ECSE 429 Software Validation Fall 2019
Mutation Testing Simulator

Smith, Brandon
McGill University
brandon.smith@mail.mcgill.ca

Piecaitis, Darius
McGill University
darius.piecaitis@mail.mcgill.ca

Yau Chung, Katherine
McGill University
katherine.yauchung@mail.mcgill.ca

I. Introduction

Software testing is an important step in the software engineering process as every additional layer added to previous modules of code introduces an exponential number of ways with which the code could malfunction. There are many different methods of testing software such as: Conditional Graph Flow Coverage; Boolean Logic Coverage; Fault and Mutation Coverage; and many others.

This paper will be covering **Fault and Mutation Coverage**. Fault and Mutation testing consists of 3 main steps:

1. Generate a list of all possible **Faults** pertaining to your Software Under Test.

2. Generate a new copy of your Software Under Test for each Fault generated in step one. Insert the fault as a **Mutation** into this new copy.

3. Run a **Simulation** that compares the output of the original Software Under Test with each Mutant Copy using a set of input vectors defined around the boundary cases of the Software Under Test.

Faults are possible typos in the code that if left unchecked could result in errors, which may or may not propagate to the output of the Software Under Test.

Mutants are, as implied by step 2 above, copies of the original Software Under Test with exactly one fault inserted into the code.

The goal of this paper is to show the pros and cons of Fault and Mutation Testing. However we have restricted our Fault and Mutation Coverage to the subset: Arithmetic Fault and Mutation Coverage on the arithmetic operators ″+″, ″-″, ″*″, and ″/″. That is we generated three faults for every arithmetic operator found in the code, inserted said three faults into 3 copies of the Software Under Test, and then compared the results between each mutant and the original Software Under Test using a set of input vectors specific to the boundary cases of our Software Under Test.

The Software Under Test for this simulation is a root finding algorithm for the intersection between a quadratic function and a linear function. Both the Software Under Test and the mutant testing software were written in Python.

II. Architectural Decisions

The problem of Fault and Mutation Testing involves numerous string manipulations to read input files, search for arithmetic operators and replace said operators with the 3 other arithmetic operators in mutant copies, i.e. for every found "+" we generate 3 mutant copies of the Software Under Test with "-", "*", and "/" respectively. Also, by the definition of the problem, we need to run $X + 1$ versions of the Software Under Test and each version may be run up to N times where $X =$ Number of Mutants generated and $N =$ Number of input test vectors. This leads to a total complexity of $O(N*(X+1))$. For these reasons we chose to work with Python as our programming language for testing software due to its extensive base library of string manipulation functions and its effective threading library for speeding up the process of running many different mutants through the simulation concurrently.

III. Testing Software

The testing software can be subdivided into two parts:

1. **Generation**
   - (a) Wait for the user to input a path to a Software Under Test.
   - (b) Generate the fault list by reading the Software Under Test line by line, writing 3 entries to the list for each mathematical operator found.
   - (c) Generate a Mutant copy of the original Software Under Test for each fault in the generated fault list.
2. **Simulation**
   - (a) Define the list of test vectors corresponding to the Software Under Test's boundary cases.
   - (b) Run the original Software under test with each input vector in the defined list over 3 different threads to speed up the process, storing the results of all the runs in a single output vector *SUTResults*.
   - (c) Execute each generated mutant for each input vector in the defined list and compare the results to *SUTResults* **killing the mutant** if we find a test vector that results in a different output from the original Software Under Test. If a mutant is killed the test vector that kills it is noted and it stops executing test vectors on said mutant, continuing on to the next mutant. This process is split into 3 different threads, each getting a third of the mutants from the mutant list.

Pseudocode:

```
Generation:
1)  String path <- None
2)  List faultList <- Empty
3)  List mutants <- Empty
4)  While (path == None or path == Invalid)
5)      path <- waitForUserInput()
6)  for each (line in file(path))
7)      for each (operator in line)
8)          addThreeEntriesToFaultList(operator, faultList)
9)  for each (fault in faultList)
10)     generateMutant(fault, file(path), mutants)
Simulation:
11) List inVectors <- defineInVectors()
12) Program sut <- exec(path)
```

```
13)     List resultVector <- Empty
14)     List killedMutants <- Empty
15)     Split into three threads with each thread getting a third of inVectors:
16)         for each (vector in inVectors)
17)             resultVector.add( sut.run(vector) )
18)     Join three threads
19)     Split into three threads with each thread getting a third of mutants:
20)         for each (mutant in mutants)
21)             Program mut <- exec(mutant)
22)             for each (vector in inVectors)
23)                 killedMutants.add( compare(mut.run(vector), resultVector) )
24)                 if (mutantKilled) break
25)     Join three threads
26)     return killedMutants
```

The user can easily determine which mutants were not killed by comparing the killedMutants list to the mutants list.

The first part of the testing software is 100% agnostic of the original type of file of the Software Under Test and will produce correct lists of the faults and mutants regardless of input.

The second part of the testing software however has two parts that are specific to the Software Under Test. It needs to have the list of vectors defining the boundary cases for the Software Under Test, and it also needs to know how to properly execute the original Software Under Test with given inputs from within the simulation.

Therefore in order to execute the developed software on any Software Under Test other than our QuadraticRoots program, one would need to change the test vectors to accurately describe the new boundary cases of said Software Under Test. Additionally, if the new Software Under Test was not executable by a single python script, then the portion of the simulation that executes the Software Under Test and the Mutants would need to be altered to abide by whatever running and/or compiling conditions that need to be met. The **simulation** script was purposely constructed as a separate file from the **generation** script so that it may be easily modified to fit any Software Under Test.

IV. Software Under Test

To test our software, we created a simple Software Under Test titled QuadraticRoots. This Software Under Test simply finds the points of intersection between a parabola and a line if any exist. The Software Under Test takes in 5 input arguments where inputs 1, 2 and 3 correspond to the constant values in the parabola in the order :

$$\textbf{Eq1.} f_1(x) = input_1 x^2 + input_2 x + input_3$$

The inputs 4 and 5 correspond to the constants of the line in the order:

$$\textbf{Eq2.} f_2(x) = input_4 x + input_5$$

The Software Under Test begins by converting the string inputs to floats and then determining if the parabola is a line by checking if $input_1$ is equal to 0. If so, we are calculating the intersection of two lines and three potential outcomes can occur. If $input_2$ is equal to $input_4$ and $input_3$ is equal to $input_5$ then these lines are equivalent and there are infinitely many intersection points along the line and the program will report that. If $input_2$ is equal to $input_4$ but $input_3$ is not equal to $input_5$ then the lines have the same slope but not the same y-intercepts and thus will never intersect. If neither of the previous conditions are met then the lines have a single point of intersection that can be calculated with either of the following equations found by setting $f_1(x) = f_2(x)$:

$$\textbf{Eq3.} x = (input_5 - input_3)/(input_2 - input_4)$$

$$\textbf{Eq4.} x = (input_3 - input_5)/(input_4 - input_2)$$

If the parabola is not a line then the Software Under Test first transforms the equivalence $f_1(x) = f_2(x)$ into $0 = f_1(x) - f_2(x)$ or equivalently $0 = input_1x^2 + input_2x + input_3 - input_4x - input_5 = ax^2 + bx + c$. Once the a,b,c variables are determined, the Software Under Test determines the roots of the equation according to the formula:

$$\textbf{Eq5.} x = (-b \pm \sqrt{mid})/(2a)$$

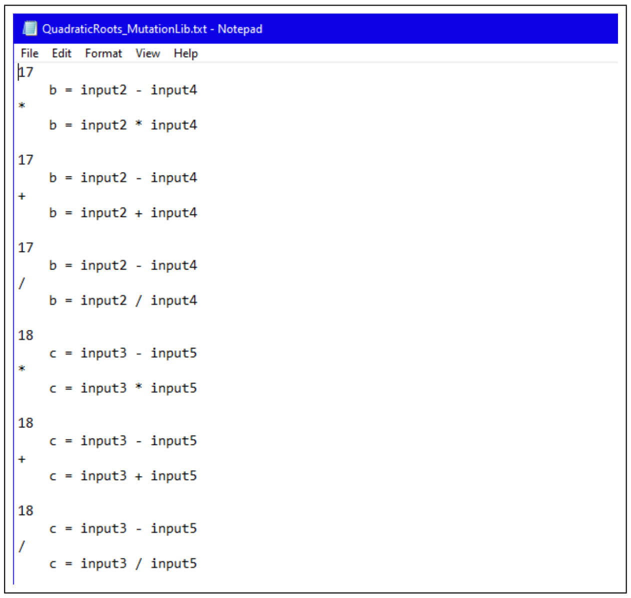
Where mid is equal to $b^2 - 4ac$. Error handling is done so that if mid is less than 0, the program does not attempt to find the roots and simply returns that the parabola and line do not intersect. The case where mid is 0, meaning there is only one point of intersection is treated the same as if it has two intersection points and the program would output the same value twice. Once the roots, or lack thereof, are determined, the program prints out the result.

The natural boundary condition of this Software Under Test is the case when all five inputs are zeroes. and as such test vectors were generated around this boundary meaning each input was set to 1,0 or -1. Our test set was every possible permutation of these options leading to $3^5 = 243$ test vectors.

V. Results and Discussion

booktabs

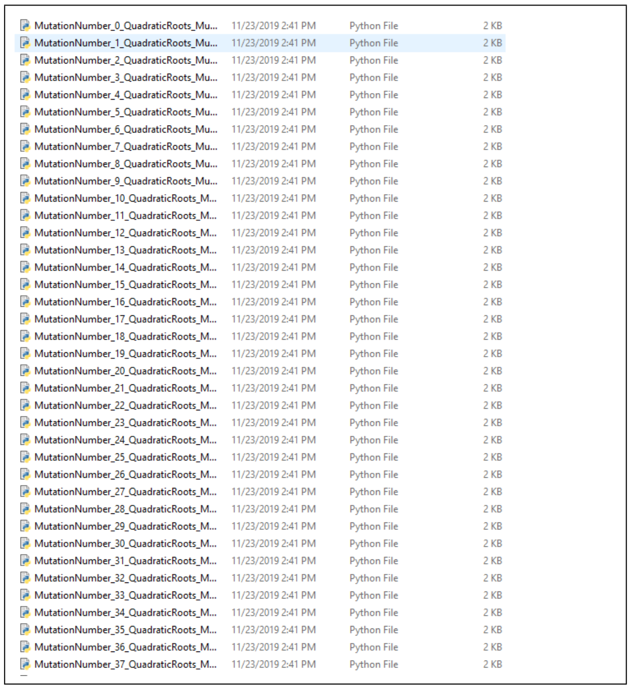
Our testing software was able to successfully produce a mutant library of 78 mutants for our Software Under Test, 3 for each of its 26 mathematical operators as shown in Figure 1.


QuadraticRoots_MutationLib.txt - Notepad
File Edit Format View Help

```
17
    b = input2 - input4
*
    b = input2 * input4

17
    b = input2 - input4
+
    b = input2 + input4

17
    b = input2 - input4
/
    b = input2 / input4

18
    c = input3 - input5
*
    c = input3 * input5

18
    c = input3 - input5
+
    c = input3 + input5

18
    c = input3 - input5
/
    c = input3 / input5
```

Figure 1: *Mutant Libary.*

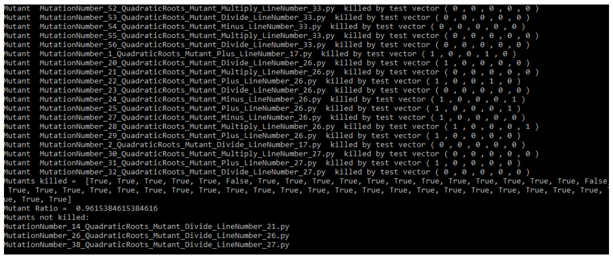
It was also able to produce mutated programs for each of these mutants and store them in a directory titled Mutated_Source_Codes_QuadraticRoots as shown in Figure 2.



```
MutationNumber_0_QuadraticRoots_Mu...   11/23/2019 2:41 PM   Python File   2 KB
MutationNumber_1_QuadraticRoots_Mu...   11/23/2019 2:41 PM   Python File   2 KB
MutationNumber_2_QuadraticRoots_Mu...   11/23/2019 2:41 PM   Python File   2 KB
MutationNumber_3_QuadraticRoots_Mu...   11/23/2019 2:41 PM   Python File   2 KB
MutationNumber_4_QuadraticRoots_Mu...   11/23/2019 2:41 PM   Python File   2 KB
MutationNumber_5_QuadraticRoots_Mu...   11/23/2019 2:41 PM   Python File   2 KB
MutationNumber_6_QuadraticRoots_Mu...   11/23/2019 2:41 PM   Python File   2 KB
MutationNumber_7_QuadraticRoots_Mu...   11/23/2019 2:41 PM   Python File   2 KB
MutationNumber_8_QuadraticRoots_Mu...   11/23/2019 2:41 PM   Python File   2 KB
MutationNumber_9_QuadraticRoots_Mu...   11/23/2019 2:41 PM   Python File   2 KB
MutationNumber_10_QuadraticRoots_M...   11/23/2019 2:41 PM   Python File   2 KB
MutationNumber_11_QuadraticRoots_M...   11/23/2019 2:41 PM   Python File   2 KB
MutationNumber_12_QuadraticRoots_M...   11/23/2019 2:41 PM   Python File   2 KB
MutationNumber_13_QuadraticRoots_M...   11/23/2019 2:41 PM   Python File   2 KB
MutationNumber_14_QuadraticRoots_M...   11/23/2019 2:41 PM   Python File   2 KB
MutationNumber_15_QuadraticRoots_M...   11/23/2019 2:41 PM   Python File   2 KB
MutationNumber_16_QuadraticRoots_M...   11/23/2019 2:41 PM   Python File   2 KB
MutationNumber_17_QuadraticRoots_M...   11/23/2019 2:41 PM   Python File   2 KB
MutationNumber_18_QuadraticRoots_M...   11/23/2019 2:41 PM   Python File   2 KB
MutationNumber_19_QuadraticRoots_M...   11/23/2019 2:41 PM   Python File   2 KB
MutationNumber_20_QuadraticRoots_M...   11/23/2019 2:41 PM   Python File   2 KB
MutationNumber_21_QuadraticRoots_M...   11/23/2019 2:41 PM   Python File   2 KB
MutationNumber_22_QuadraticRoots_M...   11/23/2019 2:41 PM   Python File   2 KB
MutationNumber_23_QuadraticRoots_M...   11/23/2019 2:41 PM   Python File   2 KB
MutationNumber_24_QuadraticRoots_M...   11/23/2019 2:41 PM   Python File   2 KB
MutationNumber_25_QuadraticRoots_M...   11/23/2019 2:41 PM   Python File   2 KB
MutationNumber_26_QuadraticRoots_M...   11/23/2019 2:41 PM   Python File   2 KB
MutationNumber_27_QuadraticRoots_M...   11/23/2019 2:41 PM   Python File   2 KB
MutationNumber_28_QuadraticRoots_M...   11/23/2019 2:41 PM   Python File   2 KB
MutationNumber_29_QuadraticRoots_M...   11/23/2019 2:41 PM   Python File   2 KB
MutationNumber_30_QuadraticRoots_M...   11/23/2019 2:41 PM   Python File   2 KB
MutationNumber_31_QuadraticRoots_M...   11/23/2019 2:41 PM   Python File   2 KB
MutationNumber_32_QuadraticRoots_M...   11/23/2019 2:41 PM   Python File   2 KB
MutationNumber_33_QuadraticRoots_M...   11/23/2019 2:41 PM   Python File   2 KB
MutationNumber_34_QuadraticRoots_M...   11/23/2019 2:41 PM   Python File   2 KB
MutationNumber_35_QuadraticRoots_M...   11/23/2019 2:41 PM   Python File   2 KB
MutationNumber_36_QuadraticRoots_M...   11/23/2019 2:41 PM   Python File   2 KB
MutationNumber_37_QuadraticRoots_M...   11/23/2019 2:41 PM   Python File   2 KB
```

Figure 2: *Generated Mutants.*

The software was then able to execute each of the mutant programs and display which were killed by which vector as well as display which mutants were not killed as shown in Figure 3.



```
Mutant  MutationNumber_52_QuadraticRoots_Mutant_Multiply_LineNumber_33.py  killed by test vector ( 0 , 0 , 0 , 0 , 0 )
Mutant  MutationNumber_53_QuadraticRoots_Mutant_Divide_LineNumber_33.py  killed by test vector ( 0 , 0 , 0 , 0 , 0 )
Mutant  MutationNumber_54_QuadraticRoots_Mutant_Minus_LineNumber_33.py  killed by test vector ( 0 , 0 , 0 , 0 , 0 )
Mutant  MutationNumber_55_QuadraticRoots_Mutant_Multiply_LineNumber_33.py  killed by test vector ( 0 , 0 , 0 , 0 , 0 )
Mutant  MutationNumber_56_QuadraticRoots_Mutant_Divide_LineNumber_33.py  killed by test vector ( 0 , 0 , 0 , 0 , 0 )
Mutant  MutationNumber_1_QuadraticRoots_Mutant_Plus_LineNumber_17.py  killed by test vector ( 1 , 0 , 0 , 1 , 0 )
Mutant  MutationNumber_20_QuadraticRoots_Mutant_Divide_LineNumber_26.py  killed by test vector ( 1 , 0 , 0 , 0 , 0 )
Mutant  MutationNumber_21_QuadraticRoots_Mutant_Multiply_LineNumber_26.py  killed by test vector ( 0 , 0 , 0 , 0 , 0 )
Mutant  MutationNumber_22_QuadraticRoots_Mutant_Plus_LineNumber_26.py  killed by test vector ( 1 , 0 , 0 , 1 , 0 )
Mutant  MutationNumber_23_QuadraticRoots_Mutant_Divide_LineNumber_26.py  killed by test vector ( 0 , 0 , 0 , 0 , 0 )
Mutant  MutationNumber_24_QuadraticRoots_Mutant_Minus_LineNumber_26.py  killed by test vector ( 1 , 0 , 0 , 0 , 1 )
Mutant  MutationNumber_25_QuadraticRoots_Mutant_Plus_LineNumber_26.py  killed by test vector ( 1 , 0 , 0 , 0 , 1 )
Mutant  MutationNumber_27_QuadraticRoots_Mutant_Minus_LineNumber_26.py  killed by test vector ( 1 , 0 , 0 , 0 , 0 )
Mutant  MutationNumber_28_QuadraticRoots_Mutant_Multiply_LineNumber_26.py  killed by test vector ( 1 , 0 , 0 , 0 , 1 )
Mutant  MutationNumber_29_QuadraticRoots_Mutant_Plus_LineNumber_26.py  killed by test vector ( 1 , 0 , 0 , 0 , 0 )
Mutant  MutationNumber_2_QuadraticRoots_Mutant_Divide_LineNumber_17.py  killed by test vector ( 0 , 0 , 0 , 0 , 0 )
Mutant  MutationNumber_30_QuadraticRoots_Mutant_Multiply_LineNumber_27.py  killed by test vector ( 0 , 0 , 0 , 0 , 0 )
Mutant  MutationNumber_31_QuadraticRoots_Mutant_Plus_LineNumber_27.py  killed by test vector ( 1 , 0 , 0 , 0 , 0 )
Mutant  MutationNumber_32_QuadraticRoots_Mutant_Divide_LineNumber_27.py  killed by test vector ( 0 , 0 , 0 , 0 , 0 )
Mutants killed =  [True, True, True, True, True, False, True, True, True, True, True, True, True, True, True, True, True, True, False,
 True, True, True, True, True, True, True, True, True, True, True, True, True, True, True, True, True, True, True, True, True, True,
ue, True, True]
Mutant Ratio =  0.9615384615384616
Mutants not killed:
MutationNumber_14_QuadraticRoots_Mutant_Divide_LineNumber_21.py
MutationNumber_26_QuadraticRoots_Mutant_Divide_LineNumber_26.py
MutationNumber_38_QuadraticRoots_Mutant_Divide_LineNumber_27.py
```

Figure 3: *Results of Mutation Testing.*

Of the 78 mutants, 75 were killed by the test vectors while 3 were not thus resulting in a mutant killed ratio of 96.15%. By displaying which mutants failed it allows us to easily examine the live mutants and examine how and why the the test vectors were inadequate or what more could be added into the code to detect these mutants.

Since each mutant simulation does not interfere with another mutant simulation, simulations are independent from one another. The total simulation time was optimized by running multiple simulations at once using three parallel threads. The table below shows a comparison of the average execution time of the program using one thread and three threads.

Table 1: *Run-time (in s) of single threaded and 3-threaded Mutation Testing Simulator*

Test Run #	1 Thread	3 Threads
1	226.44	107.70
2	231.73	107.64
3	225.42	110.23
4	227.89	110.44
5	221.33	114.70
Average	226.56	110.14

There is a difference of almost 2 minutes between the single threaded and 3-threaded execution. This is a considerable improvement.

$$\begin{aligned}\text{Overall Speedup} &= \frac{\text{ExecTime}_{\text{old}}}{\text{ExecTime}_{\text{new}}} \\ &= \frac{226.56}{110.14} \\ &= 2.06\end{aligned}$$

This is a good example of making the common case fast!

VI. Conclusion

The Software Under Test that was being used for the purposes of this paper was a prime example of how quickly the size of the list of test vectors required for Fault and Mutation Coverage grows. With only 5 inputs we ended up with $3^5 = 243$ test vectors, and by the fact that our simulation only killed 96.15% of the mutants we know that we still were missing a few boundary cases for our test vectors.

Additionally, the fault list generation of our testing software is constrained to the four arithmetic operators "+", "-", "*", and "/" which drastically reduces the sample set of mutants we are generating. A more general fault list could result in exponential increases in the number of mutants scaling with the number of sub-strings we are searching for and replacing.

Fault and Mutation Coverage is a viable option for testing simple programs because it is quick to implement, however the more complicated the Software Under Test becomes and the broader the fault coverage becomes, the more boundary cases need to be considered and the more mutations are generated. Consequently, the time required to run all of the tests increases exponentially.

References

[1] Varsity Tutors. *Solving Linear-Quadratic Systems*. https://www.varsitytutors.com/hotmath/hotmath_help/topics/solving-linear-quadratic-systems.