Analysis of Mutation Testing Tools in Aspect Oriented Software Engineering

Vineeta
Department of CSE
ASET, Amity University
Sector 125, Noida, UP, India

Preeti Lochab
Department of CSE
ASET, Amity University
Sector 125, Noida, UP, India

Abhishek Singhal
Department of CSE
ASET, Amity University
Sector 125, Noida, UP, India

ABSTRACT
Mutation testing measures the adequacy of the test suite by seeding artificial defects i.e. mutants in the program. If the mutant is not detected by the test suite, is means that the test suite is not adequate. And new test suites are added until all the mutants have been detected. AspectJ is an aspect-oriented programming language that provides the concept of pointcut and advice.

In this paper we proposed different mutation testing tools, their need and manner to implement and also at last we had developed a tabular comparison of different mutation testing tools like Ajmutator, Advice Tracer, MuAspectJ and Proteum/AJ. The uses of such tools in the a testing process enhances the feasibility of using it in real software development process and helped us to reason about the current functionalities and to identify future needs. The result includes the comparison of different testing tools and a number of parameters to judge their performance.

General Terms
Mutation Testing

Keywords
Aspect-oriented-programming, AspectJ, mutation testing, testing process, testing tools, PCD, joinpoints, advice, weaving

1. INTRODUCTION
The aim of this paper is provide brief analysis of the Aspect-oriented-programming [13] concept and a testing technique known as 'mutation testing' [14]. It also aims to analyze different mutation testing tools and a way of comparison among them. Aspect-oriented-programming [15] is a technique that separates the core concerns from the cross cutting concerns. The use of AOP changed the development process too. Here, the classes and methods are same as earlier. However the main difference lies in the fact that instead of embedding the code for cross-cutting actions into method bodies, the codes are defined into the separate aspects. Later on, these aspects are woven into the classes that represent the core concerns of the system. The weaving process deals with both the core concerns and cross cutting concerns. The cross-cutting concerns are encapsulated into two parts. In first, we have an advice that implements the cross-cutting concerns. The second part includes a pointcut descriptor (PCD) that designates a set of joinpoints in the base program where the advice should be woven. AOPs are similar to OOPs [28] as they both have classes, interfaces, methods, packages, etc. The computational model of an AOP is based on a dynamic call graph as a result of method invocations. It has joinpoints where the advice has been applied based on a pointcut match. The aspect developer must specify an appropriate set of pointcuts and advice that after weaving will results into the desired behavior. JoinPoints are the well defined moments in the execution of a program like method calls, object instantiation, variable access etc. Pointcut allows a programmer to specify joinpoints. All the pointcuts are the expressions that determines whether given pointcut matches or not. Advice specifies the code to run at a joinpoint matched by the pointcut. These actions can be performed before, after or around the specified joinpoint.

In this paper we explained the mutation testing process and analyzed different mutation testing tools and analyzed their performance based on different parameters.

2. MUTATION TESTING
Mutation testing [16,30] is a method of software testing [17], which involves modifying programs' source code or byte code in small ways. It is also known as Mutation Analysis and Program Mutation. A test suite that does not detect and reject the mutated code is considered defective. These so called mutations [23] are based on well-defined mutation operators that mimic typical programming errors (such as using the wrong operator or variable name) [20]. A number of mutation operators are defined for the mutation testing process. Some of which are summarized as follows:

- Statement deletion.
- Replace each Boolean sub expression with true or false.
- Replace each arithmetic operation with another arithmetic operation, as +,-,*, /.
- Replace each Boolean relation with another Boolean relation, as >=, <=, ==, >, <, !=.
- Replace each variable with another variable declared in the same scope.

The above defined mutation operators are known as traditional mutation operators. However, we also have mutation operators for OOPs includes concurrent constructions, complex objects like containers. These are also known as class-level mutation operators [22]. The mutation operators for AOPs include pointcuts, joincuts, advice etc.

2.1 Mutation score
When we execute a mutant using a test suite, we may have any of the following outcomes:

The result of the program is affected by the change and any test case of the test suite detects it. If this happens then the mutant is called a killed mutant.

The result of the program is not affected by the change and any test case of the test suite does not detect the mutation, the mutant is called the live mutant.

The mutation score [29] associated with a test suite and its mutants is calculated as:

Mutation score = (Number of mutants killed) / (Total number of mutants);

Where, total number of mutants = Number of killed mutants + Number of live mutants.

The mutation score measures how sensitive the program is to change and how accurate the test suite is. A mutation score is always between 0 and 1. A higher value of mutation score indicates the effectiveness of the test suite. Although effectiveness also depends on the type so faults that the mutation operators are designed to represent. The live mutants are important for us and should be analyzed thoroughly [18].

In the Mutation Testing [19] process, the new test cases are added to the mutation system. The test cases are first executed against the original program.

The result is then compared with the difference between the original output and the expected output. If the output is correct, then we resume the procedure; if the output is incorrect then the mutation process is restarted. The mutants are killed if their output does not match of the original, and remains alive if they do.

The tester must add additional test cases to kill the remaining live mutants. When all the test cases have been executed against all the live mutants, each remaining mutant can be classified into two categories. The first category is equivalent mutants. If a mutant falls under this category then there is no need for the mutant to remain in the system for further consideration. The second category includes the mutants which are killable, but if the test cases are insufficient to kill these mutants. In this situation the new test cases are created to kill the remaining live mutants. The process of adding new test cases, examining expected output, and executing mutants continues until the tester is satisfied with the number of killed mutants.

3. MUTATION TESTING PROCESS

3.1 Traditional Mutation Testing Process

The following steps were followed in Traditional Mutation Testing [18],[19]:-

1) Test Program is given as input.
2) Create the mutants of test program.
3) Add test cases automatically or manually to the mutation system.

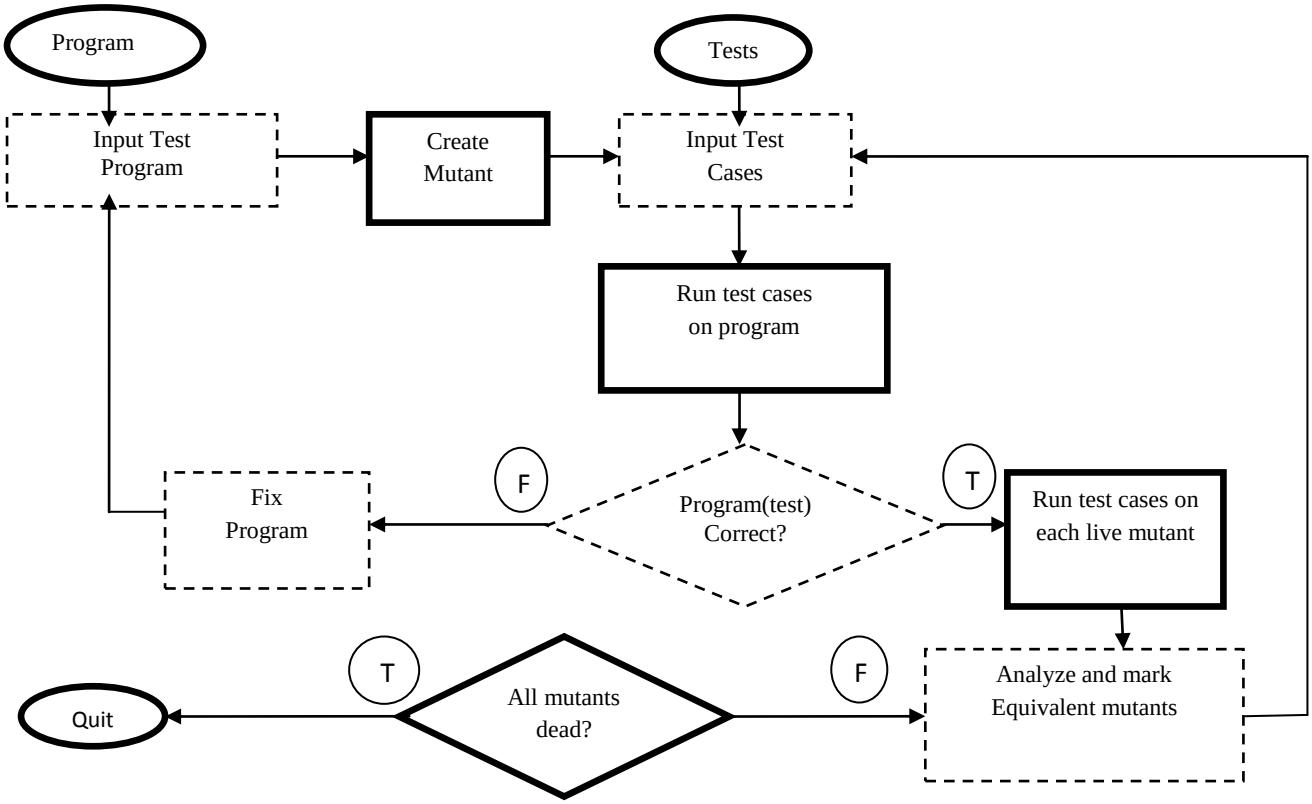

Fig.1 Traditional Mutation Testing

4) Firstly execute test case against the original version of the test program and then check the output of the test program for each test case.
5) If the output is found to incorrect, the program should be corrected and the mutation process should be restarted. If correct, that test case is executed against each live mutant present in the mutation system.
6) Compare the output of mutant program to the expected output. If the output differs from original program on the same test case, the mutant should be killed.
7) There should be only two types of mutants present in the mutation system after execution of all the test cases:

- Equivalent mutants: If mutants are equivalent there is no need to consider those mutants further.

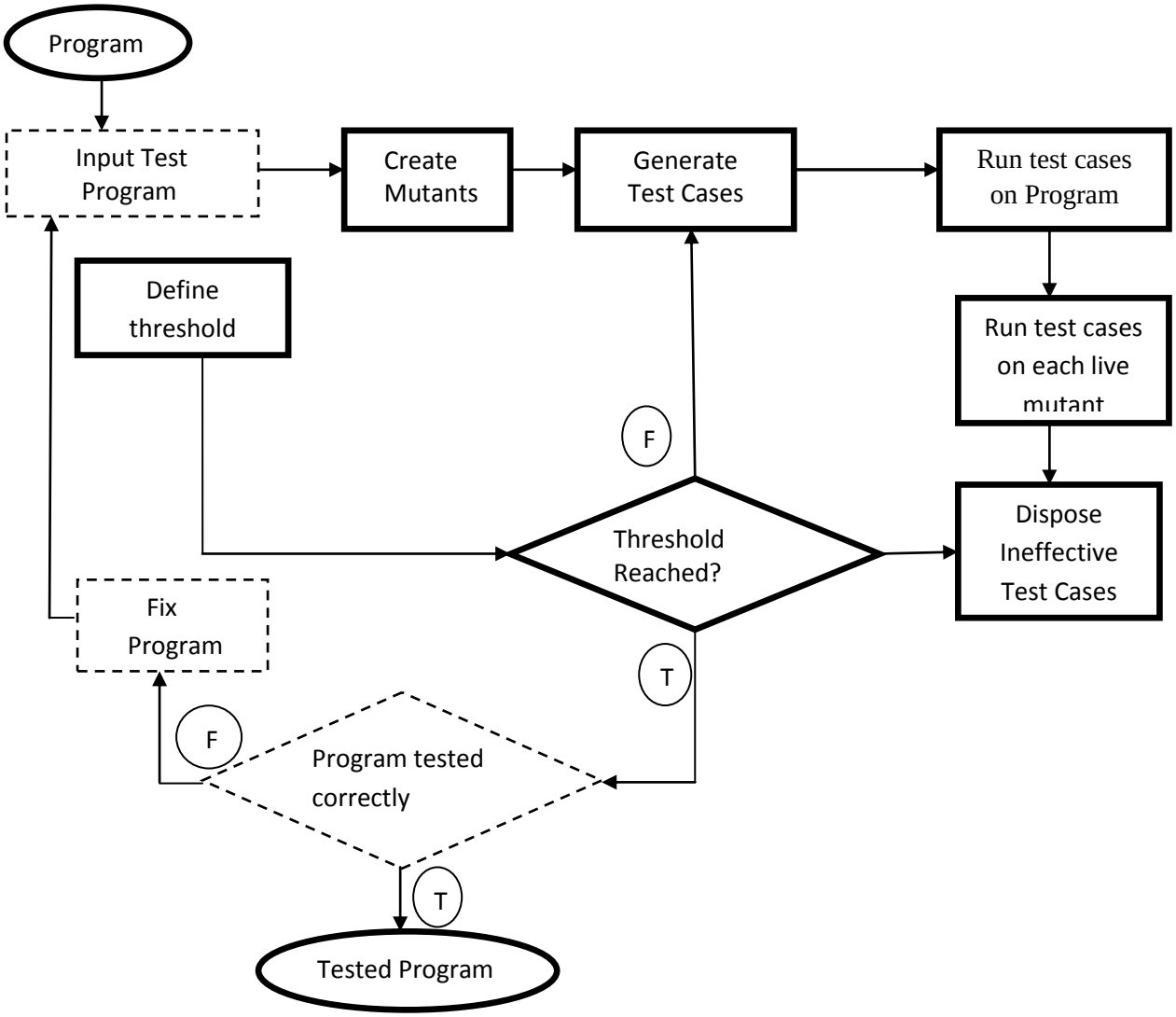

Fig.2 Modern Mutation Testing

- Killable mutants: If the mutants are killable but the test cases set is unable to kill them. For this a set of new test cases should be required

8) Test Cases should be added and mutants execution should continue until sufficient number of mutants are killed.

3.2 Modern Mutation Testing Process

The following steps were followed in Traditional Mutation Testing [18], [19]:-

1) Test Program and a define threshold value are given as input.
2) Create the mutants of test program.
3) Generate test cases automatically for the mutation system.
4) Run test cases for the input program.
5) Firstly execute test case against the original version of the test program and then check the output of the test program for each test case.
6) Run test cases on each live mutant and kill the mutants if they are found as killable mutants.
7) If threshold value is less than defined threshold then remove the ineffective test cases (ineffective tests cases are those test cases which are unable to remove any mutant).
8) If threshold value achieved equal or more than the define threshold Tested Program is obtained.
9) Lastly compare expected output of the effective test cases to the obtained output, and fix the program if difference found.

4. ASPECTJ BASED MUTATION TESTING TOOL

4.1 Proteum/AJ Tool

Proteum stands for **P**rogram **T**esting **U**sing **M**utants is a family of tools developed for Mutation Testing by the Software Engineering group at the University of São Paulo,

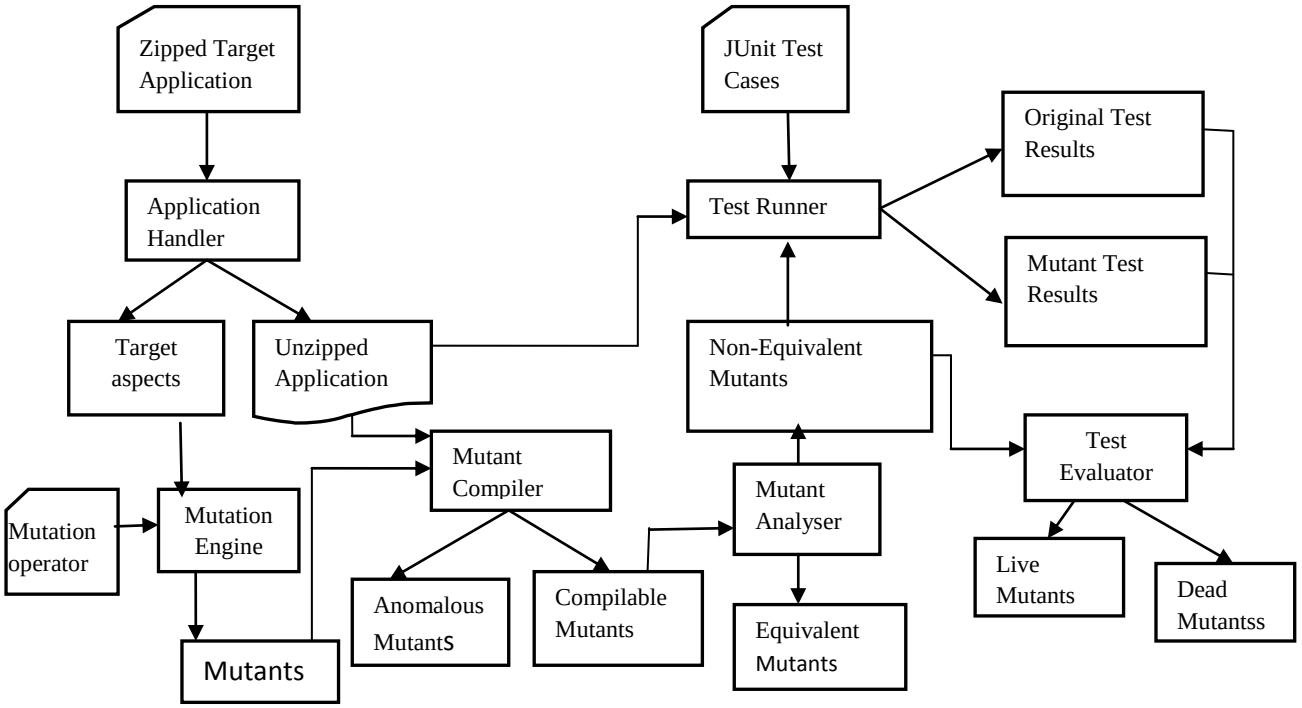

Fig.3 Proteum/AJ execution Flow Diagram

Brazil [8]. Proteum/Aj is a tool for mutation testing of Aspect Oriented Java programs written in AspectJ. The tool is implemented by using a reference architecture for software testing tools named RefTEST [9] from which all the main functional modules required for the tool are derived. RefTEST is based on mainly four things which are separation of concerns (SoC) principles, web systems pattern MVC (Model-View-Controller), 3- tier architectural patterns and ISO/IEC standard 12207 for Information Technology [9].

Proteum/AJ supports the four main steps of mutation testing, as originally described by DeMillo et al. [10]

(i) The original program is executed on the current test set and test results are stored;

(ii) The mutants are created based on a mutation operator selection that may evolve in new test cycle iterations;

(iii) The mutants can be executed all at once or individually, as well as the test set can be augmented or reduced based on specific strategies; and

(iv) The test results are evaluated so that mutants may be set as dead or equivalent, or mutants may remain alive.

Proteum/AJ handles four main concepts proposed by RefTEST [9]: testing criterion, artifact, test requirement and test case where testing criterion maps to both mutant handling and adequacy analysis requirements, artifact and test requirement comprises handling of aspect source code files and the handling of generated mutants respectively, and test case maps to the test case handling.

Testing with Proteum/AJ follows these steps [11][31]:

1) Zipped target application is submitted to Proteum/AJ as main input. This file contains all required modules like classes, aspects and libraries of the application under test. This file should always be in compressed form. The Application Handler module then runs a pre – processing step, whose outputs are decompressed original application and a list of target aspects.

2) The Decompressed application by Application Handler is sent to the Test Runner module together with the test case files. The application is executed by the Test Runner on available test set by invoking the JUnit Ant task.

3) The list of target aspects identified by the Application Handler and the set of mutation operators selected by the tester are given as input to the Mutation Engine. It produces a set of mutants which are further passed to the Mutant Compiler.

4) Each mutant sent to the Mutant Compiler invokes the ajc compiler through the iajc Ant Task provided by Aspect API [12]. The Mutant Compiler detects compilable and non-compilable mutants. The non-compilable mutants are classified as anomalous. The weaving information for compilable mutants produced by the ajc compiler is collected and the Mutant Analyzer further uses this information.

5) The Mutant Analyzer module finds the equivalent mutants from weaving information produced by ajc compiler and also updates the status of equivalent mutants. It computes the mutation score by the formula: [31]

MS= (dm) / (cm - em)

where MS is mutation score, dm is the number of dead mutants, cm is the number of compilable mutants and em is the number of equivalent mutants.

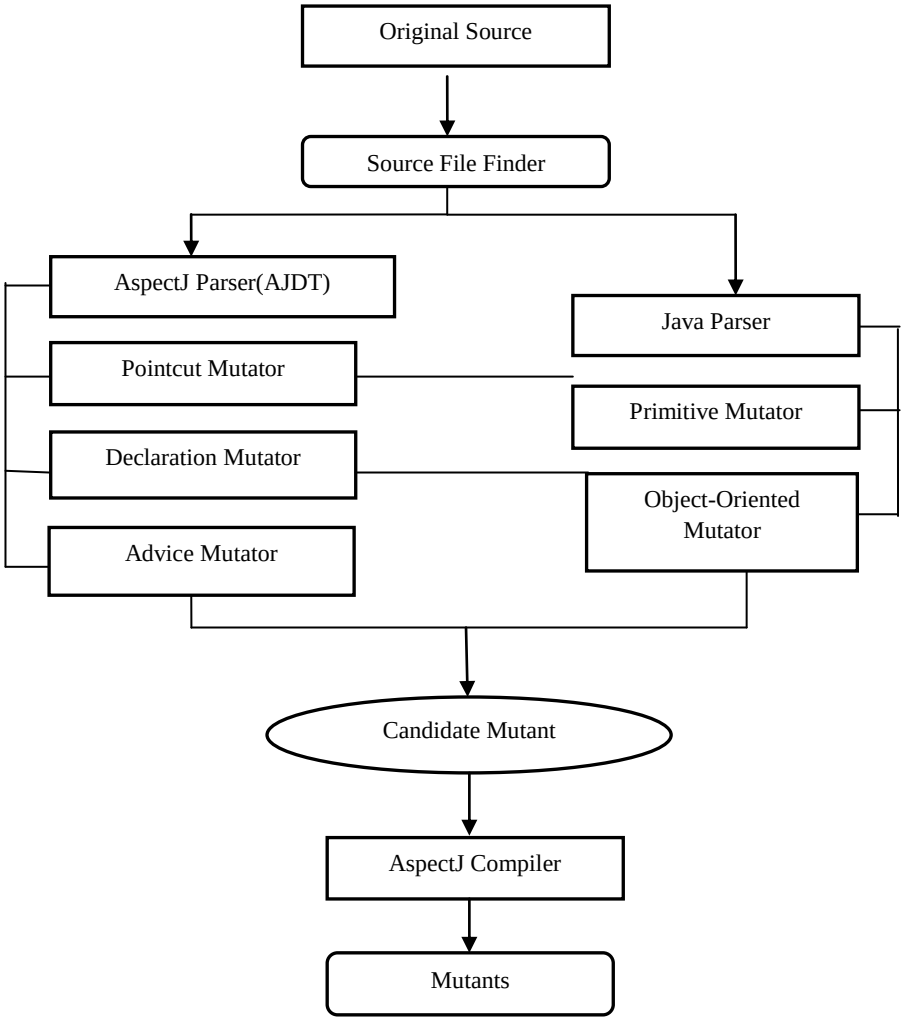

Fig.4 MuAspectJ components

The tool requires JUnit test cases to check the status of mutants which also have the ability of configuring of partial program runs.

4.2 MuAspectJ Tool

MuAspectJ is a testing tool proposed by Andrew Jackson and Siobhan Clarke. Evaluation of the tool is performed on the basis of the quality of mutants generated by it. To evaluate the quality of mutants benchmarking metrics is used that shows the quality of mutants generated by MuAspectJ against the mutants generated by Mu Java, a well known java mutation tool already used in mutation testing for Java. MuAspectJ defines the quality of mutants on the basis of location coverage and location density. Location coverage is a measure of the proportion of locations for which mutants are generated or it can be defined as the proportion of number of locations mutated over the line of code (LOC) of the module. Mutation density is a measure of the number of mutants that are generated for a location or it is measured as the number of proportion of the number of mutations over the number of locations per module [2].

The primary goal to develop MuAspectJ is to find the testability of the programs developed in AspectJ. Jackson and Clarke have compared the MuAspectJ and MuJava testing tools to check the confidence level associated with each testing tool. Both the tools possesses equivalent level of confidence with mutation analysis based on the mutants generated using their associated languages.

Health Watcher System (HWS) is taken to perform testing. It is an application having web based user interface with distributed database which allows citizens to register their complaints related to health issues [3, 4].

MuAspectJ comprises different types of mutation operators which insert faults at pointcut, declarations and advice locations [5].There are 27 AspectJ operators have been already found among which 15 are pointcut operators that add and remove wild-cards, change designator and pointcut types to super and sub types. There are 6 declarations operators which alter the error and warning instantiation declarations in AspectJ programming and rest of 6 are advice operators that changes advice or joinpoint, remove advice operator or changes the pointcut-advice bindings. This list is also extended by some additional pointcut operators that can improve pointcut negation by inserting or removing negation of pointcut.

Equivalent pointcuts(EP) are those which identify same set of join points. Equivalent pointcuts(EP) are free from error because there is no change to control flow in them. The

equivalent mutants increase the testability level and the confidence level.

The measure of testability and confidence level associated with equivalent pointcut mutants (EPM) is highly affected by the presence or absence of EPM. There are 2 types of EPM, equivalent to original pointcut(EPMO) and mutant pointcuts that diff from original but are equivalent to one another(EPMM) [2]. The absence of EPM's skews the testability and confidence level.

MuAspectJ uses eclipse plug-in to operate on AspectJ programs. It uses some components like source file finder, AspectJ parser, AspectJ operators, AspectJ compiler to perform testing The steps followed by MuAspectJ are:

1) The Source File Finder identifies all java and AspectJ source files in an AspectJ project for which analysis is required.
2) AspectJ parser creates a Document Object model (DOM).
3) The DOM is passed through some more mutator components consisting mutation operators that creates the candidate mutants. Candidate mutants are faults inserted in source files.
4) The AspectJ compiler is used to compile each mutant. After compilation mutants are considered as usable for mutation analysis.
5) The equivalent candidate mutants (ECM) skew the testability and accuracy. This issue is resolved on the basis of a strategy introduced by Anbalagan and Xie[6] in which a pre compilation step is taken to remove ECM.

Location coverage of MuAspect for HWS is found (1946) for (979) locations which is slightly greater then LC of MuJava (.1995) for (1014) locations [2]. The mutation density of MuAspectJ(6 mutants per module) for aspect locations is also slightly higher than MuJava(4.3 mutants per module) for java locations but the overall number of mutants get reduced due to the less number of mutant generated locations [2].

MuAspectJ supports high accuracy measurement of testability w.r.t other already existing testing tools. It supports mutation analysis. It reuses the pointcut generation strategy introduced by Anbalagan and Xie[7].

4.3 AdviceTracer

Delamare et al. proposed a tool for aspect oriented programming using AspectJ named, AdviceTracer [24, 25]. AdviceTracer is used to determine which advice is being executed and at what place in the base program. This information is used to determine the presence and absence of the advice along with whether it is executed correctly or not. As observed by Ferrari et al. that PCD is the most fault prone place in the aspect. One of the consequences of an incorrect PCD is that the advice is not woven at the expected place. This is known as the fragile point-cut [26] problem in AOP. There is also a well-known risk in the AOP that the PCD may match unintended joinpoints [27]. It is known as evolution paradox issue in AOP. A major challenge in order to determine the faults in the PCD is that it is an abstract declaration of the joinpoints. In order to cope with such a problem in which creation of the test cases that specifically intended and unintended joinpoints is done. In order to write these test cases, Delamare et al. developed a tool that checks whether advice is woven at a particular point in the program or not. In order to have more precise test-driven approach advice is used with JUnit.

In AdviceTracer the primitive methods used, are classified as: those that start or stop the AdviceTracer, those that configure the traced advices, and those that define assertions to specify the oracle. setAdviceTracerOn( ) method is used to call the AdviceTracer. It is to be called before an advice is to be woven. The setAdviceTracer( ) method is used to stop the tracing. And in between the two methods call, the AdviceTracer is used to store the information about which the advices were executed and their position of execution.The advice which are to be traced are specified by the tests using the AdviceTracer. If the Advice does not specify any of the advice, then there is a need to trace the advice again. addTracedAdvice( ) method is used to add the advice as a parameter to the collection of the traced advices. And setTraceAdvice( ) method is used to specify the collection of the advices to be traced.

Let us consider two test cases test1 and test2, where test1 specifies the absence of advice1 while test2 specifies the presence of advice.The test cases test1 and test2 are said to be modular because they specify a set of joinpoints where a specific joinpoints must or must not be woven. A test is not modular if it is unable to detect when is particular advice is to be woven and when not to be woven. The key concern behind using the modular test cases is that they are less affected by the addition and removal of the test cases of advices in the test cases. They are only affected by the change made on the PCD of the advices they trace. If the PCD or original program changes then there is a need to update the test cases.

4.3.1 Implementation of AdviceTracer

It deals with the actual analysis of which advice is being executed and at what place in the base program. The information regarding the name of the advice and joinpoint that triggered the advice execution is stored in the TraceElement object, which is a pair of Strings, one for the advice and other for the joinpoint. The advice is woven before each joinpoint matched by all the aspects under test. The PCD in the AdviceTracer aspect is "adviceexecution( )&& ! within (AdviceTracer) ". This PCD matches the execution of all the tested advices and not Advice-Tracer's own advice. All the TraceElement objects are stored in a list that can be retrieved by each test cases using the method (getExecutedAdvice). This list is updated each time when the AdviceTracer is set on. It contains TraceElement objects related to the advice execution between the calls of setAdviceTracerOn( ) and setAdviceTracerOff( ).

4.4 AJMutator Tool

Delamare et al. proposed a tool for aspect oriented programming using AspectJ named, AjMutator[21]. AjMutator classifies the mutant by comparing the sets of joinpoints matched by the mutant and initial PCD. Delamare

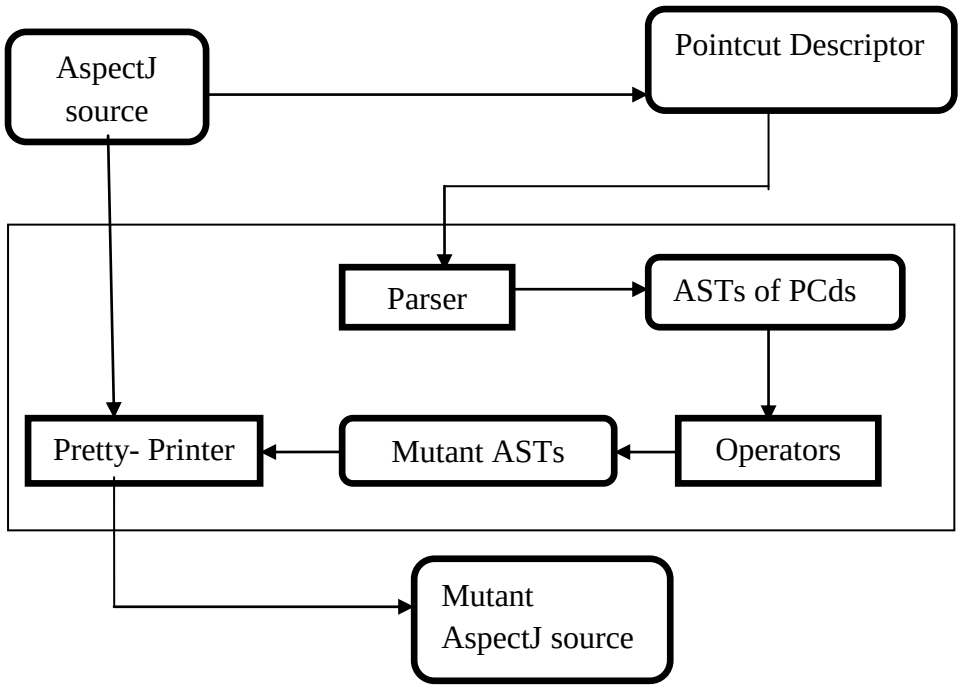

Fig. 5 The AjMutator process of generation of mutants.

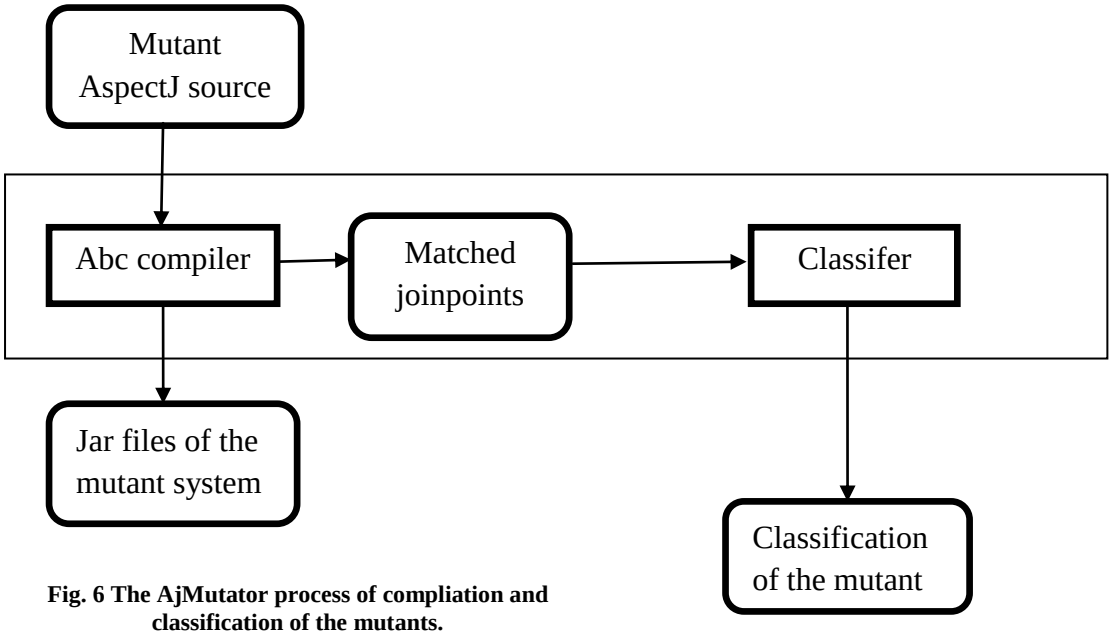

Fig. 6 The AjMutator process of compliation and classification of the mutants.

automated this classification at compile time by leveraging the static analysis performed by the compiler that computes the set of joinpoints matched by the PCDs. This classification is benefit to conclude the equivalent mutants if the mutants matches the same set of joinpoints. If the set of joinpoint is different, the advice is not correctly woven, and it can cause huge side effects. It produces mutants by inserting faults in the

PCDs. AjMutator is separated in three distinct parts [21]:

i. The generation of mutant source files from AspectJ source file.

ii. The compilation of the mutant source files.

iii. The execution of test cases on the mutant source files.

4.4.1 Generation of the mutants source files from AspectJ source file

AjMutator process for generation of the mutants is shown in the figure 2. There is an AST i.e. abstract-syntax tree for each PCD in the AspectJ source files. The operators insert faults in each of the AST, and hence there is an AST for the original PCD and also an AST for the mutant PCD. A pretty-printer is also used to produces a mutant AspectJ source file for each mutant AST. The parser has been developed using SableCC

which is an open source complier generator. This parser directly produces an AST from a PCD. To reduce the memory consumption the mutant AST is immediately printed and the reference is also not kept so that the garbage collector can free its memory space.

4.4.2 Compilation of the mutants source files

After the generation of the mutants, it is necessary to compile them. The compilation deals with the run of the test cases on the mutant and also to automatically classify them as shown in figure 3. This process relies on abc compiler which is an alternative of the AspectJ. The abc compiler produces a jar file of the system for each mutant. It also helps in finding out the joinpoints matched by the PCDs. This estimation is then used to classify the mutants.

4.4.3 Execution of the Test Cases on the mutants source files

All the test cases are executed against all the mutants. A mutant is considered killed if a test suite can experience a difference between the original and mutant system. It enables to detect the inserted fault. However if the mutant is still alive then, it is not able to detect the inserted the fault then there is a need of the new test data. Here we deal with the mutation score as ratio of the killed mutants to the total number of mutants.

5. BASIC PARAMETERS REQUIRED FOR MUTATION TESTING TOOLS

There are various requirements required for developing a mutation testing tool. Some among them are considered and compared in table1. Some work for selecting a mutation testing tool has already been done by Mayank Singh et al [14]. Some parameters required for mutation testing tools are as follows:

- Compiler used (It comprises different types of compilers used by different testing tool like ajc compiler, abc compiler).
- Generation of report about Equivalent Mutants (It includes whether report of equivalent mutants is generated by tool or not).
- Support Incremental Testing (It includes support to incremental testing type by the tool i.e. support can be fully or partially).
- Support all steps of Mutation Testing (It includes whether all steps of mutation testing are followed by the tool).
- Support Java5 Features (It comprises support to Java5 or its later versions).
- Automatic detection of mutants (It includes detection of mutants, detection can manually or automatically).
- Limitation on Size of program (It includes constraints on limitation of size of source code of program).
- Test case Activation/ deactivation(It includes whether activation and deactivation of test cases are possible).
- Implementation of Firm Mutation (It includes implementation of Firm Mutation approach. Firm Mutation approach is defined by Woodward and Halewood [32] as “the situation where a simple error is introduced into a program and which persists for one or more executions, but not for the entire program execution”)
- Large set of mutants (It includes support to large set of mutants).
- Supported Test Phase (It includes support to various types of test phases like unit or system level).
- Option for Selection of Mutation Operator (It includes features in testing tool to choose mutation operators to perform Mutation testing).
- Interface required (It includes which type of interface is required for tool i.e. menu or command line or browser plug-in).
- Support to Automatic Program Execution (It includes execution of program and mutants automatically at compile and run time).
- Test Case Handling (It includes handling of test cases during their execution and activation/ deactivation of test cases).
- Editing in Test Cases (It includes feature of editing and modification in the existing test cases).
- Generation of Anomalous Mutants (It includes generation of non executable mutants known as anomalous which are not included in mutation analysis)
- Resolve fragile point-cut problem (It includes resolution of Fragile point-cut Problem)
- Resolve paradox issue (It includes the risk in which the pcd may match to unintended joinpoints.)
- JUnit Support (This requirement includes use of JUnit for generation of mutants).

Table.1 Comparison among 4 mutation testing tools

Parameters	Proteum/ AJ	MuAspectJ	AjMutator	Advice Tracer
Complier used	Ajc compiler	Any AspectJ compiler	abc complier	AjMutator
Support Incremenal testing	Yes	No	Yes	No
Support all steps of Mutation Testing	Yes	Yes	Yes	No
Support Java5 Features	Yes	No	Partial	Partial
Automatic detection of Equivalent Mutants	Yes	Partial	Yes	No
Limitation on Size of program	No	Partial	Partial	Partial

Test case Activation/ deactivation	Partial	No	Partial	Partial
Implementation of Firm Mutation	Yes	No	Yes	No
Large set of mutants	supported	Partially	No	No
Supported Test Phase	Unit	Unit	Unit	Unit
Option for Selection of Mutation Operator	Yes	No	Yes	No
Interface required	Command line based	Eclipse plug-in	Comma-nd line based	Comm-and line based
Support to Automatic Program Execution	Yes	Yes	Yes	No
Test Case Handling	Partial	Partial	Partial	Partial
Editing in Test Cases	No	No	No	No
Generation of Anomalous Mutants	Yes	Partial	Partial	Partial
Resolve fragile point-cut problem	No	No	No	Yes
Resolve paradox issue	No	No	No	Yes
Generation of report about Equivalent Mutants	Yes	No	Yes	No

6. CONCLUSIONS AND FUTURE SCOPE

As there are so many testing tools available in market for performing Mutation Testing but there is not a single tool which comprises all the requirements mentioned in Table 1. Each tool fulfills specific set of requirements for the testing. Some weakness like complexity, interface support and support to new versions of Java are present in already existing mutation testing tools. These tools support only some specific languages and work with different interfaces. A tool which support all testing techniques and support all phases i.e. unit, integration and system level should be developed. Already exiting tools support only selective interface which make difficult to use all testing tool on the same computer and to remember each tool's interface. A tool should be developed which work on all interface i.e. menu, browser plugin and command line. There is another drawback that all testing tools work differently on same testing technique which makes difficult to choose best one tool among all. Each testing tool uses different external tools for performing testing. There is no tool which supports AspectJ System level test phase. There is no tool in which editing in test cases is possible and which performs adequacy analysis fully. Most of the tools are restricted to program size.

Our future Scope is to develop a tool which is based on AspectJ system level and works on all the interfaces whether it is menu, browser plug-in or command line. We will try to develop a mutation testing tool which use only JUnit as its external unit and unaffected from the size of program. This will also allowediting in test cases and generation of report about equivalent mutants will be possible. It will overcome all the drawbacks of existing mutation testing tools and gives a good mutation score.

7. REFERENCES

[1] Zuhoor Al-Khanjari, Martin Woodward, and Haider Ali Ramadhan. Critical analysis of the pie testability technique. Software Quality Control, 10(4):331{354, 2002.

[2] Andrew Jackson and Siobhán Clarke, "MuAspectJ: Mutant Generation to Support Measuring the Testability of AspectJ Programs", Technical report (TCD-CS-2009-38), ACM, September 2009.

[3] T. Sugeta, J. C. Maldonado, and W. E. Wong, "Mutation Testing Applied to Validate SDL Specifications," in Proceedings of the 16th IFIP International Conference on Testing of Communicating Systems, ser. LNCS, vol. 2978, Oxford, UK, p. 2741, 17-19 March 2004.

[4] Uira Kulesza, Claudio Sant'Anna, Alessandro Garcia, Roberta Coelho, Arndt von Staa, and Carlos Lucena, "Quantifying the effects of aspect-oriented programming:A maintenance study", in ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance, IEEE Computer Society, pages 223-233, Washington, DC, USA, 2006.

[5] Fabiano Cutigi Ferrari, Jose Carlos Maldonado, and Awais Rashid. Mutation testing for aspect-oriented programs. In ICST '08: Proceedings of the 2008 International Conference on Software Testing, Veri_cation, and Validation, pages 52{61, Washington, DC, USA, 2008. IEEE Computer Society.

[6] P. Anbalagan and T. Xie, "Automated Generation of Pointcut Mutants for Testing Pointcuts in AspectJ Programs," in Proceedings of the 19th International Symposium on Software Reliability Engineering (ISSRE'08). Redmond, Washington: IEEE Computer Society, pp. 239–248, 11-14 November 2008.

[7] Prasanth Anbalagan and Tao Xie. Efficient mutant generation for mutation testing of pointcuts in aspect-oriented programs. Mutation, 0:3, 2006.

[8] J. C. Maldonado et al. Proteum: A family of tools to support specification and program testing based on mutation.In Mutation 2000 Symposium (Tool Session), pages 113–116. Kluwer, 2000.

[9] E. Y. Nakagawa, A. S. Sim˜ao, F. C. Ferrari, and J. C. Maldonado, "Towards a reference architecture for software testing tools", in SEKE'07, pages 157–162, 2007.

[10] R. A. DeMillo, R. J. Lipton, and F. G. Sayward.” Hints on test data selection: Help for the practicing programmer”. IEEE Computer, 11(4):34–43, 1978.

[11] Fabiano Cutigi Ferrari, Elisa Yumi Nakagawa, José Carlos Maldonado, Awais Rashid, Proteum/AJ: a mutation system for AspectJ programs”, in Proceedings of the tenth International conference on Aspect-oriented software development companion AOSD’11), ACM New York, NY, USA , 2010.

[12] AspectJ documentation, 2010. http://www.eclipse.org/ aspectj /docs.php - accessed on 01/09/2011.

[13]] Roger T.Alexender, James M.Bieman, “Towards the Systematic Testing of Aspect-oriented Programs”, in 2004.

[14] Mayank Singh, Shailendra Mishra, Rajib Mall, “Accessing and Evaluating aspect based Mutation Testing Tools), IJCA volume 31- No.1, October 2011.

[15] David Schuler, Valentin Dallmier, and Andreas Zeller, “Efficient Mutation Testing by Checking Invariant Violations”, in ISSTA’09, July 19-23,2009,Chicago,Illinois,USA.

[16] R.A. DeMillo, R.J. Lipton, and F.G. Sayward, ”Hints on Test Data Selection: Help for the Practicing Programmer,” Computer, vol. 11,no. 4,pp. 34-41, April 1978.

[17] Beizer B., ”Software Testing Techniques”,1990.

[18] Offutt A., A Practical System for Mutation Testing: Help for the Common Programmer, Twelfth International Conference on Testing Computer Software,99-109,Washington D.C. June 1995.

[19] DeMillo R., Constraint-Based Automatic Test Data Generation, IEEE Transactions on Software Engineering, 17(9):900-910,1991.

[20] J.H. Adrews, L.C. Briand, Y. Labiche, “Is Mutation an Appropriate Tool for Testing Experiments”, in ICSE’05,May 15-21, 2005, St. Lousis, Missouri, USA.

[21] R. Delamare, B. Baudry, and Y.Le traon,”AjMutator: A Tool For The Mutation Analysis Of AspectJ Pointcut Descriptor,” in Proceedings of the 4th International Workshop on Mutation Analysis(MUTATION’09),published with Proceedings of the 2nd International Conference on Software Testing Verification and Validation Workshops, Denver, Colorado: IEEE Computer Society ,pp. 200-204, 1-4 April 2009.

[22] F.C. Ferrari, J.C. Maldonado and A. Rashid,” Mutation testing for aspect-oriented programs”. In ICST’08: Proceedings of the 1st International Conference on Software Testing, Verification, and Validation, pages 52-61, April 2008.

[23] M.R.Woodward,” Mutation Testing-An Evolving Technique”.

[24] R. Delamare, B. Baudry, S. Ghosh and Y.Le Traon, ”A Test-Driven Approach to Developing Pointcut Descriptors in AspectJ,” in Proceedings of the 2nd International Conference on Software Testing Verification and Validation (ICST’09), Davor, pp. 376-38,01-04 April 2009.

[25] R.Delamare Advicetracer. http://www.irista.fr/triskell/ Software/protos/advicetracer.

[26] M. Storzer and J. Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In ICSM’05,pages 653-656, Budapeset, Hungary, September 2005.

[27] T. Tourwe, J. Brichau and K.Gybels. On the existence of the aosd-evolution paradox. In AOSD 2003 Workshop on Software-engineering Properties of Languages for Aspect Technologies, Boston, USA,2003.

[28] Yu-Seung Ma, Mary Jean Harrold and Yong-Rae Kwon,” Evaluation of Mutation Testing for Object-Oriented Programs “ in ICSE’06, May 20-28, 2006, Shanghai, China.

[29] Roland H. untch, “ Mutation-based Software Testing Using Program Schemata” in 1992.

[30] Evan Martin and Tao Xie,” A Fault Model and Mutation Testing of Access Control Policies” in 2007.

[31] Fabiano Cutigi Ferrari, Elisa Yumi Nakagawa, José Carlos Maldonado, Awais Rashid, “Automating the Mutation Testing of Aspect – Oriented Java Programs” in Proceedings of the tenth International conference on Aspect-oriented software development companion (AOSD’11), ACM New York, NY, USA , 2010

[32] M. Woodward and K. Halewood. “From weak to strong, dead or alive? An analysis of some mutation testing issues”. In Workshop on Soft. Testing, Verification, and Analysis, pages 152–158. IEEE Computer Society, 1988.