MUTATION TESTING TOOL FOR JAVA

MUNAWAR HAFIZ

Abstract. Mutation testing takes a different approach to testing by asking questions about the efficacy of test cases. The test cases are tested by introducing bugs in the code. The test cases are then run on the original program and all the mutants. The effectiveness of a test case is determined by the percentage of mutants it kills. This report describes a mutation tool for Java programs.

1. Introduction

Mutation Testing is a mechanism to determine test set thoroughness by measuring the extent to which a test set can discriminate the program from slight variations of the program. The concept of mutation testing was introduced in as early as in 1971 in Richard Lipton's class term paper titled "Fault diagnosis of computer programs". It got popular interest in late 70s and early 80s but failed to make the final cut in industry. The reason is that generating and running the mutants has performance issues associated with it. This is despite the empirical findings of Walsh [9] that mutation testing is more powerful than branch and statement coverage and the findings of Offutt et.al.[7] that mutation testing is more more effective in finding faults with data-flow.

PIMS is an early mutation tool for Fortran IV. The basic architectural principles were bettered in Mothra in 1987. Mothra did not have an end-to-end system for doing mutation testing. Adding test cases, running test cases and comparing results to generate the report were separate tasks. Insure++ 4.0 from Parasoft was a commercial tool available for mutation testing in the late 90s. But reviews have shown that Insure++ 4.0 takes a questionable approach which does not follow the principles of mutation testing. Again SourceForge has an open source mutation tool for Java called Jester. However, the efficacy of mutation testing depends largely on the mutation operators and the mutation operators that Jester uses have proven to be rather unstable.

Yu Seung Ma and Jeff Offutt developed JMutation, a mutation system for Java programs. Later the name has been changed to MuJava. MuJava does not support test generation. It generates mutants and runs testcases on them. Our tool closely follows the MuJava architecture. The approach of Mujava was to generate the mutants using mutation operators in one phase and run the test cases on mutants in a separate phase. The ultimate goal of the tool that we are developing is to create a plugin for Eclipse framework. The current implementation has the mutation run phase. For mutant generation, we are using the MuJava tool at this moment. The mutant running and reporting program that we have developed is tolerant to various issues during execution like exceptions, infinite loop etc.

This report is oriented as follows. Section 2 gives a general architecture of a complete system. Section 3 provides a short description of the mutation operators. Section 4 and beyond describes architecture and usage of current implementation and issues involving that. Finally we conclude with future plans of extension.

2. General Architecture of a mutation system

This section builds on the idea introduced in [8]. Mutation analysis provides a testing criterion rather than a test process. A criterion is normally evaluated in terms of some coverage metric. A mutation system basically consists of two parts. The first part handles the mutation, In this stage, the original program is handed over to a mutation engine which has a collection of mutation operators. A mutation operator is a rule specifying syntactic changes. The collection of mutation operators is a crucial factor in the effectiveness of mutation testing. The most common mutation operators replace each operand with other syntactically suitable operands. The mutation engine generates a number of variants of the original program called the mutant programs.

Mutants are representative of the faults that the programmers are likely to make. The key principle of mutation based techniques is the coupling effect, which says that complex faults are coupled to simple faults in such a way that a test data set that detects all the simple faults is representative enough to detect most complex faults. This was first hypothesized in 1978[1], and later supported empirically in 1992[5].

After the mutant programs are generated, the test cases are run as input to the program. The test case is run on the original program and all the results are noted. Then the test case is run on all the mutants and the results are also tallied. These results are then compared to the original results. A non-equivalence suggests that the particular mutant is killed by the test case. Hence the test case is good enough to detect the change. Theoretically, test cases should be sufficient to kill all the programs. If some mutants are not killed, then test cases are written for that part of the original program and the results are tallied again. The steps of mutation testing are illustrated in figure 1. Mutation score is defined as the ratio of killed mutants compared to the total number of mutants. A mutation score of 100% denotes that the test cases are adequate for testing the system.

Our implementation targets a system that has both the stages combined under an eclipse plugin. However, current implementation only has the mutation execution and result comparison utility. For mutation generation, we are using the Mujava tool. Our mutation execution engine takes as input the output mutants generated by Mujava. This means that the future mutation generation part has a contract of input/output similar to Mujava.

A major technological advance in the field of mutant generation engine is the use of a schemata generator. A schemata generator produces a meta-mutant. A meta-mutant incorporates all the mutants on an original program in a single program. Each mutant is a case of setting some condition true and getting the result from the meta-mutant. Mujava is based on this technology.

One of the major problems that the traditional mutation testing tools have is the lack of automation for major parts of the process. The major tasks of entering test cases, running the test case on original input and checking the result, and running the tests on mutants and checking the results are very human-intensive. Our

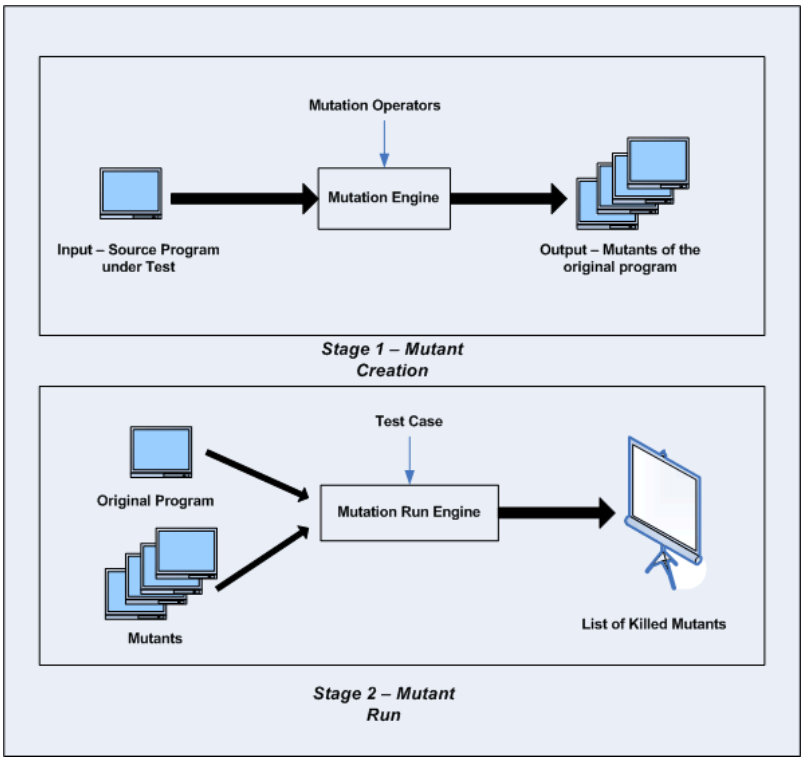

FIGURE 1. Stages of Mutation Testing

implementation is an end-to-end system and it is completely automatic. In both the traditional method and our implementation, the major overhead of mutation is in the loop of generating, running and disposing of test cases. However, removing the human intervention part from it provides major advances in terms of applicability. In future, we have plans of incorporating various hacks for getting better performance.

3. Mutation Operators

Mujava uses two types of mutation operators. The traditional mutation operators are developed from procedural languages. Object oriented languages have additional class level mutation operators. They work on the features of object oriented languages like inheritance, polymorphism and dynamic binding.

Mothra uses 22 traditional mutation operators on Fortran. However, running all these mutant operators generate a huge number of mutants and not all of them are effective because of overlaps. The idea of selective mutation was introduced by Wong and Mathur[10] and later experimentally validated by Offutt et.al.[6]. Selective mutation states that a subset of all the mutation operators is sufficient to provide same effectiveness as non-selective mutation. Table 1 has the five traditional mutation operators and example of how they are applied on the original program.

Table 1: Traditional Mutation Operators

Operator	Description	Example
ABS	Absolute Value Insertion	$a = b + c$ *to* $a = 0$
AOR	Arithmetic Operator Replacement	$a = b + c$ *to* $a = b - c$
LCR	Logical Connector Replacement	$a = b\&c$ *to* $a = b\|c$
ROR	Relational Operator Replacement	$while(a < b)$ *to* $while(a > b)$
UOI	Unary Operator Insertion	$a = b$ *to* $a = -b$

24 class mutation operators were identified for Java classes by Ma, Kwon and Offutt for testing object-oriented and integration issues. There is yet any research on applying selective mutation on these operators. A list of some candidate mutation operators is presented in table 2.

A major issue with class mutation operators is that they are applicable in different levels - intra-method, inter-method, intra-class and inter-class. Traditional mutation operators are all intra-method operators. In general the class mutation operators are intra-class, but inter-class operators are important for traditional integration testing and seldom used subsystem testing.

Table 2: Some Class Mutation Operators

Category	Operator	Description	Example
Inheritance	AMC	Access Modifier Change	public Stack s; `to` private Stack s;
Polymorphism	PNC	*new* method call with child class type	a = new A(); `to` a = new B(); *where B is subclass of A*
Overloading	OAN	Argument number change	s.Push(0.5,2); `to` s.Push(2);
Java-specific	JTD	*this* keyword deletion	this.size = size; `to` size = size;
Common Programming Mistakes	EOA	Reference assignment and content assignment replacement	list2 = list1; `to` list2 = list1.clone();

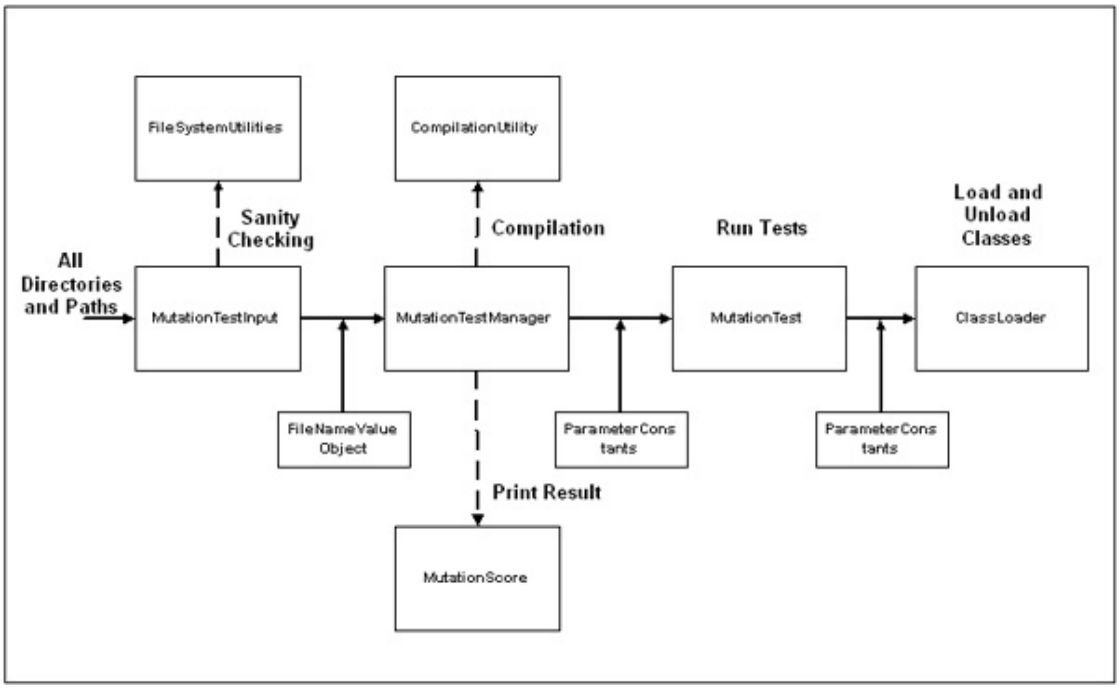

FIGURE 2. Main components and process flow

4. Architecture of the Mutation Execution Engine

Figure 2 shows the architecture of our system. The input to the system are the files and directories for original project, mutations and test projects. The *MutationTestInput* class collects and checks the filesystem for validating the existence of the directories and projects. Then it creates *FileNameValueObject* object for the inputs. A *FileNameValueObject* is the collection of filename and full path of file in the file system. This acts a Value Object for representing a data structure. The *MutationTestManager* gets the file structures from *MutationTestInput* class and uses *CompilationUtility* to compile if the input is Java source. If the input is class file, then *MutationTestManager* creates *ParameterConstants* object that has all the necessary information for loading and running the tests. These information include name of source and test classes and full path of classes in the file system. Also, information like whether the classes are original classes, or they are traditional or class mutants is passed for reporting purposes. This information is passed to *MutationTest*, which loads the classes and reflectively invokes the test method. The class loading is done by the *CustomClassLoader*. All the methods in the test class that has a prefix 'test' are run. First the test classes are run on the original classes and the results are stored in a *ResultStructure* object. Then the class mutants and traditional mutants are used and the test methods are run on them. The results are passed to *MutationScore*, that already has the original results. The results are compared and a mismatch is tallied in a list a killed mutants. The sequence of message flow between classes is shown in figure 3. Figure 2 also has a class named

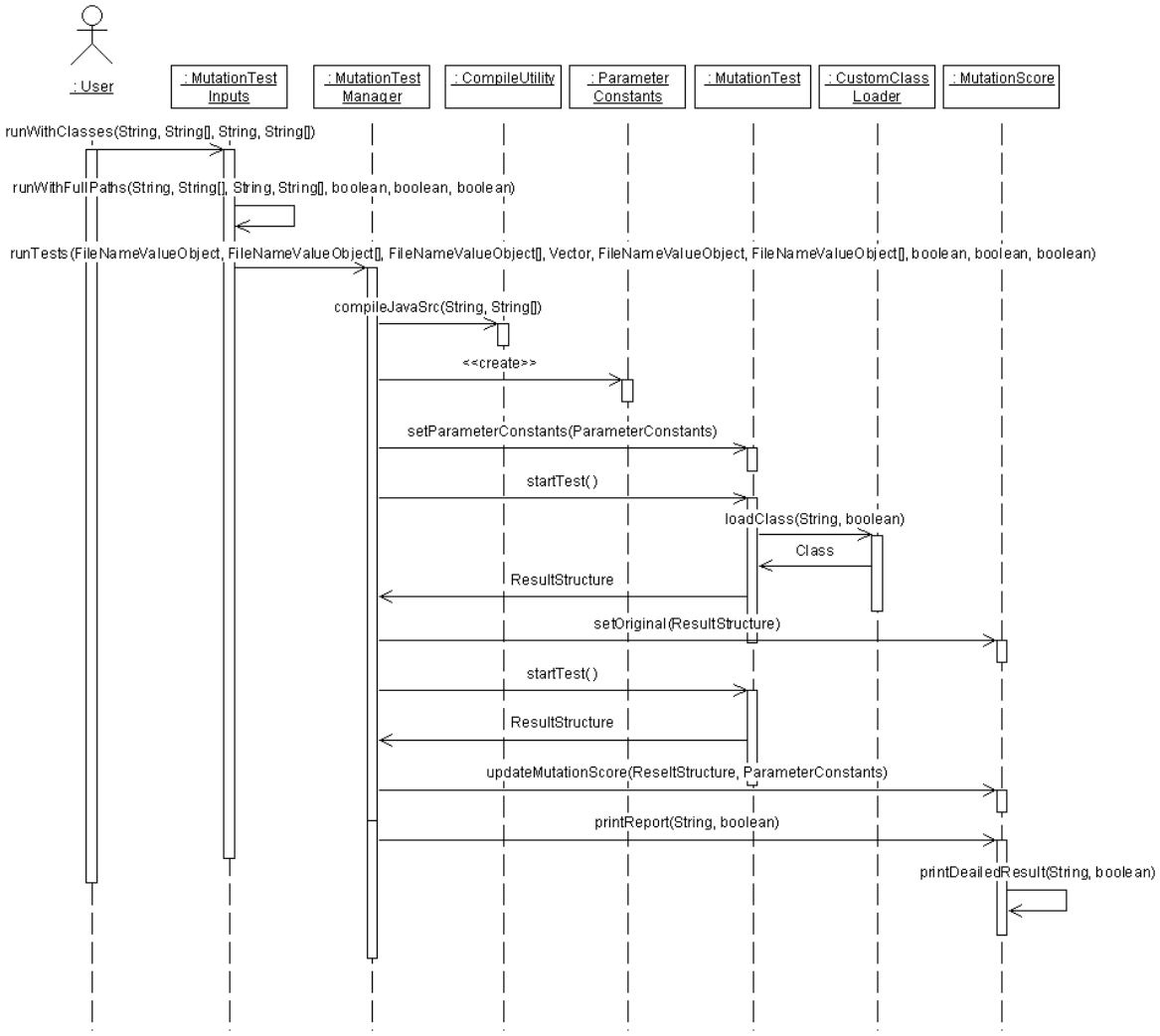

FIGURE 3. UML Sequence Diagram of Class Interactions

FileSystemUtilities. This is a utility for searching file systems for files and other file system related tasks.

5. MANUAL FOR USING

Our current implementation runs with both source and class files. An example illustrating the whole process would make things clear.

Let, there be a project *TestProject* that has two files *B.java* and *C.java* and both the files need to be tested. To get the mutants, the source project is copied to $MUJAVA_HOME\src directory. Then the command for mutation generation is run.

java mujava.gui.GenMutantsMain

This command opens up a GUI for mutation generation. The source file and the mutation operators are selected with the default being the selection of all the mutation operators. The mutants are generated under the $MUJAVA_HOME\result directory. For example, for the file C.java, the mutation directory will be $MUJAVA_HOME\result\TestProject.C. Under this directory there will be three subdirectories – "original", "traditional_mutants" and "class_mutants", each carrying the source files as the name specifies. Let us suppose the ABS operator was used twice to generate two traditional mutants. Now these files will reside under the ABS_1 and ABS_2 directory. The full path of the files would be,

$MUJAVA_HOME\result\TestProject.C\traditional_mutants\ABS_1\C.java
and
$MUJAVA_HOME\result\TestProject.C\traditional_mutants\ABS_2\C.java

The class mutants will reside under the class_mutants directory with similar file system structure.

Now this file system structure is necessary for the operation of mutant execution engine. The mutant files need not reside under $MUJAVA_HOME\result directory. However, the mutant directory should have three sub-directories under it, namely, "original", "class_mutants" and "traditional_mutants". The "class_mutants" and "traditional_mutants" directory will have further subdirectories for mutation operators and the mutants generated by applying the operators. This is the only assumption over the inputs that are passed to the system.

To run the mutants, the following method is called from the class *MutationTestInput*,

```
public void runWithSrcCompilation(String projectPath,
                                  String muDir,
                                  String testProjectPath,
                                  String testPath) throws Exception
```

Here, *projectPath* is the full path of the original source project, *muDir* is the full path of the mutation directory, *testProjectPath* is the full path of the test project and *testPath* is the full path of the test source file. Let the test file is the source file $MUJAVA_HOME\src\TestProject\TestCaseC.java and it is under the test project directory $MUJAVA_HOME\src\TestProject. Taking the running example the values would be,

```
public void runWithSrcCompilation(
                "C:\\mujava\\src\\TestProject",
                "C:\\mujava\\result\\TestProject.C",
                "C:\\mujava\\src\\TestProject",
                "C:\\mujava\\src\\TestProject\\TestCaseC.java")
        throws Exception
```

The interface can accept class files instead of source java files. In that case, it does not have to compile the java files to create the class files beforehand. The signature of that method call is,

```
public void runWithClasses(String projectPath,
                                  String muDir,
                                  String testProjectPath,
                                  String testPath) throws Exception
```

The parameters are the same except all of them now point to existing class files—the *testPath* points to a class file and all the directories have an underlying contract that class files reside in them. Going with the same example, say the class file *C.class* resides in the project *TestProject* in path , and the mutation directory *TestProject.C* contains all the mutants and the original files as class files. The test class *TestCase.class* is resided in the *TestProject* directory. In that case, the call would be,

```
public void runWithClasses(
                "C:\\mujava\\src\\TestProject",
                "C:\\mujava\\result\\TestProject.C",
                "C:\\mujava\\src\\TestProject",
                "C:\\mujava\\src\\TestProject\\TestCaseC.class")
        throws Exception
```

Our implementation also handles, multiple source classes and multiple test cases. The signature of that function is,

```
public void runWithClasses(String projectPath,
                           String[] muDir,
                           String testProjectPath,
                           String[] testPath) throws Exception
```

In this case, *muDir* parameter is not a single directory, but a collection of directories as String array, which are mixed together to create mutation profile. This is needed, when the mutation class calls some other class, and that also calls some other class, and the goal is to test the effect of mutation on all of them. This provides opportunity to do inter-class integration testing to some extent. *testPath* is a String array that contains the testcase classes to be used for testing.

Now, say we want to test both *C.class* and *B.class* in a project *TestProject*. The mutation directories for them are *TestProject.C* and *TestProject.B* correspondingly. The test project is the same as the original project and the test classes are *TestCaseC.class* and *TestCaseB.class*. *TestCaseC.class* tests the class *C.class* and *TestCaseB.class* tests the class *B.class*. Now the call would be,

```
public void runWithClasses(
                "C:\\mujava\\src\\TestProject",
                new String[] {
                    "C:\\mujava\\result\\TestProject.C",
                    "C:\\mujava\\result\\TestProject.B"},
                "C:\\mujava\\src\\TestProject",
                new String[]{
                    "C:\\mujava\\src\\TestProject\\TestCaseC.class",
                    "C:\\mujava\\src\\TestProject\\TestCaseB.class"})
        throws Exception
```

The fact is already mentioned that the path of Mujava is immaterial to run the mutation tests. The only underlying assumption is the existence of "original", "traditional_muntants" and "class_mutants" directories as subdirectories of Mutation directory. The last example would be showing this.

Let, the name of the project is "TestProject2". The *projectPath* attribute will be *"C:\\TestProject2"*. The class to be tested is *C.class* and the mutation directory is under "MuDirForC" directory. The *muDir* attribute will be *"C:\\MuDirForC"*.

Now the *MuDirForC* will have an original *C.class* under "original" directory. We assume it has 2 class mutants and 2 traditional mutants. The paths of the class mutants would be,

C:\\MuDirForC\class_mutants\MuOp_1\C.class
and
C:\\MuDirForC\class_mutants\MuOp_2\C.class

And the paths of traditional mutants would be,

C:\\MuDirForC\traditional_mutants\MuOp_1\C.class
and
C:\\MuDirForC\traditional_mutants\MuOp_2\C.class

Let the Test case be the same project as the source project. Hence the *testProjectPath* is *"C:\\TestProject2"*. And the *testPath* attribute is *"C:\\TestProject2 \\test\\testCaseForC.class"*. The class *testCaseForC.class* is under the *test* package of the test project.

The call for running the test would be,

```
public void runWithClasses(
                "C:\\TestProject2",
                "C:\\MuDirForC",
                "C:\\TestProject2",
                "C:\\TestProject2\\test\\testCaseForC.class")
        throws Exception
```

The current implementation is packaged in jar format and it is included in the project by user and then the run methods are called. Our future implementation would have this as eclipse plugin. In that case, it would have to be copied in the plugin directory of eclipse and then the eclipse IDE is restarted. This would be available as a view plugin.

The detailed API documentation is available in [3].

6. Issues of Implementation

In this section, some of the implementation idiosyncrasies are detailed.

Compilation of Source Files. The compilation utility is defined in the *CompileUtility* class. The *compileJavaSrc* method takes as input the complete path of the java source file to be compiled and a *String* array of classpath entries that are to be added to the current classpath for that compilation. The method has the following signature,

```
public void compileJavaSrc(String pathName,
                                   String[] classpathEntries)
```

The current runtime is a Singleton and it can be obtained from the *getRuntime* method. The *exec(String command)* method was used to run the command of compilation. The compilation command is created by adding the *classpathEntries* to the existing classpath and then passing that as the "-classpath" parameter of the "javac" command.

When a lot of source files are to be compiled, there are memory problems when processes are running in parallel, because a lot of processes spawn up and they

contend for memory. For this reason, the compilation process is run sequentially. The *waitFor()* method defined in the *Process* class is used for this purpose. This causes the current thread to wait, if necessary, until the process represented by the *Process* object has terminated. This method returns immediately if the subprocess has already terminated. If the subprocess has not yet terminated, the calling thread will be blocked until the subprocess exits.

Handling Multiple Files. When running mutation testing on multiple files, the files are mixed in a way such that one file is mutated and the rest are original files. This is good for integration testing and yet manages to limit the blow-up of mutants in case a more aggressive mutant strategy is used. For example, say there are three classes for mutation testing. The first one has x mutants, the second one has y mutants and the third one has z mutants. Limiting the mix to be one mutant and other original files, the total number of mutant combination is still $x + y + z$. If there are 2 mutants and 1 original, then the number of mutation combinations grow larger, because it contains a complex permutation of mutants.

This is done in the *mixAndMatchMutationDirectory* method in *MutationTestInputs* class. The method is defined as,

```
private Vector mixAndMatchMutationDirectory
                    (String[] srcPath,
                     String[] muDir)
                throws IOException
```

Here *srcPath* contains the original files and *muDir* contains the mutation directories. It returns the mutation combinations as a *Vector*.

The approach that is taken to combine the mutants follows the principle of coupling effect. Creating a mutation combination with multiple mutants makes it difficult to analyze faults in a smaller granularity level. It does not offer any additional advantages.

Creating CustomClassLoader. A custom class loader is required for running the test cases on original class and mutants. The class loader loads classes from some specific path. The Java class *ClassLoader* is an abstract class. When a specific class is required, the name of the class is passed to the *ClassLoader* and it attempts to locate and generate the definition of the class. The typical strategy is to transform the name into a filename and then read a class file of that name from the file system.

Class loaders play an important part in the security architecture of JVM. When some object is required, the JVM looks for the *Class* in the environment. If it is not present, then a *ClassLoader* is assigned to load the class. We designed a custom class loader *CustomClassLoader* that extends the *ClassLoader* of Java. It overrides the *loadClass* and *findClass* method of Java *ClassLoader*.

The *ClassLoader* uses the delegation model for to search for classes and resources. Each instance of *ClassLoader* has an associated parent. When requested to find a class or resource, a *ClassLoader* instance will delegate the search for the class or resource to its parent class loader before attempting to find the class or resource itself. The virtual machine's built-in class loader, called the "bootstrap class loader", does not itself have a parent but may serve as the parent of a *ClassLoader* instance. This means that a loaded class is not only visible within the class loader in a chain,

but also to all its descendants. It also means that a class can be loaded by multiple class loaders in a chain, but the one furthest up the tree is the one that actually loads it. All the classes loaded through the bootstrap class loader are considered to be trusted and all other classes loaded by custom class loaders are un-trusted. This distinction is important for security.

The overridden *loadClass* method first tries to delegate the task of class loading to its parent. When the parent is unable to load the class (which is normal because the requested class name is not found in the classpath), then it throws a *ClassNotFoundException*. At this point, the *findClass* method is called. The overridden *findClass* uses the final method *defineclass* to return the class. The signature of this method is,

```
protected final Class defineClass
              (String name, byte[] b, int off, int len)
          throws ClassFormatError
```

This converts an array of bytes into an instance of class *Class*. Before the class can be used it must be resolved. The *findClass* method assigns a default *ProtectionDomain* to the newly defined class. The *ProtectionDomain* is effectively granted the same set of permissions returned when *Policy.getPolicy().getPermissions(new CodeSource(null, null))* is invoked. The default domain is created on the first invocation of *defineClass*, and re-used on subsequent invocations. It throws a *ClassFormatError* if the data did not contain a valid class.

The *loadClass* method passes the full pathname of the class to the *findClass* method. The byte array parameter for the *defineClass* method is the content of the file in that specific path. Finding the name of the class is also an issue. To find the name of the class from byte array contents, the class file data structure has to be parsed and corresponding constant pool entries have to be created. A third party library is used for this purpose. The library is available as open source at [4].

Another issue of loading classes is that the original class and all the mutants have the same class name. This causes a conflict when loading because the same class cannot be loaded by the class loader. Classes by default have visibility of other classes loaded by the same class loader. One option to sidestep this is to, create another class loader into the same JVM and load the different version of the class. But this has significant memory overhead. The approach adopted in our implementation is to unload the class and re-create a class loader. The used class loader is assigned to null and then forcefully garbage collected by invocation of *System.gc()* calls. This is done twice to be absolutely make sure that it is garbage collected.

Correctness of Execution. The system has to be robust and tolerant of aberrant runtime scenarios, like exception throw, infinite loop etc.

The test cases are run by reflectively calling the *invoke(...)* method in the *Method* class of *java.lang.reflect* namespace. The signature of this method is,

```
public Object invoke(Object obj,
                     Object[] args)
              throws IllegalAccessException,
                     IllegalArgumentException,
                     InvocationTargetException
```

Here the *obj* is the underlying object from which the method is invoked from and *args* contains the argumens used for method call. The exceptions thrown by the method takes care of some aberrant running conditions. *IllegalAccessException* is thrown when the method is not visible to the invoker (a private method). *IllegalArgumentException* is there is some problem with the number or type of parameters passed to the method. *InvocationTargetException* is thrown when the underlying method throws some exceptions. This handles all the exception cases. When exception is thrown by a test method, it is trapped at the catch section of this exception and proper message is added for reporting purposes. Also this handles cases like running out of stack or running out of heap.

Table 3: Performance Profile of and results of test runs

Goal of the integration test	**Number of Mutants**	**Result**	**Percentage of time in Processes**
Run single mutant with compilation	4 class mutants and 2 traditional mutants	2 class mutants killed and 2 traditional mutants killed	*MutationTestManager* - 5.03%, *MutationTest* - 5.03%, *CompileUtility* - 0.72%
Run multiple mutants with class files	Two files B.class and C.class. For B, 4 class mutants and 96 traditional mutants. For C, 4 class mutants and 2 traditional mutants	For B, 3 class mutants killed and 47 traditional mutants killed.For C, 2 class mutants killed and 2 traditional mutants killed.	*MutationTest* - 8.97%
Run tests on classes with same name but in different namespace	Two classes C.class, in different packages. But no mutants for them.	No mutants killed.	*MutationTestManager* - 13.64%
Run with mutants that time out	1 traditional mutant	1 traditional mutant killed	*MutationTestManager* - 9.44%, *MutationTest* - 2.36%
Run with mutants that go out of stack or out of heap	2 traditional mutants	2 mutants killed	*MutationTest* - 8.88%, *MutationTestManager* - 2.44%

For infinite loops, the scenario is handled differently. When the test methods are run on the original class, the time of execution for each methods are noted. Then, when they are run on the mutant classes, they are run as separate threads and the *join(long millis)* method defined in *Thread* class is used to run the thread for a specific amount of time. The amount of time is set to ten times the execution time for the original class. The thread uses the invoke method and stores the result in some internal data structure. At the end of thread execution (by normal ending or by doing join after timeout), the internal data structure is checked to see, whether the result is present or not. If the result is not present, the test method is reported

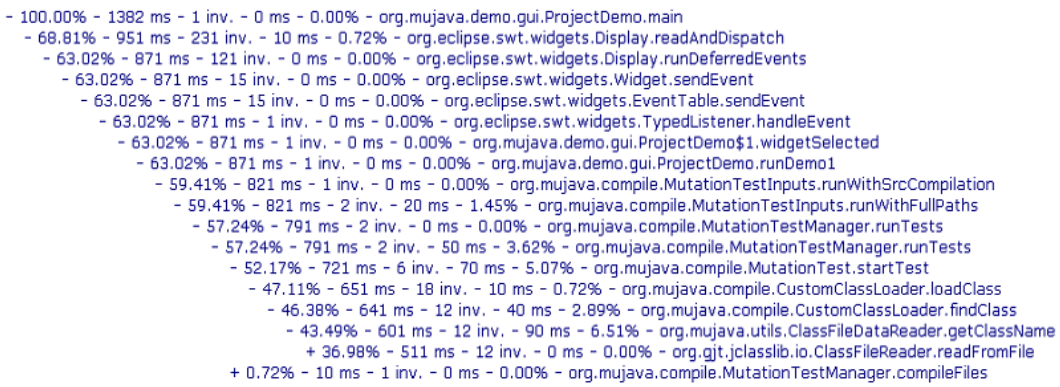

FIGURE 4. Performance result for test run of multiple classes (Row 2 of Table 3)

to have timed out.

Comparison of Result and Equality check. The original results and returned results are primarily String and native data types and we compare the results by using the *equals()* method. For complex return types, the *equals()* method has to be implemented for those types.

7. PERFORMANCE AND FUTURE WORK

Performance is the key issue for the execution engine. We used a profiler plugin [2] for eclipse to get profiles for execution. The profiler was run over the several test cases that were written to perform integration tests. Some summaries are given in Table 3.

Column 1 states the goal of the test case. Column 2 has the number of mutants that were existent. Column 3 shows the results and Column 4 shows the percentage of time taken by important processes.

The last column shows time spent in two major classes for management and execution of threads. The current goal of implementation is to improve the techniques to reduce the percentage. We have a parallel implementation using the jdt framework of eclipse, and we are finishing it to test the comparative performance. Another future goal is to implement the principle of weak mutation and test the performance results.

The results of performance profile also shows that the third party library that we are using, has significant overhead. Figure 4 shows a test profile for running multiple classes (Row 2 of Table 3). This shows that the library *org.gjt.jclasslib* has significant performance overhead associated with it. This is because this constructs the complete constant pool form byte code of a class file although we need only the name of the class. We will add a custom implementation of a smaller scope in future.

A future software engineering goal is to polish the tool into an eclipse plugin, that also has the mutation generation engine integrated and runs as an end-to-end application.

8. Conclusion

Our target is to create a feasible mutation testing tool with minimal human involvement and significant performance improvement. Our complete system would provide almost complete automation to the tester. Good coverage is an important criterion and efficient mutation testing provides significantly better coverage than other techniques.

References

[1] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, vol. 11, pp. 34-41, April 1978.

[2] Eclipse Profiler plugin. *http://sourceforge.net/projects/eclipsecolorer.*

[3] M. Hafiz. User Manual of Muatation Engine. *https://netfiles.uiuc.edu/mhafiz/www/ResearchandPublications/MutationTestingTool.htm*

[4] jclasslib byte-code viewer.*EJ-Technologies, http://www.ej-technologies.com/products/jclasslib/overview.html*

[5] A. J. Offutt. Investigation of the software testing coupling effect. *ACM Transactions on Software Engineering Methodology*, vol. 1, pp. 3-18, January 1992.

[6] A. J. Offutt, Ammei Lee, G. Rothermel, R. Untch, and C. Zapf. An experimental determination of sufficient mutation operators. *ACM Transaction on Software Engineering Methodology*, 5(2):99-118, April 1996.

[7] A. J. Offutt, J. Pan, K. Tewary, and T. Zhang. An Experimental evaluation of of data flow and mutation testing. *Software-Practice and Experience*, 26(2):165-176, February 1996.

[8] A. J. Offutt and Roland H. Untch. Mutation 2000:Uniting the Orthogonal. *Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, 45–55, San Jose, CA, October 2000.

[9] P. J. Walsh. A measure of test completeness. Ph.D. Thesis, Univ. New York, 1985.

[10] W. E. Wong and A. P. Mathur. Reducing the cost of mutation testing: An empirical study. *The Journal of Systems and Software*, 31(3):185-196, December 1995.

University of Illinois at Urbana-Champaign
E-mail address: mhafiz@uiuc.edu