ECSE 429 Final project
Mutation Testing

Elias Al Homsi, elias.alhomsi@mail.mcgill.ca , 260797449

Nihal Irmak Pakis, nihal.pakis@mail.mcgill.ca, 260733837

Fouad El-Bitar, fouad.elbitar@mail.mcgill.ca , 260719196

Jules Boulay de Touchet, jules.boulaydetouchet@mail.mcgill.ca, 260710129

Contents

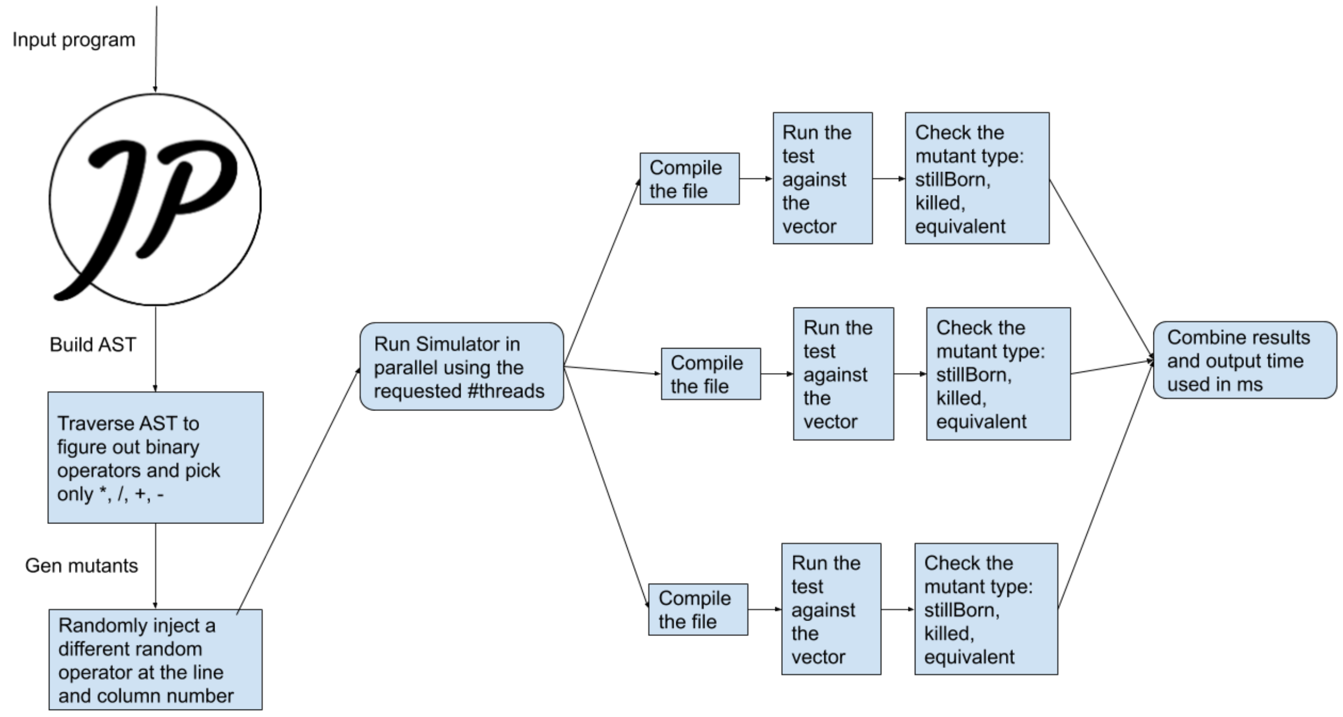
Introduction

Mutation testing is one of the most famous technique to test a piece of software and to ensure that the software is doing what it is designed to do by generating similar programs and ensuring that those programs which are known as “Mutants” are killed in or equivalent during running the software. This ensures that the program written is semantically strong and help boost the confidence in the test vector that is designed to run tests against by comparing the test vector behavior between the original software and the mutants. In case multiple mutants output the same output as the original software, this means that the test vector is not exclusive to the unit under test and we need to refine or add tests in order to make sure that the tests uniquely identify the unit and its behavior.

Mutation testing is essentially cheap in manpower since only the test vector need to be specified and the computer software would take care of generating mutants and validating the tests in order to ensure the uniqueness behavior of the unit under test. The only caveat is choosing a good test vector to ensure that all the edge cases are tested thoroughly and to identify the SUT behavior under all input. For example, specify the software behavior if the input is wrong or malformed.

Break Down

In order to do mutation testing on a software, we need to break down the software into 3 unique phases. The following diagram shows the design and the overall logic behind the software


Input program
JP
Build AST
Traverse AST to figure out binary operators and pick only *, /, +, -
Gen mutants
Randomly inject a different random operator at the line and column number
Run Simulator in parallel using the requested #threads
Compile the file
Run the test against the vector
Check the mutant type: stillBorn, killed, equivalent
Compile the file
Run the test against the vector
Check the mutant type: stillBorn, killed, equivalent
Compile the file
Run the test against the vector
Check the mutant type: stillBorn, killed, equivalent
Combine results and output time used in ms

<Figure 1: Software overall design and approach>

The Above figure shows how the input program gets parsed by a library called Javaparser. **Phase one** starts with Javaparser which is library that parses java code and builds an Abstract Syntax Tree (AST) which would help us identify the position of classes, methods and in this case binary operators. The main reason for using AST instead of simply finding all the characters matching (*, /, +, -) is that those characters can be used as non-binary operators in java code. Examples include comments, documentation, and variable names. For this reason, it was necessary to have a deeper understanding of the code to be able to replace only those operators in a meaningful way. The output of this phase is the operators position specified under **potentialFaults.txt** file. This file lists the potential faults injection locations. **Phase two** follows next by generating the requested number of mutants which is done by choosing a position of the earlier operators at random and replacing it with a random different operator. Once this done, we would generate **mutantsList.txt** which contains the definition of every mutant such as: the original operator, the mutant operator, line number, column number. In this phase we also generate mutant java classes by simply replacing the original operator with the mutant operator and output several java files that are equal in number to the number of mutants in the following form: <Name>Mutant<ID>.java where the mutant id is a number between 1 up to mutants' number. **Phase three** starts with allocating the requested number of threads and then dividing the mutation testing on them equally. Each thread has the following work to do, grab the mutant java file, compile it, load it and run the method under test with each of the test vectors. The thread would determine the result of the run test by comparing its output to the output of the original file. It would also compare the exception thrown to determine under which category does the mutant fall:

Mutants categories:

- **Still born mutant:** still born mutant is a mutant that failed to compile or run the method under test
- **Killed mutant:** this mutant was killed by a specific test vector or a mismatch in the exception thrown from the original code
- **Equivalent mutant:** this mutant is indistinguishable from the original code under the test vector specified.

Once every mutant is classified to one of these 3 categories, the program would output all these results in **simulatorOuput.txt** along with the time used to run all the threads in order to see how parallelism improve runtime.

Sample Input

In this case the sample input is a program that is trying to calculate the digits of PI by specifying the number of digits requested to calculate. At a small number the results are far away from PI however, when the number of requested digits increase, the results gets more and more accurate. This program source is specified below:

```
/**
 * This is a sample program to find the digits of pi the program is referenced
 * from: http://www.codecodex.com/wiki/Calculate_digits_of_pi#Java on Date
 * 10/14/2019
 **/
public class PiDigits {
    private static final int SCALE = 10000;
    private static final int ARRINIT = 2000;

    public static String methodUnderTest(int digits) {
        StringBuffer pi = new StringBuffer();
        int[] arr = new int[digits + 1];
        int carry = 0;

        for (int i = 0; i <= digits; ++i)
            arr[i] = ARRINIT;

        for (int i = digits; i > 0; i -= 14) {
            int sum = 0;
            for (int j = i; j > 0; --j) {
                sum = sum * j + SCALE * arr[j];
                arr[j] = sum % (j * 2 - 1);
                sum /= j * 2 - 1;
            }

            pi.append(String.format("%04d", carry + sum / SCALE));
            carry = sum % SCALE;
        }
        return pi.toString();
    }
}
```

This program was chosen simply because it has all the requested operation *, -, +, / and it is simple to understand in nature. The **methodUnderTest** is the method that would be tested in this program.

Code Parsing Phase 1

The java parser library takes a java input program like PiDigits.java and parses it into an AST that is simple to navigate and discover. A sample way to navigate the AST which his used in implementation is:

```
package mutantfactory.parser;
import com.github.javaparser.ast.Node;

public class NodeIterator {
    public interface NodeHandler {
        void handle(Node node);
    }

    private NodeHandler nodeHandler;
    public NodeIterator(NodeHandler nodeHandler) {
        this.nodeHandler = nodeHandler;
    }

    public void explore(Node node) {
        nodeHandler.handle(node);
        for (Node child : node.getChildNodes()) {
            explore(child);
        }
    }
}
```

The above code simply iterates over the nodes of the AST recursively while handling each of them using **nodeHandler.handle.** This interface is specified in another class to handle only the binary operations and output its positions. The main reason of using java parser is that those operations could be used in a way that are not necessarily binary operators. This would ensure that the replacement in the program makes sense when generating mutants. The handler would fetch all the binary operators and output their positions in order to generate **potentialFaults.txt** which is required for phase 1 to run.

Mutant Generation **Phase 2**

In this phase we would choose a random operator from the list we found in phase 1 and then inject a different random binary operator from the list +, -, *, /. This would mean that the binary operator generated would be also different than the original one to avoid generating a mutant the matches the original code. We will repeat the mentioned before for all the number of mutants requested to generate mutated files which are named as PiDigitsMutant33.java as example. Where 33 is the mutant number or id. An example of mutant file which shows in the last line which operator was modified.

```
/**
 * This is a sample program to find the digits of pi The program is referenced
 * from: http://www.codecodex.com/wiki/Calculate_digits_of_pi#Java on Date
 * 10/14/2019
 **/
public class PiDigitsMutant3 {
    private static final int SCALE = 10000;
    private static final int ARRINIT = 2000;

    public static String methodUnderTest(int digits) {
        StringBuffer pi = new StringBuffer();
        int[] arr = new int[digits + 1];
        int carry = 0;

        for (int i = 0; i <= digits; ++i)
```

```
            arr[i] = ARRINIT;

        for (int i = digits; i > 0; i -= 14) {
            int sum = 0;
            for (int j = i; j > 0; --j) {
                sum = sum * j + SCALE * arr[j];
                arr[j] = sum % (j * 2 - 1);
                sum /= j + 2 - 1; // the mutant is here it was j * 2
            }

            pi.append(String.format("%04d", carry + sum / SCALE));
            carry = sum % SCALE;
        }
        return pi.toString();
    }
}

/**
* Mutant: PiDigits: orignal:(*) -> mutant(+) at (line 23,col 26)
************************************************************
**/
```

The last couple of lines of this mutant shows the which operator was modified and how. Before it was **sum/= j * 2 -1;** the mutant modified it to **sum/= j + 2 -1;** which would change the way of how the pi digits are calculated. This phase terminates by generating all the mutant files requested.

Simulator in Parallel **Phase 3**

In this phase, the simulator program would figure out how many mutants to assign for each thread. The mutant's assignment per thread is done in a way to ensure that each thread gets the same number of mutants to work on. However, if the number of threads is larger than the mutant's number then we would simply ignore the extra threads. Read more in Simulator.java for more info.

Once each thread is assigned several mutants to run upon, the thread would simply load the java compiler and compile the files and then load the java loader to load the class file into memory and run the mutated method under test. After the test completed running the thread would return a list of mutant types depending on how the mutant ran:

Mutants categories:

- **Still born mutant:** still born mutant is a mutant that failed to compile or run the method under test
- **Killed mutant:** this mutant was killed by a specific test vector or a mismatch in the exception thrown from the original code

- **Equivalent mutant:** this mutant is indistinguishable from the original code under the test vector specified.

The results are stored in **simulatorOutput.txt** which shows under which test vector the mutant is killed. The test vector sample for this program is simply the powers of 2 up to 1024. This test vector could change to kill more mutants in case any equivalent mutants shows up during testing. The test vector is defined in the Callable Thread.

For threading, we have used callable threads that allows the threads to return output once the main thread joins them all in order to see what types of mutants we have. After all this done, we would record a time stamp in milli seconds to ensure that we can measure how increasing the number of threads would affect performance. This phase terminates by outputting the **simulatorOutput.txt** into the output directory.

Appendix

Please find the project attached with all the code and the documentation necessary.

In order to run the program please type the following in your terminal:

```
### How to Build and run
mvn clean compile assembly:single
cd target
java -jar mutantfactory <intputFile> <outputDir> <numberOfMutatns> <#threads>
```

The above requires the latest version of java installed and maven installed. Without maven you would not be able to build the program.

Once the jar file is built, we would have to specify in this order:

- **inputFile**: the original code to be mutated that has the methodUnderTest
- **outputDir:** the output directory that holds all the output files and the generated mutants
- **numberOfMutants:** the number of mutants to generate
- **number of threads:** the number of threads required to run the simulator. This will speed up the simulation phase.

Check the readme file for further details.

References:

- Java program PiDigits as original code
  http://www.codecodex.com/wiki/Calculate_digits_of_pi#Java
- Java Parser library
  https://javaparser.org/