Software Architecture and Engineering 2015
Project Part 2

Published: 23 April, 2015
Submission: 24 May, 2015 (midnight); this is a firm deadline

Your task is to implement symbolic execution engine for loop free python programs with integers and booleans. An example of such program is:

```
def twice(v):
    return v + v

def main(x, y):
    z = twice(y)
    if (x == z):
        if (x > y + 10):
            return -1
        return 1
    return 0
```

The goal of symbolic execution is to automatically explore all possible execution paths of a given function (in this case `main`) by treating input variables as symbolic rather than concrete values. In our example, there exist three possible execution paths which return the values $-1, 0$, and 1. The symbolic execution engine should be able to generate concrete inputs that execute them.

In addition to executing all feasible program paths, symbolic execution can be used to test that an assertion holds for all inputs or to find an input that violates the assertion. For example, in following program an assertion is used to check that the `abs` function is implemented properly:

```
def abs(v):
    if v >= 0:
        return v
    else:
        return -1 * v

def main(x):
    assert( abs(x) >= 0 and abs(x) == abs(-x))
    return 0
```

1 Description

Python Interpreter. You are given a simple interpreter (file `symbolic_engine.py`) that handles a subset of the python language constructs:

- Statements `If`, `Assign`, `Return` and `Assert` (implemented in `run_stmt` function)
- Expressions `BinOp`, `UnaryOp`, `BoolOp`, `Compare`, `Call` and `Tuple` (implemented in `run_expr` function)

Additionally class `FunctionEvaluator` provides a simple wrapper for executing a function. The interpreter operates on the abstract syntax tree[1]. Note that although the interpreter supports invoking functions defined by the input file, functions defined in the standard library (e.g., `hash`, `abs`) are not supported.

Running Interpreter. To run the interpreter on a python program, a script `py_sym.py` is provided with following usage:

```
$ ./py_sym.py --help
Usage:
    # execute main function with arguments initialized to 0
    ./py_sym.py run <python_file>
    # execute main function symbolically
    ./py_sym.py eval <python_file>
```

When executing a program (either concretely or symbolically) the following steps are performed:

- Parse the input program into a python abstract syntax tree representation.
- Find the function called `main`.
- Execute the function with all inputs initialized to 0 or symbolically depending on option used.

Input Files Format. The input files should contain a function called `main` consisting of the statements supported by the interpreter. Additionally, auxiliary functions can be used and called from the function `main`. All functions need to explicitly return a value, otherwise the interpreter throws an exception.

For evaluation purposes, the input file may define a function `expected_result` which simply returns a list of expected return values of the function `main`. An example of an input file is:

[1]https://docs.python.org/2/library/ast.html

```python
def main(x, y):
    if (x > y):
        return 1
    elif (x < y):
        return -1
    else:
        return 0

def expected_results():
   return [-1,0,1]
```

Symbolic Execution Engine. Your goal is to implement the provided class `SymbolicEngine`, specifically the `explore` method, which performs symbolic execution of the given input program. This method is invoked upon executing the `py_sym.py` script with the `eval` command line option. The return value of `explore` is a tuple containing two elements:

- a list of tuples `[(input#1, ret#1), (input#2, ret#2), ...]`, where `input` is a dictionary containing input values to the `main` method and `ret` is corresponding return value of `main`. Each tuple describes one feasible execution path in the `main` method (and possibly any other methods it calls internally). As an example the expected output of `SymbolicEngine.explore` for the simple program shown above is:

  ```
  [
    ({'y': 0, 'x': 0}, 0),
    ({'y': 0, 'x': 1}, 1),
    ({'y': 1, 'x': 0}, -1)
  ]
  ```

- a dictionary {`'assertion#1': input#1, ...`} that maps assertions that do not hold to concrete inputs that violate them.

Note that any input values that result in given execution path or assertion violation are considered valid. For example instead of {`'y': 0, 'x': 1`} above we could alternatively use {`'y': 0, 'x': 2`} or {`'y': 0, 'x': 3`} as they all result in executing the path `x > y`.

SMT solver You should use the Z3 SMT solver to find satisfying assignments to the formulas that you obtain during your analysis. A sample usage of Z3 from Python is shown in file `z3_test.py`. Additionally, this file includes a function `get_vars` which extracts variables from a Z3 expression. This function is not yet part of the latest Z3 release but you might find it useful in your solution.

2 Solution Requirements

Python Version The python version used should be 2.7.6.

SMT Solver You should use the Z3 SMT solver. Your solution should not invoke the SMT solver more times than there are different execution paths in the program.

Python Libraries Python libraries not in the standard distribution are not allowed.

Solution Files All your solution files should be included inside the `symbolic` folder of the template. You are free to add additional files or even modules as long as there are inside this folder. The submission will be tested by creating an instance of `SymbolicEngine` class and calling `explore` method as shown in `py_sym.py` script.

3 Deliverables

Submit your solution by email to your TA, including:

1. Symbolic execution engine implementation using the provided template that fulfils the requirements in Section 2.
2. Symbolic assertion checking that either verifies all assertions in the program or provides counterexamples that violate them.
3. Set of programs that you have used to test your implementation. For each operation handled by the interpreter there should be at least one program that uses it. The programs should be included in the `test` directory of the project and follow the format of input files. Defining the `expected_results` method is optional. Make sure to include files which contain paths that can never be executed.

4 Grading

The submissions will be graded by invoking the symbolic execution engine on a set of benchmark programs that are handled by the interpreter. A complete solution will provide concrete inputs that exercise all the feasible paths and violate assertions (if any) in the given programs while fulfilling the requirements in Section 2. Note that the solution does not need to use concolic execution to handle cases for which the Z3 SMT solver returns unknown satisfiability.

5 Resources

- Z3 Python Documentation:

  http://research.microsoft.com/en-us/um/redmond/projects/z3/z3.html

- Z3 Installation:

  Download the latest "unstable" release from http://z3.codeplex.com/releases (click on the "Planned" link on the right to get the binaries for all platforms) into folder `Z3HOME`. Add `Z3HOME\bin` to `PATH` and `PYTHONPATH` system variables.