COS 516 Final Project Report
AutoPy: An Automatic Test-Case Generation Engine for Python

Eesha Agarwal

5 December 2023

1 Introduction

Software programs today form the backbone of critical infrastructures across sectors, from healthcare to manufacturing, from education to transportation. Large-scale dependency on computer programs for the functioning of essential systems necessitates reliability, consistency, and dependency as indispensable characteristics of these programs. Comprehensive and robust software validation is therefore an unavoidable element of the software development process: the two key techniques of validation which are most widely used across the industry are those of testing and formal verification.

1.1 Comprehensive Software Validation through Testing and Formal Verification

Testing, on one hand, and formal verification on the other have always been regarded as complementary techniques. The complementarity between these techniques can be best understood by considering where each one falls short. As Dijkstra famously remarked, "Program testing can be used to show the presence of bugs, but never their absence," i.e., testing is necessarily incomplete because by the very nature of testing being case-based, it is impossible to guarantee the absence of errors with testing alone. On the flip side, while formal verification can ensure the kind of infallible, foolproof guarantees of which testing falls short, it suffers from a different but still essential kind of incompleteness: it can only be applied to those aspects of a system which can be formally modelled, whereas testing can be used to expose and eliminate errors in any part of the system under consideration. Testing and verification must hence be used in conjunction with each other to meet distinct, and often complementary, goals which are necessary to ensuring the creation of robust and reliable software systems.

1.2 Prevalence and Challenges of Software Testing

While the field of formal verification has made tremendous progress in enabling greater confidence in the consistency and dependability of computer programs, it is undeniable

that testing remains the most fundamental practice of validation in the realm of software development, so much so, that testing constitutes more than half the total costs during the development and maintenance process [1]. Of the multitude of testing techniques, when it comes to the task of vigorously inspecting the various elements of the implementation, the most popular has been unit testing. Traditionally, such tests are designed and conducted by providing particular input values to programs [2]. Each test explores a possible execution of the program, with the intent of exhaustively traversing each possible path. However, it is clear to see how this benchmark of exhaustiveness can be nearly impossible to truly achieve in practice.

1.3 Automated Test-Case Generation through Symbolic Execution

One of the chief challenges of software testing is, hence, the creation of a *good* test suite, i.e. sets of test cases. This is where formal verification methods, in addition to just complementing the robustness goals of testing, can actually also contribute to the process of testing itself, through the process of automated code-driven test-case generation. This elegant technique relies on automated exploration of the input space, achieved through symbolic execution. Symbolic execution is a program analysis technique wherein, instead of executing a program with concrete input values, we execute the program with symbols which represent arbitrary or unknown values (essentially variables). As a result, the output values for a given function are not concrete, but are instead presented in terms of the chosen symbolic representation of the input values. Symbolic execution targets the problem of test-case generation by systematically enabling the generation of high-coverage test cases, while also allowing for other tests of reliability, consistency, and correctness, such as assertion checking, identification of logic errors, etc. It analyses a program's intrinsic behaviour to find bugs, vulnerabilities, or verify desired properties of the program.

1.4 Introduction to AutoPy: A Test-Case Generation Engine

Automated test-case generation is a structural white-box approach to unit testing for software using static analysis, that combines the technique of symbolic execution with that of SMT (Satisfiability Modulo Theory) solving. Path conditions, i.e., a set of logical constraints that bring us to a specific point in the execution (necessary for a path to be feasible), are generated for each operation and conditional statement in the program. An SMT solver is then utilized to determine, for each path, whether the collection of constraints accumulated over the path are satisfiable.

For the purpose of this project, I undertake the implementation of AutoPy: a constrained automated test-case generation engine for Python. Provided with a function with any number of inputs, AutoPy analyses the different execution paths in the program. The key goals for each path are to 1) generate a set of specific input values which lead to the execution of the given path, 2) check for the possibility of a variety of different errors along the path including assertion violations, uncaught exceptions, and security vulnerabilities, or 3) declare a path to be unsatisfiable, and the corresponding code to be dead/unreachable.

Currently, the engine works to automatically generate test cases, check for assertion violations, and identify dead code for a subset of the Python programming language, fo-

cusing specifically on support for primitive numerical data types, control flow and nested conditionals, and assertions. Support for loops and detection of non-terminating iterables, more complex data structures, and the usage of external libraries and the integration and composition of functions is excluded.

2 Overview

The implementation of AutoPy can broadly be broken down into two separate components: the first of these is undertaking the symbolic execution of the function in question. This is done by:

1. Selectively parsing the Abstract Syntax Tree generated on the basis of the input source code, and then
2. Applying an SMT solver to generate test cases by solving the collection of constraints accumulated for each successful branch of the program

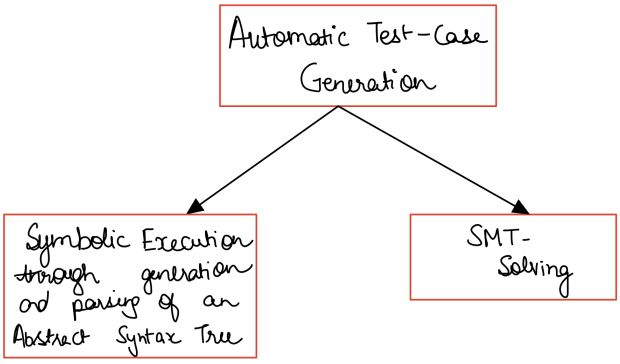


Figure 1: AutoPy: Implementation Overview

```
def two_variable_function(x, y):
    if x > 10:
        if y < 5:
            return x + y
        else:
            return x - y
    elif x == 5:
        if y > 10:
            return x * y
        else:
            return x / y
    else:
        return x ** y
```

Figure 2: Sample Python Program

An Abstract Syntax Tree is a hierarchical (tree-like) representation of the abstract syntactic structure of the source code of a language. Each node of the tree denotes a construct occurring in the source code, including simpler elements like variables, constants, and expressions, and also more complex ones like loops, conditions, try-catch blocks, assertions, etc. Insignificant or unimportant details like whitespace and comments are omitted from the source code in devising this representation. Consider the simple Python program shown above: the figure below shows the Abstract Syntax Tree representation of the same.

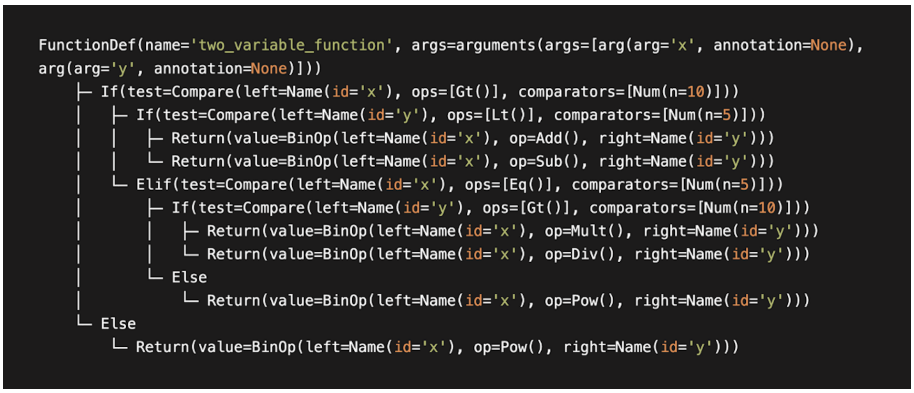


Figure 3: AST Representation for Sample Python Program

Once the input source code has been processed into an AST representation, this AST is traversed to identify and extract decision and validation conditions from the code, which, in turn, inform the formation of the different path conditions. For this project specifically, I make use of Python's AST Module [3], which allows for the generation, interpretation, and manipulation of Python's Abstract Syntax Tree. By providing classes like `NodeVisitor` and `NodeTransformer`, which may be subclassed to visit and modify the nodes in the tree during traversal, Python's AST module becomes a powerful tool in enabling static analysis, the creation of code linters, or the development of tools, like AutoPy, for code transformation and optimisation.

The second essential component of AutoPy is an SMT solver, responsible for solving the collection of constraints collected along each possible path in the source code to either devise a set of input values for each input parameter to the function or to declare the collection of constraints unsatisfiable, signifying that the program either contains dead (unreachable) code, or necessarily failing assertions.

An SMT (Satisfiability Modulo Theories) solver is a type of software tool that is used in order to determine if a given collection of logical or mathematical formulas can be satisfied. They determine whether a quantifier-free first-order formula can be satisfied with respect to some background theories [4]. Such solvers are an extension of Boolean Satisfiability (SAT) Problem solvers, which exclusively deal with pure Boolean formulae. SMT solvers, on the other hand, are able to handle more complex formulas which contain constraints arising from several other pertinent theories, such as integer arithmetic, real numbers, bit vectors, arrays, etc. They have a myriad of crucial uses across the field of formal verification, including in automated theorem proving and program analysis. For this project, I utilise the Z3 solver, an efficient, high-performance theorem proved developed by Microsoft Research which has a

synergistic relationship with software analysis, verification, and symbolic execution problems [5].

An outline of the specific transformations and tasks undertaken during the execution of the AutoPy engine is detailed in the process outline below (the details of, and the process of implementing each of these steps is further elaborated upon in the next section).

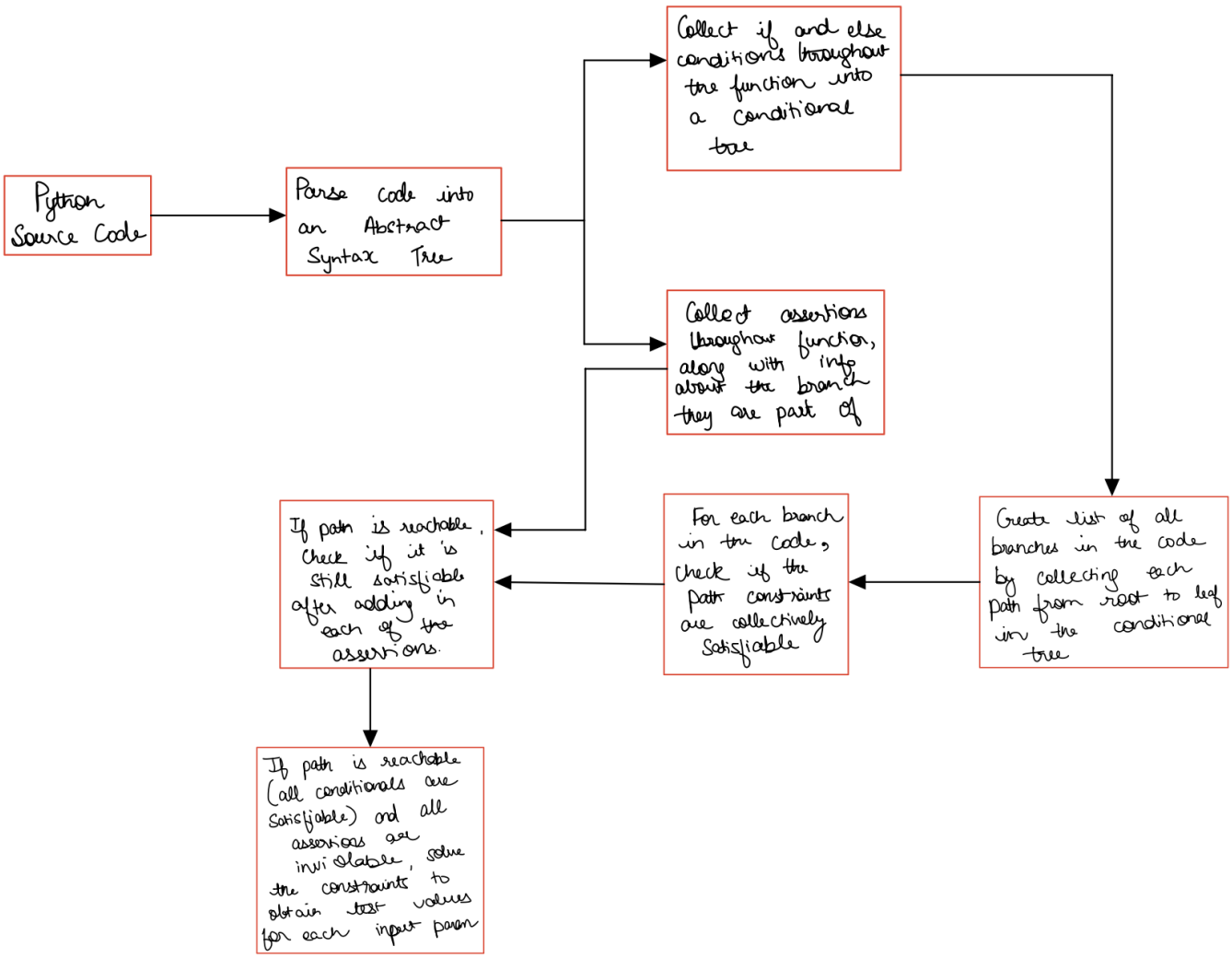


Figure 4: Detailed Implementation of AutoPy

3 Project Tasks

In this section, I highlight the specific tasks that are undertaken as part of the execution of the AutoPy test-case generation engine, in the order that they are undertaken during the actual process of test-case generation (as depicted in the diagram above).

3.1 Parsing the Source Code

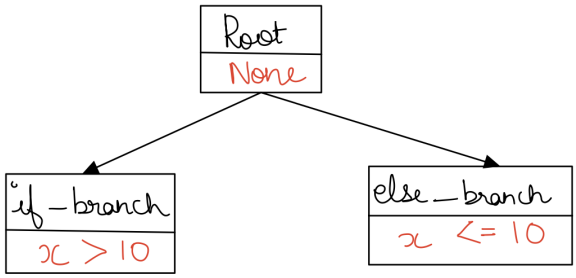
The first step of the process is to parse the Python source code into an Abstract Syntax Tree (AST) using the `ast` module. The `ast.parse(source)` function is utilised for this purpose. This parsing is crucial and is a necessary precursor to analysing the program's structure and extracting conditions and assertions because a significant part of this extraction process is contingent on modifying the `ast` module's `NodeVisitor` module in order to be able to detail the specific steps that should be undertaken to collect the assertions and conditions based on the type of node that is encountered when parsing the AST representation of the source code.

3.2 Creation of Conditional Tree

The parsed AST is then used to create a conditional tree. In order to do this, I define an `If_Node` class wherein each instance of the Node represents a single conditional block in the code. For instance, the following simple Python program would be represented as a conditional tree in the following way:

```
def nested_conditionals(x):
    if x > 10:
        return "A"
    else:
        return "B"
```

(a) Python Program #1



(b) Conditional Tree for Python Program #1

This tree is essentially built by recursively adding child nodes for each conditional statement encountered in the (if or else) branch of another conditional statement, as well as for its corresponding else-clause if one is defined. The conditional tree hence ends up resembling a (not necessarily symmetric) binary-tree: the lack of symmetry stems from the fact that it is not necessary for each `if` statement to have a corresponding `else` branch. This structure enables us to clearly trace the logical flow of the program.

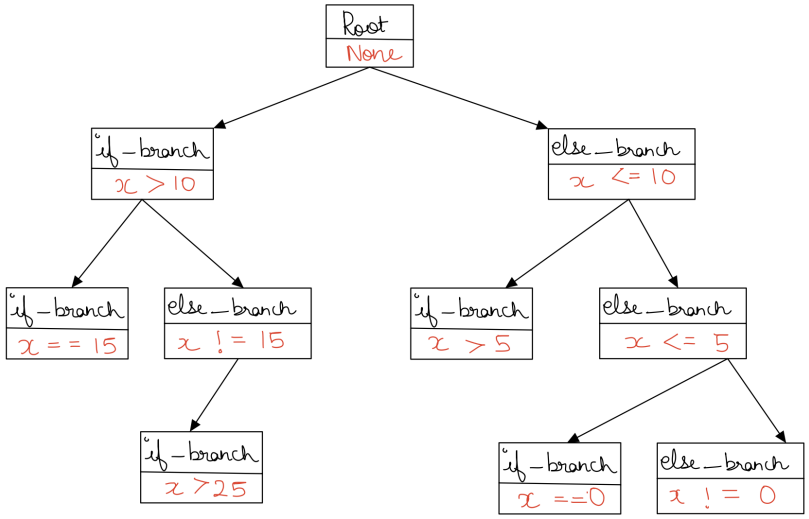
The challenge here was determining how to correctly and efficiently handle nested conditional statements. Apart from being a common construct in Python code, it was necessary to deal with these accurately, because an even more commonly-employed construct, the `elif` branch, is represented in Python ASTs as a nested conditional.

The initial implementation I undertook for the construction of this tree overrode the `ast` module's `NodeVisitor` class, specifically overriding the `visit_If` method. However, due to the nature of the NodeVisitor class and its sequential approach to visiting the Nodes of an AST (as opposed to the recursive approach necessary for this construction), this implementation was unsuccessful.

In order to overcome this, I finally implemented this construction within a separate, recursive `handle_if` method which is called from within the `visit_FunctionDef` method upon encountering a conditional node amongst any of the nodes in the relevant function body. Python's built-in `any` method serves as the key driver of this construction and is responsible for its termination, by allowing recursive checks to evaluate whether a particular conditional statement's (or the else clause's) body contains another conditional node. The code snippet below shows the conditional tree for a more complex Python program than the one depicted above and demonstrates how such a structure allows for the effective storage of complex conditioning within a program.

```
def nested_conditionals(x):
    if x > 10:
        if x == 15:
            return "A"
        else:
            if x > 25:
                return "B"
    else:
        if x > 5:
            return "C"
        else:
            if x == 0:
                return "D"
            else:
                return "E"
```

(a) Python Program #2



(b) Conditional Tree for Python Program #2

3.3 Collecting Path Constraints

Once the conditional tree has been constructed, the next step is to collect path constraints for each path in the conditional tree. These constraints, or the path conditions, are the constraints that must be met in order for the execution to reach a particular point in the program. In order to generate path constraints from the conditional tree created in the previous step, the AutoPy engine creates a list of every root-to-leaf path in the tree since each path from the root to a leaf node represents a unique execution path through the code, dictated by a specific set of conditions or path constraints. The list of paths (or branches) for the more complex Python program given above is shown below:

```
Number of branches: 5
Branch number: 1
Branch conditions: ['x > 10', 'x == 15']

Branch number: 2
Branch conditions: ['x > 10', 'not x == 15', 'x > 25']

Branch number: 3
Branch conditions: ['not x > 10', 'x > 5']

Branch number: 4
Branch conditions: ['not x > 10', 'not x > 5', 'x == 0']

Branch number: 5
Branch conditions: ['not x > 10', 'not x > 5', 'not x == 0']
```

Figure 7: Enumerated Execution Paths for Python Program #2

3.4 Converting Conditional Constraints to Z3

For each path in the program, we iterate over the list of constraints, converting each from `ast` Compare nodes to `Z3` conditions. This `convert_ast_to_z3` method is one of the two sources of the limits imposed on the subset of Python for which AutoPy currently works since it is responsible for understanding the details of both the `ast` module and `Z3` to appropriately translate constraints from one language to another; presently, the only translations that are accounted for in this version of AutoPy are comparative and logical operators applied to numerical constants and variables.

Other kinds of data types (e.g. Boolean variables, strings, lists, and more complex data structures like sets and dictionaries, etc.) as well as other kinds of operators (membership operators such as `in` and `not in`, identity operators, arithmetic operators) are not presently supported.

Once all the paths have been converted into quantifier-free first-order formulae (a collection of logical constraints imposed on the input parameters), the Z3 solver is used to determine if these constraints are collectively satisfiable. For each path, this process is undertaken incrementally, in the order that the constraints are encountered during the execution of a given path. E.g. for Python program #2 shown above, for branch 2, the satisfiability of the `x > 10` will be checked first, followed by the satisfiability of `x > 10 & x != 15`, and finally followed by the satisfiability of `x > 10 & x != 15 & x > 25`. This means of implementation allows us to detect which conditional constraint exactly, if any, renders the set of constraints unsatisfiable and the path unreachable. By storing the span of different blocks along with this information, this way, we can also easily determine which, if any, lines within a given computer program are dead or unreachable.

For instance, consider the function given below. Here, we store the lines 6-11 associated with the `elif x > 25` branch. When computing the satisfiability of constraints, this constraint is evaluated in conjunction with the ones that precede it `x > 10 & x < 20` before the conditional branches that are nested in its body `x < 40`, `x >= 40` are computed, this allows us to immediately determine that it is this constraint which results in the path being unsatisfiable, meaning lines 6-11 are now dead (unreachable) code.

```python
def single_variable_function(x):
    if x > 10 and x < 20:
        if x < 13:
            assert(x != 2)
            return "A"
        elif x > 25:
            assert(x == 15)
            if x > 40:
                return "B"
            else:
                return "C"
        else:
            return "D"
    else:
        if x == 5:
            assert(x > 15)
            return "E"
        else:
            return "F"
```

Figure 8: Python Program #3

3.5 Accumulating Assertions

When designing the process flow for AutoPy, I deemed it necessary to separately handle conditional constraints and assertions. This was because of the distinct and often complementary usage of conditional statements and assertions in a program, i.e., it is necessary to handle these different types of constraints in different ways and separately from one another because they are each usually utilised for different purposes in a given Python program. Conditional statements, on the one hand, primarily play the role of controlling program flow and meticulously delineating the program's response to a wide range of inputs and situations.

On the other hand, while sometimes temporarily used for debugging, assertions are most often used for error checking (to check for unexpected states and situations that should never occur in normal operation) and for enforcing contracts, such as preconditions that check the validity of inputs to a function, post-conditions to check the outputs, and invariants of functions or classes that ensure that the internal state of an object remains consistent. Assertions, therefore, are used to impose and ensure the validity of logical constraints beyond those incorporated into the natural logical flow of a computer program. As such, it is imperative that these constraints are evaluated separately so that if an error exists (i.e., if a particular set of constraints is unsatisfiable), one can easily pinpoint whether such an error arises from a conditional statement (in which case there is likely an error in the logic and handling of the program itself) or from an assertion (in which case some additional property required of the inputs is not satisfiable).

Originally, in order to implement a method to accumulate the different sets of assertions present in a given Python program, I attempted to override the `visit_Assert` method of the `NodeVisitor` class. This would also allow for the internal representation of the collection of assertions to remain distinct from the conditional constraints. However, this implementation would add significant overhead, both in terms of space and complexity, because of the need to associate assertions with their specific path in a program, i.e., to determine which specific assertions in a program should be inviolable when evaluating the satisfiability of a particular

path in the program.

Therefore, after attempting this implementation, I made the decision to instead incorporate information about assertions in the source code into the aforementioned conditional tree by representing each assertion in the code as an `If_Node`, albeit with no corresponding `else` branch. The `If_Node` class was augmented with an additional field to encode information about whether the node in question represented a conditional or assertion constraint. This way, we are still able to distinguish between conditional constraints and assertions on a given path while also being able to determine which assertions must be inviolable for a given path.

3.6 Ascertaining Satisfiability of Assertions

Once a particular path has been deemed to be satisfiable (by solving the constraints on the given path and obtaining a `sat` result) and the assertions for each path have been collected, AutoPy uses the Z3 solver to check the satisfiability of any pertinent assertions (assertions on a given path) under the preset path constraints. This involves adding these assertions to the Z3 solver's context and checking for satisfiability. If the path is deemed `unsat` at this point, then we know that the unsatisfiability is a consequence of a specific assertion as opposed to a necessary characteristic of a given path.

3.7 Output

The output produced by the AutoPy test-case generation engine consists of the following components:

1. The number of possible execution paths in a given Python program.
2. The collection of conditional and assertion-derived constraints necessary for a path to be satisfiable.
3. A list of test values for each input parameter for each satisfiable path.
4. Each dead path, the conditional statement or assertion that results in the path becoming unsatisfiable, and (in the case of conditional unsatisfiability), the portion of the program that is now dead or unreachable.

3.7.1 Generating Test Cases

If a particular path is deemed to be satisfiable after solving both the conditional constraints and validating the assertions on the path in conjunction with the conditional constraints, then the next step is to generate a set of test cases that results in the execution of this path. In order to enable this, AutoPy supplies each variable with a constraint placed on it with the value generated for it by `Z3`'s solver, and for the rest of the input parameters, numerical values between 0 and 100 are randomly generated. This then becomes the test case that results in the execution of a given satisfiable path.

3.7.2 Diagnosing Unsatisfiability

In the case that a particular path is discovered to be unsatisfiable, the engine first identifies it, reporting the branch number and the list of path constraints that constitute it. It then identifies the specific condition that caused the path to first become unsatisfiable. The figure below shows the output for the Python program given above.

```
Number of branches: 6
Branch number: 1
Branch conditions: ['x > 10 and x < 20', 'x < 13', 'x != 2']
Branch status: UNSAT with assertion x != 2
Status: UNSAT with assertion x != 2

Branch number: 1
Branch conditions: ['x > 10 and x < 20', 'x < 13', 'x != 2']
Branch status: SAT
Test values: {'x': 10}

Branch number: 2
Branch conditions: ['x > 10 and x < 20', 'not x < 13', 'x > 25', 'x == 15', 'x > 40']
Branch status: UNSAT
Status: Dead Code (UNSAT)
The condition that caused the dead code: x > 25. Lines 6 - 11 are unreachable.

Branch number: 3
Branch conditions: ['x > 10 and x < 20', 'not x < 13', 'x > 25', 'x == 15', 'not x > 40']
Branch status: UNSAT
Status: Dead Code (UNSAT)
The condition that caused the dead code: x > 25. Lines 6 - 11 are unreachable.

Branch number: 4
Branch conditions: ['x > 10 and x < 20', 'not x < 13', 'not x > 25']
Branch status: SAT
Test values: {'x': 49}
Branch number: 5
Branch conditions: ['not (x > 10 and x < 20)', 'x == 5', 'x > 15']
Branch status: UNSAT with assertion x > 15
Status: UNSAT with assertion x > 15

Branch number: 6
Branch conditions: ['not (x > 10 and x < 20)', 'not x == 5']
Branch status: SAT
Test values: {'x': 36}
```

Figure 9: Python Program #3

1. Path is unsatisfiable because of a conditional constraint. If this constraint is part of a conditional block, the engine identifies the specific condition, and then the line numbers corresponding to the `if` or the `or-else` block, i.e., the portion of the code that will always be dead (unreachable). This is made possible by the incremental construction of the Z3 model, i.e. because the Z3 model is built up, starting from the outermost conditions and working its way chronologically through the program, when a particular condition renders the formula unsatisfiable, it must also be the case that this innermost (of the conditional blocks traversed thus far) conditional block is unreachable.

2. Path is unsatisfiable because of an assertion. If the path is rendered unsatisfiable by an assertion, then it is already known that this path is reachable (not dead code) since we check the collective satisfiability of constraints in chronological order. In this case,

the engine simply reports the specific assertion that is failed, also reporting that the branch is rendered `unsat` only by an assertion (i.e., the collection of conditionals on that path are conjunctively satisfiable).

4 Results

This section presents the results obtained from testing this constrained automated test-case generation engine, AutoPy, and applying it to a variety of sample Python programs, with the first section containing manually authored programs and the second containing a list of programs generated by GPT 3.5, to showcase the engine's capability in handling a multitude of code structures and complexities. Each section showcases one example program complete with code, description, and results and contains a table to showcase the output for all the other programs, with the actual code for those Python programs presented in the appendix.

4.1 Manual Evaluation

4.1.1 Python Program #1: Single-Variable Function with Nested Conditionals & Assertions Results

This example consists of a simple Python program with nested conditionals and assertions.

Branch No.	Status	Conditions	Test Values
1	UNSAT (Assertion x == 2)	['x > 10 and x < 20', 'x < 13', 'x == 2']	N/A
2	UNSAT (Dead Code)	['x > 10 and x < 20', 'not x < 13', 'x > 25', 'x == 15', 'x > 40']	N/A
3	UNSAT (Dead Code)	['x > 10 and x < 20', 'not x < 13', 'x > 25', 'x == 15', 'not x > 40']	N/A
4	SAT	['x > 10 and x < 20', 'not x < 13', 'not x > 25']	{'x': 63}
5	UNSAT (Assertion x > 15)	['not (x > 10 and x < 20)', 'x == 5', 'x > 15']	N/A
6	SAT	['not (x > 10 and x < 20)', 'not x == 5']	{'x': 74}

4.1.2 Results for Remaining Manually-Crafted Examples

The actual results for each of these programs is included in the Appendix and may also be generated by running the accompanying tool on the programs provided in the same directory.

#	# Branches	# SAT Branches	# UNSAT Branches	# UNSAT (Assertion)
1	6	2	4	2
2	11	9	2	1
3	12	10	2	0
4	5	3	2	1
5	5	5	0	0

4.2 Evaluation on GPT-Generated Examples

The second half of this evaluation entailed automatically generating test-cases to test this automated test-case generation engine by prompting GPT 3.5 to generate test cases based on a list of specifications pertaining to the limitations of the engine and specifying that the programs must vary both in complexity and in terms of what they test (prompt included in Appendix).

The following table summarises the results of using AutoPy with these test cases (actual test cases presented in Appendix), wherein the first column corresponds to the Program Number (#) and the second corresponds to Branch #.

#	B.#	Conditions	Status	Test Values
1	1	['x > 0']	SAT	{'x': 92}
	2	['not x > 0']	SAT	{'x': 76}
2	1	['x > y']	SAT	{'x': 21, 'y': 39}
	2	['not x > y']	SAT	{'x': 19, 'y': 86}
3	1	['x > 0', 'y > 0']	SAT	{'x': 32, 'y': 48}
	2	['x > 0', 'not y > 0']	SAT	{'x': 65, 'y': 66}
	3	['not x > 0']	SAT	{'x': 57, 'y': 97}
5	1	['x > 0', 'x < 0']	Dead Code (UNSAT)	
	2	['x > 0', 'not x < 0']	SAT	{'x': 48, 'y': 79}
	3	['not x > 0']	SAT	{'x': 35, 'y': 23}
5	1	['x > 0', 'x < 0']	Dead Code (UNSAT)	
	2	['x > 0', 'not x < 0']	SAT	{'x': 50}
	3	['not x > 0']	SAT	{'x': 60}
6	1	['x > 0', 'y > 0', 'x == y']	SAT	{'x': 13, 'y': 3}
	2	['x > 0', 'y > 0', 'not x == y']	SAT	{'x': 24, 'y': 21}
	3	['x > 0', 'not y > 0']	SAT	{'x': 94, 'y': 95}
	4	['not x > 0']	SAT	{'x': 74, 'y': 95}
7	1	['b > a', 'c < b']	SAT	{'a': 89, 'b': 21, 'c': 14}
	2	['b > a', 'not c < b']	SAT	{'a': 31, 'b': 48, 'c': 84}
	3	['not b > a']	SAT	{'a': 15, 'b': 43, 'c': 55}
8	1	['x > 0', 'y < 0']	SAT	{'x': 84, 'y': 6}
	2	['x > 0', 'not y < 0', 'x < 0']	Dead Code (UNSAT)	
	3	['x > 0', 'not y < 0', 'not x < 0']	SAT	{'x': 79, 'y': 51}
	4	['not x > 0']	SAT	{'x': 90, 'y': 6}

9	1	['x > y', 'x > 0', 'y != 0']	UNSAT with assertion	
	2	['x > y', 'x > 0', 'not y != 0']	UNSAT with assertion	
	3	['not x > y']	SAT	{'x': 98, 'y': 93}
10	1	['x > 0', 'y > 0', 'x < 0']	Dead Code (UNSAT)	
	2	['x > 0', 'y > 0', 'not x < 0']	SAT	{'x': 25, 'y': 90}
	3	['x > 0', 'not y > 0']	SAT	{'x': 58, 'y': 48}
	4	['not x > 0', 'x > 0']	Dead Code (UNSAT)	
	5	['not x > 0', 'not x > 0']	SAT	{'x': 72, 'y': 31}

Table 1: Summary of Results for GPT-Generated Test Cases

5 Discussion

The testing of the automatic test-case generation engine revealed a comprehensive performance profile across a spectrum of Python functions. The engine successfully processed a majority of the test cases, accurately handling various conditional structures, assertions, and input configurations. Notably, the engine demonstrated robustness in tests with basic conditionals and nested structures, validating its foundational logic and processing capabilities.

For the constrained subset of Python constructs that we set out to support with this test-case generation engine, it indubitably performed well, exhibiting proficiency in handling primitive numerical data types, control flow and nested conditionals, single and multiple input parameters, and assertion violations. It was successful in **handling nested conditionals with single and multiple input parameters**, where it managed to correctly evaluate and generate test cases for multiple layers of logic. Moreover, for tests with deliberately contradictory conditions (such as $x > 0$ and $x < 0$ simultaneously), the engine was able to satisfactorily mark these **dead (unreachable)** branches as being UNSAT, even correctly identifying the specific conditional constraint responsible for this result, as well as the line numbers of the code that was dead as a consequence.

Similarly, for assertions for a given path, the engine correctly handled inviolable assertions, adding them to the collection of constraints to be accounted for by Z3 and generating appropriately-satisfactory test values. For necessarily failing assertions, the engine declared the branch UNSAT, identifying that the cause for this was not the unreachability of a particular segment of code but that the asserted condition somehow contradicted the conditional constraints on the present path.

More thorough and higher-coverage testing is undoubtedly essential to ensure this engine's robustness and ascertain its accuracy and functionality across a myriad of input programs with varied structures and constructs.

5.1 Future Work

In this project, I undertook the conceptualisation and development of AutoPy: an automated test-case generation engine for a constrained subset of the Python language, based on a combination of the techniques of symbolic execution and SMT solving. This engine's capabilities

were deliberately limited to certain crucial constraints of Python, focusing on support for functions, complex conditional statements, and assertions. As discussed in the first section of the paper, one of the most significant limitations of software validation through formal verification is that there are certain systems that cannot be formally modelled. For instance, modelling and verifying properties of probabilistic and continually-evolving machine-learning and deep-learning models, which are increasingly used in commercial software development, remains a complex and challenging task. However, in spite of this, it is incontrovertibly possible to significantly expand the scope of this project to support a wider range of Python programs with more intricate constructs that are more representative of real-world coding scenarios.

5.1.1 Loops and Non-Numeric Data Types

The first step to expand the scope of AutoPy would be to add support for non-numeric primitive Python data types, such as booleans, strings, etc. To do so would primarily entail additions to the `convert_ast_to_z3` function and more thorough parsing of the initial AST to collect such information about the program.

Adding support for loops will undoubtedly also be an important next step, due to the significance and frequency of occurrence of this programming construct. Apart from supporting test-case generation to ensure code coverage within a loop body, this engine could also be made to ensure loop termination, albeit such support may increase work on the user's end by necessitating the input of pre- and/or post-loop invariants.

5.1.2 Complex Data Structures

Implementing support for lists, sets, dictionaries, and other more complex data types is also of the essence. This would also significantly expand the types of conditional checks that AutoPy is able to undertake (e.g., membership checks). Especially given that Z3 supports the theories of arrays, including support for array declaration, indexing and updates, constraints about array elements and lengths, etc., this should undoubtedly be an achievable task, although it would call for additional work on the theoretical end to determine what comprehensive test-case generation means in the context of these constructs.

5.1.3 Support for External Libraries and Function Calls

The next step which would substantially expand the repertoire and complexity of programs that AutoPy can support would be to add support external libraries and function calls. While it may be difficult to add support for each of these libraries and functions, a much more achievable output may be to reason through the symbolic of a function provided as input to AutoPy by applying guarantees on the output values of external functions and library calls made within the input program (i.e. where these libraries and other functions supply AutoPy with constraints on output values that are guaranteed to be true given a particular input parameter).

5.1.4 Multiple Functions, Multiple Independent Conditions, Execution

An additional and easier way to increase the scope and add more intricacies to the functionality of AutoPy would be to simply add support for multiple functions within a single program and multiple independent conditions within a single function. While the former calls for a minor modification in the design of AutoPy, the latter simply calls for the multiple independent conditions in a program to be modelled as conditional trees with disjunctions between them.

6 Conclusion

In this project, I undertook the task of building an initial version of AutoPy: an automated test-case generation for Python for a constrained subset of constructs within the Python language.

Often, the quality of a test suite is assessed by using code coverage criteria. A coverage criterion aims at measuring how well the program under test is exercised by a test suite. Statement coverage, branch coverage, and path coverage are all instances of popular coverage criteria. This automatic test-case generator focuses on the third of these three, i.e. path coverage. This way, by increasing code coverage, we are able to systematically explore different paths, including edge cases that may otherwise be missed in manual testing, leading to unexpected bugs and security vulnerabilities.

Such an approach not only enhances the efficiency and effectiveness of software testing but enables us to truly reason about the completeness and hence the reliability of a testing suite. It's a vision of a development process where reliability is not left to chance, where we can assert, with a higher degree of certainty, that our software systems will perform as expected under a myriad of conditions. The implications of this shift are profound, particularly in critical systems where the cost of failure is immeasurably high, and give us an insight of what it might look like to use formal verification methods to redefine our approach to software testing, where robustness is not just hoped for, but is meticulously crafted and confidently assured.

References

[1] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. Burlington, MA, USA: Morgan Kaufmann, 2007.

[2] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[3] "ast — abstract syntax trees." https://docs.python.org/3/library/ast.html. Accessed: [Insert date here].

[4] J. Avigad, "Using smt solvers." https://avigad.github.io/lamr/using$_s mt_s olvers.html. Accessed$: $[Insertdatehere]$.

[5] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, Springer, 2008.

7 Appendix

7.1 Python Programs for Manual Evaluation

7.1.1 Python Program #1: Single-Variable Function with Nested Conditionals & Assertions

```
def single_variable_function(x):
    if x > 10 and x < 20:
        if x < 13:
            assert(x != 2)
            return "A"
        elif x > 25:
            assert(x == 15)
            if x > 40:
                return "B"
            else:
                return "C"
        else:
            return "D"
    else:
        if x == 5:
            assert(x > 15)
            return "E"
        else:
            return "F"
```

7.1.2 Python Program #2: Two-Variable Function with Nested Conditionals & Assertions

```
def two_variable_function(x, y):
    if x > 10 and y < 200:
        if y < 5:
            assert(y == 25)
            return "A"
        elif y < 10:
            return "B"
        else:
            if x < 15:
                return "C"
            else:
```

```
                return "D"
    else:
        if x == 5:
            if y > 10:
                return "E"
            elif y == 8:
                    if x > 15:
                        return "F"
                    else:
                        return "G"
            else:
                return "H"
        elif x > 3:
            if y == 0:
                return "I"
            else:
                return "J"
        else:
            return "K"
```

7.1.3 Python Program #3: Three-Variable Function with Nested Conditionals

```
def three_variable_function(x, y, z):
    if x > 10:
        if y < 20:
            if z > 5:
                return "A"
            elif y > 30:
                return "B"
        else:
            if z < 15:
                return "C"
            else:
                return "D"
    else:
        if x == 5:
            if y > 10:
                if z != 3:
                    return "E"
                else:
                    return "F"
            else:
                return "G"
        elif x > 3:
            if y < 5:
```

```
                if z >= 8:
                    return "H"
                else:
                    return "I"
            else:
                return "J"
        else:
            if z <= 2:
                return "K"
            else:
                return "L"
```

7.1.4 Python Program #4: Single-Variable Function with Nested Conditionals

```
def nested_conditionals(x):
    if x > 10:
        if x == 15:
            return "A"
        else:
            if x < 2:
                return "B"
    else:
        if x > 5:
            return "C"
        else:
            if x == 0:
                assert(x > 20)
                return "D"
            else:
                return "E"
```

7.1.5 Python Program #5: Simple Two-Variable Function with Nested Conditionals

```
def two_variable_function(x, y):
    if x > 10:
        if y < 5:
            return x + y
        else:
            return x - y
    elif x == 5:
        if y > 10:
            return x * y
        else:
            return x / y
```

```
    else:
        return x ** y
```

7.2 GPT Prompt for Test-Case Generation

I have created an automatic test-case generation engine for Python with the following constraints:

- It only takes a single function as input.
- The function can have any number of inputs.
- The function can have conditionals, nested conditionals, and assertions.
- Only numerical variables are allowed; no loops or other constructs.
- All the `if` branches should have `else` branches.
- No program should be shorter than 5 lines.
- Multiple independent conditions can't exist.
- The return value does not matter.

I want you to generate 20 sample programs for me to use to test my test-case generation engine. These programs must adhere to the constraints. Generate a mix of programs where some have assertions that will be violated by the nature of the branch, and some have dead branches/unreachable code, etc. Dead branches will only occur due to contradicting conditions between 'if' statements nested at different levels, i.e., because of the conditions not being satisfiable together. No other kind of dead code is tested for.

Give me the specific thing each program tests and the specific programs. Target specific different elements of my engine, starting more basic and building up.

7.3 GPT-Generated Python Programs

7.3.1 Python Program 1: Basic Functionality

Test Focus: Basic function with a single input and a simple conditional.

```
def test_function(x):
    if x > 0:
        return x
    else:
        return -x
```

7.3.2 Python Program 2: Dual Input

Test Focus: Function with two inputs and a conditional.

```
def test_function(x, y):
    if x > y:
        return x - y
    else:
        return y - x
```

7.3.3 Python Program 3: Nested Conditionals

Test Focus: Nested conditionals.

```
def test_function(x, y):
    if x > 0:
        if y > 0:
            return x + y
        else:
            return x - y
    else:
        return x * y
```

7.3.4 Python Program 4: Dead Branch with Assertions

Test Focus: Dead branch in a function with assertions

```
def test_function(x, y):
    assert x != y
    if x > 0:
        if x < 0:  # Dead branch
            return x + y
        else:
            return x - y
    else:
        return y
```

7.3.5 Python Program 5: Dead Branch

Test Focus: Dead branch due to contradictory conditions.

```
def test_function(x):
    if x > 0:
        if x < 0:  # This is a dead branch
            return x
        else:
            return -x
    else:
        return x
```

7.3.6 Python Program 6: Multi-level Nested Conditionals

Test Focus: Multiple levels of nested conditionals.

```
def test_function(x, y):
    if x > 0:
        if y > 0:
            if x == y:
                return 0
            else:
                return x + y
        else:
            return x - y
    else:
        return x * y
```

7.3.7 Python Program 7: Complex Assertions

Test Focus: Handling of multiple inputs with an assertion.

```
def test_function(a, b, c):
    assert a > 0
    if b > a:
        if c < b:
            return c
        else:
            return b
    else:
        return a
```

7.3.8 Python Program 8: Multi-conditional with Dead Branch

Test Focus: Multi-conditional function with a dead branch.

```
def test_function(x, y):
    if x > 0:
        if y < 0:
            return x + y
        else:
            if x < 0:  # Dead branch
                return x
            else:
                return y
    else:
        return x - y
```

7.3.9 Python Program 9: Assertion with Nested Conditionals

Test Focus: Function with assertion and nested conditionals.

```
def test_function(x, y):
    assert x > 0
    if x > y:
        if y != 0:
            return x / y
        else:
            return y - x
    else:
        return x * y
```

7.3.10 Python Program 10: Complex Dead Branch Scenario

Test Focus: Complex scenario with dead branches due to contradictory conditions.

```
def test_function(x, y):
    if x > 0:
        if y > 0:
            if x < 0:  # Dead branch
                return x
            else:
                return y
        else:
            return x + y
    else:
        if x > 0:  # Dead branch
            return x - y
        else:
            return y
```

7.4 Results for Manually-Crafted Python Programs #2 - #5

7.4.1 Results for Python Program #2: Two-Variable Function with Nested Conditionals & Assertions

Branch No.	Status	Conditions	Test Values
1	UNSAT (Assertion y == 25)	['x > 10 and y < 200', 'y < 5', 'y == 25']	N/A
2	SAT	['x > 10 and y < 200', 'not y < 5', 'y < 10']	{'x': 91, 'y': 23}
3	SAT	['x > 10 and y < 200', 'not y < 5', 'not y < 10', 'x < 15']	{'x': 68, 'y': 72}
4	SAT	['x > 10 and y < 200', 'not y < 5', 'not y < 10', 'not x < 15']	{'x': 57, 'y': 50}
5	SAT	['not (x > 10 and y < 200)', 'x == 5', 'y > 10']	{'x': 100, 'y': 70}
6	UNSAT (Dead Code)	['not (x > 10 and y < 200)', 'x == 5', 'not y > 10', 'y == 8', 'x > 15']	N/A
7	SAT	['not (x > 10 and y < 200)', 'x == 5', 'not y > 10', 'y == 8', 'not x > 15']	{'x': 87, 'y': 48}
8	SAT	['not (x > 10 and y < 200)', 'x == 5', 'not y > 10', 'not y == 8']	{'x': 35, 'y': 98}
9	SAT	['not (x > 10 and y < 200)', 'not x == 5', 'x > 3', 'y == 0']	{'x': 15, 'y': 48}
10	SAT	['not (x > 10 and y < 200)', 'not x == 5', 'x > 3', 'not y == 0']	{'x': 78, 'y': 93}
11	SAT	['not (x > 10 and y < 200)', 'not x == 5', 'not x > 3']	{'x': 89, 'y': 82}

7.4.2 Results for Python Program #3: Three-Variable Function with Nested Conditionals

Branch No.	Status	Conditions	Test Values
1	SAT	['x > 10', 'y < 20', 'z > 5']	{'x': 43, 'y': 28, 'z': 11}
2	UNSAT (Dead Code)	['x > 10', 'y < 20', 'not z > 5', 'y > 30']	N/A
3	SAT	['x > 10', 'not y < 20', 'z < 15']	{'x': 33, 'y': 87, 'z': 82}
4	SAT	['x > 10', 'not y < 20', 'not z < 15']	{'x': 37, 'y': 6, 'z': 93}
5	SAT	['not x > 10', 'x == 5', 'y > 10', 'z != 3']	{'x': 62, 'y': 76, 'z': 19}
6	SAT	['not x > 10', 'x == 5', 'y > 10', 'not z != 3']	{'x': 89, 'y': 64, 'z': 2}
7	SAT	['not x > 10', 'x == 5', 'not y > 10']	{'x': 34, 'y': 82, 'z': 74}
8	SAT	['not x > 10', 'not x == 5', 'x > 3', 'y < 5', 'z >= 8']	{'x': 11, 'y': 79, 'z': 69}
9	SAT	['not x > 10', 'not x == 5', 'x > 3', 'y < 5', 'not z >= 8']	{'x': 10, 'y': 32, 'z': 81}
10	SAT	['not x > 10', 'not x == 5', 'x > 3', 'not y < 5']	{'x': 46, 'y': 24, 'z': 69}
11	SAT	['not x > 10', 'not x == 5', 'not x > 3', 'z <= 2']	{'x': 21, 'y': 0, 'z': 28}
12	SAT	['not x > 10', 'not x == 5', 'not x > 3', 'not z <= 2']	{'x': 13, 'y': 23, 'z': 76}

Table 2: Test Case Results for a Three-Variable Function with Nested Conditionals

7.4.3 Results for Python Program #4: Single-Variable Function with Nested Conditionals

Branch No.	Status	Conditions	Test Values
1	SAT	['x > 10', 'x == 15']	{'x': 63}
2	UNSAT (Dead Code)	['x > 10', 'not x == 15', 'x < 2']	N/A
3	SAT	['not x > 10', 'x > 5']	{'x': 62}
4	UNSAT (Assertion x > 20)	['not x > 10', 'not x > 5', 'x == 0', 'x > 20']	N/A
5	SAT	['not x > 10', 'not x > 5', 'not x == 0']	{'x': 83}

Table 3: Test Case Results for Python Program #4

7.4.4 Results for Python Program #5: Simple Two-Variable Function with Nested Conditionals

Branch No.	Status	Conditions	Test Values
1	SAT	['x > 10', 'y < 5']	{'x': 31, 'y': 14}
2	SAT	['x > 10', 'not y < 5']	{'x': 58, 'y': 80}
3	SAT	['not x > 10', 'x == 5', 'y > 10']	{'x': 5, 'y': 57}
4	SAT	['not x > 10', 'x == 5', 'not y > 10']	{'x': 93, 'y': 24}
5	SAT	['not x > 10', 'not x == 5']	{'x': 38, 'y': 26}

Table 4: Test Case Results for Python Program #5