Σ-structure $\mathbf{A}$. This means in particular that if $g : \sigma\sigma_i \in \Sigma$ is a selector, the function $g^{\mathbf{A}}$ returns a value even for inputs outside the range of g's constructor. Definition 8 imposes no constraints on that value, other than it belongs to $\sigma_i^{\mathbf{A}}$. For instance, in a structure $\mathbf{A}$ with a sort for integer lists with constructors nil and insert and selectors head and tail for insert, the function $\text{head}^{\mathbf{A}}$ maps $\text{nil}^{\mathbf{A}}$ to *some* integer value. Similarly, $\text{tail}^{\mathbf{A}}$ maps $\text{nil}^{\mathbf{A}}$ to *some* integer list. This is consistent with the general modeling of partial functions in SMT-LIB as underspecified total functions — which requires a solver to consider all possible (well-sorted) ways to make a partial function total.

which is a zero wrapped in the `Int` constructor of the Any type, a result which does indeed fulfill our constraints.

Using our new grammar, we can rewrite the input expression as

$$\text{isInt?}\,(a) \land a^{\mathtt{i}} < 50$$

Notice the shorthand $a^{\mathtt{i}}$ to represent field extraction of the `i` field of the Int constructor, and the isInt function to express the constraint that a is an integer. All variables in this grammar are of Any type, thus we can assume that a has Any type.

The `Any.Int` constructor is elided in this grammar. It can be inferred from the context: when an expression is applying an operation which can only be done on an instance of Any, such as assigning a value to a variable, assume the appropriate wrapper constructor is present.

For another example, consider how to write the expression $b \land (a > 50) \lor \neg b$ using our new Any type. We will need two variables, `a` and `b`. They will both be of Any type, but `b` will need to be constrained to use the `Bool` constructor and `a` will need to be constrained to use the `Int` constructor. Thus, our expression will look like

$$\text{isInt?}\,(a) \land \text{isBool?}\,(b) \land \left(\left(a^{\mathtt{i}} > 50 \land b^{\mathtt{b}}\right) \lor \neg b^{\mathtt{b}}\right)$$

$$
\begin{aligned}
\mathit{Expr} &::= \mathit{Bool} \mid \alpha = \mathit{Any} \mid \mathit{store} = \mathit{Store} \mid \mathit{Expr} \wedge \mathit{Expr} \mid \mathit{Expr} \vee \mathit{Expr} \\
&\quad \mid \text{isInt?}\,(\mathit{Any}) \mid \text{isBool?}\,(\mathit{Any}) \mid \text{isString?}\,(\mathit{Any}) \mid \text{isReference?}\,(\mathit{Any}) \\
\mathit{Any} &::= \alpha \mid \mathit{Bool} \mid \mathit{Int} \mid \mathit{Real} \mid \mathit{String} \mid \mathit{Ref} \mid \text{Nil} \\
&\quad \mid \mathit{Pair}.\texttt{left} \mid \mathit{Pair}.\texttt{right} \\
\mathit{Bool} &::= \mathit{Any}^{\,\mathrm{b}} \mid \mathit{bool} \mid \mathit{Bool} \wedge \mathit{Bool} \mid \mathit{Bool} \vee \mathit{Bool} \mid \neg \mathit{Bool} \mid \mathit{Any} = \mathit{Any} \\
&\quad \mid \mathit{Int} < \mathit{Int} \mid \mathit{Int} \leq \mathit{Int} \mid \mathit{Int} > \mathit{Int} \mid \mathit{Int} \geq \mathit{Int} \\
\mathit{Int} &::= \mathit{Any}^{\,\mathrm{i}} \mid \mathit{int} \mid -\mathit{Int} \mid \mathit{Int} + \mathit{Int} \mid \mathit{Int} - \mathit{Int} \mid \mathit{Int} * \mathit{Int} \mid \mathit{Int}/\mathit{Int} \\
\mathit{Real} &::= \mathit{Any}^{\,\mathrm{r}} \mid \mathit{real} \\
\mathit{String} &::= \mathit{Any}^{\,\mathrm{s}} \mid \mathit{string} \mid \text{Concat}(\mathit{String}, \mathit{String}) \\
\mathit{Ref} &::= \mathit{Any}^{\,\mathrm{r}} \mid \mathit{int} \\
\mathit{Store} &::= \mathit{store} \mid \text{Empty} \mid \mathit{Store}[\mathit{Ref} \to \mathit{Pair}] \\
\mathit{Pair} &::= \text{Pair}(\mathit{Any}, \mathit{Any}) \mid \mathit{Store}[\mathit{Ref}]
\end{aligned}
$$

$$
\alpha \in \text{Names} \qquad \mathit{store} \in \text{StoreNames} \qquad \mathit{int} \in \mathbb{Z} \qquad \mathit{real} \in \mathbb{R} \qquad \mathit{string} \in \text{Strings}
$$

$$
\mathit{bool} \in \{\texttt{true}, \texttt{false}\}
$$

Figure 2.1: Simplified grammar of Z3 terms

Chapter 3

Methods

TestBuilder seeks to generate inputs to a Python function which will result in the execution of a specified line of code. By doing this, we hope to create a series of inputs which, when combined, will produce a test suite with complete line coverage.

3.1 Python

Python [61] is a well-known dynamic programming language. It is dynamically typed, although a syntax is available for type annotations [32, 33, 34]. At runtime, each value has a type. However, unlike a more strictly typed programming language, the types of the values are not typically used to determine what operations are legal.

Instead, Python idiomatically uses duck typing [38] and believes that it is "easier to ask for forgiveness than permission [39]" (EAFP). Duck typing is often expressed by the saying "if it looks like a duck, and quacks like a duck, then it is a duck." It treats the operations and fields defined on an object as more important than the concrete type of that object. This means that, as long as an operation is defined on a value at runtime, it is legal to execute it. Python does not require that a class providing a particular interface is known at compile time, only that, when run, the objects provided do, in fact, have the methods which are called on them.

3.2 Types

In contrast, as we mentioned before, Z3 requires that values be typed. Thus, we need to compare the types Python and Z3 use and determine the best mapping between them.

3.2.1 Z3 Types

Z3 provides a helpful range of supported types, along with custom data types, which are tagged unions. Of the built-in types, we use Int, Real, Bool, and String values. The Z3 documentation does not seem to make it clear what the precise semantics of its types are, but Moura and Bjørner's original paper indicates that "the SMT-LIB input format" is supported [29]. We assume, therefore, that the semantics of Z3's types are those described by the SMT-LIB standard, and experience seems to confirm this.

For the Int type, the SMT-LIB spec requires that models interpret "the sort Int as the set of all integer numbers [47]"[1] It also requires that division be implemented "according to Boute's Euclidean definition" [9] (citation in original) and that negation, addition, subtraction, and multiplication be implemented "as expected [47]". Thus, we conclude that Z3's Int type is intended to have the standard semantics of the integers, with the definitions for integer division and modulus being spelled out. Note that there are no bitwise operators defined for Int.

Similarly, for the Real type, the SMT-LIB spec requires that models interpret "the sort Real as the set of all real numbers [48]", division "as a total function that coincides with the real division function for all inputs x and y where y is non-zero [48]", and "the other function symbols of Reals as expected [48]".

[1]Recall that *sort* is the term used in logic for something like what is commonly called a *type* in computer science [7].

For booleans, the SMT-LIB spec requires the set of values to be $\{true, false\}$ [46], and that "the other function symbols of Core denote the standard Boolean operators as expected [46]"

There is not a theory for strings in SMT-LIB. The Z3 C API provides a function `Z3_mk_string_sort`, which "creates a sort for ASCII strings. Each character is 8 bits [62]." This seems to indicate that Z3 strings are sequences of ASCII characters. We support only length and concatenation on strings. The functions for those seem to support the expected semantics of those operations: the `Length` function will "Obtain the length of a sequence 's' [63]", and the C API function called to actually execute concatenation is said to "Concatenate sequences [62]."

3.2.2 Python Types

Python has an extensive set of built-in types [54]. However, the only ones of interest to us are None, integers, floats, booleans, and strings.

Python's None value is a singleton of a unique type with one value [50].

Python's Integer type provides "numbers in an unlimited range, subject to available (virtual) memory only [50]."

Python floats "represent machine-level double precision floating point numbers. You are at the mercy of the underlying machine architecture (and C or Java implementation) for the accepted range and handling of overflow [50]."

In Python, "...Boolean values behave like the values 0 and 1, respectively, in almost all contexts, the exception being that when converted to a string, the strings "False" or "True" are returned, respectively [50]."

Finally, Python strings are officially "immutable sequences of Unicode code points [54]."

3.2.3 Type Mapping

We need to represent Python types with the appropriate types from Z3, in order to be able to translate between the two languages. The mapping we chose is described below and summarized in table 3.1 on the next page.

We chose to represent Python's integers with Z3 Ints. This is the most correct type available from a theoretical standpoint, because Python integers, as noted above, are of unlimited length, and therefore no overflow effects should be observable.

Having chosen `Ints` to represent Python integers, we were forced to model Python floats with Z3 Reals. This is not ideal, as Z3 supports a true floating-point type, which should more closely align to the semantics of the Python implementation. However, there are no functions available in Z3 to translate directly between integer expressions and floating-point expressions. [2] Thus, we would be unable to intermix the types in expressions, which would prevent us from translating some expressions which are legal in Python. Z3 Reals seem like the best compromise, particularly as we do not anticipate user code relying on the differences between floating-point numbers and true reals.

Naturally, Python's Boolean values were modeled with Z3's Boolean values.

The None type is translated to a `Nil` constructor of no arguments which is present on the Z3 datatype we use as an Any type, following the pattern described in section 2.3 on page 8.

Python strings are translated to Z3's String type. Since only two operations are available for strings, concatenation and length, they are translated directly to their Z3 equivalents. As mentioned before, Z3 strings are ASCII. If a string literal is used which contains non-ASCII characters, it will crash TestBuilder. We accept this. If a

[2] There do exist functions to convert integer values to real values, and functions to convert real values to floating-point values, but some basic testing suggests there may be problems with using the composition of both, as Z3 was unable to find a solution to $a > 4$ when a was an integer value converted to a floating-point value.

Table 3.1: Mapping from Python to Z3 types.

Python type		Z3 Types
Integer	⇒	Int
Float	⇒	Real
Boolean	⇒	Bool
String	⇒	String
None	⇒	Nil

string were to somehow be constrained to contain non-ASCII characters without using a literal, TestBuilder would simply be unable to find a test case.

3.3 Cartesian Product Algorithm

We do this inference using the Cartesian product algorithm for type inference, as described by Agesen [1] and applied to Python by Cannon [11]. We restrict the types of each variable to those given by the intersection of the type constraints placed upon it by the operations executed on that variable. If no types remain, we have found a bug in the user's code. The Cartesian product type inference algorithm begins with the insight that a value can only have one concrete type in our system. Thus, while there may be several possible types for each argument to a function or an operator, when the code actually runs, each of those arguments must have a single type. It follows that the Cartesian product of the sets of possible types for each variable is a complete listing of all possible type tuples for the function involved. This allows us to determine the result type of a function or operator for each type tuple independently. In our context, we restrict the potential types of the variables based on which type tuples are implemented by the operator involved. For example, in Python, the `+` operator is defined for both integers and strings. It is not defined for an integer and a string. Thus, if we encounter code such as `a+b`, we can infer that `a` and `b` are either both strings or both integers, but `a` cannot be a string while `b` is an integer without causing a type error. Since we wish to reach a later line of code,

triggering a type error would be counter-productive. Thus, we choose only type tuples such that no type errors will occur. By doing this, we are able to reduce the number of possible cases from the square of the number of available types (currently five, for a total of 5^2 possible type tuples) to two: `a` and `b` are integers and `a` and `b` are strings.

3.4 Type System

As described in section 3.3 on the previous page, TestBuilder does Cartesian product type inference in order to handle Python's dynamic type system. In theory, we could do Cartesian product inference separately for every partial expression, returning values to Any type between operations, but by tracking the type of each subexpression, we can generate much cleaner expressions. As we will show below, this optimization is almost free.

Note that this type tracking is only necessary for optimization purposes. We do not need to track types for correctness, as Z3 internally requires that variables have consistent types throughout an expression.

Also note that the typing judgments presented below are non-standard. Rather than indicating that an expression has a type, they indicate that an expression may have a type. Because the goal of this type system is to allow solving for values which satisfy all the type constraints upon an expression, this is sufficient information.

For this reason, variables of Any type behaves as unconstrained values. This is different from the typical treatment of Any as a union of all possible types, but arises naturally when one considers the concrete types which might be present in a variable with Any type. Since any of the possible concrete types could be present, any concrete type is a satisfactory value for a variable of Any type.

$$\frac{}{i \Downarrow \text{Int}(i)} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{\text{BinOp}(e_1, \text{Add}, e_2) \Downarrow \begin{cases} v_1^i + v_2^i & \text{isInt?}\,(v_1) \wedge \text{isInt?}\,(v_2) \\ \text{Concat}(v_1^s, v_2^s) & \text{isString?}\,(v_1) \wedge \text{isString?}\,(v_2) \end{cases}}$$

Figure 3.1: Rules without typing information

Note: values are Any structures, as defined in fig. 2.1 on page 15 and the expressions are e structures, as defined in fig. 4.1 on page 31.

3.4.1 No Type System

In the completely dynamic case, we would wrap each sub-expression in an Any type, and then extract the correct type for the next operation. Simple addition, such as 1 + 2 would use the two rules shown in fig. 3.1 to generate an expression something like this:

$$\begin{cases} \text{Int}\left((\text{Int}(1))^i + (\text{Int}(2))^i\right) & \text{isInt?}\,(\text{Int}(1)) \wedge \text{isInt?}\,(\text{Int}(2)) \\ \text{String}\left(\text{Concat}\left((\text{Int}(1))^s, (\text{Int}(2))^s\right)\right) & \text{isString?}\,(\text{Int}(1)) \wedge \text{isString?}\,(\text{Int}(2)) \end{cases} \tag{3.1}$$

The case for strings might surprise you, but because `+` is the concatenation operator in Python, and we have no typing information, we are forced to generate cases for both possible types which could be in the Any values (even though those values are statically `Int` Any values).

Notice how we're forced to dynamically check the types of values we can statically determine the type of.

3.4.2 Statically Checking `Any` Variants

An obvious optimization would change the rule for addition to that in fig. 3.2 on the next page, resulting in the following expression:

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad \text{isInt?}(v_1) \qquad \text{isInt?}(v_2)}{\text{BinOp}(e_1, \text{Add}, e_2) \Downarrow \text{Int}\left(v_1^i + v_2^i\right)}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad \text{isString?}(v_1) \qquad \text{isString?}(v_2)}{\text{BinOp}(e_1, \text{Add}, e_2) \Downarrow \text{String}\left(\text{Concat}\left(v_1^s, v_2^s\right)\right)}$$

Figure 3.2: Rules for `BinOp Add` with static type checking
Values are as in fig. 3.1 on the preceding page; the isInt? and isString? functions checks at compile time that a value is an Any Int or String wrapper, respectively.

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad \text{isInt?}(v_1) \qquad \text{isInt?}(v_2) \qquad i_1 = \text{int}(v_1) \qquad i_2 = \text{int}(v_2)}{\text{BinOp}(e_1, \text{Add}, e_2) \Downarrow i_1 + i_2}$$

Figure 3.3: Rule for addition with static extraction of values from wrappers
Values and functions are as in fig. 3.2; the int function extracts an integer value from an Int wrapper at compile time.

$$\text{Int}\left((\text{Int}(1))^i + (\text{Int}(2))^i\right) \tag{3.2}$$

Notice that we are now able to omit the `isInt?` checks, because we can statically determine that the values are integers.

3.4.3 Static Expression Extraction

We might want to execute the extraction from the *Int* wrapper at compile time as well, by swapping our addition rule to that in fig. 3.3. This gives us the result expression

$$\text{Int}(1 + 2) \tag{3.3}$$

$$\frac{}{\vdash i : int} \qquad \frac{\vdash e_1 : int \qquad \vdash e_2 : int}{\vdash \mathrm{BinOp}(e_1, \mathrm{Add}, e_2) : int} \qquad \frac{\vdash e_1 : str \qquad \vdash e_2 : str}{\vdash \mathrm{BinOp}(e_1, \mathrm{Add}, e_2) : str}$$

Figure 3.4: Typechecking rule for addition

$$\frac{}{\vdash \alpha : any} \qquad \frac{\vdash e : any}{\vdash e : int}$$

Figure 3.5: Additional typechecking rules for addition with Any typed variables

3.4.4 Type Checking

From here, it is a simple observation to note that we could omit the wrapper and check the type of the subexpressions directly using the typechecking rule in fig. 3.4, obtaining this expression:

$$1 + 2$$

We started with no types and built a simple type system in order to reduce clutter in the generated expressions. Introducing variables of indeterminate type, such as function arguments, causes an interesting wrinkle. We can no longer statically check which form of addition is intended by the author, so we need to maintain two possible interpretations of the expression, one for strings and one for integers. In addition, we need to constrain the type of the variables to be in accord with the interpretation used in the expression. By adding the rule shown in fig. 3.5 to the rules in fig. 3.4, we enable type checking of expressions with Any type:

$$\begin{cases} \mathrm{Int}\left(a^i + b^i\right) & \mathrm{isInt?}\,(a) \land \mathrm{isInt?}\,(b) \\ \mathrm{String}\left(\mathrm{Concat}\left(a^s, b^s\right)\right) & \mathrm{isString?}\,(a) \land \mathrm{isString?}\,(b) \end{cases} \tag{3.4}$$

Recall that isInt and isString are Z3 constraints as in eq. (3.1) on page 22, rather than compile-time checks.

Now we have two possible expressions, and we need to carry them around together and handle both possibilities when compiling. If we are assigning the result of this expression to a variable, as in $c = a + b$, we would get the expression

$$\begin{aligned} &\left(c = \text{Int}\left(a^i + b^i\right) \wedge isInt(a) \wedge isInt(b)\right) \vee \\ &\left(c = \text{String}\left(\text{Concat}\left(a^s, b^s\right)\right) \wedge isString(a) \wedge isString(b)\right) \end{aligned} \tag{3.5}$$

Suppose, on the other hand, that the entire line was $c = 1 + (a + b)$. Now we can tell that only the integer solution will be well-typed, and we can generate the following:

$$\begin{cases} c = \text{Int}\left(1 + \left(a^i + b^i\right)\right) & isInt(a) \wedge isInt(b) \end{cases} \tag{3.6}$$

There is no other possible value. Since, as explained above, we need to avoid generating type errors, we cannot use the String interpretation.

One additional optimization is used in our code. When making an assignment such as the $c = 1 + (a + b)$ shown in eq. (3.6), we associate the variable with the set of types we are assigning into it. This allows later uses of the variable to have a more limited set of types available to them. For example, if we were to have both the line $c = 1 + (a + b)$ from eq. (3.6) and the line $e = c + d$, without any typing information for c or d, we would have to assume that, as in eq. (3.5), c could be an integer or a string. But since the previous assignment to c could only be of type int, we are able to conclude that c is an integer and the only type which $c + d$ can have is integer. Thus, we can eliminate an entire case we would otherwise have to generate.

3.5 Standard of Truth

As described in section 9.2, many concolic testing tools rely on crashes to determine the correct result of user code. In our case, we want to generate test cases for the user. Further, we would like to identify incorrect code which doesn't crash. Thus, we need a stronger standard of truth than merely "does the code crash." We chose to ask the user for the expected results of each test case. This allows us to generate test cases which are based on examples and check that the code produces the expected result. By asking the user what the expected output is, we encourage them to think about what their code is actually expected to do in a concrete context. We hope that this will help them discover even more bugs, via a similar process to rubber duck debugging [26]. In addition, even though we don't have any way to generate test cases for missing conditions, there is a possibility that, by automatically generating test cases, we will hit upon values which the user would not have considered and will expose unknown bugs.

3.6 Function Substitution

Some preliminary testing suggested that defining functions in Z3 led to significantly longer solving times than substituting their definitions into the result expressions. It would be preferable to define the functions in Z3, as this would allow proper support for recursive functions (at the cost of no longer being a decision procedure, since it seems function definitions require universal quantification). In the interest of execution speed, however, TestBuilder does substitution of called functions to a depth of one.[3] When a function is encountered which has not been substituted with its definition, either because of the depth limit or because of a lack of a definition,

[3] I.e., it substitutes known functions in the code under test, but not functions appearing within the substituted code.

TestBuilder treats the function as producing whatever return value is desirable. Note that this is not treating the function as returning an arbitrary value; to do so would result in type errors which would only be present in TestBuilder when the real function implementation does not produce some values. An alternative to this would be to refuse to generate tests for functions with missing definitions, but, due to the limitation on substitution depth, this would result in TestBuilder being unable to generate test cases for recursive functions.

3.7 Preprocessor

TestBuilder provides support for a `Pair` class which stores two values. This `Pair` has two attributes: `left` and `right`. Student code, however, may use internally-consistent but different names for its equivalent data structures. In order to support testing such code, we provide a preprocessor. This preprocessor is able to rewrite parts of the AST of the user's code before test case generation, in order to adjust it to the configuration used by TestBuilder. For example, the user may define a rewrite rule which specifies that any attribute named "first" is to be rewritten to "left", and that any attribute named "rest" is to be rewritten to "right." Adding these as preprocessor rules means that the preprocessor will rewrite the code before tests are generated.

One potential problem exists: if the user used two classes in their code which were both mapped to pairs, TestBuilder will be unable to distinguish between the two. This could lead to incorrect results if the original code would have crashed due to an error where it treated the same value as both classes. TestBuilder will analyze the code as though there was one type present. This is a problem in one sense, but unless the programmer's expectation of multiple types was made clear through the use of the `type` function, it may not be possible to confirm that the code would not work as expected. This is due to the potential for a user to provide the

code with an object with answers to all the methods which are called on it, perhaps by subclassing both the classes the value is treated as. Such as subclass would pass even `isinstance` tests. See fig. 3.6 on the following page for an example.

As a result, the preprocessor should be viewed as a stop-gap measure designed to increase compatibility with user code, rather than as a permanent solution. It does not map attributes semantically, changing the internal name of the attribute in TestBuilder. Rather, it modifies code on-the-fly to follow TestBuilder's expected conventions, without permanent changes, allowing semantically-equivalent constructions to be unified to the format supported by TestBuilder. Ultimately, support for custom objects would eliminate the need for this preprocessor.

While TestBuilder generates tests using its built-in Pair class, the Pair class has been designed with many common names available as aliases to its own attributes. Thus, it is able to be used in contexts where the user's Pair class used different names than TestBuilder.

Code

```
class TypeA:
    def __init__(self, a):
        self.a = a


class TypeB:
    def __init__(self, b):
        self.b = b


def uses_both(val):
    if isinstance(val, TypeA):
        print(val.a)
    if isinstance(val, TypeB):
        print(val.b)


class TypeBoth(TypeA, TypeB):
    def __init__(self, a, b):
        super().__init__(a)
        self.a = a
        self.b = b


val = TypeBoth(1, 2)
uses_both(val)
```

Result

```
python3 multireference.py
```

1
2

Figure 3.6: Using a variable as both of its parent types.

Chapter 4

Model

4.1 Overview

We convert a Python AST into a loop-free SSA [42, 2] IR, adding phi nodes at join points and substituting functions with their definitions. The resulting IR grammar is defined in fig. 4.1 on the next page.

4.1.1 SSA IR Reduction

Reduction on the IR is defined by **v**, as defined in fig. 4.2 on page 32 using **e** as defined in fig. 4.3 on page 33. **v** is a mapping from a basic block, a list of basic blocks, a store (σ), and an environment (ρ) to a basic block, a list of basic blocks, a store, and an environment. **e** is a mapping from an expression, a store (σ), and an environment (ρ) to modified versions. Note that the environment is never modified by an expression, although the store may be. This occurs when a Pair is created, which requires allocation from the store.

TestBuilder takes a program in this SSA IR form, along with a desired line number to execute, and computes a set of inputs such that the specified line will be executed when the inputs are provided to the function.

$$
\begin{aligned}
\mathit{BasicBlock} &::= \text{BB}([\mathit{Statement}, \dots], \mathit{next}) \mid \text{Result}(v) \\
\mathit{Statement} &::= \text{Set}(\mathit{id}, e) \mid \text{Assert}(e) \mid \text{Setattr}(e, \mathit{access}, e) \\
\mathit{next} &::= \text{Next}(i) \mid \text{ReturnExpr}(e) \mid \text{Return} \mid \text{Conditional}(e, i, i) \\
e &::= v \mid \text{BinOp}(e, op, e) \mid \text{USub}(e) \mid (e) \mid \alpha \\
&\quad\mid \text{Attr}(e, \mathit{access}) \\
v &::= i \mid s \mid b \mid f \mid \text{None} \mid \text{Pair}(e, e) \\
op &::= \texttt{Add} \mid \texttt{Sub} \mid \texttt{Mult} \mid \texttt{Div} \mid \texttt{Lt} \mid \texttt{LtE} \mid \texttt{Gt} \mid \texttt{GtE} \mid \texttt{Eq} \mid \texttt{And} \mid \texttt{Or} \\
\mathit{access} &::= \texttt{left} \mid \texttt{right}
\end{aligned}
$$

$$
\alpha \in \mathit{id} \qquad i \in \mathbb{Z} \qquad s \in \mathit{longstringitem}^* \qquad b \in \{\texttt{true}, \texttt{false}\} \qquad f \in \text{Float}
$$

Figure 4.1: Grammar of SSA IR

Definitions:
id is the set of valid Python identifiers, as defined by [51].
longstringitem is the set of valid characters in a Python triple-quoted string, as defined by [52].
Next and Conditional use indices into the list of basic blocks to indicate the block to jump to.

4.1.2 Compilation of SSA IR to Z3 Expressions

During compilation, the enclosing function for the specified line is converted into the language of the Z3 SMT solver. Z3 is then used to solve the resulting expression, and the variables representing the function arguments are extracted. A simplified grammar of Z3 expressions was presented in fig. 2.1 on page 15 and will be reused here.

For the purposes of exposition in this section, all sub-expressions have been treated as being executed and converted immediately back into Z3 Any types. This leads to many small assignments and avoids the need to model a type system. In addition, this means that larger expressions become a chain of assignments, all Anded together.

Unlike the model, the actual implementation uses a type system, as described in section 3.4 on page 21.

$$\langle [\beta_1, \dots], [], [], [] \rangle \ \mathbf{v} \ \langle \beta_1, [\beta_1, \dots], [], [] \rangle$$

$$\langle \text{BB}([\text{Set}(id, e), s, \dots], next), \beta, \sigma, \rho \rangle \ \mathbf{v} \ \langle \text{BB}([s, \dots], next), \beta, \sigma, \rho[id \to v] \rangle$$
$$\text{if } \langle e, \sigma, \rho \rangle \ \mathbf{e} \ \langle v, \sigma_1, \rho \rangle$$

$$\langle \text{BB}([\text{Setattr}(t, \texttt{left}, e), s, \dots], next), \beta, \sigma, \rho \rangle \ \mathbf{v} \ \langle \text{BB}([s, \dots], next), \beta, \sigma_2[r \to v], \rho \rangle$$
$$\text{if } \begin{array}{l} \langle e, \sigma, \rho \rangle \ \mathbf{e} \ \langle v, \sigma_1, \rho \rangle \\ \langle t, \sigma_1, \rho \rangle \ \mathbf{e} \ \langle tv, \sigma_2, \rho \rangle \end{array}$$

$$\langle \text{BB}([\text{Setattr}(t, \texttt{right}, e), s, \dots], next), \beta, \sigma, \rho \rangle \ \mathbf{v} \ \langle \text{BB}([s, \dots], next), \beta, \sigma_2[r \to v], \rho \rangle$$
$$\text{if } \begin{array}{l} \langle e, \sigma, \rho \rangle \ \mathbf{e} \ \langle v, \sigma_1, \rho \rangle \\ \langle t, \sigma_1, \rho \rangle \ \mathbf{e} \ \langle tv, \sigma_2, \rho \rangle \end{array}$$

$$\langle \text{BB}([\text{Assert}(e), s, \dots], next), \beta, \sigma, \rho \rangle \ \mathbf{v} \ \langle \text{BB}([s, \dots], next), \beta, \sigma_1, \rho \rangle$$
$$\text{if } \langle e, \sigma, \rho \rangle \ \mathbf{e} \ \langle \texttt{true}, \sigma_1, \rho \rangle$$

$$\langle \text{BB}([\text{Assert}(e), s, \dots], next), \beta, \sigma, \rho \rangle \ \mathbf{v} \perp \text{ if } \langle e, \sigma, \rho \rangle \ \mathbf{e} \ \langle \texttt{false}, \sigma_1, \rho \rangle$$

$$\langle \text{BB}([], \text{Next}(i)), \beta, \sigma, \rho \rangle \ \mathbf{v} \ \langle \beta_i, \beta, \sigma, \rho \rangle$$

$$\langle \text{BB}([], \text{ReturnExpr}(e)), \beta, \sigma, \rho \rangle \ \mathbf{v} \ \langle \text{Result}(v), \beta, \sigma_1, \rho \rangle$$
$$\text{if } \langle e, \sigma, \rho \rangle \ \mathbf{e} \ \langle v, \sigma_1, \rho \rangle$$

$$\langle \text{BB}([], \text{Return}), \beta, \sigma, \rho \rangle \ \mathbf{v} \ \langle \text{Result}(\texttt{None}), \beta, \sigma, \rho \rangle$$

$$\langle \text{BB}([], \text{Conditional}(e, i, j)), \beta, \sigma, \rho \rangle \ \mathbf{v} \ \langle \beta_i, \beta, \sigma_1, \rho \rangle \text{ if } \langle e, \sigma, \rho \rangle \ \mathbf{e} \ \langle \texttt{true}, \sigma_1, \rho \rangle$$

$$\langle \text{BB}([], \text{Conditional}(e, i, j)), \beta, \sigma, \rho \rangle \ \mathbf{v} \ \langle \beta_j, \beta, \sigma_1, \rho \rangle$$
$$\text{if } \langle e, \sigma, \rho \rangle \ \mathbf{e} \ \langle \texttt{false}, \sigma_1, \rho \rangle$$

Figure 4.2: Reduction of SSA IR

Note: Values are as defined in fig. 4.1 on the preceding page; **e** is defined in fig. 4.3 on the next page. See text for description of **v**, **e**, σ and ρ.

$$
\begin{aligned}
&& \langle left, \sigma, \rho\rangle \,\mathbf{e}\, \langle lv, \sigma_1, \rho\rangle \wedge \\
\langle Pair(left, right), \sigma, \rho\rangle \,\mathbf{e}\, \langle k, \sigma_2[k \rightarrow Pair(lv, rv)], \rho\rangle &\text{ if }& \langle right, \sigma_1, \rho\rangle \,\mathbf{e}\, \langle rv, \sigma_2, \rho\rangle \wedge \\
&& k \text{ is a fresh index} \\
\langle \alpha, \sigma, \rho\rangle \,\mathbf{e}\, \langle \sigma[\alpha], \sigma, \rho\rangle && \\
\langle \text{BinOp}(l, op, r), \sigma, \rho\rangle \,\mathbf{e}\, \langle \delta(op, lv, rv), \sigma_2, \rho\rangle &\text{ if }& \langle l, \sigma, \rho\rangle \,\mathbf{e}\, \langle lv, \sigma_1, \rho\rangle \wedge \\
&& \langle r, \sigma_1, \rho\rangle \,\mathbf{e}\, \langle rv, \sigma_2, \rho\rangle \\
\langle \text{USub}(e), \sigma, \rho\rangle \,\mathbf{e}\, \langle \delta(-, v), \sigma_1, \rho\rangle &\text{ if }& \langle e, \sigma, \rho\rangle \,\mathbf{e}\, \langle v, \sigma_1, \rho\rangle \\
\langle \ (e), \sigma, \rho\rangle \,\mathbf{e}\, \langle \delta(\texttt{not}, v), \sigma_1, \rho\rangle &\text{ if }& \langle e, \sigma, \rho\rangle \,\mathbf{e}\, \langle v, \sigma_1, \rho\rangle \\
\langle \text{Attr}(e, \texttt{left}), \sigma, \rho\rangle \,\mathbf{e}\, \langle l, \sigma_1, \rho\rangle &\text{ if }& \langle e, \sigma, \rho\rangle \,\mathbf{e}\, \langle \text{Pair}(l, r), \sigma_1, \rho\rangle \\
\langle \text{Attr}(e, \texttt{right}), \sigma, \rho\rangle \,\mathbf{e}\, \langle r, \sigma_1, \rho\rangle &\text{ if }& \langle e, \sigma, \rho\rangle \,\mathbf{e}\, \langle \text{Pair}(l, r), \sigma_1, \rho\rangle
\end{aligned}
$$

Figure 4.3: Reduction of SSA IR expressions

Note: δ is defined in fig. 4.4. Values are as defined in fig. 4.1 on page 31. See text for description of **v**, **e**, σ and ρ.

$$
\begin{aligned}
\delta(\texttt{Add}, \ulcorner m \urcorner, \ulcorner n \urcorner) &= \ulcorner m + n \urcorner \\
\delta(\texttt{Sub}, \ulcorner m \urcorner, \ulcorner n \urcorner) &= \ulcorner m - n \urcorner \\
\delta(\texttt{Mult}, \ulcorner m \urcorner, \ulcorner n \urcorner) &= \ulcorner m \cdot n \urcorner \\
\delta(\texttt{Div}, \ulcorner m \urcorner, \ulcorner n \urcorner) &= \ulcorner m / n \urcorner \\
\delta(\texttt{Lt}, \ulcorner m \urcorner, \ulcorner n \urcorner) &= \ulcorner m < n \urcorner \\
\delta(\texttt{LtE}, \ulcorner m \urcorner, \ulcorner n \urcorner) &= \ulcorner m \leq n \urcorner \\
\delta(\texttt{Gt}, \ulcorner m \urcorner, \ulcorner n \urcorner) &= \ulcorner m > n \urcorner \\
\delta(\texttt{GtE}, \ulcorner m \urcorner, \ulcorner n \urcorner) &= \ulcorner m \geq n \urcorner \\
\delta(\texttt{Eq}, \ulcorner m \urcorner, \ulcorner n \urcorner) &= \ulcorner m = n \urcorner \\
\delta(\texttt{And}, \ulcorner m \urcorner, \ulcorner n \urcorner) &= \ulcorner m \wedge n \urcorner \\
\delta(\texttt{Or}, \ulcorner m \urcorner, \ulcorner n \urcorner) &= \ulcorner m \vee n \urcorner
\end{aligned}
$$

Figure 4.4: Definition of δ for SSA IR

$\ulcorner \cdot \urcorner$ denotes the value resulting from the expression inside.

$\epsilon [\![x]\!]$ is a function mapping the location tag $[\![x]\!]$ to a variable. At times, this location tag will be represented with Python syntax, which is to be understood to refer to the tag on that expression after conversion to SSA form.

The $\gamma(\alpha)$ function maps from a Python name to a Z3 name. To distinguish variable names deriving from Python from those added by TestBuilder, the implementation adds a prefix of `pyname_` to each of the original variables when converting to the SSA IR. $\gamma(\alpha)$ may be considered to add such a prefix.

Compilation of reference types into the SMT domain requires a store, in order to handle aliasing of references. If we model a mutable reference type without a store, mutation of the referenced value in one aliased variable will not mutate the value referenced by its alias. For the current version of TestBuilder, we have only implemented support for pairs, although it should be possible to extend to tuples of other lengths.

The store is modeled as a Z3 array indexed by an integer i. Allocation of store indices occurs statically at compile time for variables with store accesses: those where one of the subscripts is referenced. Variables for which no type restriction prevents them from being references, but which also do not have store accesses, will not have store locations allocated. As the store is never accessed, this does not affect the result of the computation.

We begin with function compilation. Function compilation is the result of compiling the first basic block in the function and following labels from it to other basic blocks, which are then also compiled.

$$C_{Fn}([\beta_1, \dots, \beta_n] = C_{BB}(\beta_1, [\beta_1, \dots, \beta_n]) \tag{4.1}$$

Several forms of basic block exist. We can have a simple code block, which also holds the label of the next block:

$$C_{BB}(\text{BB}(stmts, \text{Next}(j)), [\beta_1, \ldots, \beta_n]) = \begin{array}{c} C_{Body}(stmts) \\ \wedge C_{BB}(\beta_j, [\beta_1, \ldots, \beta_n]) \end{array} \tag{4.2}$$

Or we can have a return expression block, which terminates the function and returns a value:

$$C_{BB}(\text{BB}(stmts, \text{ReturnExpr}(e)), [\beta_1, \ldots, \beta_n]) = C_{Body}(stmts) \wedge (ret = \epsilon [\![e]\!]) \wedge C_E(e) \tag{4.3}$$

Or the simple return block, which terminates the function and does not return a value:

$$C_{BB}(\text{BB}(stmts, \text{Return}), [\beta_1, \ldots, \beta_n]) = C_{Body}(stmts) \tag{4.4}$$

Finally, we may have a conditional block, which evaluates an expression and goes to one of two labels depending on its value. The conditional block is converted into a disjunction, with one expression for each of its branches:

$$\begin{aligned} C_{BB}(\text{BB}(stmts, \text{Conditional}(e, i, j)), [\beta_1, \ldots, \beta_n]) = & \\ & C_{Body}(stmts) \\ & \wedge C_E(e) \\ & \wedge \left(\begin{array}{l} (\epsilon [\![e]\!] \wedge C_{BB}(\beta_i, [\beta_1, \ldots, \beta_n])) \\ \vee \left(\neg \epsilon [\![e]\!] \wedge C_{BB}(\beta_j, [\beta_1, \ldots, \beta_n])\right) \end{array} \right) \end{aligned} \tag{4.5}$$

To compile the body of a basic block, we form a conjunction of all the assignment statements in it, along with the assertions in it:

$$C_{Body}(stmts) = \bigwedge_{set \in stmts} C_{Set}(set) \wedge \bigwedge_{Assert(e) \in stmts} C_E(e) \wedge \epsilon [\![e]\!] \tag{4.6}$$

The right hand side of each assignment statement is compiled:

$$C_{Set}(\mathrm{Set}(\alpha, e)) = (\alpha = \epsilon [\![e]\!] \wedge C_E(e)) \tag{4.7}$$

Compilation of simple values proceeds as expected: Integers, strings, and booleans all compile to their corresponding Z3 objects:

$$C_E(i) = (\epsilon [\![i]\!] = i) \tag{4.8}$$

$$C_E(s) = (\epsilon [\![s]\!] = s) \tag{4.9}$$

$$C_E(b) = (\epsilon [\![b]\!] = b) \tag{4.10}$$

$$C_E(\alpha) = (\epsilon [\![\alpha]\!] = \gamma(\alpha)) \tag{4.11}$$

$$\alpha \in Strings$$

Binary operators compile to the equivalent operations in Z3, as shown in fig. 4.5 on the following page.

In effect, we compile each sub-expression to an assignment to a variable determined by a call to ϵ with the code location. This assignment is Anded with any constraints required to make it valid — such as a requirement that the left and right values both be integers — and then the resulting expression is Ored with any alternative expressions resulting from other interpretations of the component parts.

$$C_E(\text{BinOp}(l,+,r)) =$$
$$\bigvee \begin{bmatrix} \left(\epsilon\,[\![\text{BinOp}(l,+,r)]\!] = \epsilon\,[\![l]\!]^{\text{i}} + \epsilon\,[\![r]\!]^{\text{i}}\right) \wedge \text{isInt?}\,(\epsilon\,[\![l]\!]) \wedge \text{isInt?}\,(\epsilon\,[\![r]\!]) \\ \left(\epsilon\,[\![\text{BinOp}(l,+,r)]\!] = \text{Concat}(\epsilon\,[\![l]\!]^{\text{s}}, \epsilon\,[\![r]\!]^{\text{s}})\right) \wedge \text{isString?}\,(\epsilon\,[\![l]\!]) \wedge \text{isString?}\,(\epsilon\,[\![r]\!]) \end{bmatrix}$$
$$\wedge\, C_E(l) \wedge C_E(r) \tag{4.12}$$

Figure 4.5: Compilation of binary operator.

4.2 Examples

4.2.1 Identity Function

Consider the identity function defined in Python:

```
def ident(x):
    return x
```

Translating this function into the SSA IR results in the following list of basic blocks:

$$[\text{BB}([\,], \text{ReturnExpr}(x))] \tag{4.13}$$

Conversion of this SSA IR to an expression in the Z3 language is very direct:

$$(ret = \epsilon\,[\![x]\!]) \wedge (\epsilon\,[\![x]\!] = \gamma(x)) \tag{4.14}$$

This is an exceptionally simple example, as no typechecking need occur for this assignment. In the next example, we will show how the conversion from dynamically-typed Python to statically typed Z3 is achieved.

4.2.2 Basic Expression in Function

We now demonstrate the conversion of the following simple function into a Z3 expression:

```
def twothings(a, b):
    return a + b
```

Translating this function into the SSA IR results in the following expression:

$$[\text{BB}([], \textit{ReturnExpr}(\textit{BinOp}(a, \texttt{Add}, b)))] \tag{4.15}$$

We now begin the conversion process to Z3 expressions. This proceeds in a depth-first fashion. Thus, we first convert both the variables, then convert the addition operation on them. The variable conversions looks like this:

$$(\epsilon [\![a]\!] = \gamma(a)) \land (\epsilon [\![b]\!] = \gamma(b)) \tag{4.16}$$

Each Python variable is converted to a Z3 constant of *Any* type. *Any* is a Z3 datatype which defines wrapper constructors for each possible datatype in Python. This allows us to represent a value for which the current type constraints are not sufficiently strong to restrict it to a single type. Abstract interpretation across the possible types of an input expression could in theory produce enormous expressions: every element of the Cartesian product of the possible types of the input variables could need to be represented. But because assignments are always into variables of *Any* type, every expression will be reduced to a constant number of potential value accesses from a variable. Thus, no exponential growth in the number of expressions occurs.

Having converted the variables, we now convert the addition operation:

$$\begin{bmatrix} \begin{pmatrix} \left(\epsilon [\![a+b]\!] = \epsilon [\![a]\!]^{\mathrm{i}} + \epsilon [\![b]\!]^{\mathrm{i}}\right) \\ \wedge\, \text{isInt?}\left(\epsilon [\![a]\!]\right) \wedge \text{isInt?}\left(\epsilon [\![b]\!]\right) \end{pmatrix} \\ \vee \begin{pmatrix} \left(\epsilon [\![a+b]\!] = \text{Concat}\left(\epsilon [\![a]\!]^{\mathrm{s}}, \epsilon [\![b]\!]^{\mathrm{s}}\right)\right) \\ \wedge\, \text{isString?}\left(\epsilon [\![a]\!]\right) \wedge \text{isString?}\left(\epsilon [\![b]\!]\right) \end{pmatrix} \end{bmatrix} \\ \wedge \left(\epsilon [\![a]\!] = \gamma(a)\right) \wedge \left(\epsilon [\![b]\!] = \gamma(b)\right) \tag{4.17}$$

We have now processed the entire expression which is supposed to be returned. Thus, we now produce a non-deterministic return of the possible values from the result. Each expression is wrapped in the constructor for the *Any* wrapper of its type:

$$ret = \epsilon [\![a+b]\!] \wedge (\text{eq. } (4.17)) \tag{4.18}$$

4.2.2.1 What about passing type sets down?

Rather than initially allowing variables to take on any type and then restricting the types according to the definition of the operation, it might seem we could restrict the variables to the types for which the operation is defined. This would reduce the number of Cartesian products required, as it would reduce the number of potential types for the variable. However, it would not eliminate the potential need for further restriction on the available variable types. Consider the case where one of the two arguments to the addition operator is an integer literal. Then it must have type *int*. If we pass the possible argument types from the addition operation to the arguments, we will pass both *int* and *str* to each argument. But the instantiation of $+$ as $1 + b^{\mathrm{s}} : b : str$ is impossible in this case, as 1 will never be a string; thus, the formerly-unconstrained variable b can now be constrained to *int*. To do this, we still require the Cartesian product algorithm, as described above. Thus, restricting the

initial instantiation types of the variables falls into the category of optimizations, rather than alternative solutions.

4.2.3 Conditional

Consider this code:

```
def conditional(a, b):
    if a < 3:
        c = a + 2
    else:
        c = a + 3
    return c
```

Translating this to SSA IR results in this:

$$\begin{aligned}[\,&\text{BB}([\,],\text{Conditional}(a<3,2,3)),\\ &\text{BB}([c=a+2],\text{Next}(4)),\\ &\text{BB}([c=a+3],\text{Next}(4)),\\ &\text{BB}([\,],\text{ReturnExpr}(c))]\end{aligned} \tag{4.19}$$

Expression evaluation over $\text{BB}(\dots)$ proceeds as standard:

$$\epsilon\,[\![a]\!] = \gamma(a) \tag{4.20}$$

and the addition:

$$\begin{aligned}&\left(\epsilon\,[\![a+2]\!] = \epsilon\,[\![a]\!]^{\text{i}} + 2\right) \wedge \text{isInt?}\,(\epsilon\,[\![a]\!])\\ \wedge\ &(\epsilon\,[\![a]\!] = \gamma(a))\end{aligned} \tag{4.21}$$

Note that, because we are adding an integer to a, the only valid interpretation of a is as an *int*, so we only have one possible expression.

The compilation of the test expression proceeds as follows:

$$\left(\epsilon\,[\![a < 3]\!] = \left(\epsilon\,[\![a]\!]^{\mathrm{i}} < 3\right)\right) \wedge \text{isInt?}\,(\epsilon\,[\![a]\!]) \wedge (\epsilon\,[\![a]\!] = \gamma(a)) \tag{4.22}$$

Compilation of the conditional to Z3 results in the following:

$$\begin{aligned}
&\begin{pmatrix} \epsilon\,[\![a < 3]\!] \wedge \gamma(c) = \epsilon\,[\![a + 2]\!] \vee \\ \neg\epsilon\,[\![a < 3]\!] \wedge \gamma(c) = \epsilon\,[\![a + 3]\!] \end{pmatrix} \\
&\wedge\ \epsilon\,[\![a + 2]\!] = \epsilon\,[\![\text{BinOp}(a, \texttt{Add}, 2)]\!] \\
&\wedge\ \epsilon\,[\![a + 3]\!] = \epsilon\,[\![\text{BinOp}(a, \texttt{Add}, 3)]\!] \\
&\wedge \left(\epsilon\,[\![a < 3]\!] = \left(\epsilon\,[\![a]\!]^{\mathrm{i}} < 3\right)\right) \wedge \text{isInt?}\,(\epsilon\,[\![a]\!]) \\
&\wedge\ (\epsilon\,[\![a]\!] = \gamma(a))
\end{aligned} \tag{4.23}$$

Finally, compilation of the return statement yields

$$ret = \epsilon\,[\![\gamma(c)]\!] \tag{4.24}$$

4.2.4 Pairs and Mutation

Support for a Pair datatype is present. Consider this Python code:

```
a = Pair(1, 2)

b = a

a.left += 1

assert b.left == 2
```

Translating the Python above into SSA IR results in the following list of expressions:

$$\begin{aligned}
&[\text{BB}([\\
&\quad \text{Set}(a, \text{Pair}(1,2)),\\
&\quad \text{Set}(b, a),\\
&\quad \text{Set}(\text{Attr}(a, \texttt{left}),\ Attr(a, \texttt{left}) + 1)\\
&\quad \text{Assert}(\text{Attr}(b, \texttt{left}) = 2)\\
&])]
\end{aligned} \tag{4.25}$$

We compile the pair creation into an assignment of the store index to the *a* variable and a mutation of the store array to assign a new pair to that index. The pair itself is a Z3 datatype.

$$\begin{aligned}
&(\epsilon [\![a]\!] = 0)\\
&\wedge (store_1 = \text{Store}\,(store, 0, \epsilon [\![Pair(1,2)]\!]))\\
&\wedge (\epsilon [\![Pair(1,2)]\!] = \text{Pair}\,(\epsilon [\![1]\!], \epsilon [\![2]\!]))\\
&\wedge (\epsilon [\![1]\!] = 1)\\
&\wedge (\epsilon [\![2]\!] = 2)
\end{aligned} \tag{4.26}$$

The assignment into the left field of the *a* variable is converted into a replacement of the pair currently in the store with a new pair which has been mutated

appropriately. It is worth noting that the value in a does not change, because only the referenced value changes.

$$
\begin{aligned}
&\text{isReference?}\left(\epsilon\left[\!\left[a\right]\!\right]\right)\\
&\wedge\epsilon\left[\!\left[a\right]\!\right] = \gamma(a)\\
&\wedge\epsilon\left[\!\left[a.left\right]\!\right] = \left(store[\epsilon\left[\!\left[a\right]\!\right]^{\mathrm{r}}]\right).left \wedge \text{isReference?}\left(\epsilon\left[\!\left[a\right]\!\right]\right)\\
&\wedge\epsilon\left[\!\left[a.left+1\right]\!\right] = \epsilon\left[\!\left[a.left\right]\!\right] + 1 \wedge \text{isInt?}\left(\epsilon\left[\!\left[left_1\right]\!\right]\right)\\
&\wedge\left(store_2 = \text{Store}\left(\begin{array}{l} store_1, \epsilon\left[\!\left[a\right]\!\right]^{\mathrm{r}}, \\ \text{Pair}\left(\begin{array}{l}\epsilon\left[\!\left[a.left+1\right]\!\right], \\ \left(store[\epsilon\left[\!\left[a\right]\!\right]^{\mathrm{r}}]\right).right \\ \wedge\ \text{isReference?}\left(\epsilon\left[\!\left[a\right]\!\right]\right)\end{array}\right)\end{array}\right)\right)
\end{aligned}
\tag{4.27}
$$

As compilation occurs, the current $store_n$ variable is tracked, and the appropriate one is written. Because all loops are unrolled, the current store can be determined statically.

Chapter 5

Evaluation

In order to determine the effectiveness of TestBuilder, three evaluations were undertaken. For all of these assignments, a “golden master” implementation of the assignment was used which was produced by the author. In practice, the “golden master” might be the implementation created by the instructor in developing the assignment.

Mutation testing is a technique for evaluating the quality of a test suite. It involves modifying the code of a program to deliberately insert errors, and then counting how many of the errors are caught by the test suite. For the evaluations conducted in this thesis, a mutation testing tool was written which implements a variant of the deletion mutation operator presented by Deng, Offutt, and Li [15].

For testing purposes, we used a collection of 347 anonymous student projects from the Spring 2017 edition of CSC 202 (officially 103 at the time). We ran TestBuilder against each project, and the ones which successfully ran without crashes were gathered as the sample code for this evaluation.

The cases in which TestBuilder does not run are caused by several factors. We gathered the output from the failing runs, and table 5.1 on the next page presents a table of the top three most common final lines of the output, along with a probable cause of the error.

Table 5.1: Top three most common final lines of output from failed TestBuilder runs.

Count	Final line of output
74	RuntimeError: Don't know what to do with a For(<class '_ast.For'>); no such attribute exists *TestBuilder does not support Python* For (*see chapter 7 on page 52*)
37	Timeout! *TestBuilder was stopped if it ran for more than 5 minutes on a file.*
11	RuntimeError: Unknown target type <class '_ast.Call'> *These appear to be assignments to tuples. The* Call *in the error is due to the pre-processor we applied, which converts Python tuples into Pair instances, including when they are the targets of assignments.*

5.1 Grade Boosting

We ran TestBuilder on submitted student code with the goal of discovering bugs which were also found by the professor's grading test suite. To simulate a professor's grading system, we used the test suite from our "golden master" implementation of the students' project.

We began with 347 student projects. Only 343 of those had a linked_list.py file, which was our target file for this experiment. Due to limitations or bugs in the current implementation of TestBuilder, not all student submissions could be processed without crashes. We removed 185 student submissions which caused TestBuilder to crash while attempting to generate tests. We ran TestBuilder on the remaining 158 student submissions, providing correct answers to its queries by running our golden master implementation on the requested inputs. This simulates a student giving correct answers to the questions posed by TestBuilder. We run the resulting test cases against the student's code and find any failing tests. We then run the grading test suite against the same code and compare the bugs it finds. Each bug found by both TestBuilder and the grading suite represents an opportunity for the student to lose fewer points by running TestBuilder on their code.

We compared the bugs found by our grading test suite and TestBuilder. In those cases where both test suites found the same bug, we can conclude that running TestBuilder on the student's code could have saved them the points they lost due to the bugs found by our simulated grading suite. This quantifies TestBuilder's direct benefit to the student. Potentially, this kind of test could also be run on submissions regularly, with the results reported upon assignment submission.[1]

To implement this technique in an educational setting would require the interfaces to the tested methods to exist in both the student's and professor's code, but as this limitation already exists with grading test suites it is not expected to pose a problem.

5.2 Student Tests vs. TestBuilder Tests

In order to compare the quality of hand-written tests written by students and test cases generated by TestBuilder, we ran a mutation testing procedure on each of ten student linked list projects. We selected the student projects by shuffling the list of projects on which TestBuilder ran successfully and selecting the top ten working projects from the shuffled list. If the student's tests did not run as submitted, we omitted that project and chose the next project on the shuffled list.

For one run of the mutation testing, we used the TestBuilder generated test suite to attempt to kill the mutants. For the other run, we used the student's handwritten tests to do the same. This allowed us to compare the relative performance of the two test suites in killing mutants.

[1] For example, as a message of the form "If you use TestBuilder, you could find errors that may lose you five points!"

5.3 Refactoring Legacy Code

Rather than attempting to write tests around legacy code, TestBuilder can be used to canonize[2] the current behavior of the program in order to prepare for refactoring. To test its effectiveness in generating a test suite for code, a test suite was generated for the golden master program, treating the current behavior of the code as correct. Mutation testing was then applied to each program, and the number of mutations not caught by the test suite was counted. This number, as a fraction of the total number of mutations generated, quantifies the performance of the generated test suite in catching potential regressions. This is a valid measurement, as every mutation is by definition a potential regression from the previous behavior of the program.

[2]To make canonical [14]; c.f. *differences - "Normalization" vs. "canonicalization" - English Language & Usage Stack Exchange* [16]

Chapter 6

Results

6.1 Grade Boosting

We ran the tests written for the golden master across all the student projects in our list of 158 working student projects. We then gathered the 22 projects which failed the golden master's tests,[1] and ran TestBuilder on each of them, using the golden master linked list implementation to determine expected outputs. Only three of the resulting test cases failed (two from one project and one from another), and all of them failed due to a difference in implementation of the empty list between the project they were testing and the golden master. Thus, they would not have failed if the user had given the correct expected output. See fig. 6.1 on the next page for a table of project counts.

6.2 Student Tests vs. TestBuilder Tests

We shuffled the list of student projects which work with TestBuilder and selected projects from the top of the list until we had ten which had both tests and student code and worked well with TestBuilder. We then ran mutation tests on both sets of

[1]This does not include projects which broke the test runner. Typically, this seemed to happen due to missing functions which the golden master tests expected.

347	projects
343	with `linked_list.py`
185	crash TestBuilder
158	build successfully
22	failed golden master tests
2	failed master-driven TestBuilder tests

Figure 6.1: Project breakdown

tests: the tests written by the students as part of the course requirements and the tests generated by TestBuilder. The percentage of mutations which were caught by each test suite was calculated for each project, and the results may be seen summarized in fig. 6.2 on the following page.

A Shapiro-Wilk normality test was performed on the results for each test suite, but the results were not normally distributed ($p < 0.0002$). A Wilcoxon signed-rank test was used to determine that the mean fraction of mutations[2] killed by each test suite differed ($p < 0.002$) with a 95% confidence interval of 24 to 39 percentage points difference. We conclude that the tests written by students are better at killing mutants than those written by TestBuilder, at least in its current configuration.

Python includes a built-in `compile` function which catches some basic errors in code and converts it into a Python code object [53].[3] To confirm that TestBuilder tests were doing more than merely eliminating the mutations with such basic errors, we gathered data on the total number of mutations which were eliminated by failure to compile. A Wilcoxon signed-rank test was used to confirm that the mean fraction of mutations killed by `compile` differed from the mean fraction of mutations killed by running the TestBuilder tests ($p < 0.002$) with a 95% confidence interval of 36 to 49 percentage points difference. We conclude that TestBuilder tests are better than

[2] I.e., for each project, we divide the number of mutations killed by the total number of mutations, producing a percentage. The test is on the mean of these percentages.

[3] I believe this is similar to the operations which are executed when a file is loaded by the Python interpreter.

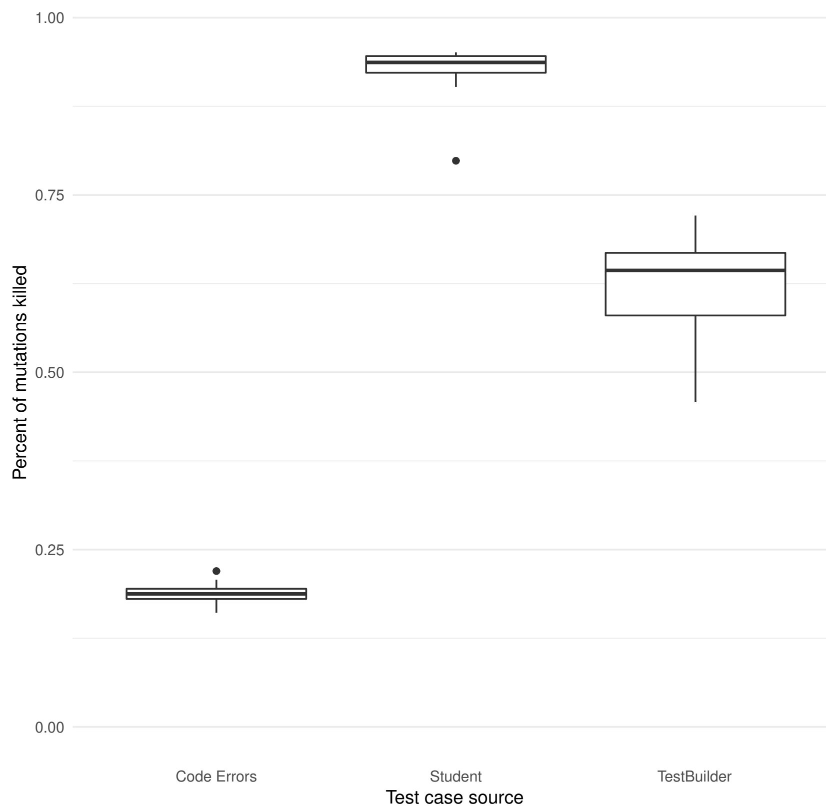

Figure 6.2: Mutation killing performance of student and TestBuilder tests.

merely running `compile` on the code.[4] The number of mutations eliminated by the `compile` function is shown in fig. 6.2 under the name "Code Errors".

While reviewing these results, we wondered what effect the lack of statement coverage by TestBuilder was having on the results. We found that the median coverage of the TestBuilder test suites used in this section was 78%, whereas the median coverage of the students' test suites was 100%.

6.3 Refactoring Legacy Code

We ran a mutation test using the generated test suite for our golden master linked list implementation. The results are presented in table 6.1 on the following page.

[4]To the best of my understanding, this should be equivalent to saying, TestBuilder is better than loading the code and executing nothing, because, as mentioned before, I believe Python does the equivalent of running the built-in `compile` function on code as it is loaded from files.

Table 6.1: Results of mutation testing.

Test suite	Percent of mutations killed
TestBuilder	76.1
Golden master	94.3

In order to give a better standard for comparison of the results, we also ran a mutation test using the test cases which were written for the golden master linked list implementation. This test suite had full coverage.

We also ran coverage tests on both test suites. As mentioned above, the hand-written tests had full coverage. The generated test suite had 76% coverage, which was surprising, as its statement-by-statement technique for generating tests would have been expected to result in full coverage. Further investigation revealed two factors which contributed to this surprisingly low result. First, TestBuilder does not generate tests which trigger errors, leaving all but one of the lines which raise `IndexErrors` untested.[5] This accounted for three out of eleven of the uncovered lines. Second, TestBuilder does not understand functions with higher-order arguments. Two of the functions—`foreach` and `sort`—both involved a higher-order function, and in both cases TestBuilder provided an integer instead of a function. As a result, the test cases it generated for those parts of the functions merely assert that an error is thrown, rather than executing the code meaningfully. Finally, four of the lines which were not covered were in the `remove` function. This seems to have been due to the recursive call in `remove`, which, after the first substitution, would have been treated as an arbitrary return value (see section 3.6 on page 26).

[5]It is unclear why one of the test cases was generated to trigger an `IndexError`; I believe it is due to TestBuilder not tracking the conditions of conditionals for the code following them.

Chapter 7

Capabilities and Limitations

7.1 Capabilities

TestBuilder currently supports the following constructs from Python (this listing is derived from [56]):

Builtins: `int`, `string`, `bool`

> Python treats booleans as a subtype of `ints`; we currently treat them as a distinct type. If the user intentionally uses a boolean as an integer, TestBuilder will find a type error. However, it seems more likely that in student code, booleans used as integers are a mistake. This decision could be revisited at a later point if desired.

Singletons: `True`, `False`, `None`

Statement: `FunctionDef`, `Return`, `Assign`, `AugAssign`, `While`, `If`, `Pass`, `Raise`, `Expr`, `Assert`

> We allow bare expressions, referred to in the Python AST as `Exprs`, although if they are not involved in the computation of the target for a particular test case, they are likely to be ignored.

Expression: `BoolOp`, `BinOp`, `UnaryOp`,`Compare`, `Call`, `Num`, `Str`, `NameConstant`, `Attribute`, `Name`

`BoolOp`s are not short-circuiting.

`BinOp` is only defined for the operations listed in section 7.1.

`UnaryOp` is only defined for the operations listed in section 7.1.

`Call`s to names are ignored if the called function is unknown, and the return value is treated as an unknown Any value, as with a function argument. This may seem imprecise, but just as we must infer the type of function arguments from their use and must assume that the author does not intend type errors, so we must assume that the function will, in fact, return a desirable value. For recursive calls, the call is only expanded to a depth of one. Keyword arguments are not supported.

`Compare` is supported for all the comparison operators other than `In` and `NotIn`.

Boolean Operators: `And`, `Or`

Operators: `Add`, `Sub`, `Mult`, `Div`, `FloorDiv`, `Mod`

Unary Operators: `Not`, `USub`

Comparison Operators: `Eq`, `NotEq`, `Lt`, `LtE`, `Gt`, `GtE`, `Is`, `IsNot`

In addition, some Python syntax is ignored without triggering an error:

Statement: `ClassDef`, `Import`, `ImportFrom`

TestBuilder supports integers, booleans, and `Pair`s, which have `left` and `right` accessors. A limited amount of support is available for floating-point values: floating-point constants may be defined, floating-point values may be assigned to variables, and division results in floating-point values (following Python 3 semantics), but no

```
val = None
if val == None:
   a = True
else:
   a = val.left
```

Figure 7.1: Not-quite type error

operations are currently supported on floating-point values. Similarly, strings are available for definition and assignment, but the only operation available on them is concatenation.

7.2 Limitations

TestBuilder does not support reference types other than a pair, which is indexed with a `left` and `right` attribute.

TestBuilder does not currently support closures. It is likely that support could be added using the techniques presented by Nguyễn et al. [31].

An obscure corner case exists which currently prevents TestBuilder from generating a test case. If a user writes code which has the form shown in fig. 7.1, it will confuse TestBuilder, because the "else" portion is a static `AttributeError`. It does not cause an `AttributeError` at runtime, of course, due to the conditional preventing the execution of the false branch. Since we do not evaluate the conditionals at compile time, we are not currently able to detect these cases and react reasonably. Instead, TestBuilder gives up on writing tests for this function.[1]

[1] Actually, we crash with an assertion failure, but when running Python code with the Python optimizer turned on, assertions are removed.

Chapter 8

Tools

Our goal with TestBuilder was to help beginning programmers write better tests. While TestBuilder has not been as successful in generating test cases as I wish, I have come across a number of tools during the process of development which I believe might be of use to beginning programmers. This chapter is a list of tools which I hope might be of use in helping beginning programmers write better code. Most of these have been used in the construction of some part or other of TestBuilder.

8.1 Libraries

8.1.1 Dataclasses

Python 3.7 added the dataclasses library, which helps in creating classes for data storage. In its simplest application, the resulting classes are something like JavaBeans or structs from other languages. Nevertheless, they are still full Python classes and can have arbitrary methods added.

The dataclass library provides a decorator, `@dataclass`, which generates appropriate special methods to implement various operations on a class with fields defined by the PEP526 syntax for variable annotations[36, 34]. By default, this includes

initialization, equality, and string representation [36]. It also supports immutable classes or classes with ordering, as well as additional features.

A Lisp-style cons class could be defined as follows:

```
from __future__ import annotations

from dataclasses import dataclass
from typing import Any, Optional


@dataclass
class Cons:
    car: Any
    cdr: Optional[Cons]
```

We can create Cons instances like so:

```
from dataclass import Cons

print(Cons(3, Cons(4, None)))
```

Running the program prints

```
Cons(car=3, cdr=Cons(car=4, cdr=None))
```

See [25] for a more complete summary and [57] for full documentation.

8.2 Testing

8.2.1 pytest

pytest is a test runner for Python. Tests are defined as ordinary functions with names beginning with `test` and which may be intermixed with other code. pytest gathers

such functions and runs them, printing a standard pass/fail summary, as seen in fig. 8.1 on the next page. Rather than a specialized assert library, it uses Python's built-in `assert` statement with introspection to provide better information on why tests fail.

pytest supports skipping tests or marking them as expected to fail, which provides flexibility in implementation order.

8.2.2 Hypothesis

Hypothesis is a QuickCheck implementation for Python. It allows the user to annotate tests with the format of desired values and it then generates inputs of that form and provides them as arguments to the function. See fig. 8.2 on page 59 for an example implementation of tree depth computation with a Hypothesis-driven test case, and see fig. 8.3 on page 60 for some example inputs generated by Hypothesis with the configuration from the test case.

8.2.3 unittest.mock

Python has a built-in mocking library. It supports an "action → assertion" style [55] where the user defines a mock, runs the code using the mock, and then asserts that the expected methods were called on the mock. An example is shown in fig. 8.4 on page 61. The `create_autospec` function demonstrated there accepts a real-world object to mock, and refuses to allow the user to mock non-existent attributes, helping to ensure the mock and the real interface do not diverge.

pytestexample.py

```
def double(a):
    return a * 2


def triple(a):
    return a * 3


def test_double():
    assert double(0) == 0
    assert double(1) == 2
    assert double(9) == 18


def test_triple():
    assert triple(3) == 9
    assert triple(0) == 0
```

```
pytest pytestexample.py
====================== test session starts ======================
platform linux -- Python 3.7.2, pytest-4.2.1, py-1.7.0, pluggy-0.8.1
hypothesis profile 'default' -> database=DirectoryBasedExampleDatabase
rootdir: /home/andrew/Education/masters/thesis/examples, inifile:
plugins: cov-2.6.1, hypothesis-4.6.0
collected 2 items

pytestexample.py ..                                          [100%]

=================== 2 passed in 0.01 seconds ====================
```

Figure 8.1: Example pytest run on a module with inline tests

```python
from hypothesis import given
from hypothesis.strategies import builds, deferred, integers, one_of

from dataclass import Cons

pair_tree = deferred(
    lambda: one_of(integers(), builds(Cons, pair_tree, pair_tree))
)


@given(pair_tree)
def test_pair(tree):
    assert breadth_depth(tree) == rec_depth(tree)


# Implementation of the system under test follows


def breadth_depth(tree):
    total = 0
    todo = [(tree, 0)]
    while len(todo) > 0:
        item, depth = todo.pop()
        if isinstance(item, int):
            total = max(total, depth + 1)
        else:
            todo.insert(0, (item.car, depth + 1))
            todo.insert(0, (item.cdr, depth + 1))
    return total


def rec_depth(tree):
    if isinstance(tree, Cons):
        return 1 + max(rec_depth(tree.car), rec_depth(tree.cdr))
    else:
        return 1
```

Figure 8.2: Example Hypothesis test

```
0

-5271882463388798187

-1838109540

Cons(car=39, cdr=Cons(car=-45, cdr=77))

Cons(
    car=Cons(
        car=Cons(
            car=Cons(
                car=Cons(
                    car=-7481887819142686162,
                    cdr=Cons(car=0, cdr=0),
                ),
                cdr=Cons(car=965705312, cdr=Cons(car=0, cdr=0)),
            ),
            cdr=Cons(car=14, cdr=-841895590),
        ),
        cdr=Cons(
            car=82201542168895823290416849619829073407,
            cdr=Cons(
                car=-80,
                cdr=84924153051237575165043362870992218977,
            ),
        ),
    ),
    cdr=Cons(car=13, cdr=21),
)

Cons(car=-20383, cdr=-1916)
```

Figure 8.3: Example Hypothesis inputs
(such as might be passed to the test in fig. 8.2 on the preceding page)

```python
from unittest.mock import create_autospec

from dataclass import Cons


def test_with_mock():
    mockbox = create_autospec(Box(None, None))
    mockbox.length.return_value = 3

    box = Box(2, mockbox)
    assert box.length() == 4

    mockbox.length.assert_called_once()


class Box(Cons):
    def length(self):
        if isinstance(self.cdr, Box):
            return 1 + self.cdr.length()
        else:
            return 1
```

Figure 8.4: Example test using `unittest.mock`

8.3 Typing

8.3.1 mypy

mypy is an optional static typechecker for Python [30] which implements static typechecking of the annotations defined by [33]. It supports mixing typed and untyped code: functions without types are not typechecked [22].[1] Using the `typing` module built in to Python 3.5, the user can define more complex types, including union and generic types. It understands most of standard Python, including the new dataclasses discussed in section 8.1.1 on page 55.

Unfortunately, the support of generics in mypy is not as good as could be wished. Basic generics, such as the type of a list, are supported and work fine. However, typing for more complex functions, such as `partial`, from the standard library `functools` module, is messy: the current implementation [58] does not check that the arguments provided are of the expected type, which means that it is possible for some of the inputs to be the wrong type without a static error.

In addition, recursive types such as

```
tree = Union[int, Tuple[tree, tree]]
```

are not supported properly. Instead, the type must be defined as

```
tree = Union[int, Tuple[Any, Any]]
```

An example of an annotated class is shown in fig. 8.5 on the next page. The `Cons` class defined there is slightly different from that shown in section 8.1.1 on page 55, as it is generic in the type of its `car`, rather than being completely dynamically typed for the `car` field. The `Stack` class is generic in the type of the values it contains, with the push and pop operations being defined as expected.

[1] It is possible to require types to be present everywhere, if this is desired.

```python
from __future__ import annotations

from dataclasses import dataclass
from typing import Generic, Optional, TypeVar

# This __future__ import enables annotations to include types which
# will be defined later


E = TypeVar("E")
# This defines a type variable which can be used by any generics
# present.


@dataclass
class Cons(Generic[E]):
    """Defines a Cons pair which is generic in the type of its car, in
    order to type it in the manner of a standard Lisp list. `None` is
    used as the empty list value.

    """

    car: E
    cdr: Optional[Cons[E]]


class Stack(Generic[E]):
    def __init__(self) -> None:
        self.lst: Optional[Cons[E]] = None

    def push(self, val: E) -> None:
        self.lst = Cons(val, self.lst)

    def pop(self) -> Optional[E]:
        if self.lst is None:
            return None
        else:
            val = self.lst.car
            self.lst = self.lst.cdr
            return val
```

Figure 8.5: Example of Python class with PEP 484 type annotations

```python
from helper import Pair
from typing import Callable, Optional

def add(lst: Optional[Pair], idx: int, val: int) -> Pair:
    ...

def empty_list() -> None:
    ...

def foreach(lst: Optional[Pair], f: Callable) -> None:
    ...

def get(lst: Optional[Pair], idx: int) -> int:
    ...

def length(lst: Optional[Pair]) -> int:
    ...

def remove(lst: Optional[Pair], idx: int) -> Pair:
    ...

def set(lst: Optional[Pair], idx: int, val: int) -> Pair:
    ...

def sort(lst: Optional[Pair], compare: Callable) -> Optional[Pair]:
    ...
```

Figure 8.6: Stubs for golden master `LinkedList` implementation generated by MonkeyType

8.3.2 MonkeyType

Instagram has released a type annotation generator for Python named MonkeyType. It monitors a run of the program and gathers information about the types of the values involved. It can then generate stubs for the functions in a module of the program or annotate the module itself with typing information.

8.4 Debugging

8.4.1 `breakpoint()`

Python 3.7 introduced the `breakpoint()` built-in function [35]. This causes Python to drop to a debugger when it is executed, making it easy to mark breakpoints directly in code without the need to import any particular modules. Further, the action on execution is configurable: by setting the environment variable `PYTHONBREAKPOINT` it is possible to run any desired function at breakpoints, including alternative debuggers [35].

8.4.2 pdb

Python provides a built-in debugger named `pdb`. It is basic but adequate, and it does not require installing anything besides Python. It can be launched by passing `-m pdb` to the Python interpreter. This will drop into the debugger, rather than running the program directly. The debugger can catch exceptions and allow inspection of the broken state as well.

8.4.3 Pdb++

This is an enhanced version of pdb; when installed, it generally replaces pdb. One of its most useful features is `sticky` mode, which provides a continuous view of the code being debugged as it's being run (almost like running `l` after every step). See fig. 8.7 on the next page for an example of pdb++ running.

8.4.4 Embedded IPython

IPython is a Python REPL similar to the one provided with Python. Unlike the built-in REPL, however, it includes colorized output, syntax highlighting, and tab completion. It can be very helpful to drop to a REPL from within a program, allowing

```
  4     def sort(lst):
  5         if lst is None:
  6             return None
  7         less, more = split(lst.cdr, lst.car)
  8  ->     less = sort(less)
  9         more = sort(more)
 10         return append(less, Cons(lst.car, more))
(Pdb++) less
Cons(car=5, cdr=None)
(Pdb++) more
Cons(car=4, cdr=Cons(car=1, cdr=Cons(car=3, cdr=Cons(car=2, cdr=Cons(car=4, cdr=
Cons(car=1, cdr=Cons(car=3, cdr=None)))))))
(Pdb++) sort(less)
Cons(car=5, cdr=None)
(Pdb++)
```

Figure 8.7: The pdb++ user interface

experimentation with the current state without having to manually reproduce the state in an external REPL. IPython provides an `embed` function which can be used to produce a REPL at any point in a program.

This code will drop to an IPython REPL in the middle of a program:

```
from IPython import embed
embed()
```

8.5 Formatting and Style

8.5.1 Black

Black is a Python formatter which is designed to have almost no configuration. Potentially this makes it more suitable for student use, as it does not require configuration files and cannot be configured incorrectly.

According to the documentation [8], Black follows a PEP-8 like style. The Python examples in this document are formatted with Black, typically to 65 columns, to keep them fitting well on the page.

Code:

```
def forgot_to_call():
    print("All the results!")
```

Vulture Output:

```
vultureexample.py:1: unused function 'forgot_to_call' (60% confidence)
```

Figure 8.8: Example of Vulture catching an uncalled function

8.5.2 isort

isort sorts Python imports into a sensible order. Running it and Black on code almost completely eliminates any formatting decisions.

8.5.3 flake8

flake8 combines several other Python linters and style checkers. It is very configurable, and when combined with the `flake8-bugbear` package, can be configured to be more cooperative with Black's formatting, which does not quite meet its requirements.

8.5.4 Vulture

Vulture is able to detect unused code. Ordinarily, this would not seem helpful on a project as small as most student projects. However, integrating Vulture into a standard tool set could help alert beginning programmers to code they have unintentionally failed to call. An example of Vulture's use is shown in fig. 8.8.

8.6 CLI

8.6.1 Docopt

Docopt is a library which significantly simplifies writing CLI tools which take arguments. While less-related to the topic of this thesis than many of the other tools in this section, like other libraries this can be helpful in reducing the amount of code which needs to be written and tested to achieve a goal.

Chapter 9

Related Work

9.1 Automatic Test Generation

Zhang et al. showed how tests could be generated based on a short trace of program use [64]. They demonstrated the value of combined static and dynamic analysis to achieve higher coverage from the generated tests. By watching an example execution sequence, their tool was able to learn legal call sequences, which it combined with static analysis results to generate method call sequences which could result in properly initialized objects. In our work, we do not attempt to handle generation of test cases for stateful classes, as the target audience are not authoring general classes which rely on multiple method calls for correct instantiation.

Saff, Boshernitsan, and Ernst worked on a more general form of test case which they termed a theory [44]. "A theory generalizes a (possibly infinite) set of example-based tests [44]." In the same paper, they describe an additional tool they developed to work with theories, which they called "Theory Explorer." This tool generates inputs to theories in an attempt to find a case for which the theorem does not hold. While our work involves input generation for test cases, we attempt to increase test coverage, rather than explicitly searching for failing examples.

In [10], Cadar, Dunbar, Engler, et al. present KLEE, a tool for test case generation. It relies on abstract interpretation of LLVM IR, combined with some concrete interaction with the environment. For example, when a program under test attempts to open a file with a concrete name, the existing file on disk will be opened. However, if the program attempts to open a file with an abstract value as the name, KLEE will instead create a simulated, abstract file, allowing it to continue operating abstractly. Ultimately, KLEE tracks symbolic values and creates path constraints, which enable it to construct proofs about the program. TestBuilder does not do abstract execution of the user's code, rather, it translates the user's code entirely to the SMT solver's domain and allows the solver to construct any desired path conditions. This has the disadvantage of not allowing KLEE's techniques for handling the environment, but it has the advantage of allowing proofs to be constructed for all possible paths through the program, modulo loop unrolling.

Cannon, in his master's thesis [11], implemented a type inference system for Python using the Cartesian Product type inference algorithm. One of the restrictions placed on this work was that it be implemented "without any semantic changes to the compiler or language." As the rest of the thesis demonstrates, this is an onerous restriction, as Python is a very dynamic language; the net result is that only the local control flow in the file currently being compiled can be depended on.

Ceccato et al. present a series of experiments on the effectiveness of manually written versus automatically generated tests in debugging [12]. They found that, for less-experienced developers, automatically generated tests enabled more effective debugging, while for the most experienced developers, the type of test did not have any significant effect. They suggest that this may be due to the less-experienced developers tending to attempt to understand the test code, while the more-experienced developers did not. These results are encouraging for the present project, as they

suggest that automatically generated tests, such as the ones we produce, can be more helpful to beginning programmers than manually written tests.

Dybjer, Haiyan, and Takeyama present an automatic test generation system integrated into the Agda theorem prover [17]. This enables the user to choose between proving and testing theorems depending on which is most appropriate. This approach enables annoying-to-prove but obvious theorems to be tested rather than proved. Lemmas can be tested before a proof attempt is made, in order to reduce the time wasted attempting to prove an incorrect lemma. The counterexample resulting from a failed test case can provide guidance in refining the lemma to make it valid.

Feng et al. propose conducting program synthesis via a CDCL-like algorithm which attempts to generate programs which meet user-provided examples [19]. Using the grammar of a DSL and an abstract semantics of its terminals, it searches for contradictions between a partial program and the user's example. Any contradictions which are found allow it to eliminate the program and similar alternative programs which share the conflicting characteristics.

Gulwani, Radiček, and Zuleger present Clara, a tool for finding repairs to student programs to make them correct, particularly designed for use in the context of a MOOC [24]. It compares an incorrect student program to clusters of correct programs from previous student submissions. Using a cluster which structurally matches the incorrect program, it determines a set of changes which make the program equivalent under renaming. Fundamentally, Clara relies on the availability of correct student programs from which to derive corrections.

Khurshid, Păsăreanu, and Visser describe symbolic execution via a model checker of a program containing complex data structures [27]. They instrument a program written in Java to create a modified version which can be run via a model checker which runs Java source. Inputs to the program are created lazily, only instantiating

the input data structure as needed. They demonstrate how their system can be used, with suitable constraints, for test case generation in order to achieve various coverage metrics. Unlike TestBuilder, they seem to rely on constraining the symbolic execution to take desired paths, rather than choosing a path and computing a symbolic path condition for it as TestBuilder does.

Might [28] presents a number of improvements to the basic kCFA developed by Shivers [45]. First, he presents abstract garbage collection, which allows the elimination of unreferenced abstract values in order to prevent them muddying later analyses. Second, he presents abstract counting semantics, which tracks the number of concrete values which abstract to a given abstract value in a given state. The goal is to discover abstract values which are equal to one another, and whose sets of concrete values only contain one element. This then proves that their concrete values are equal.

Nguyễn et al. present a technique for statically checking the validity of contracts on code written in an untyped, higher-order language with mutation [31]. They statically execute the program, tracking the conditionals passed at any given point, and confirm that either there is no way a particular path could run, or that no action it could take would violate the function's contract. Having shown this, they can confirm that a function is never at fault for crashes, allowing blame to be redirected to other functions.

The tool presented in this paper uses an SMT solver for checking feasibility of path conditions. If a path condition is infeasible, all code controlled by it is dead and will never be executed, thus guaranteeing that it cannot cause problems.

Our tool differs from that of Nguyễn et al. in its use of an SMT solver. While their tool searches for execution paths which are infeasible, our tool seeks to find inputs to execute a given path.

Pike introduced SmartCheck, a property testing tool capable of automatically shrinking discovered counterexamples and generalizing across parts of the example, including both generalizing out subexpressions and determining that a counterexample exists for every constructor of a sum type [37]. Like other property-based testing tools, SmartCheck differs from our work in its use of properties to define correct behaviour, rather than concrete test cases. It is potentially better able to discover bugs than TestBuilder, but it requires the user to know and be able to state properties of the code they have written.

Ryu proposes a design aid for beginning programmers based on both the design recipies proposed by Felleisen et al. [18] and an outlining process derived from that used in the language arts [43]. As part of this, Ryu proposes a tool known as DRCOP which can convert design recipies into function stubs and test suites. It generates test suites using the input–output pairs provided by the student in the design outline. Whereas DRCOP relies on the student to generate the inputs and outputs for all the tests it will generate, Testbuilder generates inputs and prompts the student for the expected output only.

Vorobyov and Krishnan present a test generation tool called Batg which uses a static analyzer to determine potentially-buggy parts of a program and generate test cases to expose those bugs [59]. They focus on buffer overflows in C code, using Valgrind to confirm during test case execution that the buffer overflow actually exists.

Static analysis allows the test case generation to focus on those parts of the program likely to contain bugs. Test case generation allows the automatic verification of the potential bugs detected by the static analysis.

Wang, Singh, and Su present sarfgen, a tool to correct programs submitted to a MOOC [60]. Minimal changesets for repair are generated by comparing the incorrect program to correct submissions and choosing a minimal set of changes

which make the program pass the tests. As with Clara, this tool relies on correct submissions to provide a golden master.

Multiple papers have been written on automatically generating test cases. Rayadurgam and Heimdahl present a test-generation method using model checkers, along with a theoretical model of programs suitable for specifying testing requirements. They also cite two other groups which are doing similar work (Gargantini and Heitmeyer [20] and Ammann, Black, and Majurski [4, 3])

9.2 Concolic Testing

In 2015, Ball and Daniel presented a technique for concolic testing of Python code [5]. By subclassing the standard Python types, they are able to create instances of those types which can track their execution history and compile a set of conditionals that affect execution flow. When operations are applied to them, they can return specialized instances of the ordinary result type, allowing them to handle more complex cases.

Concolic testing as presented in [23] uses code crashes to determine buggy behavior. This is somewhat less applicable in Python. Python code can crash due to type errors. But as mentioned in section 3.1 on page 16, Python idiom makes strong use of Duck typing. This means that there are an enormous number of possible type errors in every Python function, but that it is idiomatic to expect the user of a function not to trigger them by supplying incorrectly-typed data. Thus, type error crashes due to violating the implicit interface of the code are not particularly interesting. On the other hand, plain Python should not suffer from buffer overflows or segfaults. Thus, there are fewer available ways for a Python program to crash dangerously. In addition, as TestBuilder is intended to work with student code from beginning programmers, the complexity of that code is expected to be low, and the anticipated number of places where the code dereferences a None is expected to

be low. Thus, basic concolic testing of functions seems unlikely to expose many bugs. Finally, as mentioned above, we wanted to support the generation of test cases which could reach specific lines in a user's code. For all these reasons, we chose not to use a concolic testing system.

Chapter 10

Future Work

10.1 Obvious Deficiencies

The current version of TestBuilder only supports a subset of Python. It would be desirable to add support for the rest of the language, along with support for proper function calls and recursive functions (perhaps via better support for functions in the SMT solver). Notable challenges include support for custom classes and custom operators. Perhaps we can reimplement the standard operators in terms of custom operators, to unify their implementation. It would also be desirable to use some library to abstract over SMT solvers[1] so that we can potentially swap out SMT solvers for the best performing one.

It would also be good to convert target lines to target basic blocks, in order to reduce the number of test cases that are generated. Further, if two test cases are identical, the system should omit one.

Finally, any type annotations present on the user's code should be used to force type inference in a particular direction. This would enable users to avoid having their code tested with unintended types and could potentially reduce the size of the SMT expressions generated by TestBuilder.

[1] E.g., pySMT [21]

10.2 Alternative Applications

10.2.1 Property Proving

Potentially we can use the generated expressions to allow us to prove properties of the user's code. This is particularly interesting as it makes the Python source the single source of truth, rather than proving properties in terms of a prover language and then translating to Python.

10.2.2 Test Case Minimization

The decision to ask the user about the expected results means that we want to generate a minimal number of test cases. In order to achieve this, it would be desirable to add a system which generated an excess of test cases and kept a set which covered all the lines of the program. Of course, set cover is an NP-complete problem, but an approximation which overcovers is not a problem.

It would also be interesting to look into the work that has been done in choosing test vectors for IC designs using SAT solvers. I think this work is interested in minimizing the number of vectors required; if so, perhaps we can use a similar technique to choose a minimal set of test inputs.

10.3 Additional Capabilities

10.3.1 Boundary Test Case Generation

In order to better support users in discovering problems with their code, it would be desirable to support type-specific boundary testing: for example, when finding a range of integers which satisfy a path, choose a few different integers around the boundaries of that range and make sure the code behaves as expected in those cases.

How to avoid asking the user for expected outputs for a huge number of these is not clear. Perhaps they can be safely combined (see test case minimization above)

10.3.2 Concolic-style Testing

It would be nice to incorporate elements of some of the concolic testing work that has been done, in order to allow better handling of library code, without having to convert everything to SMT expressions. Could such handling be integrated into a solver as a theory, where code is run to determine whether a proposed solution is valid?

10.3.3 Mutation-Driven Test Generation

Rather than simply solving for inputs which reach a point in the program, we could solve for inputs which result in different outputs from the original and a mutated version of the program. This would be expected to increase the strength of the generated test suites, as it guarantees that they will kill certain mutants, rather than hoping that the examples which are generated will happen to be the ones on which the mutant differs from the correct code.

Chapter 11

Conclusion

TestBuilder solves constraints generated by control structures in order to direct execution in a desired direction. We presented a model of its operation, demonstrating how a dynamic language such as Python can be converted into typed expressions which an SMT solver can work with. Although we were unable to show that TestBuilder would help students avoid losing points, the tests generated by TestBuilder required almost no work to write, and were still able to catch 61% of mutations on average. We anticipate that, with more work, TestBuilder could potentially improve in performance.

Bibliography

[1] Ole Agesen. “The Cartesian Product Algorithm”. In: *ECOOP’95 — Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995*. Ed. by Mario Tokoro and Remo Pareschi. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 2–26. ISBN: 978-3-540-49538-3.

[2] B. Alpern, M. N. Wegman, and F. K. Zadeck. “Detecting Equality of Variables in Programs”. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’88. San Diego, California, USA: ACM, 1988, pp. 1–11. ISBN: 0-89791-252-7. DOI: 10.1145/73560.73561. URL: http://doi.acm.org/10.1145/73560.73561.

[3] P. E. Ammann and P. E. Black. “A specification-based coverage metric to evaluate test sets”. In: *Proceedings 4th IEEE International Symposium on High-Assurance Systems Engineering*. 1999, pp. 239–248. DOI: 10.1109/HASE.1999.809499.

[4] P. E. Ammann, P. E. Black, and W. Majurski. “Using model checking to generate tests from specifications”. In: *Proceedings Second International Conference on Formal Engineering Methods* (*Cat.No.98EX241*). Dec. 1998, pp. 46–54. DOI: 10.1109/ICFEM.1998.730569.

[5] Thomas Ball and Jakub Daniel. *Deconstructing Dynamic Symbolic Execution*. Tech. rep. Proceedings of the Sixth Conference on Uncertainty in Artificial Intelligence, Boston, MA. Jan. 2015. URL: https://www.microsoft.com/en-us/research/publication/deconstructing-dynamic-symbolic-execution/.

[6] Leon Bambrick. Jan. 1, 2010. URL: https://twitter.com/secretGeek/status/7269997868.

[7] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.6*. Tech. rep. Department of Computer Science, The University of Iowa, 2017. URL: http://smtlib.cs.uiowa.edu/language.shtml (visited on 03/08/2019).

[8] *Black: The uncompromising Python code formatter*. URL: https://github.com/ambv/black (visited on 02/18/2019).

[9] Raymond T. Boute. "The Euclidean Definition of the Functions Div and Mod". In: *ACM Trans. Program. Lang. Syst.* 14.2 (Apr. 1992), pp. 127–144. ISSN: 0164-0925. DOI: 10.1145/128861.128862. URL: http://doi.acm.org/10.1145/128861.128862.

[10] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." In: *OSDI*. Vol. 8. 2008, pp. 209–224.

[11] Brett Cannon. "Localized type inference of atomic types in Python". MS. California Polytechnic State University, 2005. URL: https://cpslo-primo.hosted.exlibrisgroup.com/primo-explore/fulldisplay?docid=01CALS_ALMA713757478000029
L&vid=01CALS_PSU&search_scope=EVERYTHING&tab=everything&lang=en_US.

[12] Mariano Ceccato et al. "Do Automatically Generated Test Cases Make Debugging Easier? An Experimental Assessment of Debugging Effectiveness and Efficiency". In: *ACM Trans. Softw. Eng. Methodol.* 25.1 (Dec. 2015), 5:1–5:38. ISSN: 1049-331X. DOI: 10.1145/2768829. URL: http://doi.acm.org/10.1145/2768829.

[13] "Conversation with David Parkinson". Feb. 13, 2018.

[14] *Definition of Canonize by Merriam-Webster*. URL: https://www.merriam-webster.com/dictionary/canonize (visited on 03/21/2019).

[15] L. Deng, J. Offutt, and N. Li. "Empirical Evaluation of the Statement Deletion Mutation Operator". In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. Mar. 2013, pp. 84–93. DOI: 10.1109/ICST.2013.20.

[16] *differences - "Normalization" vs. "canonicalization" - English Language & Usage Stack Exchange*. URL: https://english.stackexchange.com/questions/35860/normalization-vs-canonicalization (visited on 03/21/2019).

[17] Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. "Verifying Haskell programs by combining testing, model checking and interactive theorem proving". In: *Information and Software Technology* 46.15 (2004). Third International Conference on Quality Software: QSIC 2003, pp. 1011–1025. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2004.07.002. URL: http://www.sciencedirect.com/science/article/pii/S0950584904001077.

[18] M. Felleisen et al. *How to Design Programs, Second Edition*. MIT Press, 2014.

[19] Yu Feng et al. "Program Synthesis Using Conflict-driven Learning". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2018. Philadelphia, PA, USA: ACM, 2018, pp. 420–435. ISBN: 978-1-4503-5698-5. DOI: 10.1145/3192366.3192382. URL: http://doi.acm.org/10.1145/3192366.3192382.

[20] Angelo Gargantini and Constance Heitmeyer. "Using Model Checking to Generate Tests from Requirements Specifications". In: *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC/FSE-7.

Toulouse, France: Springer-Verlag, 1999, pp. 146–162. ISBN: 3-540-66538-2. URL: http://dl.acm.org/citation.cfm?id=318773.318939.

[21] Marco Gario and Andrea Micheli. "PySMT: a solver-agnostic library for fast prototyping of SMT-based algorithms". In: *SMT Workshop 2015*. 2015.

[22] *Getting started — Mypy 0.670 documentation*. URL: https://mypy.readthedocs.io/en/stable/getting_started.html (visited on 02/20/2019).

[23] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: directed automated random testing". In: *ACM Sigplan Notices*. Vol. 40. 6. ACM. 2005, pp. 213–223.

[24] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. "Automated Clustering and Program Repair for Introductory Programming Assignments". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2018. Philadelphia, PA, USA: ACM, 2018, pp. 465–480. ISBN: 978-1-4503-5698-5. DOI: 10.1145/3192366.3192387. URL: http://doi.acm.org/10.1145/3192366.3192387.

[25] Geir Arne Hjelle. "The Ultimate Guide to Data Classes in Python 3.7". In: (May 15, 2018). URL: https://realpython.com/python-data-classes/.

[26] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, 2000.

[27] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. "Generalized Symbolic Execution for Model Checking and Testing". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Hubert Garavel and John Hatcliff. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 553–568. ISBN: 978-3-540-36577-8.

[28] Matthew Brendon Might. "Environment analysis of higher-order languages". PhD thesis. Georgia Institute of Technology, 2007.

[29] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78800-3.

[30] *mypy — Optional Static Typechecking for Python*. URL: `http://mypy-lang.org/index.html` (visited on 02/19/2019).

[31] Phúc C. Nguyễn et al. "Soft Contract Verification for Higher-order Stateful Programs". In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017), 51:1–51:30. ISSN: 2475-1421. DOI: `10.1145/3158139`. URL: `http://doi.acm.org/10.1145/3158139`.

[32] *PEP 483 – The Theory of Type Hints*. URL: `https://www.python.org/dev/peps/pep-0483/`.

[33] *PEP 484 – Type Hints*. URL: `https://www.python.org/dev/peps/pep-0484/`.

[34] *PEP 526 – Syntax for Variable Annotations*. URL: `https://www.python.org/dev/peps/pep-0526/`.

[35] *PEP 553 – Built-in breakpoint()*. URL: `https://www.python.org/dev/peps/pep-0553/` (visited on 02/21/2019).

[36] *PEP 557 – Data Classes*. URL: `https://www.python.org/dev/peps/pep-0557/`.

[37] Lee Pike. "SmartCheck: Automatic and Efficient Counterexample Reduction and Generalization". In: *SIGPLAN Not.* 49.12 (Sept. 2014), pp. 53–64. ISSN: 0362-1340. DOI: `10.1145/2775050.2633365`. URL: `http://doi.acm.org/10.1145/2775050.2633365`.

[38] *Python Glossary; entry for "duck typing"*. URL: `https://docs.python.org/3/glossary.html#term-duck-typing`.

[39] *Python Glossary; entry for "EAFP"*. URL: https://docs.python.org/3/glossary.html#term-eafp.

[40] S. Rayadurgam and M. P. E. Heimdahl. "Coverage based test-case generation using model checkers". In: *Proceedings. Eighth Annual IEEE International Conference and Workshop On the Engineering of Computer-Based Systems-ECBS 2001*. 2001, pp. 83–91. DOI: 10.1109/ECBS.2001.922409.

[41] Jack W Reeves. "What is software design". In: *C++ Journal* 2.2 (1992), pp. 14–12.

[42] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. "Global Value Numbers and Redundant Computations". In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '88. San Diego, California, USA: ACM, 1988, pp. 12–27. ISBN: 0-89791-252-7. DOI: 10.1145/73560.73562. URL: http://doi.acm.org/10.1145/73560.73562.

[43] Mike Dongyub Ryu. "Improving Introductory Computer Science Education with DRaCO". In: (2018).

[44] David Saff, Marat Boshernitsan, and Michael D. Ernst. "Theories in Practice: Easy-to-Write Specifications that Catch Bugs". In: (Jan. 2008). URL: http://hdl.handle.net/1721.1/40090.

[45] Olin. Shivers. "Control-flow analysis of higher-order languages: or taming lambda". en. PhD thesis. Pittsburgh, Pa, 1991.

[46] *SMT-LIB: The Satisfiability Modulo Theories Library: Core*. URL: http://smtlib.cs.uiowa.edu/theories-Core.shtml (visited on 03/08/2019).

[47] *SMT-LIB: The Satisfiability Modulo Theories Library: Ints*. URL: http://smtlib.cs.uiowa.edu/theories-Ints.shtml (visited on 03/08/2019).

[48] *SMT-LIB: The Satisfiability Modulo Theories Library: Reals*. URL: http://smtlib.cs.uiowa.edu/theories-Reals.shtml (visited on 03/08/2019).

[49] O. Tange. "GNU Parallel - The Command-Line Power Tool". In: *;login: The USENIX Magazine* 36.1 (Feb. 2011), pp. 42–47. DOI: http://dx.doi.org/10.5281/zenodo.16303. URL: http://www.gnu.org/s/parallel.

[50] *The Python Language Reference: Data model*. URL: https://docs.python.org/3/reference/datamodel.html (visited on 03/07/2019).

[51] *The Python Language Reference: Lexical analysis*. URL: https://docs.python.org/3/reference/lexical_analysis.html#identifiers.

[52] *The Python Language Reference: Lexical analysis*. URL: https://docs.python.org/3/reference/lexical_analysis.html#string-and-bytes-literals.

[53] *The Python Standard Library: Built-in Functions*. URL: https://docs.python.org/3/library/functions.html (visited on 03/21/2019).

[54] *The Python Standard Library: Built-in Types*. URL: https://docs.python.org/3/library/stdtypes.html (visited on 03/07/2019).

[55] *The Python Standard Library: Development Tools: unittest.mock — mock object library*. URL: https://docs.python.org/3/library/unittest.mock.html (visited on 02/19/2019).

[56] *The Python Standard Library: Python Language Services: ast — Abstract Syntax Trees*. URL: https://docs.python.org/3/library/ast.html.

[57] *The Python Standard Library: Python Runtime Services: dataclasses — Data Classes*. URL: https://docs.python.org/3/library/dataclasses.html.

[58] *typeshed/functools.pyi*. URL: https://github.com/python/typeshed/blob/95eff73ab2092f2c3158198d404a921447172418/stdlib/3/functools.pyi#L43 (visited on 02/19/2019).

[59] K. Vorobyov and P. Krishnan. “Combining Static Analysis and Constraint Solving for Automatic Test Case Generation”. In: *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. Apr. 2012, pp. 915–920. DOI: 10.1109/ICST.2012.196.

[60] Ke Wang, Rishabh Singh, and Zhendong Su. “Search, Align, and Repair: Data-driven Feedback Generation for Introductory Programming Exercises”. In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2018. Philadelphia, PA, USA: ACM, 2018, pp. 481–495. ISBN: 978-1-4503-5698-5. DOI: 10.1145/3192366.3192384. URL: http://doi.acm.org/10.1145/3192366.3192384.

[61] *Welcome to Python.org*. URL: https://www.python.org/.

[62] *Z3: C API*. Feb. 8, 2019. URL: http://z3prover.github.io/api/html/group__capi.html (visited on 03/08/2019).

[63] *Z3: z3py Namespace Reference*. Feb. 8, 2019. URL: http://z3prover.github.io/api/html/namespacez3py.html (visited on 03/08/2019).

[64] Sai Zhang et al. “Combined Static and Dynamic Automated Test Generation”. In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ISSTA ’11. Toronto, Ontario, Canada: ACM, 2011, pp. 353–363. ISBN: 978-1-4503-0562-4. DOI: 10.1145/2001420.2001463. URL: http://doi.acm.org/10.1145/2001420.2001463.