MuAspectJ: Mutant Generation to Support Measuring the Testability of AspectJ Programs

Andrew Jackson and Siobhán Clarke
Distributed Systems Group
School of Computer Science and Statistics
Trinity College Dublin,Ireland
+353 1 896 1756
{firstname.lastname}@cs.tcd.ie

ABSTRACT

The impact of Aspect-Oriented Software Development (AOSD) on testability must be quantified before it can be considered for widespread adoption by industry. One way to measure testability is through mutation analysis (MA). In MA, a mutation tool generates faults for locations in software. Each fault is created in a new version of the software called a mutant. Testability of a location is measured by executing tests against mutants and counting the proportion of mutants that cause test failure. To quantify the testability of approaches to AOSD through MA, mutant generation tools are needed. This paper introduces MuAspectJ, a tool for generating mutants for AspectJ programs, to satisfy this need.

The tool is evaluated in terms of the quality of mutants it generates. Assertions reached about the testability of the software under MA are derived by aggregating the testability of each location. The quality of the assertions that can be derived from MA results is only as good as the mutants on which the analysis is based. MuAspectJ is evaluated by benchmarking metrics that indicate the quality of generated mutants against the existing well known Java mutation tool, MuJava. The results validate the quality of the mutants generated by MuAspectJ.

1. INTRODUCTION

A significant proportion of the total cost of software is attributed to testing over its lifetime [24]. One way to reduce this cost is to increase software testability. Testability is an measure of how easily software exposes faults when tested [2]. By improving testability the cost of testing is reduced. There is anecdotal evidence to suggest that Aspect-Oriented Programming (AOP), such as AspectJ, may improve testability [9]. However, for industry to consider the widespread adoption of AOP, the impact of AOP on testability must be quantified.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

Mutation Analysis (MA) is an accurate approach to measuring testability [2]. There are other approaches that advocate measuring testability in terms of the complexity of the programs static structure. Analysis of such measures suggests that they are good indicators of testability [16] but are too coarse grained [26] to provide a reliable and accurate measure. To ensure that the impact of AOP on testability is accurately quantified support to apply mutation analysis in comparative experiments is needed.

In MA, mutation operators are used to systematically create faults at locations in a programs source code [19]. These faults simulate the types of real errors that a competent programmer would make. A location is any element in the code at which a fault can occur. In AspectJ faults can occur at locations including expressions, statements, methods, fields, declarations, advice and pointcuts. Mutant generation tools identify locations at which to apply mutation operators. Each mutation operator applied to a location inserts a particular type of fault. Each fault is created in a new mutant version of the software. The testability of a location is measured by executing tests against mutants generated for the location. The testability of the location is measured as the proportion of the mutants that cause the test to fail [19, 2].

The results of MA are interpreted by analysing the testability of the programs locations. Assertions reached about the testability of the program under analysis are made by aggregating the testability of each location [2]. The confidence associated with the measure of testability for each location is based on the number of mutants generated for each location level. The higher the number of mutants generated for a location the tighter the confidence interval around each testability measure. The confidence associated with the measure of testability for the program is based on the confidence associated with individual measures and the proportion of locations in the program that are considered in the analysis. Increasing the proportion of locations covered means that assertions reached through aggregation is more representative of the entire program.

To employ MA in experiments to quantify the impact of AOP on testability, mutant generation tools that can generate mutations for AO and non-AO specific constructs are required. This paper introduces MuAspectJ, a tool for generating mutants for AspectJ programs. The tool is an extension of an existing tool for generating mutants for Java called MuJava[19]. The tool provides a complete set of mutation operators that can be applied to a broad range of AO and non-AO specific locations in AspectJ programs.

MuAspectJ evaluated in terms of the quality of mutants it generates. The quality of mutants is measured by the confidence that can be associated with the results of mutation analysis based on the mutants. In this paper an quality is in terms of location coverage and mutation density. Location coverage is a measure of the proportion of locations for which mutants are generated. Mutation density is a measure of the number of mutants that are generated for a location. This evaluation compares the location coverage and mutation density achieved by MuAspectJ against that achieved by a benchmark set by MuJava. These measures are derived from analysing the mutants generated by applying each tool to a well known case study implemented as an equivalent AspectJ and Java programs.

The results of this comparison shows that the location coverage and mutation density of MuAspectJ are equivalent to MuJava. This means that an equivalent level of confidence can be associated with the results of mutation analysis based on using mutants generated from either tool. The results are a positive validation of the quality of the mutants generated by MuAspectJ. This validation ensures that researches who want toconduct experiments on AspectJ program using MA can do so with confidence.

The remainder of the paper is structured as follows. First, the background for this tool is presented. Then, the MuAspectJ tool is introduced and described. This is followed by the evaluation of the tool and a discussion of the results. The work is positioned then in relation to existing work. Finally, the paper is concluded and our future work is outlined.

2. BACKGROUND

In this section the motivation that underlies the choice of mutation analysis as a means to measure testability is outlined. The audience that are expected to use MuAspectJ are then identified. The background to the mutation operators implemented by MuAspectJ is presented and the case study on which the tool is explained and evaluated are presented.

2.1 Mutation Analysis

The goal of MuAspectJ is to provide support to accurately quantify the impact of AOSD on testability. Although a number of existing approaches that provide means for measuring testability, this paper describes a tool to provide support for mutation analysis. These approaches employ static measures of complexity based on program structure [5, 15, 25, 6] as measures of testability. These measures are indended for use at the design stage to indicate areas that are estaimated to be of low testability early in development. Mutation analysis provides an accurate measure of testability because it simulates real faults and measures the programs ability to expose those faults under test. Static measures do provide some indication of where faults may arise [16] they are described as too coarse grained to provide a reliable and accurate measure of testability [26]. To ensure an accurate for quantify the impact of AOSD on testability the provision of support for mutation analysis is preferred.

2.2 Usage of MuAspectJ

MuAspectJ is a tool that generates mutants that can be used in mutation analysis. Mutation analysis can be used to measure testability but can also be used in testing experiments. Mutants generated from programs using MuJava have been used in various experiments, including to assessments of different testing strategies [20], of mutation analysis effecienct [21, 23] and frameworks to support the testing process[27]. Our primary goal in creating MuAspectJ is to provide a means for researchers to measure the testability of AspectJ programs through experiments. MuAspectJ can however also be used by researchers to generate mutants that can be used a wide array of testing experiments based on AspectJ programs.

2.3 Case Study

The case study we apply MuAspectJ to is the Health Watcher (HW) system that has been used in empirical studies that investigate the benefits of AOSD [14, 17, 8]. The HW is a distributed, database driven application with a web based user interface that allows citizens to register complaints regarding health issues. In this case study there are AspectJ and Java, implementations of the HW system. Both implementations satisfy equivalent requirements and are both equally well designed.

2.4 Mutation Operators

MuAspectJ generates mutants for AspectJ programs. AspectJ is an extension of the Java language. To ensure that the testability of AspectJ and Java specific locations in AspectJ programs can be measured a set of mutation operators that can generate mutants at these locations is needed. MuAspectJ provides Java and AspectJ mutation operators to support the mutation of locations in AspectJ progress.

2.4.1 Java Operators

There are an existing set of well known mutation operators for Java implemented in a tool called MuJava [19]. MuJava adapts mutant operators from existing works and tools [1, 22, 19] for Java. MuJava supports 44 mutation operators. 14 of these operators are primitive operators and the remaining 30 are object-oriented operators. The primitive operators create mutants with faults by replacing, inserting and deleting Java operators in expressions. The 30 object-oriented operators are broken down into operators that generate inheritance, polymorphism and Java-specific. Inheritance operators change reference to inherited members by adding or removing overriding members or adding or removing calls to super. Polymorphism operators generate faults in references to classes or members that are polymorphic by changing class and member references or specification. Java specific operators cause an assortment of faults, ranging from inserting or deleting Java specific keywords i.e., *this* and *static*, to replacing reference and content assignment or comparison. MuAspectJ adapts the Java mutation operators to support the mutation of Java locations in AspectJ progress. The adaptation of Java mutation operators for AspectJ within MuAspectJ avoids reinventing the wheel. An additional advantage is that the application of Java operators by both tools can be directly compared.

2.4.2 AspectJ Operators

MuAspectJ provides a set of mutation operators that insert faults at pointcut, advice and declarations locations. MuAspectJ implements the mutation operators that have been identified by Ferrari et at al [12]. In their work Ferrari et at al. identify a comprehensive set of mutation operators based on a fault models [1, 7, 9, 11], fault classifications [18] and bug reports [28]. They identify 15 pointcut operators that insert or remove wild-cards, change designator types, change pointcut types to super and sub types, and alter flow and contextual designators types. They identify 6 declaration operators that remove or alter precedence, soft, error or warning and aspect instantiation declarations. They also identify 6 advice operators that alter advice or join point handle (this to enclosing), remove advice implementation and proceed statements and change pointcut-advice bindings. MuAspectJ extends this list with additional pointcut operators to insert and remove pointcut negation. Work on identifying an appropriate set of mutation operators for AOP just beginning. We expect new works will emerge that identify new operators and refine the existing set. Extensions or alterations can easily be made to the set supported by MuAspectJ. The implementation of the AspectJ operators identified by Ferrari et al ensures that MuAspectJ supports a comprehensive set of mutation operators that generate realistic faults.

2.5 Equivalent Pointcut Mutants

Pointcuts are locations in AspectJ programs that can be mutated by removing or adding wild-cards. The systematic mutation of pointcuts in this way results in a large number of mutants many of which are equivalent [4]. The use of equivalent mutants in Mutation Analysis (MA) causes the measure of testability becomes skewed and associated with a high level of confidence.

A pointcut identifies a set join point at run-time. Pointcuts are equivalent if they identify the same set of join points. Faults in mutated pointcuts can change the set of join points selected by a pointcut. A different join point selection can change the control flow of the program, which may cause program error. If pointcuts are equivalent, there is no possibility of program error because there is no change to control flow. If mutant pointcuts select the same sets of join points then these represent the same error.

Testability is measured as the proportion mutants that fail when tested. The accuracy of testability measure is indicated by the with of the confidence interval around the measure. The width of the interval is measured as follows $interval = 2\sqrt{\frac{p(1-p)}{n}}$ where p is the measure of testability and n is the number of mutations from which p is derived. A smaller width indicates a more accurate result. From this we can see that increases in n has the effect of increasing the interval width, reducing our confidence in the accuracy of the testability measure.

The problem is illustrated by showing how the measure of testability and the indicator or accuracy change when equivalent pointcut mutants (EPM) are used in MA. There are two types of EPM, mutant pointcuts that are equivalent to the original pointcut (EPMO) and mutant pointcuts that differ from the original but are equivalent to one another (EPMM). Table 1 is an example of how the testability of a pointcut can be effected if EPMOs and EPMMs are not excluded from use in MA.

The first row shows that when there are no EPMs that testability is high (.67), but accuracy is low (Width .85). The second row shows that when EPMOs are not excluded the testability is reduced (.02), but the accuracy in this measure is increased (Width .07). The third row shows that when EPMMs are not excluded the testability is further reduced (.009), and the accuracy in this measure is further increased (Width .03).

Scenario	MD	Equiv	Testability	Width
No EPM	3	0	.67	.85
EPMO	100	97	.02	.07
EPMM	224	221	.009	.03

Table 1: Equivalent Pointcut Problem

This simple example clearly shows that if equivalent pointcuts are not removed that they can dramatically skew the measure of testability and confidence associated with that measure. In this example we could conclude that the testability of the pointcut is very low and that this measure was very accurate when in fact the opposite is true. This issue could, if not addressed, skew the overall result of program testability when skewed measures pointcut testability are subsumed, through aggregation, into the measure of program testability.

3. TOOL

This paper introduces MuAspectJ, a tool for generating mutants for AspectJ programs. The mutants generated by this tool can be used in mutation analysis to quantify the testability of AspectJ programs. In the following sections an overview of the tool is presented. Following this, the implementation of the Java and AspectJ mutation operators is briefly described. The application of these operators to the Java and AspectJ implementations of the Health Watcher case study is outlined. The mutants generated by the MuAspectJ mutation operators are compared with those generated by MuJava. The similarities and differences observed are identified and explained. This means that an equivalent level of confidence can be associated with results of mutation analysis based on using mutants generated from either tool.

3.1 Overview

MuAspectJ is implemented as an eclipse plug-in that operates on AspectJ projects. The high level components that make up the plug-in are presented in Figure 1. The *Source File Finder* component identifies all Java and AspectJ source files in an AspectJ project under analysis. Java and AspectJ source files are parsed using the relevant *Parser* (*AspectJ* or *Java*). Each parser creates a Document Object Model (DOM)[1] representation of the source. The DOM is then passed to a series of *Mutator* components. Each *Mutator* component identifies all locations in the DOM at which the set mutation operators it controls can be applied. *Mutator* components apply operators at locations to generate candidate mutants. Candidate mutants are new versions of the source file in which a fault has been inserted. The *Primitive* and *Object-Oriented Mutators* control the Java mutation operators, described in Section 2.4.1. The *Pointcut, Advice* and *Declaration Mutators* control the AspectJ mutation operators, also introduced in Section 2.4.2. Candidate mutants must be compiled before they can be used in mutation analysis. The fault inserted into a candidate mutant may cause compile time errors. The *AspectJ Compiler* component is used to compile each candidate mutant. Candidate mutants

[1]http://www.w3.org/DOM/

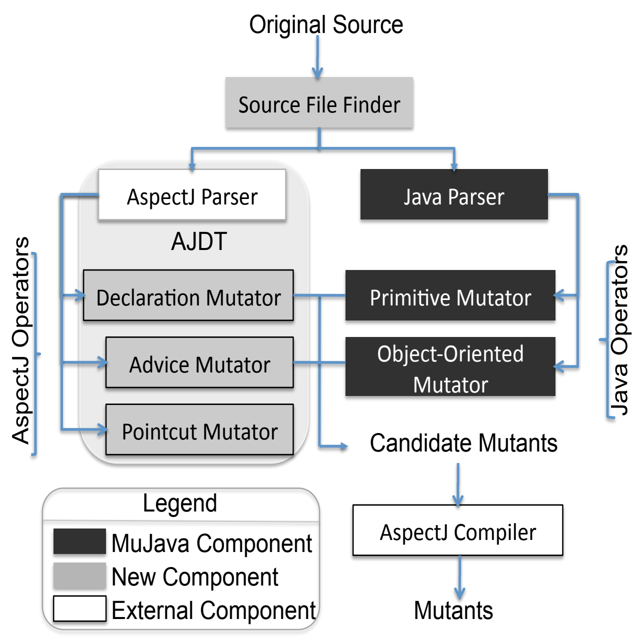

Figure 1: MuAspectJ Components

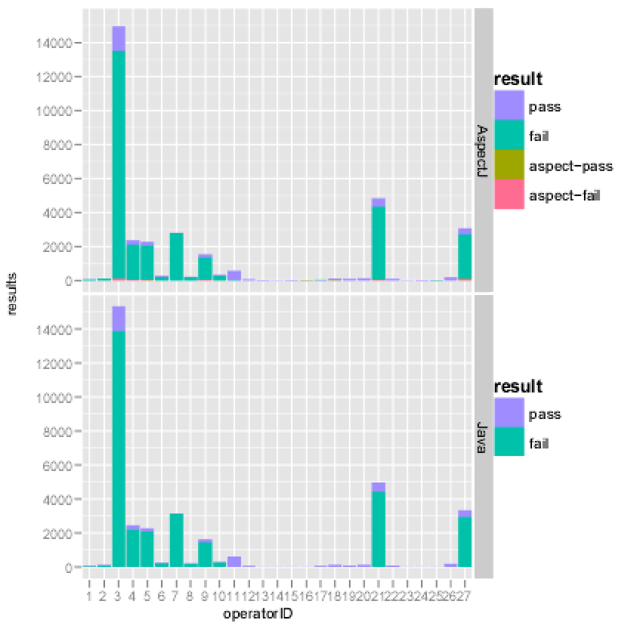

Figure 2: Java Operators - Results

that do compile are usable in mutation analysis. Those that fail to compile cannot be used in mutation analysis.

3.2 Java Mutation Operators

In this section we briefly outline how the Java mutation operators are implemented and detail how

3.2.1 *Implementation*

The MuAspectJ tool adapts the Java operators outlined in Section 2.4.1 to enable their application to locations in AspectJ programs. MuAspectJ reuses and adapts some components from the MuJava tool that implement these operators. The adapted components are clearly identified in Figure 1. These components support the application of Java operators to locations in classes or aspects. The reused components are altered to use the AspectJ compiler to compile candidate mutants.

3.2.2 *Case Study Application*

These Java operators generate candidate mutants which must pass compilation to be usable in mutation analysis. Figure 2 shows the number of candidate mutants generated by the Java mutant operators within the MuJava and MuAspectJ tools when applied to the Java and AspectJ implementation of the HW case study, introduced in Section 2.3. The proportion of candidates that pass and fail are indicated for each operator. This figure validates that both tools generate the same results generate roughly the same number of candidates and mutants per-operator. It also shows that although the tools are applied to Java and AspectJ implementations that change the distribution of functionality, the number of candidates and mutants generated stays the same. In the AspectJ implementation crosscutting functionality is encapsulated within aspects and is scattered within the Java implementation. The small number of candidates generated within aspects that pass compilation are identified in the figure to highlight that these tools are applied to different implementations. By differentiating these, we can also conclude that there are very few mutants generated by Java operators applied to AspectJ programs.

3.3 AspectJ Mutation Operators

3.3.1 *Implementation*

The implementation of the AspectJ *Mutator* components and the mutation operators they based on the eclipse Java (EJ) and AspectJ (EAJ) APIs [10]. The tool makes use of the aspect parser from the EAJ API to create a DOM representation of the source. The various *Mutator* components then search the DOM for locations (pointcuts, declarations and advice) to apply AspectJ mutation operators. The mutation operators generate candidate mutants by creating new versions of the DOM in which the location to which the operator is applied. The operator then inserts a fault at the location in the new DOM. The DOM is then transformed back into source and is compiled by the AspectJ compiler.

3.3.2 *Basic & Contextual Operators*

For the most part, mutation of aspect-oriented locations involves the simple removal from or alteration of an element in the DOM. For instance, the removal of a *proceed* statement from an around advice requires that the statement be removed from the DOM. The alteration of the ordering of the aspects declared in a *precedence* declaration requires that the declaration be extracted, the order changed and then replaced. There are however, various mutation operators that require contextual information from which mutants can be generated. For instance, operators that strengthen pointcuts by replacing elements that contain wild cards with concrete elements require some contextual access to valid elements that can be used as replacements.

Table 2 presents an example of a contextual strengthening mutation of an execution pointcut designator. The original pointcut specifies the type concretely as the EmployeeRe-

Pointcut	*execution*(pattern)
Original	** EmployeeRecord.*(..)*
Mutant-1	*Employee search(String)*
Mutant-2	*void insert(Employee)*
Mutant-3	*void update(Employee)*

Table 2: Contextual Mutation

Pointcut	*execution*(pattern)	JPS	Compile
Original	*HealthWatcherFacade.*(..)*	48	-
Mutant-1	*Health*WatcherFacade.*(..)*	48	no
Mutant-2	*Health*Facade.*(..)*	48	no
Mutant-3	**.*(..)*	999	yes

Table 3: Equivalent Candidate Mutations

cord type and specifies the method signature as wild-cards. This indicates that any only methods of that the EmployeeRecord type can be used to generate strengthening mutants. The EmployeeRecord type contains three methods. The tool uses the search mechanism in the EJ API to identify these methods. The strengthening operator then uses this contextual information to generate mutants 1, 2 and 3 in Table 2. Other examples of contextual operators include operators that replace types with their super or sub types and operators that change the types of method parameters.

3.3.3 Equivalent Pointcut Mutants

As demonstrated in Section 3, the use of equivalent mutants in Mutation Analysis (MA) causes the accuracy and measure and of testability to become skewed. To avoid this issue, a pre-compilation step is taken to remove equivalent candidate mutations. This step is based on existing strategy introduced by Anbalagan and Xie [4]. The EAJ API provides a mechanism to identify the join point shadows associated with advice their associated pointcuts. To test for equivalency, the join point shadows associated with the original pointcut and the mutations of that pointcut are recorded. If mutations result in a set of join point shadows that are the same as the original pointcut or an existing mutation the candidate mutant is not compiled.

Table 3 illustrates an example of equivalent candidate removal. The original pointcut is presented in the first row of the table. The pointcut weakening operator alters the pointcut by inserting wild-cards into the pointcut. The result of applying the operator are 224 candidate mutants, 3 of which are presented in Table 3. Candidate mutants one and two result in the same set of join point shadows as the original. These mutations and the original match the exact same 48 join point shadows. Candidate mutant three in contrast matches 999 join point shadows. Candidate mutants three is compiled and if compilation is successful can be used in mutation analysis.

3.3.4 Case Study Application

Figure 3 shows the number of candidate mutants generated by the AspectJ operators. Operators 1-5 are advice based operators, 6 is a declaration based operator and 7-15

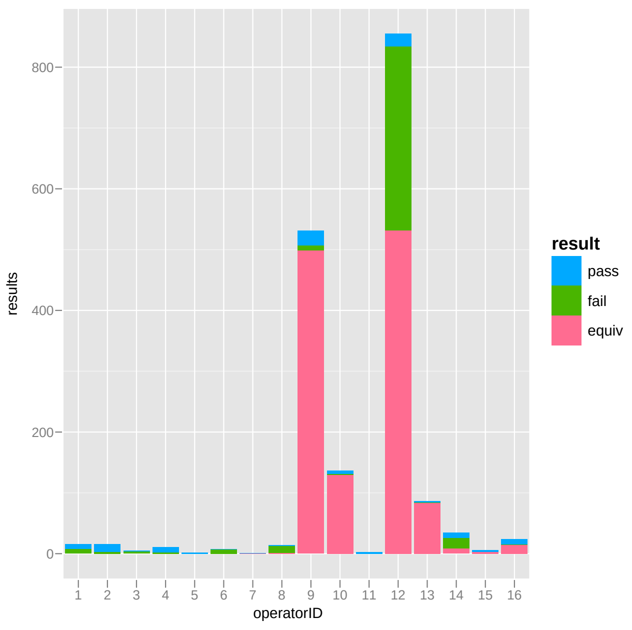

Figure 3: AspectJ Operators - Result

are pointcut based operators. It can immediately be noted that 5 of the 6 advice operators are used, 3 of the 6 declaration operators is used and 9 of the 15 pointcut operators are used. Lower usage rates of declaration and pointcut operators is due to a lack of locations in the aspects at which these operators could be applied. For instance, no initialization/preinitialization/get/set pointcut designators were found , no annotations were found and no throwing clauses were found in pointcuts. No error/warning declarations or deployment clauses were found and as such operators that generate mutations at locations of these types cannot be applied.

In Figure 3 the number of mutants that pass and fail compilation can be identified by colour. For the pointcut operators the number of mutants that are found to be equivalent are also identified by colour. The pointcut operators that show high numbers of equivalent mutants (9,10,12 and 13) are those that weaken pointcuts and strengthen the pointcuts. From this figure clearly shows that the introduction of a step to remove these equivalent mutants before compilation has a dramatic reduction effect on the number of candidates that are reach compilation.

The scale of candidate mutant generation differs between Figure 3 and Figure 2. It is clear that the Java operators have produced far more mutant candidates than those produced by the AspectJ operators. This is due to the difference in the number of aspects when compared to classes. In the AspectJ implementation there are 12 aspects and 89 classes. Section 3.4 clearly demonstrates that size has a direct impact on the number of candidate mutants generated.

3.4 Generated Mutants

The number of mutants generated by a mutant generation tool is the sum of mutants generated by the mutation operators it supports. Table 4 presents the proportion of mutants generated from candidates (PMG) by operators in

Operator	Mu-Tool	Mutants	Candidates	PMG
Java	Java	4367	35325	.124
Java	AspectJ	4303	34048	.126
AspectJ	AspectJ	110	1748	.062
Both	AspectJ	4411	35796	.125

Table 4: Mutant Generation

both tools. From examining this table the contribution of the Java and AspectJ mutation operators to the number of mutants generated by MuJava and MuAspectJ when applied to the Java and AspectJ implementations can be identified.

Both MuJava and MuAspectJ support the same set of Java mutation operators. There is slight but insignificant difference (.124 - row one v .126 - row two, p-value 0.2774) between the PMG of Java operators in the MuJava and MuAspectJ tools. The big difference between the MuJava and MuAspectJ tools is that the MuAspectJ supports a set of AspectJ mutation operators. The effect of AspectJ operators on the overall PMG for the AspectJ implementation by examining the PMG of the AspectJ and Java operators. The PMG for AspectJ operators is quite low (.62 - row three) and this reduces the overall PMG for the AspectJ implementation (from .126 - row two, to .125 row four). Despite this reduction the difference in PMG with the Java implementation (.124 v .125, p-value 0.881) remains insignificant.

In summary, although AspectJ operators do reduce the overall PMG of the AspectJ implementation, the reduction does not lead to a significant difference. The AspectJ operators produce a relatively small number of mutants and candidates because there are a relatively small number of aspects and consequently aspect locations in the AspectJ implementation. The AspectJ implementation is made up of 101 (aspect and class) modules with 5006 lines of code LOC. 11 of these modules are aspects and between them they account for 450 LOC. Considering that a small proportion of those lines of code will contain locations that AspectJ operators can be applied to, the small number of mutants and candidates produced by AspectJ operators is understandable. Because the number of mutants and candidates is so low compared with the Java operators that the impact of the AspectJ operators is minimal.

4. EVALUATION

MuAspectJ evaluated in terms of the quality of mutants it generates. The confidence that can be associated with the results of mutation analysis is bounded by the quality of mutants used. In this paper, quality is measured is in terms of location coverage and mutation density. Location coverage (LC) is a measure of the proportion of locations for which mutants are generated. Mutation density (MD) is a measure of the number of mutants that are generated for a location. This evaluation compares the location coverage and mutation density achieved by MuAspectJ against that achieved by a benchmark set by MuJava. These measures are derived from analysing the mutants generated by applying each tool to a well known case study implemented as an equivalent AspectJ and Java programs. The goal of the evaluation is to validate that the same level of confidence can be associated with results of mutation analysis based on using mutants generated from either tool.

Measure	Mean	SD	Sum
Java (modules = 61)			
location	16.6	23.5	1014
mutation	71.6	119.8	4367
LOC	90.9	101.3	5543
AspectJ (modules = 65)			
location	15.1	22.4	979
mutation	67.9	114.7	4411
LOC	76.7	84.4	4984

Table 5: Descriptive Statistics

4.1 Comparison

In this evaluation the LC and MD achieved by both tools are compared on a per-module basis. This is feasible because both implementations contain may classes of set the same name. This allows a direct comparison between these classes in both implementations. This approach also serves to highlight those modules that are not covered in both implementations. As there is no way that these modules can be directly compared in the same way, LC and MD are compared at the overall program level. This two pronged approach allows both a detailed and overall perspective of LC and MD.

Table 5 presents descriptive statistics that characterise the data from which the LC and MD are derived. The table characterises two data sets, one for the MuJava and MuAspectJ tools. In each tools data set the number of locations, number of mutations and LOC are counted for each module covered by the tool.

4.2 Location Coverage

LC is a measure of the proportion of locations for which mutants are generated for each module. The total number of locations for each module is indicated by the Lines Of Code (LOC) per module. LC of a module is measured as a proportion of the number of locations mutated over the LOC of the module. Figure 4, visualises the comparison of LC between tools. There are two smoothed lines representing the LC over modules, one for each tool. These lines are surrounded by a shaded band representing a confidence interval for each smoothed line. From this figure we can assert that there is no significant difference between directly comparable classes (12-64). This is because each line is within the confidence interval of the other, indicating no significant difference. This figure also shows the LC of classes (1-11) that are covered only in the Java implementation and the LC of aspects (65-75) that are covered in the AspectJ implementation only. From visual inspection it seems that overall the LC of each program is very similar but may be slightly lower for the AspectJ implementation. This is confirmed by noticing that the total number of locations covered by the Java (1014) and AspectJ (979) implementations are very similar. This is further confirmed when the mean LC of MuJava (.1995) is found to be slightly, but not significantly, lower than MuAspectJ (.1946).

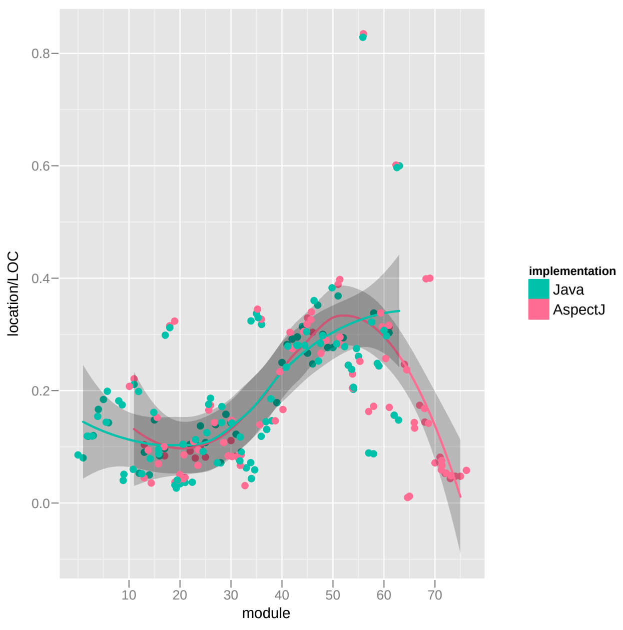

Figure 4: Location Coverage

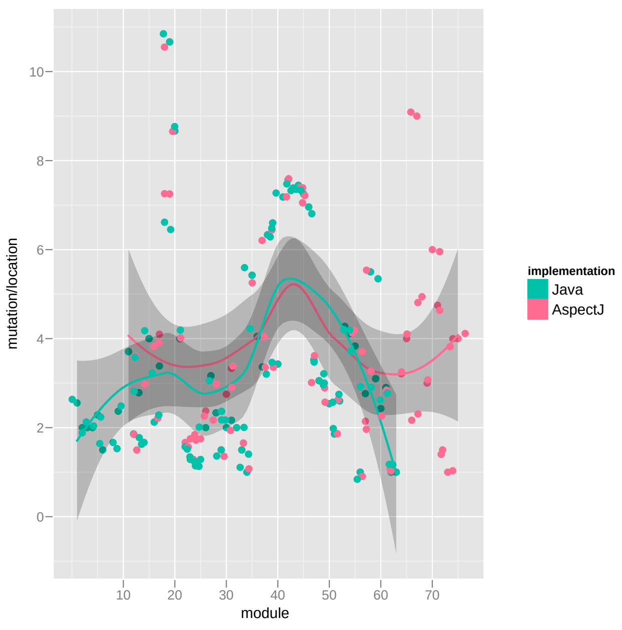

Figure 5: Mutation Density

The lower LC for aspect modules is explained by the low number of aspect locations. There are a small number of aspect locations for which mutants are generated. The avergae number of aspect locations in these modules is approximately 4 per aspect. The number of java locations in aspects is low. In Section 3.2.2, the cause of this is the manner in which aspects are written. In general aspects specify crosscutting and then delegate behaviour to supporting classes. As the analysis shows, the impact of aspects is a reduction of LC that is insignificant when subsumed into the overall measure of LC for MuAspectJ.

4.3 Mutation Density

MD is a measure of the number of mutants that are generated for a location. It is measured as a proportion of the number of mutations over the number of locations per module. Figure 5, visualises the comparison of MD. From this figure we assert that there are no significant differences between directly comparable classes (12-64). From visual inspection it seems that overall the MD of each program is almost identical for both implementations. The LC associated with the non-overlapping classes (1-11) and aspects (65-75) covered only by one implementation seem to balance one another. The fact that there are more mutations generated for the AspectJ (5543) provides some evidence back-up this observation. To provide a firm confirmation, the mean MD of the Java (3.45) and AspectJ (3.77) implementations are compared. The result (p-value 0.2815) indicates that although the mean MD achieved by MuAspectJ is slightly higher than MuJava, that this difference is not at all significant.

As identified in 4.2, the number of locations in aspect modules is low. As can be seen in Figure 3, there are a number of mutation operators that generate mutants for this small number of locations. This resuts in a relatively large number of mutants per-aspect location which equates to a relatively high MD for aspect locations. The MD when subsumed into the overall figure does increase the overall MD per-module for MuAspectJ but not significantly.

Mu-Tool	Operator	N	Mean	Sum
Java	Java	26	79.42	2065
AspectJ	Java	25	80.4	2010
AspectJ	AspectJ	13	5.39	72
AspectJ	Both	38	54.74	2082

Table 6: Location Coverage

5. DISCUSSION

In this section, the results of evaluation are discussed. In 3.4 the results of applying MuJava and MuAspectJ are explained by examining the contribution of mutation operators they support to the overall number of mutants generated by each tool. In this section this strategy is repeated to explain how the measures of LC and MD presented in Sections 4.2 and 4.3 were arrived at.

5.1 Location Coverage

Table 6 provides mean and related information to shows how mutation operators contribute to the LC in both implementations. The first row of this table presents the mean LC of the Java operators when applied to the Java implementation by MuJava. The remaining three rows show the mean LC for the Java operators, AspectJ operators and their combination when applied to the AspectJ implementation by MuAspectJ.

Both MuJava and MuAspectJ support Java mutation operators which can be directly compared. The mean LC of Java operators is slightly, but not significantly (p-value 0.9025) higher for MuAspectJ. MuAspectJ also supports As-

Mu-Tool	Mods	Locat	N	Oper	Mutant
Java	class	Java	1014	2.04	4.3
AspectJ	class	Java	935	2.09	4.5
AspectJ	aspect	Java	27	2.07	3.1
AspectJ	aspect	aspect	18	3.89	6
AspectJ	both	both	980	2.12	4.5

Table 7: Operators & Mutants - Per Location

pectJ operators that contribute to the overall LC for the AspectJ implementations to which MuAspectJ is applied. The mean LC of the AspectJ operators is low, relative to the LC of the Java operators. When the LC of both AspectJ and Java operators is combined the low LC of the AspectJ operator serves to reduce the overall mean for MuAspectJ. The result is that the overall LC per-operator is barely significantly lower for the AspectJ implementation (p-value 0.1007). This finding matches our finding in Section 4.2 that the LC per-module is lower for the MuAspectJ tool. Based on these findings we can infer a causal relationship between these results. The the lower LC per-module observed for the AspectJ implementation in Section 4.2 is caused by the lower LC per-operator.

The lower LC per-operator is due to the smaller number of aspect locations that can be covered by AspectJ mutation operators. There are a very small number of aspect locations when compared to the number of java locations in the AspectJ implementation of the case study. This reduces the LC per-operator which in turn reduces the LC per-module.

5.2 Mutation Density

Mutation Operators increase mutation density by applying more operators locations to generate more mutants per-location. Table 7 presents the mean number of operators (Oper) and mean number of mutants generated (Mutant) per-location for the MuJava and MuAspectJ tools. To help isolate how each tool contributes to mutation density, each row of the table shows how many mutants are created for each type of location.

The MuJava tool generates mutations for Java locations in classes. The MuAspectJ tool generates for Java locations in classes and aspects as well as aspect locations in aspect. The first row presents the results of the MuJava tool. The second, third and fourth present results for the Java and AspectJ locations treated by the MuAspectJ tool. The fifth and final row presents the overall results for MuAspectJ tool.

The table shows that the locations identified by the MuAspectJ tool have a higher number of operators applied to them. The Java locations in classes and aspects are only very slightly higher but it is obvious that the mean number of operators covered for AspectJ locations is significantly higher (p-value 4.987e-06). Due to the small number of AspectJ locations this difference is scaled down in the overall result. The overall mean number of operators covered by MuAspectJ is not significantly higher than MuJava. The same pattern is observed for the mean number of mutants generated per-locations. The number of mutants generated by MuAspectJ for AspectJ locations is significantly higher than that generated by MuJava for the Java locations. Again this difference is reduced when these numbers are subsumed into the overall result, which indicates MuAspectJ produces more but not significantly more mutants per-location than MuJava.

These results provide an explanation of the finding in Section 4.3 that MuAspectJ achieved higher Mutation Density per module. We can conclude from the information presented above that this is the result of more operators being applied to locations in AspectJ programs resulting in more mutations per-location, increasing mutation density.

5.3 Case Study Validity

This section describes how we addressed the validity threats posed to our evaluation. A biased comparison threatens the internal validity of our study. To counter bias toward either tool we have chosen to apply the tools under evaluation to an existing case study, introduced in Section 2.3, widely used in comparative research evaluations [14, 17, 8]. The fact that we are drawing conclusions from one case study threatens the external validity of the results of our case evaluation. Although the case study presented is a very realistic example of how AspectJ programs are developed, it is difficult to generalise from the results of the evaluation.

6. RELATED WORK

The work introduces a tool for the generation of mutants for AspectJ programs. The mutants generated by this tool can be used in mutation analysis to measure the testability of AspectJ programs. In this section other tools that may be used to support measuring testability are described and related to this tool. It also introduces a new means for evaluating this tool. The tool is evaluated in terms of the quality of mutants it generates. This differs from the evaluation of MuJava, the mutant generation tool most related to MuJava. In this section we describe how MuJava is evaluated and justify the evaluation undertaken in this paper.

6.1 Tooling

MuAspectJ provides support for generating mutants based on which mutation analysis employed to quantify the testability of AspectJ programs. There are existing tools such as AJATO and Jinghu, that support the collection of static metrics from aspect-oriented programs. AJATO and Jinghu support the collection of static metrics from aspect-oriented programs. AJATO supports a varied suite of metrics including traditional, object-oriented and concern separation metrics [13]. Jinghu [29]supports the collection of coupling metrics. As noted in Section 2.1, some of the static metrics that these tools collect could act as coarse grain indicators of testability [5, 15, 26, 29, 25]. They could be used to gather metrics to provide a low accuracy - low cost assessment of testability. Cost in this instance is the computational and interpretation time involved. The cost is lower using static methods because metric can gathered and interpreted quickly. MuAspectJ complements these tools by providing support for high accuracy measurement of testability. MuAspectJ supports mutation analysis which is computationally expensive. All mutants must be run for a number of tests to get results. When results are complete there is a large volume of measurements that is time consuming to interpret.

MuAspectJ generates mutants for AspectJ programs. There are existing tools that also generate mutants for AspectJ. Anbalagan and Xie [4, 3] provide tooling to automatically generate non-equivalent mutant pointcuts by the insertion or removal wild cards in pointcuts. Their framework does not support a full set of mutation operators of MuAspectJ. MuAspectJ provides a full range of mutation operators that can be applied to pointcut, advice and declaration as well as Java locations. MuAspectJ is an extension of this work as it reuses the pointcut generation strategy introduced by Anbalagan and Xie [4].

6.2 Evaluation

MuAspectJ is evaluated in terms of the quality of mutants it generates. MuJava, which is very related to this MuAspectJ, is evaluated in terms of tool performance [19] rather than mutant quality. The tool is evaluated in terms of how fast mutants can be generated and executed. This does provide some sense of the length of time that it will take to get to a result but does not provide any indication of the quality of generated mutants. This type of evaluation does not however provide any sense of how the mutants will impact on the assertions that can be made from analysing the results. Although speed of generation and execution are practical issues that must be considered when performing MA, then can be easily addressed through parallel execution of mutants in a distributed mutant execution approach. The ability to measure and compare the confidence that can be associated with assertions reached by using a mutant generation tool is a more goal focused measure. The performance measure will tell the user how long it will take to get a result. The measure of the quality of mutants generated by a tool bounds how good the results from using the tool can be. The type of evaluation introduced here does not replace the performance based evaluation presented in [19]. It expands the numbers of factors that can be used to evaluate mutant generation tools and through expansion enables a broader understanding of the quality of generated mutants.

7. CONCLUSIONS

The results of this comparison show that the location coverage and mutation density of MuAspectJ are equivalent to MuJava. This means that an equivalent level of confidence can be associated with results of mutation analysis based on using mutants generated from either tool. The results are a positive validation that MuAspectJ achieves the mutant generation quality benchmark set by MuJava.

This makes two contributions. The primary contribution is the provision of the MuAspectJ that can be used to generate mutants for AspectJ programs. A secondary contribution is the introduction of location coverage and mutation density as a means to measure the quality of generated mutants.

The next step in this work is to use the mutants generated by MuAspectJ in experimentation. To quantify the impact of AOSD on testability experiments that apply mutation analysis to equivalent Java and AspectJ implementations are planned. In these experiments the mutants generated by MuAspectJ and MuJava will be used. In this work we have shown that the mutants generated by both tools are of equivalent quality indicating that they can be used in comparative experiments.

8. REFERENCES

[1] Hiralal Agrawal, Richard A. Demillo, Bob Hathaway, William Hsu, Wynne Hsu, E. W. Krauser, R. J. Martin, Aditya P. Mathur, and Eugene Spafford. Design of mutant operators for the c programming language. Technical report, 1989.

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

[3] Prasanth Anbalagan and Tao Xie. Apte: automated pointcut testing for aspectj programs. In *proceedings of the 2nd workshop on Testing aspect-oriented programs, International Symposium on Software Testing and Analysis*, pages 27–32, New York, NY, USA, 2006. ACM Press.

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

[5] Benoit Baudry and Yves Le Traon. Measuring design testability of a uml class diagram. *Information and Software Technology*, 47(13), October 2005.

[6] M. Bruntink and A. van. Predicting class testability using object-oriented metrics, 2004.

[7] Jon BŸkkeny and Roger Alexander. A candidate fault model for aspectj pointcuts. In *ISSRE '06: Proceedings of the 17th International Symposium on Software Reliability Engineering*, pages 169–178, Washington, DC, USA, 2006. IEEE Computer Society.

[8] Sant'Anna C., Figueiredo E., Garcia A., and Lucena C. On the modularity assessment of software architectures: Do my architectural concerns count? In *Proceedings of the 1st Workshop on Aspects in Architectural Description (AARCH), at the International Conference on Aspect-Oriented Software Development (AOSD)*, Vancouver (Canada), March 2007.

[9] Mariano Ceccato, Paolo Tonella, and Filippo Ricca. Is aop code easier or harder to test than oop code?, 2005.

[10] Adrian Colyer, Andy Clement, George Harley, and Matthew Webster. *Eclipse aspectj: aspect-oriented programming with aspectj and the eclipse aspectj development tools.* Addison-Wesley Professional, 2004.

[11] Marc Eaddy, Alfred V. Aho, Weiping Hu, Paddy McDonald, and Julian Burger. Debugging aspect-enabled programs. In Markus Lumpe and Wim Vanderperren, editors, *Software Composition*, volume 4829 of *Lecture Notes in Computer Science*, pages 200–215. Springer, 2007.

[12] 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, Verification, and Validation*, pages 52–61, Washington, DC, USA, 2008. IEEE Computer Society.

[13] E. Figueiredo, C. Sant'Anna, A. Garcia, T.T. Bartolomei, W. Cazzola, and A. Marchetto. On the maintainability of aspect-oriented software: A concern-oriented measurement framework. *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*, pages 183–192, April 2008.

[14] Phil Greenwood, Thiago Bartolomei, Eduardo Figueiredo, Marcos Dósea, Alessandro Garcia, Nélio Cacho, Cláudio Sant'Anna, Sérgio Soares, Paulo

Borba, Uirá Kulesza, and Awais Rashid. On the impact of aspectual decompositions on design stability: An empirical study. In *proceedings of the 21st European Conference on Object-Oriented Programming*, pages 176–200, 2007.

[15] Stefan Jungmayr. Identifying test-critical dependencies. In *proceedings of the 18th IEEE International Conference on Software Maintenance*, page 404, November 2002.

[16] T.M. Khoshgoftaar, R.M. Szabo, and J.M. Voas. Detecting program modules with low testability. *icsm*, 00:242, 1995.

[17] 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*, pages 223–233, Washington, DC, USA, 2006. IEEE Computer Society.

[18] Otávio Augusto Lazzarini Lemos, Fabiano Cutigi Ferrari, Paulo Cesar Masiero, and Cristina Videira Lopes. Testing aspect-oriented programming pointcut descriptors. In *WTAOP '06: Proceedings of the 2nd workshop on Testing aspect-oriented programs*, pages 33–38, New York, NY, USA, 2006. ACM.

[19] Yu S. Ma, Jeff Offutt, and Yong R. Kwon. Mujava: an automated class mutation system. *Softw. Test. Verif. Reliab.*, 15(2):97–133, 2005.

[20] Johannes Mayer and Christoph Schneckenburger. An empirical analysis and comparison of random testing techniques. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 105–114, 2006.

[21] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland Untch, and Christian Zapf. An experimental determination of sufficient mutation operators. *ACM Transactions on Software Engineering and Methodology*, 5:99–118, 1996.

[22] A. Jefferson Offutt, Jeff Voas, and Jeff Payne. Mutation operators for ada. Technical report, 1996.

[23] B.H. Smith and L. Williams. An empirical evaluation of the mujava mutation operators. *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. TAICPART-MUTATION 2007*, pages 193–202, Sept. 2007.

[24] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. Technical Report Planning Report 02–3, National Institute of Standards and Technology, Program Office Strategic Planning and Economic Analysis Group, May 2002.

[25] Jeffrey Voas, Larry Morrel, and Keith Miller. Predicting where faults can hide from testing. *IEEE Softw.*, 8(2):41–48, 1991.

[26] Jeffrey M. Voas, Keith W. Miller, and Jeffery E. Payne. A comparison of a dynamic software testability metric to static cyclomatic complexity. In *in Second International Conference on Software Quality Management*, pages 431–445. Publications, 1994.

[27] Tao Xie, D. Notkin, and D. Marinov. Rostra: a framework for detecting redundant object-oriented unit tests. *Automated Software Engineering, 2004. Proceedings. 19th International Conference on*, pages 196–205, Sept. 2004.

[28] Sai Zhang and Jianjun Zhao. On identifying bug patterns in aspect-oriented programs. *Computer Software and Applications Conference, Annual International*, 1:431–438, 2007.

[29] Jianjun Zhao. Measuring coupling in aspect-oriented systems. In *Information Processing Society of Japan (IPSJ*, pages 14–16, 2004.