CS453: Coursework 2
Search Based Test Data Generator Tool: SBTGen

Bekzat Tilekbay ID: 20180884

Contents

1 Introduction

SBTGen is the automated test data generation tool for the subset of Python programs. It consumes source code file written in Python with functions and generates input arguments for branches in those functions. In other words, it gives input parameters for a particular function that will visit the specific branch. It uses *Iterated Pattern Search* for data generation but can be extended to use other methods.

Assumptions

Since developing test data generation tools for Python is a difficult task because of the dynamic typing aspect and etc, I focused on a specific subset of Python programs. Here are several assumptions:

- functions in the target *.py* file take only integer arguments, possibly multiple of them
- can contain *for* and *while* loops
- contains predicates that only involve relational operations ($==, !=, \Leftarrow, >=, <, >$), integer variables, and calls to functions with integer return type
- can call external functions, including external libraries
- considers only *If-else* statements for branching, but does not include such statements nested in *for-else* or *while-else.*

2 Documentation

Usage Information

The usage of the tool is trivial, you download the **tool** directory, which contains all the crucial components and use the command line:
python3 ./tool/covgen.py <input.py> -a <method> -v <0/1> -f <function name>
where:

- **./** is the directory where downloaded **tool** folder is placed
- <**input.py**> is the directory of target *.py* file for which you want to generate input data
- **-a** <**method**> is the method that can be used to generation of data, only supported method is *Iterated Pattern Search* and it is set as the default.
- **-v** <**0/1**> is the option that if **1** or **true** entered **./tool/covgen.py** will also print the **line number** of each true branch along the test data generated.
- **-f** <**function_name**> is the option with which you can specify the target function in the target Python code. You can specify only one function.

Environment

- OS: **Linux 16.04+**
- **Python 3.6.10**
- Installed **Astor - AST observe/rewrite** module

PS: These are my laptop configurations

Directory

```
tool
 ├── covgen.py Default.py
 ├── worker
 │    ├── Searcher.py
 │    ├── Runner.py
 │    ├── Modifier.py
 │    ├── Evaluator.py
 │    ├── Transformer.py
 │    └── Checker.py
 ├── methods
 │    ├── Generic.py
 │    └── PatternSearch.py
 └── data
      ├── Vector.py
      └── ObjectiveVal.py
```

Default Values

```
# Usage of covgen.py
USAGE_DEF = 'usage: python3 covgen.py <input.py> -a <method> -v <0/1> -f <function
    name>'


# Timeout (not implemented)
TIMEOUT_DEF = 15


# Method of search, currently only Iterated Pattern Search is available
AVM_DEF = 'IteratedPatternSearch'

# Acceleration factor for Pattern Search
```

```python
ACC_FACTOR_DEF = 2

# K that is used for Approach Level calculations
K_DEF = 1

# Normalization function of Approach Level. Must be an integer from [0,1]
NORMALIZATION_DEF = 1

# Maximum iterations of explorations in the Pattern Search
MAX_ITERATIONS_DEF = 1000


# Some kind of verbose feature, print out line-numbers for True branches
ADD_LINENO = False


# Function name for which we generate inputs, by default, we generate inputs for
    all functions in the given source code
FUNC_NAME_DEF = None


# Rerun of Method (more specifically, Iterated Pattern Search by default)
RERUN_DEF = 30

# Modified source code
source = ''


# Do not print output
NO_OUTPUT = False

# File where we collect answers
collect = 'collector_999999.txt'

# File where we put output of Default.source
modify = 'modify_999999.txt'
```

- I wanted to use *NO_OUTPUT* value for testing purposes but did not have enough resources to accomplish the task
- I have used .txt (*collect*, *modify*) files to store output data for ease of development. However, it can be easily turned into JSON objects, or have random file names to avoid confusion and the same-named files in the directory. They will be created in *current directory (./)* and will be deleted after the tool finishes its work unless the tool ends with some error or timeout.

3 Approach

Generally, the approach is straightforward and resembles **AVMf Framework**. Overview of the procedures of the tool is the following:

1. **covgen.py** gets the command line arguments and parses it, to take the target source code directory and options;
2. **covgen.py** calls **Modification.py** which will adjust the source code (add some print statements) so that it would contain all the elements that are needed;
3. (Minor) **Modification.py** interacts with **Transformer.py** in order to get changed expressions for approach level calculations;
4. Then **covgen.py** calls **Searcher.py** that will fix the branches one by one and find appropriate inputs to cover it;
5. For each branch **Searcher.py** will call **Generic.py** which performs *Iterated Pattern Search* and returns the generated data to **Searcher.py**;
6. **Generic.py** along performing *Iterated Pattern Search* interacts with **Runner.py** to run the source code with generated input data and evaluates the output in **modify** file by calling **Evaluator.py**.
7. **Searcher.py** receives the generated data and checks its validity by **Check.py** which reruns the **Runner.py** with generated data.
8. **Searcher.py** writes the returned data into **collect** file.
9. **covgen.py** prints the output by parsing the **collect** file.

Main files

- **covgen.py**
  - The main program that controls the tool, it parses the input, then modifies the target source code. Also it calls the crucial **Searcher.py** and prints the output.
  - It deletes the **collect** and **modify** files.
- **Default.py**
  - Stores all the default values needed for the tool. The values can be adjusted appropriately.

Workers

- **Searcher.py**
  - Contains class that recursively traverses the AST and fixes the branch.
  - Along traversing the tree, stores *path_branch* which is sequence of branches visited until current branch.
  - Calculates the input data needed by calling **Generic.py** can run it *RERUN_DEF* times and every time the *Vector* object is initialized differently.
  - After receiving answer from **Generic.py** and checks it with **Check.py**.
  - Forms the output format that will be collected in **collect** file.
- **Runner.py**
  - Runs the target function with the generated arguments.
  - Stores the output in **modify** file.

- **Modifier.py**
  - Contains class that traverses AST and modifies the source code by integrating appropriate print functions.
  - It also calls **Transformer.py** to change the conditionals into Distance Functions to calculate *Approach Level* for *Iterated Pattern Search.*
- **Evaluator.py**
  - Evaluates the output after running target code (i.e finds *Branch Distance* and *Approach Level*).
  - Parses Distance Functions and gets the **Approach Level**.
  - Interacts with **ObjectiveVal**.py to store the **Branch Distance** and **Approach Level**.
  - Contains parentheses parser for parsing Distance Function.
- **Transformer.py**
  - Rewrites the conditional expression to Distance Functions concatenated as conditionals.
  - Example: $(a < b \text{ and } a == b) \rightarrow a - b + K \text{ and } (abs(a - b))$.
  - Does the reversed procedure for *else* branches. For example, treats $a < b$ as $a >= b$ and *and* as *or* and vice versa.
- **Checker.py**
  - Reruns the source code with the generated data to ensure its correctness.
  - Uses **Runner.py** for this purpose.

Methods

- **Generic.py**
  - Generic function that encompasses all search methods.
  - By default, it interacts with **PatternSearch.py** and perform *Iterated Pattern Search.*
  - Uses **Vector.py** to create appropriate vector from arguments of the target function.
  - Here we randomly initialize the *Vector* object, thus supporting **Searcher.py**.
  - Can be extended by implementing new search methods here.
- **PatternSearch.py**
  - Performs *Pattern Search*, implements exploration and pattern moves
  - Interacts with **ObjectiveVal.py** to compare *Branch Distance + Approach Level*

Data

- **Vector.py**
  - Implements object class *Vector* that are convenient to use in search methods.
- **ObjectiveVal.py**
  - Implements objective value, in this case, *Branch Distance* and *Approach Level.*
  - First initializes them with large numbers.

Performance

It is really difficult to calculate the practical time and memory performance of the tool, but I can only theoretically predict the time consumption as:

$$O(n * m * RERUN * MX\_ITER * log_{ACC}(MX))$$

$n = \#of\_branches$
$m = \#of\_args$
$RERUN = RERUN\_DEF$
$MX\_ITER = MAX\_ITERATIONS\_DEF$
$ACC = ACC\_FACTOR\_DEF$
MX =maximum possible value of arguments

4 Ideas for future upgrades

Termination Policy: There is no global monitoring system of the tool that could have calculated time and perform procedures before and after the execution of the tool. In fact, having such a system is a great advantage in the sense that, it will increase the quality and sustainability of the product and even can help for developing purposes (You will not care about what if tool catches an error, unexpected behavior, or timeout). Thus, it is a top-priority feature that must be implemented and integrated into the tool.

Run-time Calculation: If we will not perform this with **Termination Policy** above, it may be difficult to implement, store, and use it, this comes as a consequence of **Termination Policy**.

Cache for target file executions: It is an optimization idea to lower the time consumption of the tool. From the tool design, it is clear that the targeted source code runs many times and also arguments may be repeated. Thus, by storing the outputs of such runs, I expect that tool can optimize its time consumption, but using more memory.

More methods of search-based data generation can be implemented: I tried to design the tool so that it is easy to integrate new search methods, this is also one of the major paths of the tool development that should be considered.

Widening assumptions, i.e including a broader subset of Python programs: I feel like, this upgrade may become arbitrary difficult, but as a starting point, we can integrate the support of *for-else* and **while-else** statements.

Some UI: It is obvious that users need better interaction with the tool.

Adapting for IDEs and Windows: Ideally, if the tool would be integrated into popular IDEs, it could make testing and debugging procedures easier. Also, we mainly depend on **Linux** right now, so supporting **Windows** in the future is a good idea to increase the number of users.

5 Summary

Overall, the coursework was very interesting and I hope I will find the power to develop **SBT-Gen** further. Aside from theoretical knowledge gained from lectures, it was a great opportunity to practice and widen my knowledge of Python. Ironically, the only pity side of the work, is that I could not test the tool properly. I hope sample tests will be enough.

6 Reference

Astor - AST observe/rewrite
AST - Abstract Syntax Tree
AVMf Framework Github
KAIST - CS453 Slides