Python test generator

Paolo Tonella (Software Institute, Università della Svizzera italiana, Lugano, Switzerland)

Goal of the project

Write a search based automated test generator for Python. *The generator shall maximize condition coverage of the functions under test and will be compared against a random fuzzer used as baseline.*

1. **Write an instrumentation script that transforms the Python code under test to enable computation of the coverage fitness function [mandatory: 6/10]**
2. Develop a fuzzer that generates new test cases randomly or by mutating/crossing over previously created tests [mandatory: 6/10]
3. Use the library Deap to define a genetic algorithm that evolves test case inputs so as to maximize condition coverage [optional: 8/10]
4. Use the tool MutPy to inject artificial faults (mutations) into the benchmark functions under test and evaluate the fault detection capability of the genetic algorithm, considering the random fuzzer as baseline [optional: 10/10]

Instrumentation

example.py

```python
def f(a: int, b: int) -> int:
    if a > 0:
        if b < 0:
            return a
    if b > 0:
        if a < 0:
            return b
    if a > b:
        return a
    else:
        return b
```

instrumentor.py

example_instrumented.py

```python
from instrumentor import evaluate_condition


def f_instrumented(a: int, b: int) -> int:
    if evaluate_condition(1, 'Gt', a, 0):
        if evaluate_condition(2, 'Lt', b, 0):
            return a
    if evaluate_condition(3, 'Gt', b, 0):
        if evaluate_condition(4, 'Lt', a, 0):
            return b
    if evaluate_condition(5, 'Gt', a, b):
        return a
    else:
        return b
```

- Define a subclass of ast.NodeTransformer and define a visit method for node type FunctionDef (to add "_instrumented" at the end of the function name) and a visit method for node type Compare (to replace a Compare expression with a Call to evaluate_condition).
- Define a function evaluate_condition(num, op, lhs, rhs) that computes the branch distance to the true and to the false branch, to be stored into some global variables, and returns True when the distance to the true branch is zero (i.e., the true branch is satisfied); False otherwise.
- The global variables containing the distances to the true/false branches are updated with the minimum between the previously stored value (if any) and the new one.
- The benchmark of functions under test to be instrumented is available inside the folder `benchmark`.

Instrumentation: implement evaluate_condition

The following relational operators should be supported by function evaluate_condition(num, op, lhs, rhs)

Types	Cmp	Op	True dist	False dist
`int`, `str` with len == 1	<	Lt	lhs - rhs + 1 if lhs >= rhs; 0 otherwise	rhs - lhs if lhs < rhs; 0 otherwise
`int`, `str` with len == 1	>	Gt	rhs - lhs + 1 if lhs <= rhs; 0 otherwise	lhs - rhs if lhs > rhs; 0 otherwise
`int`, `str` with len == 1	<=	LtE	lhs - rhs if lhs > rhs; 0 otherwise	rhs - lhs + 1 if lhs <= rhs; 0 otherwise
`int`, `str` with len == 1	>=	GtE	rhs - lhs if lhs < rhs; 0 otherwise	lhs - rhs + 1 if lhs >= rhs; 0 otherwise
`int`, `str` with len == 1	==	Eq	| lhs - rhs |	1 if lhs == rhs; 0 otherwise
`int`, `str` with len == 1	!=	NotEq	1 if lhs == rhs; 0 otherwise	| lhs - rhs |
`str` with len > 1	==	Eq	edit_distance(lhs, rhs)	1 if lhs == rhs; 0 otherwise
`str` with len > 1	!=	NotEq	1 if lhs == rhs; 0 otherwise	edit_distance(lhs, rhs)

For comparisons between string variables:

- if the strings being compared have length one, convert them to integers (e.g., via ord(s)) and apply the branch distances already defined for integers
- if the strings being compared have length greater than one, use `nltk.metrics.distance.edit_distance` as distance metric; in such a case, only equality/inequality need to be supported

Instrumentation

Conditions inside assertions or return statements should not be replaced by a call to evaluate_condition

```
def f(a: int, b: int) -> int:
    assert a > 0 and b > 0
    if a > b:
        return a > b
    else:
        return a > b
```

```
from instrumentor import evaluate_condition


def f_instrumented(a: int, b: int) -> int:
  assert a > 0 and b > 0
  if evaluate_condition(1, 'Gt', a, b):
        return a > b
  else:
        return a > b
```

Recursive calls to an instrumented function should use the new function name, with suffix "_instrumented"

```
def f(a: int, b: int) -> int:
    if a < b:
        return f(b, a)
    return a - b
```

```
from instrumentor import evaluate_condition


def f_instrumented(a: int, b: int) -> int:
    if evaluate_condition(1, 'Lt', a, b):
        return f_instrumented(b, a)
    return a - b
```

Python test generator

Paolo Tonella
Software Institute, Università della Svizzera italiana, Lugano, Switzerland