Automated Testcase Generation using Z3 SMT Solver
for simple Python programs

Thejesh Venkata (114963271) - *MS CS 2022*
Sai Shashank Gaddam (114777984) - *MS CS 2022*
Vivek Maurya (115133804) - *MS CS 2022*

Supervision

– Prof. C.R. Ramakrishnan

Motivation

Code testing is as vital a part of the code deployment pipeline as code writing itself. In an ideal world, code would be perfect and error-free from the get go, but as long as the human element is present, there are bound to be bugs. Some of these bugs may appear in the form of e.g. unreachable statements, resulting from inconsistent logic. As code grows in complexity, so does the difficulty in spotting these insidious bugs.

Unit testing involves covering each branch in code, ensuring that all intended reachable statements are verified. Good unit tests have 100% coverage. As long as code branches have a 1:1 mapping with business logic usecases, 100% unit test coverage translates to 100% business logic coverage, which is an obviously desirable state.

When the human element is factored in, achieving 100% code coverage becomes an error-prone undertaking. Generating test cases automatically that ensure full coverage can help reduce the effect of the human element. Logic errors in code can be discovered looking at the test cases.

Objectives

The primary objective is to create a proof of concept for the usage of Z3, an SMT solver, to automate the generation of test cases. Only simple Python programs are in scope for this project – single function, primitive variables, few branches. A secondary objective is to generate helpful analytical artifacts in the form of graphs that show possible paths of execution. These are used to verify if Z3-generated test cases provide sufficient coverage.

Pieces of the Puzzle

SMT

SMT stands for *Satisfiability modulo theories*. The "Theories" part of SMT represents "additional theories governing the inputs". e.g. if the inputs are real numbers, "additional theories" may refer to the semantics of real numbers - arithmetic operations, comparison, etc. The SMT problem is a generalization of the boolean satisfiability (SAT) problem to non-boolean inputs. SMT can therefore deal with predicates that take as inputs complex objects (in contrast with simple boolean objects) such as integers and vectors. Such data objects are used extensively in even the most basic programs. [1]

Z3

Z3 Theorem Prover is an SMT-solver developed by Microsoft. Z3 provides its own scripting language for general-purpose use. In our project however, we have used the python3 bindings for Z3. Since we'll be analysing and generating test cases for python programs, python bindings turned out to be the most convenient.

Python

Python is an interpreted, dynamically-typed, high-level programming language. Its most popular implementation is CPython, which compiles to bytecode. Execution involves an interpreter interpreting this bytecode.

Modules

1. dis: dis is a disassembler for CPython programs. It allows for analysis of disassembled python bytecode, and provides a more human-friendly view of it.
2. z3-solver: We have used the z3-solver python module which encapsulates python3 bindings for z3.
3. graphviz: Graphviz has been used to generate helpful visualization of a given function's execution paths.

Work Done

Here is an outline of the Test Case Generator.

1. Identify blocks. A block is a set of instructions that is executed sequentially, without branching or jumping. Blocks in bytecode can be identified easily by looking at JUMP statements. The first instruction of the bytecode marks the start of the first block. Other starting statements are statements pointed to by JUMP statements, and statements right after a JUMP statement.
2. Arrange blocks to form a program execution flow graph. This graph shows multiple sequential execution flows from the first block to any terminal block (a terminal block is a block with no successors).
3. On this graph, enumerate all the paths from the first block to every terminal block. For a well-written program with no unreachable statements, there must be a unit test for each path.
4. For each path, identify constraints along the path by analysing the python bytecode that makes up the blocks that make up this path. The individual constraints depend upon the python program statement.
5. Pass these constraints to the SMT solver. If the solver says that these constraints are satisfiable, generate a unit test conforming to these constraints. To reiterate, this is done for each path.
6. Output the unit tests along with some auxiliary program analysis information such as the program execution flow graph and the disassembled python bytecode.

For further reading, refer to [2] `https://youtu.be/xa5oFc7DLU0`.

Current Scope

There are some restrictions on the kind of python programs that can be analyzed and for which test cases can be generated by our generator. These felt sufficient for our proof of concept.

1. Support has been added only for boolean, int, and float function arguments
2. Variable reassignment **is** supported through versioning.
3. Analysis is scoped to a single function. This function may not call other functions.
4. Supported operations for ints and floats are $>, <, >=, <=, ==, !=, +, -, *$. To keep things simple, division and modulo are currently not supported to avoid extra checks for division by zero.
5. Function parameter types must be mentioned.

Future Scope

This test case generator can be extended to support the following.

1. Functions that call other functions.
2. Composite data types like dicts and lists.
3. Division and modulo
4. Leveraging the unsat output from z3 to identify unreachable statements

Outcome

Examples

Example 1

Python Function

```
def func1(a:int,b:int):
    a = a * 1
    b = b + 4
    a = a + 4
    c = a * a
    return a
```

Generated Testcases

```
[('a', '-4'), ('b', '-4')]
```

Bytecode

```
 6           0 LOAD_FAST                0 (a)
             2 LOAD_CONST               1 (1)
             4 BINARY_MULTIPLY
             6 STORE_FAST               0 (a)

 7           8 LOAD_FAST                1 (b)
            10 LOAD_CONST               2 (4)
            12 BINARY_ADD
            14 STORE_FAST               1 (b)

 8          16 LOAD_FAST                0 (a)
            18 LOAD_CONST               2 (4)
            20 BINARY_ADD
            22 STORE_FAST               0 (a)

 9          24 LOAD_FAST                0 (a)
            26 LOAD_FAST                0 (a)
            28 BINARY_MULTIPLY
            30 STORE_FAST               2 (c)

10          32 LOAD_FAST                0 (a)
            34 RETURN_VALUE
```

Code execution flow graph

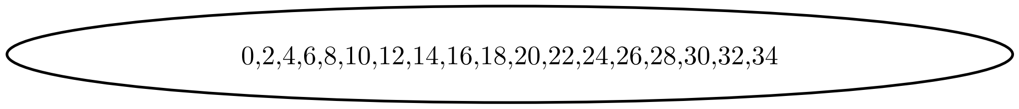

Figure 1: References line numbers from bytecode

Each node in this graph represents a single code block as defined in the "Work Done" section. Since func1 is completely without branches, there is only one node in this graph, and only one test case needed to cover all statements.

Example 2

Python Function

```
def func2(a:int,b:float):
    if a == 1 and b==2.3:
        b = 2
        return 233
    return 333
```

Generated Testcases

```
[('a', '2'), ('b', '0.47')]
[('a', '1'), ('b', '33/10')]
[('a', '1'), ('b', '23/10')]
```

Bytecode

```
 13           0 LOAD_FAST                0 (a)
              2 LOAD_CONST               1 (1)
              4 COMPARE_OP               2 (==)
              6 POP_JUMP_IF_FALSE       12 (to 24)
              8 LOAD_FAST                1 (b)
             10 LOAD_CONST               2 (2.3)
             12 COMPARE_OP               2 (==)
             14 POP_JUMP_IF_FALSE       12 (to 24)

 14          16 LOAD_CONST               3 (2)
             18 STORE_FAST               1 (b)

 15          20 LOAD_CONST               4 (233)
             22 RETURN_VALUE

 16     >>   24 LOAD_CONST               5 (333)
             26 RETURN_VALUE
```

Code execution flow graph

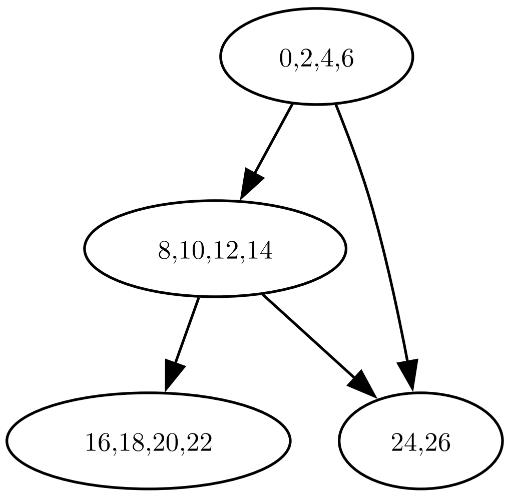

Figure 2: References line numbers from bytecode

Here we can see some branching. At line 6, there is a conditional jump to line 24. If this jump is not taken, program execution continues from line 8, even though it's part of another block since line 8 is a statement right after a JUMP statement.

Note that there are three possible paths from the first block to any leaf block. The three test cases correspond to each of these blocks. Also note that the short-circuited *and* is accounted for - there is a jump straight to the last statement after checking just $a == 1$.

Example 3

Python Function

```
def func3(a:int,b:int):
    if a == 1 and b==2:
        b = 2
        return 233
    elif b==3:
        b = 4
        return
    else:
        a = 1
    return 333
```

Generated Testcases

```
[('a', '2'), ('b', '3')]
[('a', '2'), ('b', '4')]
[('a', '1'), ('b', '3')]
[('a', '1'), ('b', '0')]
[('a', '1'), ('b', '2')]
```

Bytecode

```
 19           0 LOAD_FAST                0 (a)
              2 LOAD_CONST               1 (1)
              4 COMPARE_OP               2 (==)
              6 POP_JUMP_IF_FALSE       12 (to 24)
              8 LOAD_FAST                1 (b)
             10 LOAD_CONST               2 (2)
             12 COMPARE_OP               2 (==)
             14 POP_JUMP_IF_FALSE       12 (to 24)

 20          16 LOAD_CONST               2 (2)
             18 STORE_FAST               1 (b)

 21          20 LOAD_CONST               3 (233)
             22 RETURN_VALUE

 22     >>   24 LOAD_FAST                1 (b)
             26 LOAD_CONST               4 (3)
             28 COMPARE_OP               2 (==)
             30 POP_JUMP_IF_FALSE       20 (to 40)

 23          32 LOAD_CONST               5 (4)
             34 STORE_FAST               1 (b)

 24          36 LOAD_CONST               0 (None)
             38 RETURN_VALUE

 26     >>   40 LOAD_CONST               1 (1)
             42 STORE_FAST               0 (a)

 27          44 LOAD_CONST               6 (333)
             46 RETURN_VALUE
```

Code execution flow graph

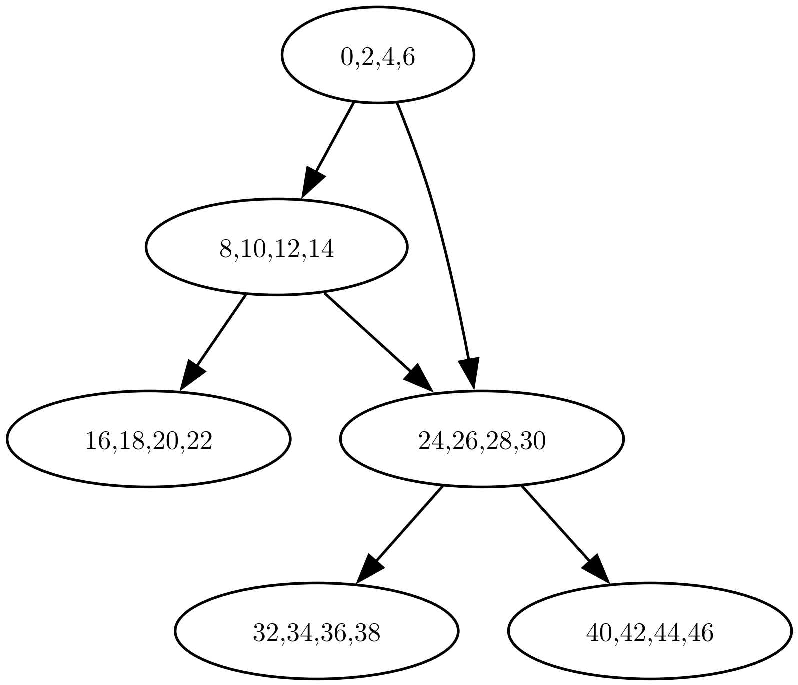

Figure 3: References line numbers from bytecode

Example 4

Python Function

```
def func4(a:int,b:int,c:bool):
    if a==3 or c:
        if b==4:
            return
        else:
            return
    elif a==2 and b==4:
        return
    else:
        return
```

Generated Testcases

```
[('a', '3'), ('b', '4'), ('c', 'True')]
[('a', '3'), ('b', '5'), ('c', 'False')]
[('a', '2'), ('b', '5'), ('c', 'False')]
[('a', '2'), ('b', '4'), ('c', 'False')]
[('a', '0'), ('b', '6'), ('c', 'False')]
[('a', '4'), ('b', '4'), ('c', 'True')]
[('a', '4'), ('b', '5'), ('c', 'True')]
```

Bytecode

```
 30           0 LOAD_FAST                0 (a)
              2 LOAD_CONST               1 (3)
              4 COMPARE_OP               2 (==)
              6 POP_JUMP_IF_TRUE         6 (to 12)
              8 LOAD_FAST                2 (c)
             10 POP_JUMP_IF_FALSE       14 (to 28)

 31     >>   12 LOAD_FAST                1 (b)
             14 LOAD_CONST               2 (4)
             16 COMPARE_OP               2 (==)
             18 POP_JUMP_IF_FALSE       12 (to 24)

 32          20 LOAD_CONST               0 (None)
             22 RETURN_VALUE

 34     >>   24 LOAD_CONST               0 (None)
             26 RETURN_VALUE

 35     >>   28 LOAD_FAST                0 (a)
             30 LOAD_CONST               3 (2)
             32 COMPARE_OP               2 (==)
             34 POP_JUMP_IF_FALSE       24 (to 48)
             36 LOAD_FAST                1 (b)
             38 LOAD_CONST               2 (4)
             40 COMPARE_OP               2 (==)
             42 POP_JUMP_IF_FALSE       24 (to 48)

 36          44 LOAD_CONST               0 (None)
             46 RETURN_VALUE

 38     >>   48 LOAD_CONST               0 (None)
             50 RETURN_VALUE
```

Code execution flow graph

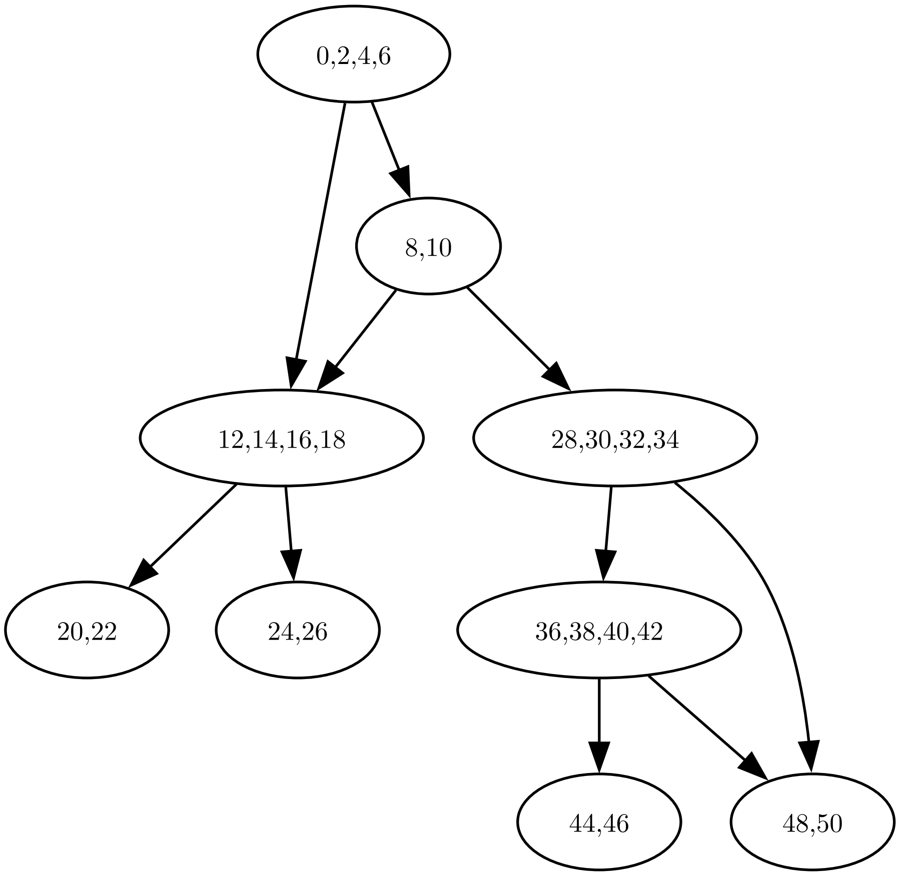

Figure 4: References line numbers from bytecode

Note again that there are 7 possible paths from the first block to any terminal block, and there are 7 test cases generated.

Further Reading

1. z3: `https://github.com/Z3Prover/z3/wiki`
2. z3-python: `https://z3prover.github.io/api/html/z3.html`
3. dis: `https://docs.python.org/3/library/dis.html`
4. graphviz: `https://graphviz.readthedocs.io/en/stable/manual.html`
5. The Fuzzing Book: `https://www.fuzzingbook.org/html/00_Table_of_Contents.html`

References

[1] Wikipedia - satisfiability modulo theories. [Online]. Available: https://en.wikipedia.org/wiki/Satisfiability_modulo_theories

[2] J. M. at PyCon AU, "Let's Build a Symbolic Analyser and Automatically Find Bugs" - Jon Manning (PyCon AU 2019)," Uploaded to YouTube, August 2019.