CS569 Project

Name: Wen-Yin Wang
Student ID: 932-482-357

Background

Developing software needs multiple steps to achieve the goal that could be published. Expect for programming, testing is another important part that every software engineers should take it seriously. Testing helps developers to find bugs and handling exceptions, which is necessary for comprehensive and perfection. So, I took this testing course and trying to design my novel algorithm and implement my first test generation program. Before I took this course, I have never learned anything about TSTL. So, the first step for me is doing research job. I read some papers related to TSTL on the Internet and also read the papers that instructor provided on class website. And I also do some researches on test generation methods. After academic paper research, I decided to implement my algorithm using random testing. My algorithm is based on random testing, so, we can expect that the coverage and efficiency will similar with random test generation.

Algorithm and Implement

Main function handles all procedures. At first, I set up all variables used in my program. Then, calling Parse_args and Make_config for setting environment. Next, I divided the testing into two phases. Phase 1 is normally do randomAct in a for loop that executes config.Depth times. It does some checking jobs. First, if the result from randomAct is not ok then break. Second, if the config.Running is 1, then print out new branches and new statements we found in randomAct. Third, it checks if it config.Timeout during the loop multiple times, if the execution time exceeds the config.Timeout, then break. After we finished the loop, it collects all coverages. In the phase 2, it similar with phase 1, but it used a while loop to test. Within the while loop, it saves test when new branches or statements found. If the random function returns value greater than 0.03 which means there are 30% tests will backtrack to previous state, it call sut.backtrack() to previous saved test. Then, it executes as phase 1. In this phase, I am trying to cover more branches and statements. Since the coverage is slightly more than normal random testing, my program will get better results. After all testing, it checks three things. First, if config.coverdetail is 1, then it calls printCoverage which produces all collected coverage in detail. Second, if config.coverage is 1, then it calls internalReport in SUT module. Third, if config.resultdetail is 1, then print out the testing results.

```
def main():
        initialize variables
        arsed_args, parser = parse_args()
        config = make_config(parsed_args, parser)
        print "Starting Phase 1"
        if not config.timeout:
                for i in 1 to config.depth:
                        if not randomAct():
                                break
                        if config.running:
                                print out new branches and statements
```

```
                        if config.timeout:
                                break
                collect coverage
        print “Starting Phase 2”
        while not config.timeout:
                savedTest = None
                if (savedTest != None) and (R.random() > 0.3):
                        sut.backtrack(savedTest)
                for i in 1 to config.depth:
                        if not randomAct():
                                break
                        if config.running:
                                print out new branches and statements
                        if config.timeout:
                                break
                collect coverage
        if config.coverage:
                sut.internalReport()
        print results
```

Parse_args creates a parser that contains all required parameters and storing all command line parameters into a variable named parsed_args. And then, returning parser and parsed_args. It is a simple function that helps user to use this program easily.

```
def parse_args():
        parser = argparse.ArgumentParser()
        parser.add_argument('-t', '--timeout', type = int, default = 300, help = 'Timeout in seconds’)
        parser.add_argument('-s', '--seed', type = int, default = None, help = 'Random seed.')
        parser.add_argument('-d', '--depth', type = int, default = 10, help = 'Maximum search depth')
        parser.add_argument('-w', '--width', type = int, default = 10, help = 'Maximum search width')
        parser.add_argument('-f', '--faults', type = int, default = 0, help = 'Faults display.')
        parser.add_argument('-c', '--coverage', type = int, default = 0, help = 'Coverage display.')
        parser.add_argument('-r', '--running', type = int, default = 0, help = 'Running display.')
        parser.add_argument('-p', '--check', type = int, default = 0, help = 'Checking property.')
        parser.add_argument('-sc', '--silence', type = int, default = 0, help = 'Coverage silence.')
        parser.add_argument('-cd', '--coverdetail', type = int, default = 0, help = 'Display coverage in detail.')
        parser.add_argument('-rd', '--resultdetail', type = int, default = 0, help = 'Display result in detail.')
        parsed_args = parser.parse_args(sys.argv[1:])
        return (parsed_args, parser)
```

Make_config is another function for setting environment. It takes parser and parsed_args from function, parse_args. After calling this function, parameter’s values can be easily called by config. It is easy to understand in developing and maintaining.

```
def make_config(parsed_args, parser):
        pdict = parsed_args.__dict__
        key_list = pdict.keys()
        arg_list = [pdict[k] for k in key_list]
        Config = namedtuple('Config', key_list)
        nt_config = Config(*arg_list)
        return nt_config
```

RandomAct is a function doing random test. In this function, it calls the randomenabled function in SUT module using R as a random seed and storing the result value in variable, act. After getting act, it adds 1 to variable actCount, which records the number of acts already happened. And then, it checks whether the act is

safe by safely function in SUT module and it checks whether the sut property is ok. If the act is not safe or config.check is 1 and property is not pass check, then calling the failure function, which handles situations when bug found. Else situations, which means act is safe, will going to savedTest function. After all, it will return the safely result of act. This function is mainly handling the random testing and related works when bugs found or exceptions occurred.

```
def randomAct():
        act = sut.randomenabled(R)
        actCount += 1
        ok = sut.safely(act)
        propok = sut.check()
        if (not ok) or ((not propok) and (config.check)):
                failure()
        else:
                savingTest()
        return ok
```

Failure is used to handle situation when bug found. At first, it counts 1 to variable, bugs, and collecting coverage. Using reduce function in SUT module. If config.faults is 1, then using sut.saveTest() to save this test into file. Print out fail's situation and restart sut, then returns.

```
def failure():
        bugs += 1
        collect coverage
        R = sut.reduce(sut.test(), sut.fails, True, True)
        if config.faults :
                filename = 'failure%d.test'%bugs
                sut.saveTest(sut.test(), filename)
        sut.prettyPrintTest(R)
        sut.restart()
        return
```

SavingTest is using when new branch or new statement is found. It will save current test. And it mainly effects the testing in phase 2.

```
def savingTest():
        if new branche is found or new statement is found:
                savedTest = sut.state()
        return
```

Executing Results

1. Executing with incorrect command line, it shows the usage with –h as below.

```
Wen-Yins-MacBook-Pro:~ Phoebe$ python mytester.py -h
usage: mytester.py [-h] [-t TIMEOUT] [-s SEED] [-d DEPTH] [-w WIDTH]
                   [-f FAULTS] [-c COVERAGE] [-r RUNNING] [-p CHECK]
                   [-sc SILENCE] [-cd COVERDETAIL] [-rd RESULTDETAIL]

optional arguments:
  -h, --help            show this help message and exit
  -t TIMEOUT, --timeout TIMEOUT
                        Timeout in seconds (300 default).
  -s SEED, --seed SEED  Random seed (default = None).
  -d DEPTH, --depth DEPTH
                        Maximum search depth (10 default).
  -w WIDTH, --width WIDTH
                        Maximum search width (10 default).
  -f FAULTS, --faults FAULTS
                        Faults display (default = 0).
  -c COVERAGE, --coverage COVERAGE
                        Coverage display (default = 0).
  -r RUNNING, --running RUNNING
                        Running display (default = 0).
  -p CHECK, --check CHECK
                        Checking property (default = 0).
  -sc SILENCE, --silence SILENCE
                        Coverage silence (default = 0).
  -cd COVERDETAIL, --coverdetail COVERDETAIL
                        Display collected coverage in detail (default = 0).
  -rd RESULTDETAIL, --resultdetail RESULTDETAIL
                        Display result in detail (default = 0).
```

2. After execution, the program shows the configuration on the top.

```
10-249-165-78:~ Phoebe$ python mytester.py 60 1 1000 100 1 1 1
Testing using config=Config(timeout=60, faults=1, running=1, width=100, depth=10
00, seed=1, coverage=1)
```

3. After execution, it shows result in the following format.

```
Covered 191 branches
Covered 141 statements
Failed 1 times
Total tests 160
Total actions 159494
Total runtime 61.8309619427
```

4. When the input fault is 1, it results multiple failure files in current directory and the content likes below picture.

```
self.p_avl[1] = avl.AVLTree()
self.p_val[2] = 1
self.p_avl[1].insert(self.p_val[2])
self.p_val[1] = 2
self.p_avl[1].insert(self.p_val[1])
self.p_val[2] = 3
self.p_avl[1].insert(self.p_val[2])
```

5. When the input coverage is 1, it shows the internal report.

```
TSTL INTERNAL COVERAGE REPORT:
/Users/Phoebe/avl.py ARCS: 191 [(-1, 6), (-1, 11), (-1, 16), (-1, 31), (-1,
197), (-1, 246), (-1, 258), (-1, 267), (6, -5), (11, 12), (12, 13), (13, -1
, (56, -55), (71, 73), (73, 74), (73, 76), (74, -70), (76, 77), (76, 79), (
, 91), (90, 92), (91, 100), (92, 93), (92, 94), (93, 100), (94, 95), (94, 9
7), (108, 109), (108, 111), (109, -85), (109, 117), (111, 112), (111, 115),
8), (127, 136), (128, 129), (128, 132), (129, 130), (130, 131), (131, 132),
), (140, 141), (141, 142), (142, 143), (143, 126), (149, 150), (150, 151),
), (166, 167), (167, 168), (168, -159), (172, 173), (172, 181), (173, 174),
93), (185, 186), (185, 191), (186, 187), (187, 188), (188, 189), (189, 191)
203), (201, 217), (203, 204), (203, 205), (204, 217), (205, 206), (205, 210
195), (220, 221), (220, 222), (221, 225), (222, 223), (223, 225), (225, -19
, 249), (258, 259), (258, 262), (259, -257), (262, 263), (263, 264), (264,
75, 277), (277, 278), (278, 279), (278, 281), (279, 278), (281, -266)]
/Users/Phoebe/avl.py LINES: 141 [6, 11, 12, 13, 16, 17, 31, 35, 36, 37, 38,
, 104, 105, 106, 108, 109, 111, 112, 115, 117, 124, 125, 126, 127, 128, 129
 162, 163, 164, 166, 167, 168, 172, 173, 174, 175, 176, 177, 179, 181, 184,
217, 218, 219, 220, 221, 222, 223, 225, 227, 246, 247, 249, 250, 251, 252,
TSTL BRANCH COUNT: 191
TSTL STATEMENT COUNT: 141
```

6. When the running is 1, it shows format likes below picture when new branch and new statement found.

```
0.00241899490356 6 New branch (u'/Users/Phoebe/avl.py', (35, 36))
0.00241899490356 6 New branch (u'/Users/Phoebe/avl.py', (-1, 35))
0.00241899490356 6 New branch (u'/Users/Phoebe/avl.py', (37, 38))
0.00241899490356 6 New branch (u'/Users/Phoebe/avl.py', (38, 40))
0.00241899490356 6 New branch (u'/Users/Phoebe/avl.py', (36, 37))
0.00241899490356 6 New branch (u'/Users/Phoebe/avl.py', (40, -34))
0.00241899490356 5 New statement (u'/Users/Phoebe/avl.py', 37)
0.00241899490356 5 New statement (u'/Users/Phoebe/avl.py', 36)
0.00241899490356 5 New statement (u'/Users/Phoebe/avl.py', 38)
0.00241899490356 5 New statement (u'/Users/Phoebe/avl.py', 35)
0.00241899490356 5 New statement (u'/Users/Phoebe/avl.py', 40)
```

7. When a bug found, it shows the detail as below.

```
avl1 = avl.AVLTree()                                    # STEP 0
val2 = 1                                                # STEP 1
avl1.insert(val2)                                       # STEP 2
val1 = 2                                                # STEP 3
avl1.insert(val1)                                       # STEP 4
val2 = 3                                                # STEP 5
avl1.insert(val2)                                       # STEP 6
val3 = 4                                                # STEP 7
avl1.insert(val3)                                       # STEP 8
```

Comparison

Since my algorithm was based on random test generation, I made a comparison between my algorithm and my previous algorithm using pure random test generation. I used same configuration (timeout = 60, seed = 1, depth = 1000, width = 100, faults = 1, coverage = 1, running = 1) to execute both programs, and the results shows in below table. In my program, it covered 191 branches and 141 statements, found 1 bug, test 160 times, total action is 159494, and the total running time is 60.2527 seconds. And in pure program, it covered 180 branches and 135 statements, found 0 bugs, test 148 times, total action is 147317, and the total running time is 60.0035 seconds. We can easily found that my algorithm has higher coverage and executes more times, but it takes more time on execution.

	mytester.py	puretester.py
# of covered branches	191	180
# of covered statements	141	135
# of bugs	1	0
# of tests	160	148
# of total actions	159494	147317
# of total running time	60.2527	60.0035

Conclusion and Future Work

The objective for this project is implementing a test generator using TSTL package. After multiple weeks, I finally produced my program that testes correctly. As I mentioned before, I have never learned TSTL before and it is my first time taking course in software engineering field. Unlike most classmates who have taken CS562 in winter term, TSTL is a brand new language for me. I spent a lot of time on reading papers and trying to understand how it works. In the previous submission, my program was really unstable and it crushed occasionally, especially when bugs found. So, I reconsidered my structure of codes. I added a failure function to handle exceptions, and adopted backtrack function in SUT module to restore previous saved test. And my program is stable after the structure changed. To sum up, my program using random test generation plus random backtrack mechanism produces appropriate results in testing. Although, it is too simple and lack of consideration in this step, we can still expected it can be implement more comprehensive in the future by adding more effective algorithm and functions.

References

1. A. Groce, and J. Pinto, “A little language for testing,” in the 7th NASA Formal Methods Symposium (NFM), 2015.
2. A. Groce, A. Fern, J. Pinto, T. Bauer, A. Alipour, M. Erwig and C. Lopez, “Lightweight Automated Testing with Adaptation-based Programming,” in the Proceedings of the 23rd IEEE International Symposium on Software Reliability Engineering (ISSRE), 2012.
3. A. Arcuri, M. Z. Iqbal and L. Briand, "Random Testing: Theoretical Results and Practical Implications," in IEEE Transactions on Software Engineering, vol. 38, no. 2, pp. 258-277, March-April 2012.