SENG3320

Software Verification and Validation

Assessment #2

Automated Test Data Generation

25 marks (25% of the whole course assessment)
Due 11:59pm, Monday, 15th June 2020

1. Fuzz Testing (12.5%)
2. Symbolic Execution (12.5%)

C3303000 | C3305509 | C3279166 | C3282229

1. Fuzz Testing

Blackbox testing:

Partition(no input)
Partition(Only contains words)
Partition(Only contains Number)
Partition(Only contains special Characters)
Partition(Contains Letters and Numbers)
Partition(Contains Numbers and special characters)
Partition (Contains Letters and special characters)
Partition(Contains Letters, numbers and special characters)

Partition(no input)

- /no input /

Result:

- Works

Partition(contains letters and special characters)

- = a d !

Result:

- Fail

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 4 out of bounds
for length 4
    at KWIC.partition(KWIC.java:790)
    at KWIC.quickSort(KWIC.java:774)
    at KWIC.quickSort(KWIC.java:778)
    at KWIC.newAlphabetizing(KWIC.java:759)
    at KWIC.main(KWIC.java:858)
```

Partition(contains letters and special characters)

- !fghf
- A

Result:

- Works

Partition(contains only special characters)

- !
- {{{{
- {{{

Result:

- Works

Partition(contains letters and special characters)

- A
- !fghf

Result:

- Fail

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 2 out of bounds for length 2

    at KWIC.partition(KWIC.java:790)
    at KWIC.quickSort(KWIC.java:774)
    at KWIC.newAlphabetizing(KWIC.java:759)
    at KWIC.main(KWIC.java:858)
```

Partition(contains letters, numbers and special characters)

- {
- fghf12

Result:

- Fails

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 2

at KWIC.partition(KWIC.java:790)
at KWIC.quickSort(KWIC.java:774)
at KWIC.newAlphabetizing(KWIC.java:759)
at KWIC.main(KWIC.java:858)
```

Partition(contains letters and numbers)

- R4
- 421

Result:

- Fails

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 2

at KWIC.partition(KWIC.java:790)
at KWIC.quickSort(KWIC.java:774)
at KWIC.newAlphabetizing(KWIC.java:759)
at KWIC.main(KWIC.java:858)
```

Partition(contains only numbers)

- 333

Result:

- Works

Partition(Only contains words)

- Ahahdjhaga
- Ajhagjha

Result:

- Works

Partition(contains only special characters)

input : WORKS

- $$$)
- $%#&&#)

Partition(Contains Numbers and special characters)

- $123
- $%#884

Result:

- Works

Test Report:

Readme:

This section is just outlining how to run each of the programs.

Performance Testing:

For the performance testing section you need to open the folder up in intellij and change the arguments to make sense for the file you're testing. The same arguments you would give the kwic program. The only other thing you may have to change is the java settings in the run configuration. For the sake of simplicity I've included the two text files inside the src folder. It should by default be set to run books1. If you want to run books2 all you have to do is swap the 1 to a 2 in the run configuration.

Fuzz Testing:

For the Fuzz testing tool you should be able to just open it up in intellij and run it the only exception being that you may have to change your java settings in the run configuration. The arguments for the fuzz testing are the amount of iterations you want to run for the test.

Test Tool Design:

Functionality Overview:

The tool accepts an argument that represents the amount of times the fuzz testing will be run. The program generates random files on each iteration that have a max line count from 0 to 999. This file is inputted into the KWIC program main method as a parameter. The KWIC program is run in a try catch loop with the exception being checked to see if it has occurred before, and if it has then it's skipped and nothing happens. If it hasn't then the error and it's locations are saved into a file called output. This happens for each loop until the amount of inputted times has occurred. Then the user can access the results in the output file.

ExceptionObj:

ExceptionObj is an object that stores the Exception that occurs. It has a method called compare that compares any two ExceptionObj objects and returns true if they are the same and false if they're different.

These objects are stored inside a linkedlist in the main class and are used to compare against for new exceptions to see if they're unique.

Compare Method:

The compare method takes in a ExceptionObj and compares it against the object running the method. It does this firstly by checking the length of the StackTrace Array. If they're different it means that the exceptions are different so false is returned.

If they're the same then all the each individual stacktrace is compared if at any point they're different then false is returned.

The exception message is checked to see if it contains a : and if it does then each exception 1 and exception 2 are checked separately if it's true then the other one is checked to see if it contains the subset and true is returned if it does. For my clarification check the example below.

example:

These two exceptions are considered the same for deciding whether an exception is unique.

java.lang.StringIndexOutOfBoundsException: offset 969, count 38, length 982

java.lang.StringIndexOutOfBoundsException: offset 9796, count 25, length 9817

They are determined to be the same by checking if the first exception has a : in the name because if it did then it checks whether 1 is a subset of 2. If it is it returns true. The test is also done the opposite way as well.

1.java.lang.StringIndexOutOfBoundsException

2.java.lang.StringIndexOutOfBoundsException: offset 969, count 38, length 982

If these two fail then the final test is to see if the messages aren't the same if they aren't then a false is returned.

If non of the if statements succeeded then true is returned as all of the tests except the : ones were checking to see if they were different.

RandomFileContentGenerator

The RandomFileContentGenerator class contains a number of methods used in order to create and fill a file with random data The class when called will take two arguments, being the file name and line count which will be used when generating the file

When the method generate() is called it will create a new text file and will use a variable named root to place it in the correct directory and the name of the file will be the value of variable fileName

A for loop will then be used in order to repeatedly call the method createRandomCode until the loop counter reaches the max number of lines in the text file

The method createRandomCode will take the code length and the characters available to be used in the string as parameters and use this to generate a random string which in our case will be used as our fuzz data.

Test Cases

test case examples can be found in the folder named data under the text file reportOutputs

Test Results

Running our our fuzz testing tool produced a total of 2 unique errors. A summary of all unique errors and their equivalent stack trace can be found in the folder named data under the text file reportOutputs

Performance Tester:

Functionality Overview:

The performance tester program just takes in the text file of books as an input.(The directory) and just runs it normally except that the time taken to run the program is recorded in nanoseconds.

This is printed in the console saying that the program has finished and shows the time taken.

Logger:

The logger works by initialising the start time in the constructor which is the current time in nanoseconds.

Once the stop method is called then the finish time is made equal to current time minus the start time. This is the total time taken for the kwic program to execute.The getFinished method is then used to return the time taken.

Performance example:

In the src folder of Performance Logger there are two text files called Books1 and Books2. Books1 is the original that is provided with the assignment. Books 2 is a copy of Books1 that has half the amount of books included. The time taken to execute can be seen below:(Note that the time taken can vary run by run).

Books1:

```
Kwic testing completed in:395376900 nanoseconds
```

Books2:

```
Kwic testing completed in:66983300 nanoseconds
```

2. Symbolic Execution

Code Created To Test The Given Function

```
int main() {
	int a,b,c;
	klee_make_symbolic(&a, sizeof(a), "a");
	klee_make_symbolic(&b, sizeof(b), "b");
	klee_make_symbolic(&c, sizeof(b), "c");
	return triangle(a,b,c);
}
```

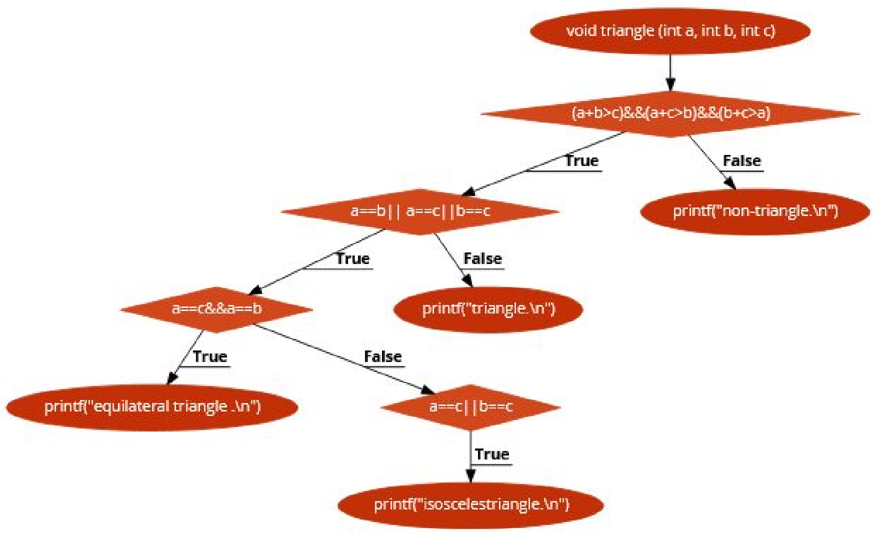
KLEE Generated Data

```
KLEE: done: total instructions = 136
KLEE: done: completed paths = 8
KLEE: done: generated tests = 8
```

TEST	A	B	C	Outcome
Test 1	0	0	0	non-triangle
Test 2	2	16777216	0	non-triangle
Test 3	16777216	0	0	non-triangle
Test 4	1023410176	1023410176	33554432	*
Test 5	16777216	16777216	16777216	Equi-Triangle
Test 6	29360257	29360259	33554432	Triangle
Test 7	1024	4	1024	isoscelestriangle
Test 8	29	4096	4096	isoscelestriangle

** Test 4 shows no result*

KLEE Control-Flow Diagram


void triangle (int a, int b, int c)
(a+b>c)&&(a+c>b)&&(b+c>a)
True
False
a==b|| a==c||b==c
printf("non-triangle.\n")
True
False
a==c&&a==b
printf("triangle.\n")
True
False
printf("equilateral triangle .\n")
a==c||b==c
True
printf("isoscelestriangle.\n")

KLEE Coverage

Statement Coverage:

TEST	A	B	C	Outcome
Test 1	0	0	0	non-triangle
Test 5	16777216	16777216	16777216	Equi-Triangle
Test 6	29360257	29360259	33554432	Triangle
Test 7	1024	4	1024	isoscelestriangle

Decision Coverage:

TEST	A	B	C	Outcome
Test 1	0	0	0	non-triangle
Test 4	1023410176	1023410176	33554432	*
Test 5	16777216	16777216	16777216	Equi-Triangle
Test 6	29360257	29360259	33554432	Triangle
Test 7	1024	4	1024	isoscelestriangle

Condition Coverage:

TEST	A	B	C	Outcome
Test 1	0	0	0	non-triangle
Test 4	1023410176	1023410176	33554432	*
Test 5	16777216	16777216	16777216	Equi-Triangle
Test 6	29360257	29360259	33554432	Triangle
Test 7	1024	4	1024	isoscelestriangle

Condition/Decision Coverage:

TEST	A	B	C	Outcome
Test 1	0	0	0	non-triangle
Test 4	1023410176	1023410176	33554432	*
Test 5	16777216	16777216	16777216	Equi-Triangle
Test 6	29360257	29360259	33554432	Triangle
Test 7	1024	4	1024	isoscelestriangle

Multiple Condition Coverage:

TEST	A	B	C	Outcome
Test 1	0	0	0	non-triangle
Test 2	2	16777216	0	non-triangle
Test 3	16777216	0	0	non-triangle
Test 4	1023410176	1023410176	33554432	*
Test 5	16777216	16777216	16777216	Equi-Triangle
Test 6	29360257	29360259	33554432	Triangle
Test 7	1024	4	1024	isoscelestriangle
Test 8	29	4096	4096	isoscelestriangle

Applying Fuzz Testing/Steps to achieve Fuzz Testing

To test this program we did some initial porting and setup:
1) Firstly we ported it to java, as it is our preferred language
2) We created a program that generates random data for inputs a,b,c
3) Tracks the overall results & will stop the program once all outputs have been met
4) Outputs the sum of results, errored results, time to compute & total runs

Program Specs & Design

- Generate int between 0 -> 10
- Input 3 values into test function
- Get output of function
- Test if all outcomes have been received
- Print Results

Results of Fuzz Testing:

Equilateral triangle: 1
Isosceles triangle: 1
Triangle: 4
Non-triangle: 18
Errored Results: 1

Total Runs: 25
Total Time to compute (milliseconds): 30

Full data can be found in the 'fuzz data.txt' file in the submission folder

Experimental Report Overview

The use of KLEE testing showed 100% statement coverage, branch decision coverage, which also produced a major bug in the code, where an input was given however no output was received, showing that the code was not complete in terms of handling all possible data.

Fuzz testing showed the same, however instead of making 8 calls of the function to create 100% coverage, it took 25 runs, which can differ depending on the random numbers generated. Some other test runs took between 12 -> 750 calls, which makes random data far less desirable than symbolic execution.

KLEE Steps
Inorder to run our KLEE test, we set up the KLEE program via docker on a google cloud compute server (https://klee.github.io/docker/), however our test can be ran on any version of KLEE. To view all data from KLEE (*.ktest files), we used KLEE's ktool command line, specifically we ran the command:

```
ktest-tool klee-last/test00000*.ktest > exportedData.dat
```

To get KLEE's runtime stats we viewed the info file:

```
cat klee-last/info
```

Which Showed:

```
Started: 2020-06-13 03:16:06
BEGIN searcher description
<InterleavedSearcher> containing 2 searchers:
RandomPathSearcher
WeightedRandomSearcher::CoveringNew
</InterleavedSearcher>
END searcher description
Finished: 2020-06-13 03:16:06
Elapsed: 00:00:00
KLEE: done: explored paths = 8
KLEE: done: avg. constructs per query = 112
KLEE: done: total queries = 9
KLEE: done: valid queries = 1
KLEE: done: invalid queries = 8
KLEE: done: query cex = 9
KLEE: done: total instructions = 136
KLEE: done: completed paths = 8
KLEE: done: generated tests = 8
```