Towards the Establishment of a Sufficient Set of Mutation Operators for AspectJ Programs

Jésus Thiago Sousa Lacerda[1], **Fabiano Cutigi Ferrari**[1]

[1]Computing Department
Federal University of São Carlos (UFSCar) – São Carlos – SP – Brazil

{jesus.lacerda, fabiano}@dc.ufscar.br

Abstract. *Mutation testing is an effective test selection criterion that has been explored in the context of aspect-oriented (AO) programs written in the AspectJ language. Despite its effectiveness, mutation testing is expensive. To reduce its application cost, some strategies based on mutation operator selection are available in the literature. This paper describes the results of a study that aimed to reduce the costs of applying mutation testing to AspectJ programs by identifying a reduced set of mutation operators known as sufficient operators. To achieve the proposed objective, we applied the Sufficient Procedure, which has resulted in expressive cost reductions when applied to procedural programs. Indeed, the procedure led to a small set of operators and to a cost reduction of 53% in terms of number of mutants to be handled.*

1. Introduction

Aspect-oriented programming (AOP) is an approach to improve the modularisation of software in the face of the difficulties faced by existing techniques, including object-oriented programming (OOP). These difficulties refer to the behaviour of software that is usually scattered in different modules of the system or the other intertwined behaviour. Such behaviour, which may be related to functional or non-functional properties of the software, constitute the so-called crosscutting concerns [11]. Although AOP represents a possible solution to deal with crosscutting concerns, the adoption of this technique (or paradigm) may introduce new types of faults. Thus, it is necessary design specialised tests to ensure the quality of software that uses this paradigm.

From a variety of testing criteria that can be applied for the validation of AO programs, mutation testing [4] has shown to be a promising alternative [3, 7, 9, 15, 17]. In spite of the high effectiveness of mutation testing, this criterion has been historically characterised as very expensive due to large number of the mutants generated, given that manual tasks are required to analyse the mutants and improve the test set. Therefore, establishing strategies for cost reduction of mutation testing of AO programs becomes essential for the sake of feasibility of the criterion.

Amongst the techniques for reducing the costs of mutation testing, some focus on reducing the number of generated mutants based on the characteristics of mutation operators. Examples are constrained mutation [13], selective mutation [14] and sufficient operators [2]. From these techniques, sufficient operators has resulted in the highest cost reduction rates when applied to procedural programs.

To date, few studies explored the cost reduction of mutation testing for AO programs, mainly focusing on reducing the number of equivalent mutants [15, 17]. In this context, the objective of this study is to identify a sufficient set of mutation operators which is able to reduce substantially the cost of mutation testing when compared to the full set of operators.

To accomplish the proposed objective, we apply the Sufficient Procedure [2] to a set of 12 small-sized AO programs written in the AspectJ language. The Sufficient Procedure aims to identify a set of sufficient mutation operators that keeps the effectiveness of the

test suite with respect the whole set of mutation operators. Note that AspectJ represents the mainstream AOP technology, upon which research and practical usage have developed. We choose the Sufficient Procedure since it has yielded expressive results for procedural programs. Indeed, our results show that a reduced set of mutation operators for AspectJ programs may produce a high mutant coverage with a cost reduction of 53%.

The remaining of this paper is organised as follows: Sections 2 and 3 describe basic concepts and related work, respectively. The plan of the exploratory study to identify sufficient operators is described in Section 4, whereas Section 5 details the achieved results and limitations. Conclusions and future work come in Section 6.

2. Background

2.1. Aspect-Oriented Programming

The goal of aspect-oriented programming (AOP) is improving the modularisation of the so-called crosscutting concerns [11]. In doing so, such concerns will not appear tangled with other concerns or spread across various parts of the code. Adopting AOP, the developer is able to separate and better organise the code and thus improve software quality through the concept of separation of concerns [5, 11]. In AOP, the element responsible to make the modularisation of these concerns is the aspect. Ideally, an aspect can change the behaviour of the remaining system functionalities without their knowledge of the aspect action. In one aspect, behaviour related to a crosscutting concern is termed advice, and it can be executed at a specified time. The moment of the execution of an advice is determined by a join point and can be defined in predicate-based expressions known as pointcuts. Mainstream AOP technologies such as AspectJ[1], SpringAOP[2] and CaesarJ[3] provide mechanisms for defining pointcut expressions to identify joint points throughout the software code.

Despite the benefits of AOP brings, using its programming constructs may result in new types of faults in the code [1, 6]. Therefore, to ensure the quality of produced software, several testing approaches have been explored, involving different techniques and testing criteria. Among these approaches, mutation testing has attracted the research community's attention and has been on focus in recent research [7, 9, 15, 17].

2.2. Mutation Testing for AO Programs

The Mutant Analysis criterion – also known as mutation testing – was first described by DeMillo et al. [4]. This criterion inserts small defects in software with the aim of simulating the common mistakes made by programmers. Mutation testing consists in generating several versions of the program P under testing with small syntactic changes (P', P'', ...); these modified versions are called mutants. Tests are executed on P and on (P', P'', ...). Given a mutant, say P', if the result of at least one test case differs between P and P', P' is said to be killed (*i.e.* the simulated fault was revealed). Otherwise, the mutant remains alive so P' must be analysed to check for equivalence with P or to require a new test case that reveals the difference between P and P'. The mutation score (MS) measures the test coverage based on the ratio of killed mutants with respect to the total number of non-equivalent mutants.

For AO programs, Ferrari et al. [7] defined a set of mutation operators considering the AspectJ language. The operators are based on fault types specific of the AO paradigm. The operators are organised in three groups; each group simulates three categories of faults related to main AOP concepts and constructs of AspectJ, namely: pointcut expressions

[1] http://www.eclipse.org/aspectj/ – accessed in 27/06/2014

[2] http://docs.spring.io/spring/docs/2.5.4/reference/aop.html – accessed in 27/06/2014

[3] http://caesarj.org/ – accessed in 27/06/2014

(Group 1); declare-like expressions (Group 2); and advice definition and implementation (Group 3). In total, 26 operators are defined (15 in Group 1, 5 in Group 2, and 6 in Group 3), and the *Proteum/AJ* tool [8] automates 24 of them.

2.3. The Sufficient Procedure

Many studies have investigated different approaches for mutation testing in order to reduce costs of such criterion. A survey reported by Jia and Harman [10] describes the main contributions in this field. Examples of cost reduction strategies are selective mutation, higher order mutation, weak mutation, and sufficient mutation. Amongst them, sufficient mutation was proposed by Barbosa et al. [2] and, by applying a procedure called Sufficient, has shown the highest reduction costs without effectiveness loses.

The Sufficient Procedure consists of six steps that aim to select the best mutation operators to compose an essential (or sufficient) set. The steps are described as follows:

1. **Consider operators that determine high score mutation:** select operators which produces the highest mutation scores in relation with the whole set of operators.
2. **Consider at least one operator for each class of mutation:** since each class of operators models specific types of faults, it is desirable that the set of operators contains at least one operator for each class.
3. **Evaluate empirical inclusion among operators:** check if operators in the sufficient set are empirically included by other operators; if so, remove them from the set.
4. **Establish incremental implementation strategy:** due to the high cost of operators, it is necessary to establish an incremental strategy for their application.
5. **Consider operators that provide an increase in the mutation score:** seek for operators not yet selected, but have the ability to increase the mutation score.
6. **Consider operators with high strength:** consider operators which have high strength, since they are usually not empirically included by other operators.

3. Related Work

Barbosa et al. [2] applied the Sufficient Procedure (described in Section 2) to procedural programs developed in the C language. The procedure was applied to two different sets of applications. In both sets the authors achieved expressive cost reduction of around 65%, on average. Motivated by such results, we apply the Sufficient Procedure in the study presented in this paper. The results are compared with Barbosa et al.'s results in Section 5.2.

In more recent studies focus on different approaches for testing AO programs using mutation [15, 17]. Wedyan and Ghosh proposed the use of simple object-based analysis to prevent the generation of equivalent mutants for some mutation operators for AspectJ programs. They argue that reducing the amount of equivalent mutants generated by some operators would consequently reduce the cost of mutation testing as a whole. The authors use three testing tools (AjMutator, Proteum/AJ, MuJava) to assess their technique. The results are that, on average, 80% of the equivalent mutants could be avoided of being generated, proving the efficiency of their approach. Note that Wedyan and Ghosh's approach consists in changing the mutation rules encapsulated into the operators with the aim of not generating equivalent mutants. Such strategy requires tool adaptive maintenance. Once implemented, it could be applied in conjunction with the Sufficient Procedure to reduce even more the cost of mutation testing applied to AO programs.

Omar and Ghosh [15] presented four approaches to generate higher order mutants for AspectJ programs. The approaches were evaluated in terms of the ability to create mutants of higher order resulting in higher efficacy and less effort when compared with first order

mutants. Some of these approaches can reduce the amount of equivalent mutants generated or reduce the total number of mutants, *i.e.* they can reduce the cost of mutation testing in this paradigm. Furthermore, all approaches proposed can produce higher order mutants that can be used to increase test effectiveness and reduce test effort.

Ferrari et al. [9] evaluated the same set of applications we analyse in this study. They apply a whole set of mutation operators proposed in their previous work [7]. Two analyses are presented: (i) they compared the cost of applying mutation with the Systematic Functional Testing (SFT) criterion, which has shown to be very effective in detecting mutants for C programs [12]; and (ii) they measured the effort required to achieve mutation adequate test sets. In short, the results are that the application of SFT was not enough to achieve adequate mutation coverage; on average, 5% of additional tests was required.

4. Study Setup

4.1. Research Question

This study aims to investigate how to reduce the cost of applying mutation testing in AO programs. We define two research questions, the first related to the feasibility of achieving significant cost reduction, and the second related to the amount of effort we can save.

- RQ1: Can one identify a reduced set of mutation operators which are effective to reveal AO-specific faults simulated by a complete set of operators?
- RQ2: Can one achieve an expressive cost reduction in mutation testing for AO programs by applying the reduced set of operators?

We selected the Sufficient Procedure proposed by Barbosa et al. [2] as the approach to be applied to AO programs in order to obtain a reduced set of operators for AspectJ programs. The key measure considered by the Sufficient Procedure is the mutation score associated with an individual mutation operator or to a set of mutation operators. Since such procedure has resulted in the highest effort savings when applied to procedural programs when compared to other operator-based cost reduction approaches [2], we believe this procedure should also provide significant results for AO programs as well.

4.2. The Dataset

In our study we reused mutation-adequate test sets originally designed by Ferrari et al. [9] for 12 small AO software applications developed in the AspectJ language. A short description and some coarse-grained metrics of these applications are shown in Table 1. We highlight that the design of test sets as well as the mutant analysis were thoroughly performed by those authors with support of the *Proteum/AJ* tool [8].

General test data for each application is shown in Table 2. Column "#Total Mut." shows the total number of mutants per application. The remaining columns represent, respectively: the number of mutants automatically classified as equivalent ("#Autom.Equiv") and anomalous[4] ("#Anom."); the number of mutants that should be killed by tests or manually classified as equivalent ("#Alive Mut."); the number of mutants manually classified as equivalent ("#Man.Equiv."); the size of the mutation-adequate test set ("$|TM|$"); and the final mutation score ("M.S").

4.3. Dataset Preparation

Based on the results generated in the study of Ferrari et al. [9], we implemented and executed a series of SQL queries and Java routines to obtain a cross-comparison table amongst

[4] A mutant is classified as anomalous if it does not compile.

Table 1: Description of the applications used in the study (adapted from Ferrari et al. [9])

Application	Description	LOC	Classes	Aspects
1. BankingSystem	System that manages transactions for bank accounts. Aspects in BankingSystem implement logging, minimum balance control and overdraft operations.	199	9	6
2. Telecom	Phone system simulator developed in AspectJ and distributed by The Eclipse Foundation. Manage phone calls, and calculations of time and cost of each call are made for aspects	251	6	3
3. ProdLine	Software product line for graph applications that includes a set of common functionalities of the graph domain. The aspects in this application are responsible for introducing the selected features in a determined instance SPL.	537	8	8
4. FactorialOptimizer	Mathematical system that implements an optimization on the factorial of a number. The calculation is managed by an aspect, it stores all cached calculations to eventually be recovered and reused in other calculations.	39	1	1
5. MusicOnline	Management system from an online music store. The aspects are responsible for managing user accounts and billing system.	150	7	2
6. VendingMachine	System that manages the interaction of sales in a drinks machine. The aspects managed manage the sale, returning the user coins when needed and releasing the drink requested.	64	1	3
7. PointBoundsChecker	Two-dimension point constraint checker. The aspect checks if the coordinates of the points belong to a given interval, otherwise an exception is thrown	44	1	1
8. StackManager	It's a simple stack that improves the operations of push and pop. The aspects perform audit of the stored numbers and count the number of operations of push type.	77	4	3
9. PointShadowManager	Application for managing two-dimension point coordinates. One aspect create and manage the shadow of two points, the shadows most have exactly same coordinates of the points.	66	2	1
10. Math	Math utility application that calculates the probability of successes in a sequence of n independent yes/no experiments (Bernoulli trial), each yielding success with probability p. The aspect logs exponentiation operations, identifying the type of the exponent.	53	1	1
11. AuthSystem	Simpler version of a bank system, the system requires the user to authenticate before simple operations such as debit, credit and check balance. The aspects are responsible for authentications and transaction management.	89	3	2
12. SeqGen	Implements a generator from a sequence of numbers and characters. The aspects modularize the policy generation and logging concerns	205	8	4

Table 2: Summary of the application of mutation testing on the set of programs

Application	#Total Mut.	#Autom.Equiv.	#Anom.	#Alive Mut.	# Man.Equiv.	$\|TM\|$	M.S.
1. BankingSystem	136	68	18	50	-	58	1.00
2. Telecom	111	46	12	53	10	67	1.00
3. ProdLine	199	125	16	58	-	36	1.00
4. FactorialOptimizer	29	8	6	15	1	19	1.00
5. MusicOnline	57	25	5	27	2	48	1.00
6. VendingMachine	113	58	8	47	13	23	1.00
7. PointBoundsChecker	70	32	10	28	-	14	1.00
8. StackManager	45	24	0	21	-	15	1.00
9. PointShadowManager	50	25	4	21	5	14	1.00
10. Math	20	13	0	7	2	54	1.00
11. AuthSystem	52	28	3	21	1	19	1.00
12. SeqGen	40	19	3	18	8	42	1.00
Total	922	471	85	366	42	409	1.00

operators. For each mutation operator, we identified the respective adequate subset of test cases and checked the coverage of mutants generated by all other operators. Part of the comparison table appears in Table 3.

Similarly to Barbosa et al.'s study [2], using such a table we can check the average mutation score obtained by each operator's adequate test set with respect to all other operators. For instance, the subset of adequate test cases for operator ABHA (line 2) is almost considered adequate for operator ABAR (column 1), achieving a mutation score of 0.9960. In this case, it is said that the ABAR operator is included empirically by the ABHA operator.

Table 3: Part of the cross between mutation operators

Op/Op	ABAR	ABHA	ABPR	APER	APSR	DAPC	DAPO	PCCE	...	Average
ABAR	**1**	0.9352	0.9431	1	0.8963	1	0.9853	0.8595	...	0.9246
ABHA	0.9960	**1**	0.9431	1	0.9931	1	0.9853	0.9372	...	0.9598
ABPR	0.6169	0.6508	**1**	1	0.7234	1	0.9853	0.5275	...	0.7730
APER	0.1422	0.2964	0.5583	**1**	0.3872	0.6107	0.5317	0.1946	...	0.3359
APSR	0.5230	0.6361	0.7967	1	**1**	1	0.9853	0.5042	...	0.7196
DAPC	0.4410	0.5008	0.7154	1	0.5106	**1**	0.9853	0.4019	...	0.6204
DAPO	0.1327	0.2360	0.3631	0.4820	0.2851	0.4497	**1**	0.1603	...	0.2744
PCCE	0.8708	0.8913	0.8238	1	0.6935	1	0.9853	**1**	...	0.9005
Average	0.5392	0.5518	0.5938	0.6634	0.5178	0.7363	0.7010	0.4761	...	-

With this comparison table, we can perform some preliminary analysis required to run the Sufficient Procedure. For example, we can rank the operator with the highest mutation score (MS), highest strength and highest cost. This data can be observed in Table 4. Note that strength is obtained with the formula (1 - operator's average mutation score with respect to all other mutation operators). For example, the strength of ABHA operator is 0.4482 (*i.e.* 1 - 0.5518; see ABHA column in Table 3).

Table 4: Mutation score, strength and cost of operators.

MS		Strength		Cost	
Operator	Value	Operator	Value	Operator	Value
ABHA	0.9598	PWIW	0.5564	PWIW	456
PWIW	0.9534	PCCE	0.5239	ABAR	75
ABAR	0.9246	PCTT	0.5082	ABPR	62
PCCE	0.9005	POAC	0.5048	PCCE	57
POAC	0.8728	APSR	0.4822	POPL	54
ABPR	0.7330	PCGS	0.4706	ABHA	52
APSR	0.7196	ABAR	0.4608	POAC	50
DAPC	0.6204	ABHA	0.4482	PCCT	31
POPL	0.5718	POPL	0.4147	PCLO	30
PSWR	0.5635	ABPR	0.4062	APSR	18
...	...	...	...	...	...

5. Results and Analysis

In this section the Sufficient Procedure is applied step-by-step ad the results are described. At the end of each step the partial result of the sufficient set is presented. After describing all partial results, we present the final sufficient set and an analysis of the cost reduction achieved with the procedure.

5.1. The Sufficient Procedure Step-by-Step

As described in Section 2.3, the Sufficient Procedure requires six steps to result in a sufficient set of mutation operators for programs written in a particular programming language. Following we describe the execution of each step and the partial results we achieved.

Step 1: Consider mutant operators that determine a high mutation score

To apply this step, we defined the value of $0.75^{+}_{-}0.02$ for mutation score (MS) as a minimum threshold to select operators to compose the SS_{pre} set, which is the preliminary sufficient mutant operators sets. That is, only operators that produce MS equal or higher than the threshold are selected to compose SS_{pre}.

The minimum threshold for inclusion was defined based on the study of Barbosa et al. [2]. Although those authors have chosen a higher threshold (MS = $0.9^{+}_{-}0.005$), we decided for a lower value in order to work with a similar SS_{pre} size as they did. More

specifically, our threshold enabled us to create the SS_{pre} set with six operators, as follows:

SS_{pre} = {ABHA, PWIW, ABAR, PCCE, POAC, ABPR}

Step 2: Consider at least one operator for each class of mutation

As the reader can notice, SS_{pre} lacks operators of Group 2, which are operators that model faults related to declare-like expressions in AspectJ. From the five available operators (namely, DAPC, DAPO, DSSR, DEWC, DAIC), only DAPC and DAPO generated mutants for the target applications. However, running mutants of DAPC and DAPO on SS_{pre}-adequate test cases results in MS = 1.000 and MS = 0.9852 in regard to those mutants, respectively. For this analysis, MS $\geq$ 0.95 is defined as an optimal index of mutation score to consider a mutation operator empirically included by the others, likewise in Barbosa et al.'s study [2]. We then conclude DAPC and DAPO are empirically included by SS_{pre} and that at this point, SS_{pre} remains unchanged.

Step 3: Evaluate empirical inclusion among operators

From the SS_{pre} set obtained so far, we can analyse the empirical inclusion amongst its elements and possibly reduce the preliminary set. Table 5 supports the analysis, in which operators one-by-one are contrasted with the remaining operators from SS_{pre}. Similarly to the previous step, MS $\geq$ 0.95 is considered an optimal index for empirical inclusion.

Table 5: Empirical inclusion considering SS_{pre} operators one-by-one.

Relation Empirical Inclusion	Highlight Op.	Mutation Score
{'ABHA','PWIW','ABAR','PCCE','POAC'} → {'ABPR'}	ABPR	0.943089
{'ABPR','PWIW','ABAR','PCCE','POAC'} → { 'ABHA'}	**ABHA**	**0.999489**
{'ABPR','ABHA','ABAR','PCCE','POAC'} → { 'PWIW'}	PWIW	0.742682
{'ABPR','ABHA','PWIW','PCCE','POAC'} → { 'ABAR'}	**ABAR**	**0.996038**
{'ABPR','ABHA','PWIW','ABAR','POAC'} → { 'PCCE'}	PCCE	0.937153
{'ABPR','ABHA','PWIW','ABAR','PCCE'} → { 'POAC'}	POAC	0.921406

Operators to be considered for removal from SS_{pre} are in ABHA and ABAR. Since this step is performed incrementally, the first operator to be removed will be one that is more empirically included by the other operators, in this case, ABHA, since it is the operator with the highest score listed in Table 5 (MS$_{ABHA}$ = 0.999489).

After removing the ABHA operator, SS_{pre} was re-evaluated and we verified that the values of the empirical inclusion remained the same. The values of this new revaluation of inclusion relations are shown in Table 6.

Table 6: Empirical inclusion after removing ABHA.

Relation Empirical Inclusion	Highlight Op.	Mutation Score
{'PWIW','ABAR','PCCE','POAC'} → {'ABPR'}	ABPR	0.943089
{'ABPR','ABAR','PCCE','POAC'} → { 'PWIW'}	PWIW	0.742682
{'ABPR','PWIW','PCCE','POAC'} → { 'ABAR'}	**ABAR**	**0.996038**
{'ABPR','PWIW','ABAR','POAC'} → { 'PCCE'}	PCCE	0.937153
{'ABPR','PWIW','ABAR','PCCE'} → { 'POAC'}	POAC	0.921406

The next operator to be removed is ABAR. After its removal, there were changes in some mutation scores, but none of these changes lead to the empirical inclusion of any operator in SS_{pre}. This new revaluation is shown in Table 7.

At the end of step 3, we have the following SS_{pre} set:

SS_{pre} = {PWIW, PCCE, POAC, ABPR}

Table 7: Empirical Inclusion for the operator after exclude ABAR operator

Relation Empirical Inclusion	Highlight Op.	Mutation Score
{'PWIW','PCCE','POAC'} → {'ABPR'}	ABPR	0.861788
{'ABPR','PCCE','POAC'} → { 'PWIW'}	PWIW	0.688023
{'ABPR','PWIW','POAC'} → { 'PCCE'}	PCCE	0.937153
{'ABPR','PWIW','PCCE'} → { 'POAC'}	POAC	0.921406

Step 4: Establish incremental implementation strategy

This step does not change the composition of SS_{pre}; instead, it defines an order for the application of the pre-selected operators, based on their cost in terms of number of generated mutants. Operators must be applied from the lowest-cost operator to the highest-cost one, resulting in the following order of application: **POAC** (50 mutants), **PCCE** (57 mutants), **ABPR** (62 mutants), **PWIW** (456 mutants).

Step 5: Consider operators that provide an increase in the mutation score

In this step, Barbosa et al. [2] defined the minimum increment index (MII) (in this case, MII ≥ 0.02), as a minimum increment an operator should provide to overall mutation score if it was included in SS_{pre}.

Observing all mutation operators (except those already in SS_{pre}), only PCLO and POPL are not empirically included by SS_{pre}. Thus, these are candidate operators to be inserted into SS_{pre}. However, neither PCLO nor POPL have achieved the MMI index, *i.e.* none of them have increased substantially the mutation score of SS_{pre}, so the preliminary set remains the same.

Step 6: Consider operators with high strength

In Step 6, Barbosa et al. [2] defined a minimum index of strength (MSI) for the inclusion of an operator in SS_{pre}. The authors have set MSI = $0.4^{+}_{-}0.02$.

Based on this index, we have the following operators: PWIW, PCCE, PCTT, POAC, APSR, PCGS, ABAR, ABHA, POPL, ABPR and PCLO. From these, PWIW, PCCE, POAC and ABPR are not considered as they already compose the SS_{pre} set. From the remaining operators, only POPL and PCLO are not empirically included, as noticed in Step 5. Thus, they again become candidates for inclusion in SS_{pre}.

Since POPL and PCLO belong to the same class of operators, only one will be included in SS_{pre}. Thus, the POPL operator (strength = 0.4147, see Table 4) was included in SS_{pre} because it has higher strength than PCLO (strength = 0.3850, not listed in Table 4 due to space limitations).

The resulting sufficient set of operators

At the end of the six steps, we have defined the following sufficient operator set, thus we answer positively the first research question defined in Section 4.1:

$$\boxed{SS = \{\text{POAC, PCCE, ABPR, PWIW, POPL}\}}$$

5.2. Analysis

We now analyse the cost reduction we can obtain by applying the achieved sufficient set. For this, we must compare the cost of applying only the *SS* operators with the cost of applying the full set of operators. Table 8 shows the application costs for each *SS* operator. The table columns include, respectively: the operator name; the number of mutants; the number of compilable mutants; the number of equivalent mutants automatically detected by the

Proteum/AJ tool; and the number of significant mutants (*i.e.* killed by tests or manually classified as equivalent).

At a first sight, we obtain a cost reduction of 26.4%, considering all 922 mutants generated (see Table 2) and the 679 mutants generated with *SS* operators. In a further analysis, we discard anomalous mutants and mutants that are automatically detected as equivalent. In both cases, no manual effort is required, since identifying such mutants only depends on automated routines performed by the tool. Therefore, we consider only the last column of Table 2 and the results are as follows:

From the 922 mutants generated by all operators, 471 are automatically classified as equivalent and 85 are anomalous. This gives us a balance of 366 significant mutants. Comparing this number with the value of 173 (which is the number of significant mutants generated by the *SS* operators), the cost reduction reaches 53%.

This cost reduction is slightly lower than the reduction of 65% reached by Barbosa et al. [2]. Nevertheless, it is undoubtedly an expressive value. Therefore, we can also answer positively the second research question defined in Section 4.1.

Table 8: Cost of the Sufficient Set

Operator	Cost	Comp.Mut.	Equiv.Mut.	Total Mutants
POAC	50	44	0	44
PCCE	57	55	0	55
ABPR	62	20	0	20
PWIW	456	446	402	44
POPL	54	50	40	10
Total	679	615	442	173

5.3. Limitations

According to Barbosa et al. [2], the Sufficient Procedure depends on the application domain and the set of applications used. Given that there is no other study reporting the use of this procedure to determine sufficient sets of mutation operators in AO programs, it would be interesting to apply the Sufficient Procedure to other application domains in order to contrast the results with the findings of this study. It is also interesting to apply this procedure using a larger set of application to investigate the scalability of the procedure in this paradigm, because the set used in this study consists of only 12 applications.

Another limitation concerns the variety of mutation operators applied. The DSSR, DEWC and DAIC operators did not generate any mutant for the target applications. Indeed, a previous cost estimation study performed by Ferrari et al. [9] showed that these operators are likely to produce few (if none) mutants even when applied to medium-sized systems.

6. Conclusion and Future Work

This paper reported an exploratory study that aimed to identify a reduced set of mutation operators that require effective tests to reveal faults artificially introduced into programs. To achieve the results, we applied the Sufficient Procedures, which is an incremental technique to identify the most relevant mutation operators from a given set of operators. The relevance of an operator relies on its produced mutant coverage (*i.e.* the mutation score) and on its strength when compared to other operators. After concluding the study, we provided positive answers to two research questions that regarded the feasibility of the procedure for the AO programs, and the level of cost reduction that can be achieved. In short, a set compound by 5 out of 24 operators requires 53% less effort in terms of the number of generated mutants when contrasted with the full set of operators.

The results of this study are useful, for instance, for researchers and testers who intend to apply mutation testing in AspectJ programs from the same application domain. Beyond that, the results can build a historical database of sufficient operators that can grow with new rounds of experiments. As future work, we plan to apply the Sufficient Procedure to larger sets of AspectJ applications, so that we can contrast the results to check if there is any variation in the sufficient set of operators achieved.

Acknowledgements

The authors would like to thank the financial support received from CNPq (Universal Grant #485235/2013-7), Federal University of São Carlos (RTN grant) and CAPES.

References

[1] Alexander, R. T., Bieman, J. M., and Andrews, A. A. (2004). Towards the systematic testing of aspect-oriented programs. Tech. Report CS-04-105, Dept. of Computer Science, Colorado State University, Fort Collins/Colorado - USA.

[2] Barbosa, E. F., Maldonado, J. C., and Vincenzi, A. M. R. (2001). Toward the determination of sufficient mutant operators for C. *The Journal of Software Testing, Verification and Reliability*, 11(2):113–136.

[3] Delamare, R., Baudry, B., Ghosh, S., and Le Traon, Y. (2009). A test-driven approach to developing pointcut descriptors in AspectJ. In *Proceedings of the 2^{nd} International Conference on Software Testing, Verification and Validation (ICST)*, pages 376–385, Denver/CO - USA. IEEE Computer Society.

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

[5] Dijkstra, E. W. (1976). *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs/NJ - USA.

[6] Ferrari, F. C., Burrows, R., Lemos, O. A. L., Garcia, A., and Maldonado, J. C. (2010a). Characterising faults in aspect-oriented programs: Towards filling the gap between theory and practice. In *Proceedings of the 24^{th} Brazilian Symposium on Software Engineering (SBES)*, pages 50–59, Salvador/BA - Brazil. IEEE Computer Society.

[7] Ferrari, F. C., Maldonado, J. C., and Rashid, A. (2008). Mutation testing for aspect-oriented programs. In *Proceedings of the 1^{st} International Conference on Software Testing, Verification and Validation (ICST)*, pages 52–61, Lillehammer - Norway. IEEE.

[8] Ferrari, F. C., Nakagawa, E. Y., Rashid, A., and Maldonado, J. C. (2010b). Automating the mutation testing of aspect-oriented Java programs. In *Proceedings of the 5^{th} ICSE International Workshop on Automation of Software Test (AST)*, pages 51–58, Cape Town - South Africa. ACM Press.

[9] Ferrari, F. C., Rashid, A., and Maldonado, J. C. (2013). Towards the practical mutation testing of AspectJ programs. *Science of Computer Programming*, 78(9):1639–1662.

[10] Jia, Y. and Harman, M. (2009). Higher order mutation testing. *Inf. Softw. Technol.*, 51(10):1379–1393.

[11] Kiczales, G., Irwin, J., Lamping, J., Loingtier, J.-M., Lopes, C., Maeda, C., and Menhdhekar, A. (1997). Aspect-oriented programming. In *Proceedings of the 11^{th} European Conference on Object-Oriented Programming (ECOOP)*, pages 220–242 (LNCS v.1241), Jyväskylä - Finland. Springer-Verlag.

[12] Linkman, S., Vincenzi, A. M. R., and Maldonado, J. C. (2003). An evaluation of systematic functional testing using mutation testing. In *Proceedings of the 7^{th} International Conference on Empirical Assessment in Software Engineering (EASE)*, pages 1–15, Keele - UK.

[13] Mathur, A. P. and Wong, W. E. (1993). Evaluation of the cost of alternative mutation strategies. In *Proceedings of the 7^{th} Brazilian Symposium on Software Engineering (SBES)*, pages 320–335, João Pessoa/PB - Brazil.

[14] Offutt, A. J., Rothermel, G., and Zapf, C. (1993). An experimental evaluation of selective mutation. In *Proceedings of the 15^{th} International Conference on Software Engineering (ICSE)*, pages 100–107, Baltimore/MD - USA. IEEE Computer Society.

[15] Omar, E. and Ghosh, S. (2012). An exploratory study of higher order mutation testing in aspect-oriented programming. In *Proceedings of the 23rd International Symposium on Software Reliability Engineering (ISSRE)*, pages 1–10. IEEE.

[16] The Eclipse Foundation (2010). AJDT Eclipse plugin. Online. `http://www.eclipse.org/ajdt/` - last accessed on 27/06/2014.

[17] Wedyan, F. and Ghosh, S. (2012). On generating mutants for aspectj programs. *Information and Software Technology*, 54(8):900–914.