Automated Evolutionary Test Data Generation with Domain Reduction for Aspect-Oriented Programs

Mark Harman, Fayeziin Islam
King’s College London
Centre for Research on Evolution, Search and Testing (CREST)
Strand, London, WC2R 2LS, UK

Tao Xie
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-8206

ABSTRACT
Aspect-Oriented Programming is an emerging technique that helps improve separation of concerns in software systems. It has received a great deal of recent interest. However, algorithms and empirical results for testing of aspect-oriented programs are lagging some way behind this upsurge in interest. To date, there are few published approaches to automated test data generation for aspect-oriented programs and, therefore, there remain few empirical results on testing aspect-oriented programs. In this paper, we present a new approach to automated test data generation for aspect-oriented programs and its supporting system, providing empirical evidence to validate its applicability. The new approach uses domain reduction to further improve the performance of test data generation, providing empirical evidence that domain reduction can significantly reduce the computational expense of test data generation for aspect-oriented programs. The paper also presents the results of a study into effort reduction when focusing on testing behavior in aspects instead of all behavior in the whole program.

Keywords
Test Data Generation, Aspect-Oriented Software Development, Evolutionary Testing, Search Based Software Engineering

1. INTRODUCTION
Aspect-Oriented Programming (AOP) is a new technique that helps to improve separation of concerns in software systems [4, 15, 18, 28]. The goal of this approach to software engineering is to make it possible to modularize the crosscutting concerns of a software system, thereby making the system easier to maintain and evolve. Research in AOP has received a great deal of recent interest from both the software engineering and programming language communities. Aspect-oriented programming languages, such as AspectJ [15], introduce new language constructs (such as join points, advice, intertype declarations, and aspects). This creates new challenges for software testing in general, and for automated software test data generation in particular.

The behavior of an aspect in AspectJ programs can be categorized into two types [19]: aspectual behavior (behavior implemented in pieces of advice) and aspectual composition behavior (behavior implemented in pointcuts for composition between base and aspectual behavior). To assure the correct construction of aspect-oriented programs, developers need to conduct adequate testing on aspect-oriented programs to make sure that aspectual and aspectual composition behavior perform as expected. However, like other testing activities, this testing activity is often laborious, expensive and error-prone when conducted manually. Therefore, there is a pressing need for automated test data generation techniques.

To support adequate testing of aspect-oriented programs, several approaches have been proposed recently, including fault models and coverage criteria [1, 2, 40], test selection [36, 39], model-based test data generation [37, 38], and white-box test data generation [15]. However, few approaches target automated test data generation (ATDG), and even fewer provide tool automation and empirical results concerning aspect-oriented program testing. This current lack of AOP test automation progress and the associated empirical paucity raises barriers to increased uptake and practical application of AOP techniques.

One of the few tools available that is applicable to aspect-oriented program test data generation is Aspectra [35] developed by Xie and Zhao. It provides a wrapper mechanism to leverage the existing object-oriented test generation tool Parasoft Jtest [24]. Parasoft Jtest generates default values for primitive-type arguments and generates random method sequences. However, Aspectra provides no special support for the generation of test data for aspect-oriented programs. Furthermore, the underlying Parasoft Jtest tool was neither especially designed nor optimized for testing aspect-oriented programs and its test generation mechanisms are limited. Essentially, it supports the generation of default or random data. This is known to be highly sub-optimal for testing non-aspect-oriented programs [3, 17, 23, 25, 31], and there is no reason to believe that it will be any better, simply because of the presence of aspects. Indeed, this paper presents results that support the claim that it is not.

Two important questions in ATDG for aspect-oriented programs remain unanswered in any previous approach including the Aspectra approach. First, how (and how effectively) can advanced ATDG techniques be adapted to test aspect-oriented programs? It is natural to exploit more advanced ATDG techniques (such as evolutionary-ary techniques and domain reduction) rather than merely the simple ones such as generating default values (as done by Parasoft Jtest in the Aspectra approach). Second, how (and how effectively) can the overall test generation cost be reduced by focusing automation efforts on aspects? If our test objective is aspect behavior coverage then it may not be economical to spend time on the non-aspectual parts of the system under test.
This paper addresses these questions, providing an approach to fully automated aspect-oriented test data generation using evolutionary techniques and dependence-based domain reduction. The paper presents empirical results that support the claim that the approach is effective, efficient and that domain reduction further improves the performance of the approach.

This paper makes the following primary contributions:

- We develop a system for ATDG for aspect-oriented programs. This is the first ATDG system for aspect-oriented programs and represents the first application of search-based testing to the aspect-oriented paradigm.
- We demonstrate the effectiveness of the system with an empirical study that provides strong evidence to support the claim that Evolutionary Testing for aspect-oriented programs can achieve higher coverage and with less effort than Random Testing.
- We develop domain reduction techniques in our system to improve the performance of ATDG. The techniques use a dependence analysis based on slicing to identify irrelevant parts of the test input that cannot affect the outcome of branch evaluation in aspects. This type of domain reduction has recently been shown to improve Search-Based Testing for programs written in C [12]. We investigate whether domain reduction can also improve Evolutionary Testing of aspect-oriented programs. The findings of this empirical study are encouraging:
  - The study revealed many predicates in real-world aspect-oriented programs where dependence analysis was able to determine a proper subset of the input domain, upon which the outcome of a predicate depends. This finding indicates that there is scope for domain reduction as a mechanism for test-effort reduction when branch coverage is the test adequacy criterion.
  - For all programs where there were coverable branches, the test effort decreased when domains were reduced.
  - The number of covered branches was increased by domain reduction. This finding is important because in the study of non-aspect-oriented programs [12], it was found that only the test effort was reduced, while branch coverage was not increased by domain reduction.
  - An interesting co-lateral coverage effect was also noticed. That is, the results provide evidence to suggest that the overall branch coverage was improved when targeting branches for which the domain reduction information was available to the ATDG system. This paper is the first to report on this co-lateral effect. It may find wider applications in test data generation.
- We present the results of a study of focusing testing efforts on those aspetual branches instead of all branches in a program. The results show that the test effort was reduced while the same or better aspetual branch coverage was achieved.

The rest of the paper is organized as follows. Section 2 uses examples to illustrate our approach to ATDG for aspect-oriented programs. Section 3 introduces the approach and how it was implemented in order to produce the results reported in this paper. Section 5 describes the experimental set up: the subject programs being studied as well as the research questions posed by the research and addressed by the empirical studies that follow. Sections 6, 7, and 8 present the results of the three empirical studies. These three studies are, respectively, concerned with validating that

```java
public class Account {
    private float _balance;
    private int _accountNumber;
    private Customer _customer;
    public Account(int accountNumber, Customer customer) { ... }
    public void debit(float amount) { ... }
}
```

```java
public aspect ODRuleAspect
    pointcut debitExecution(Account account, float withdrawalAmount) : execution(void Account.debit(float) && this(account) && args(withdrawalAmount));
    before(Account account, float withdrawalAmount) : debitExecution(account, withdrawalAmount) {
        if (account.getAvailableBalance() < withdrawalAmount) {
            float transferAmountNeeded = withdrawalAmount - account.getAvailableBalance();
            Customer customer = account.getCus
```
our evolutionary tester generates various test data for the parameters; without losing generality, we consider the receiver object (the Account object) as one parameter. The evolutionary tester uses a search-based software testing approach [22] based on the theory of evolution.

It is quite costly for the evolutionary tester to try various combinations of the data for the parameters. We can observe that, in order to cover an aspectual branch, often, only a subset of parameters are relevant, which are required to hold certain data values. Our test generation should explore the domain for these relevant parameters instead of investing time on all parameters. Therefore, to reduce the test-generation cost, we use a domain reduction technique to exclude irrelevant parameters in the search space of test data. In particular, we perform backward slicing. The slice criterion is the predicate that is involved with the target aspectual branch. The resulting program slice contains only statements that can influence the coverage of our target branch. For example, the resulting slice of our target branch is shown below:

```java
Customer customer=account.getCustomer();
if(customer == null) return;
```

We next identify which parameters in the target method are not relevant to our target branch by looking for the name and type of each parameter in the resulting slice. As the program slice contains all statements that can be executed within the target method to influence the target branch, any parameter that is not contained within the slice is considered to be irrelevant. In our example, the parameter “account” (receiver object) of the “debit” method occurs within the slice but the parameter “amount” of the “float” type does not. Therefore, the float parameter is considered to be irrelevant for the target branch.

After all irrelevant parameters have been identified, we instruct the evolutionary tester not to try various data for these irrelevant parameters. By excluding the irrelevant parameters, we are essentially reducing the search space for testing the target branch.

3. APPROACH

We develop an approach for automated test data generation (ATDG) for aspect-oriented programs. Its test objective is to generate test data to cover aspectual branches (i.e., achieving high aspectual branch coverage). The input to the framework is aspects written in AspectJ as well as the Java classes where the aspects are woven, being called base classes. As proposed in Aspectra [35], for the given aspects under test, our approach generates test data for the base classes and these test data indirectly exercise or cover the aspects. We can view these base classes as providing the scaffolding necessary to drive the aspects. The generated test data is a type of unit test for the base classes but with respect to aspect code, the generated test data can be also viewed as a type of integration tests, testing the interactions between the aspects and the base classes. To measure coverage of aspectual behavior, our approach uses the metric of aspectual branch coverage [35], which measures the branch coverage within aspect code.

Our approach consists of four major components:

- **Aspectual-branch identity**. Given the AspectJ source code (including both aspects and base classes), the component of aspectual branch identifier identifies branches inside aspects, which are the coverage targets of our approach. There are two types of aspectual branches: predicate branches and pointcut branches. Predicate branches are the branches involved with conditional predicates inside aspects and pointcut branches are call sites of advices defined in aspects. Pointcut branches are included because covering them can help cover advices defined in aspects and thus cover predicate branches in aspects. The identified aspectual branches are to be specified as test goals to the component of the evolutionary tester.

- **Relevant-parameter identifier**. Because not all parameters of the methods of the base classes would be relevant to covering a target aspectual branch, the component of relevant-parameter identifier identifies only those relevant method parameters. This component implements a type of domain reduction in test data generation.

- **Evolutionary tester**. Given the relevant parameters produced by the relevant-parameter identifier, the component of evolutionary tester conducts evolutionary testing on the relevant parameters.

- **Aspectual-branch-coverage measurer**. After the tests are generated by the evolutionary tester, the component of aspectual-branch-coverage measurer measures the coverage of aspectual branches and selects test data that can cover a new aspectual branch that is not covered by earlier selected test data.

We next present more details on two key techniques in our approach: input-domain reduction and evolutionary testing, conducted by the components of the relevant-parameter identifier and the evolutionary tester, respectively.

3.1 Input-Domain Reduction

The input-domain reduction technique [12, 13] was introduced for constraint-based testing. It typically involves simplifying constraints using various techniques and generating random inputs for the variables with the smallest domain. The process is repeated until the target structural entity such as a branch has been covered.

**Input domain.** The input domain in program testing is generally considered as global variables and the set of input parameters of a method (called a target method) that contains the target branch or whose callees contain the target branch (in testing object-oriented programs, we can view the receiver object of the method under test as an input parameter). In our problem context, the input domain is the set of input parameters of a method (called a target method) in a base class that invokes the aspect containing the target aspectual branch. In existing approaches such as Aspectra [35], this target method is directly fed as the method under test to an existing ATDG tool for object-oriented programs, and consequently the tool would generate various data values for all the parameters of the target method. As we can see, the number of parameters where various data values shall be tried determine the size of the search space of test data. Usually this test data generation process (i.e., search process) is quite expensive, inducing high cost.

However, in testing aspectual behavior such as generating test data to cover aspectual branches, we can observe there are two main opportunities for reducing this cost. First, not all parameters of the target method would affect whether the advice containing the target
aspectual branch would be invoked. Second, not all parameters of the target method would affect whether the target aspectual branch would be covered. Based on this observation, our approach uses the input-domain reduction technique to reduce the input domain by identifying irrelevant parameters and excluding them from the scope of testing in order to reduce testing efforts.

**Program slicing.** To identify such irrelevant parameters, we use program slicing [32]. Program slicing is a static analysis technique that helps to create a reduced version of a program by placing its attention on selected areas of semantics. The process removes any part of the program that cannot influence the semantics of interest in any way. The reduced version of the program is called a slice and the semantics of interest is known as slice criterion. Based on the slice criterion, it is possible to produce backward or forward slices. A backward slice consists of the set of statements that can influence the slice criterion based on data or control flow. A forward slice contains the set of statement that are control or data dependent on the slice criterion. That is, a forward slice includes any statement that can be affected by the slice criterion.

Search space reduction by program slicing has been used as the technique for identifying irrelevant parameters for each aspectual branch. After aspects have been woven to base classes and aspectual branches have been identified, the line of the conditional statement associated with the target aspectual branch is used as the slicing criterion for backward slicing. Then we check whether a parameter of the target method is within the slice to determine its relevancy. If a parameter does not appear within the slice, then it is considered as an irrelevant parameter.

### 3.2 Evolutionary Testing

Evolutionary testing [22] is a search-based software testing approach based on the theory of evolution. The idea that underpins evolutionary algorithms is based on maintenance of a population of test data called individuals. The population is changed with a series of generations. The fitness of each individual is calculated using a fitness function that gives greater values for good test data. Every generation is produced by applying genetic operators to individuals which imitate the mating and transformation of natural genetics. As the number of generations increases, the population contains more individuals with high fitness values. The procedure stops when an adequate amount of fitness has been achieved or the maximum number of generations have been reached. This testing approach has been found to achieve better performance than random testing since it concentrates the search towards finding test data with high fitness values.

**Structural testing.** For structural testing such as branch testing, the fitness value is usually determined using the distance based approach or based on how close the test data came to cover the target branch. Approximation level depicts the distance from target in terms of nesting level of branches and local distance represents the distance in terms of test data value. Typically approximation level and local distance are used in combination for fitness calculation of individual test data. For branch coverage, a fitness value closer to 0 is desired as a fitness value of 0 means that the branch has been covered.

**Class testing.** When testing a class, evolutionary testing [30, 31] transforms the task of creating test or method sequences that lead to high structural coverage of the code under test to a set of optimization problems that a genetic programming algorithm then tries to solve. Each uncovered code element, such as a branch when performing branch testing, becomes an individual test goal for which an evolutionary search will be carried out. A tree-based representation of test sequences is used to account for the call dependencies that exist among the methods of the classes that participate in the test. This representation combats the occurrence of non-executable test sequences. Method call trees are evolved via sub tree crossover, demotion, promotion, and the mutation of the primitive arguments.

Our approach conducts evolutionary testing for structural testing and class testing on the target method of the base class for a target aspectual branch. Our approach specially narrows down the search space for evolutionary testing by instructing the evolutionary tester not to explore those identified irrelevant parameters.

### 4. IMPLEMENTATION

We have implemented the proposed approach for ATDG of aspect-oriented programs in a prototype tool called EvolutionaryAspectTester (EAT).

**Aspectual-branch identifier.** To identify aspectual branches and measure the coverage of aspectual branches, we modified the aspectual branch coverage measurement tool called Jusc from the Aspectra approach [35]. In particular, based on Jusc, we identify branches related to aspects by scanning and matching method names in the bytecode (produced by an AspectJ compiler) against pre-defined signatures related to aspects.

**Relevant-parameter identifier.** We used the Indus Java slicer [26] to produce backward slices from Java code. Because an AspectJ compiler by default produces bytecode instead of source code after weaving, we convert AspectJ woven bytecode to Java source code by using the ajc AspectJ Compiler 1.0.6; it offers an option of producing an equivalent Java version of the AspectJ code. We modified the Indus API to store the information of original source line numbers in Jimple [29] code, the format of slices generated by Indus. After slicing, line numbers of the slices are extracted from Jimple output and corresponding code from those lines were used to construct slices in the source code. Once a target method is sliced for an aspectual branch and the method’s parameters are identified as relevant or not, our tool produces a new version of Java code for each branch, with the irrelevant parameters removed and declared as local variables within the method. This new version of Java code is fed to the evolutionary tester as input.

**Evolutionary tester.** In our implementation, we used EvoUnit [30, 31] from Daimler Chrysler to implement the evolutionary testing technique as well as the random testing technique, which is used as comparison base in our experiments being described in the rest of the paper. We also extended EvoUnit to implement the concept of reducing the input domain for evolutionary testing. EvoUnit generated JUnit test suites containing one test case for each target branch.

**Aspectual-branch-coverage measurer.** We used Jusc [35] to measure the aspectual branch coverage achieved by the JUnit test suite generated by EvoUnit. For each covered branch, our tool also produces the name of the first-encountered JUnit test case that covers that branch.

### 5. EXPERIMENTAL SETUP

We next describe the experiment setup, including the programs used for the empirical study and the research questions to be answered. The paper provides empirical results that provide evidence to support the claims made in answering the questions.

In the empirical studies, we applied the proposed approach to a suite of 14 aspect-oriented programs written in AspectJ [15]. Figure 3 shows the details of these programs with Columns 1-6, showing the program name, the lines of code (LoC) of the whole program, the LoC of the test driver (the base classes used to drive the aspects under test) together with the aspects, the number of aspectual branches (including both branches related to predicates in as-
aspects and methods in aspects where the entry of a method in aspects is counted as one branch to accommodate covering a method without any branching points [35]), the number of aspectual branches from predicates in aspects, and a brief description, respectively. Note that only the aspectual branches related to predicates are used to conduct domain reduction in the empirical studies.

Most of these programs were used by Xie and Zhao in evaluating Aspectra [35]. These programs include most of the programs used by Rinard et al. [27] to evaluate their classification system for aspect-oriented programs. The programs also include most of the programs used by Dufour et al. [9] in measuring performance behavior of AspectJ programs. These programs also include one of the aspect-oriented design pattern implementations used by Hanne mann and Kiczales [10]. Although the programs are relatively small, they represent a range of different real uses of aspect code including instrumentation, monitoring, contract enforcement, exception handling, logging, updating, and filtering. There are a total of 658 different branches considered, each of which represents a different search optimization problem. The involved search space is large in many cases, leading to non-trivial search problems for search-based test data generation.

The research questions addressed in the three empirical studies are described as below:

**Assessment of evolutionary testing**
- RQ 1.1. Can evolutionary testing outperform random testing for testing aspect-oriented programs?

**Effect of domain reduction**
- RQ 2.1. What is the number of branches in each program that have irrelevant parameters and how many parameters are irrelevant for each of these branches?
- RQ 2.2. What is the computational effort reduction for each branch that has irrelevant parameters removed and for how many of these branches is the improvement statistically significant?
- RQ 2.3. For each program that contains branches that have irrelevant parameters, what is the overall improvement in computational effort for test data generation?
- RQ 2.4. When generating test data for a particular target branch, what is the co-lateral effect on the level of coverage achieved for not-targeted branches?

**Effect of focusing on testing aspectual behavior**
- RQ 3.1. What is the computational effort reduction for test data generation for each program if aspectual behavior instead of all behavior is focused?

### Metrics

**Metrics.** We use aspectual branch coverage (the number of covered aspectual branches divided by the total number of aspectual branches) to measure how well the advices in aspect-oriented programs have been tested.

As is standard in experiments on evolutionary and search-based computation algorithms, we measure the effort (i.e., the computational cost) in terms of the number of fitness evaluations used by each algorithm to find a test input that covers the target branch. This measurement avoids implementation bias and the difficulties associated with reliable, robust, and replicable assessment of computation effort; the evaluation of fitness is the core atomic unit of computational effort for search-based approaches. The upper bound for the number of fitness evaluations was set to 10,000. If no solution (i.e., no covering test input) was found after this number of fitness evaluations, then the algorithm was terminated. To cater for the inherently stochastic nature of the search-based algorithms, each algorithm was executed 30 times on each test objective, facilitating the assessment of the statistical significance of the results. For each branch to be covered, the t-test was used to assess the significance in the difference of means over these 30 runs, at the 95% confidence level.

### 6. EMPIRICAL STUDY: ASSESSING EVOLUTIONARY TESTING

We applied evolutionary testing and random testing on the 14 programs and compared their results in terms of the achieved code coverage and effort taken for testing. Figure 4 shows the improvement in coverage achieved by evolutionary testing over random testing. The x axis represents each program and the y axis represents the improvement in branch coverage as a result of using evolutionary testing. We observed that we achieved the same branch coverage on 9 out of 14 programs with evolutionary and random testing. We achieved better branch coverage on the remaining 5 programs with evolutionary testing. The maximum improvement of 42.67% in branch coverage is observed on the program SavingsAccount.

Figure 5 shows the effort reduction per program for evolutionary testing over random testing. The x axis shows the 14 programs and the y axis shows the percentage reduction in effort with evolutionary testing. We observed that 5 out of 14 programs had no difference in effort for evolutionary testing and random testing, and the remaining 9 programs had a reduction in effort for using evolutionary testing. The maximum reduction of 61.28% is achieved by the program Queue. Overall we can deduce that evolutionary testing takes the same or less of effort for testing the same programs when compared to random testing.

An interesting observation is that, when the results of branch coverage improvement and effort reduction are compared, all 5 programs that had an improvement in branch coverage also had
only aspecual branches from predicates (indeed, we can similarly consider the entry of a method in aspects as the slicing criterion for conducting domain reduction. But in our empirical studies, we focus on those real branches in aspects).

For only 2 of the remaining 8 programs (2 smaller programs: NonNegative and NullChecker), there were no branches that had irrelevant parameters; all parameters potentially affected the outcomes of all predicates for these 2 programs according to the Java slicing tool, Indus, used in the implementation. Of the remaining 6 programs with branches that have irrelevant parameters, there were a total of 90 branches with irrelevant parameters and which could, therefore, potentially benefit from the exploitation of domain reduction. Of these 90 branches, 66 were testable using the evolutionary tester. Of these 66, it was possible to generate test data reliably (on all of the 30 runs of the test data generation system) for 48.

Figure 7 shows the reduction in parameters achieved for each of the 48 branches for which some non-zero reduction was possible. The size of reduction is represented using the percentage of irrelevant parameters. The x axis shows all 48 branches using their branch identifiers and y axis shows the percentage of irrelevant parameters. Overall, a considerable amount of domain reduction was possible for all 48 branches. We observed that for several branches we have achieved 100% domain reduction. This complete reduction is possible because only the search space related to input parameters is represented here. 100% domain reduction implies that the methods that contain these branches do not have any parameters that can help to cover these branches. However, a class may define public fields whose values can affect the coverage of these branches and the search space for these public fields is not considered as part of the search space related to input parameters in our measurement.

7.2 Research Question RQ 2.2

Figure 8 shows the effect of domain reduction in test data generation effort for each branch with non-zero irrelevant parameters. Recall that the number of fitness evaluations required during evolutionary testing has been used as the measure of effort. In the figure, the x axis shows each branch with non-zero irrelevant parameters and the y axis shows the percentage reduction in effort after input domain reduction.

We observed that 25% (12 of 48) of the branches had an increase in effort, 6.25% (3 of 48) of the branches had no change, and 68.75% (33 of 48) of the branches had a reduction in effort.
due to input domain reduction. The maximum reduction achieved is 93.67% by a branch in NullCheck and the minimum reduction achieved is -87.98% by a branch in Queue. Although the maximum and minimum reduction values are far apart, we observed that the majority of the branches respond positively to input domain reduction.

The results indicate that input domain reduction can also cause increased effort of up to 87.98%. Further investigation revealed that 11 out of 12 branches with an increase in effort are trivial branches whose average effort size is so small that random effects predominate.

We also performed a t-test to identify the percentage of branches where change in effort before and after input domain reduction is statistically significant. The results of the t-test (not presented here in detail due to space limitations) show that there are 11 out of 48 branches where the change in effort after input domain reduction is statistically significant at the 95% level. The change in effort for the remaining 37 branches was found to be statistically insignificant.

### 7.3 Research Question RQ 2.3

Figure 9 shows the effect of domain reduction (shown on the x axis) on each program (shown on the y axis) that contains branches with irrelevant parameters. Note that for some (comparatively trivial) branches presented earlier in Figure 8, there was an increase in effort. But this effect does not translate into an increase in effort to test the program. In all cases, the overall effort to test the program is reduced, by 17.34% to 99.74%. This finding answers RQ 2.3 and suggests that domain reduction may be a useful technique for improving test data generation for aspect-oriented programs.

In summary, for the programs that contained branches with irrelevant inputs that were testable, the overall impact of domain reduction on the effort required to test the program was positive. Note that these results are not purely a reflection of the number of branches to be covered, indicating that the complexity of the testing problem of covering the branch is the issue rather than the sheer number of branches to be covered. For instance, the program NullCheck has only four branches, yet enjoys a 92.86% reduction in test effort through domain reduction.

### 7.4 Research Question RQ 2.4

RQ 2.4 addresses the issue of co-lateral coverage, which we explain next. In testing a program, each branch is targeted in turn and an attempt is made to cover the branch. However, in common with other work on search-based testing [3, 17, 23, 25, 31], it is not uncommon for non-target branches to be covered as a byproduct of test data generation. This non-target coverage typically results from situations where the target branch is hard to cover and requires a certain number of intermediate branches to be covered before the target branch can be reached. This non-target coverage also results from the natural stochastic nature of the test data generation process using search-based optimization; there always remains the possibility for some run of the algorithm to cover any branch. This stochastic nature is the reason for the careful control denoted by the repeated execution of the algorithm (30 times) and the application of statistical techniques to investigate significance of results.

In RQ 2.4, the question is whether the reduction of a domain for a target branch can help to generate test data for other non-target branches. In the study, this effect was indeed found to happen, though not always. The results are presented in Figure 10 where the x axis shows the branches and the y axis shows the branch coverage improvement. We observed that there is a ‘positive effect’ of domain reduction on other branch objectives; co-lateral coverage tends to increase with domain reduction. This interesting finding suggests possible further research. It is possible that the target branches are on a path that contains other controlling predicates that share similar sub-domains of the input with the target branch. In this situation, it could be expected that attaching the target branch with test data generation would also hit the non-target branches on paths to the target. However, more research is needed to explore these possibilities.
In summary, test data generation aimed at covering a chosen target branch can result in other non-target branches being covered. Interestingly, by reducing the domain for the target branch there is a tendency to improve this ‘co-lateral coverage’. The figure shows the improvement in co-lateral coverage ordered by the strength of improvement. For only a few branches does the co-lateral coverage decrease, whereas for the majority, it increases.

8. EMPIRICAL STUDY: IMPACT OF FOCUSING ON ASPECTUAL BEHAVIOR

We compared the results of generating test data for covering aspectual branches only and generating test data for covering all branches in both the aspects and base classes. Figure 11 shows the impact of testing aspectual behavior in terms of effort per tested program. The x axis shows the tested programs and the y axis shows the percentage reduction in effort for these programs. As shown by the results, in all 14 programs, a reduction in effort has been achieved as a result of testing aspectual branches as opposed to testing the full program. The maximum overall reduction of 99.99% was possible in the QuickSort program. The minimum reduction of 3.02% was observed in the NullCheck program.

Figure 12 shows the improvement in aspectual branch coverage for all 14 programs as a result of testing aspectual branches as opposed to testing all branches in the program. We observed that the improvement in aspectual branch coverage is quite small. The minimum improvement is 0% for 8 out of 14 programs, indicating that there was no change in aspectual branch coverage. The maximum improvement is on the Queue program where the improvement was 62.20%.

Further investigation revealed that the improvement in coverage caused by the random behavior of evolutionary testing as some branches in the Queue program were not covered while testing all branches in the program. However, while testing aspectual branches only, some of these branches were randomly covered more often resulting in the spike in branch coverage. The improvement was random and not caused by a better technique; otherwise, similar results would have been observed in other programs.

We performed t-test independently on all 65 classes under test (included in the 14 programs) by taking the effort data collected from all 30 runs as input for the statistical test. The results of t-test (not presented in this paper due to space limit) show that there are 47 classes where the reduction in effort is statistically significant. There were 13 branches for which p values could not be calculated as the formula calculation encounters division by zero in those cases. 5 out of 65 classes were found to be statistically insignificant. This analysis indicates that the reductions in effort for majority of the classes (and the programs) are statistically significant. Therefore, it can be concluded that testing only aspectual branches results in effort reduction and at the same time achieves same or better aspectual branch coverage.

9. THREATS TO VALIDITY

The threats to external validity primarily include the degree to which the subject programs and testing techniques under study are representative of true practice. The subject programs are relatively small. We collected AspectJ benchmarks from the web and reused benchmarks used in the literature in testing and analyzing aspect-oriented programs. The subject programs under study do represent the real aspects being used in practice, which are often not large. But the small size of these aspects does not devalue the importance of testing these aspects because these aspects are woven into many locations of the base classes and once there are defects in these aspects, the impact would often be substantial. We studied the application of evolutionary testing and random testing in testing aspect-oriented programs, because they are common testing techniques used in practice, being able to be widely used in various types of programs without being constrained by the characteristics of the programs under test, unlike some other testing techniques such as those based on symbolic execution [8, 14, 16]. These threats could be reduced by more experiments on wider types of subject programs and testing techniques in future work. The threats to internal validity are instrumentation effects that can bias our results. Faults in our own prototype, its underlying adapted Jusc coverage
measurer [35], the underlying adapted Indus Java slicer [26], and the underlying adapted EvoUnit [30, 31] might cause such effects. To reduce these threats, we manually inspected the intermediate results of each component for selected program subjects.

10. RELATED WORK

Our approach adapts evolutionary testing and program slicing as components of the test data generation strategy for aspect-oriented programs. Evolutionary testing is an example of search-based software engineering [11]. The most common applications of evolutionary testing have been to structural test data generation [3, 6, 17, 20, 21, 23, 25, 31, 33, 34]. However, these applications focused largely on procedural programs such as those written in C, with a few of them on structural testing of object-oriented programs and none at all on structural testing of aspect-oriented programs. Our new approach is, therefore, the first to apply evolutionary search to the structural testing of aspect-oriented programs.

Our new approach uses program slicing [32] as an underlying technique in order to conduct dependence analysis for domain reduction. Binkley and Harman [5] studied the application of domain reduction on procedural programs and, more recently, Harman et al. [12] showed that it could be useful in reducing test effort for procedural programs. Our new approach further makes a significant new contribution to the knowledge body of applying domain reduction on testing aspect-oriented programs and achieved quite positive results.

The Aspectra framework developed by Xie and Zhao [35] is the most related work to our approach. Aspectra uses a wrapper synthesis mechanism to produce wrapper classes for aspects and base classes. Then Aspectra leverages an existing object-oriented test generation tool Parasoft Jtest [24], which generates default values for primitive-type arguments and generates random method sequences. However, Aspectra provides no special support for generating test data for aspect-oriented programs. Furthermore, the underlying Parasoft Jtest tool was neither especially designed nor optimized for testing aspect-oriented programs and its test generation mechanisms are quite limited. Our results show that with the application of more advanced techniques such as evolutionary testing, we achieve better results (higher aspectual branch coverage and reduced testing efforts). Our approach makes new significant contributions to Aspectra by adapting and studying evolutionary testing as well as domain reduction in testing aspect-oriented programs. In addition, our approach enhances Aspectra in that our approach provides a more powerful underlying test data generation engine to exploit Aspectra’s wrapper mechanism.

Xu et al. [37,38] presented specification-based testing approaches for aspect-oriented programs. The approaches create aspectual state models and then includes two techniques for generating test data from the model. Their approach requires models to be specified whereas our approach does not. In addition, their approach does not provide automation, implementation, or empirical results whereas our approach does.

Zhao [40] proposed a data-flow-based unit testing approach for aspect-oriented programs. For each aspect or class, the approach performs three levels of testing: intra-module, inter-module, and intra-aspect or intra-class testing. His approach focuses on data-flow coverage criteria and test selection without providing any support to automated test data generation, which is focused by our approach. In addition, his work does not provide any automation, implementation, or empirical results.

Some other related work on the general area of testing aspect-oriented programs [1, 2], which could potentially be used to help assess the quality of the test data generated by our approach in addition to the aspectual branch coverage being used currently. Test selection for result inspection [36] can be applied on the test data generated by our approach when specifications are not available for asserting program behavior. Test selection for regression testing [39] can be also applied on the test data generated by our approach in the regression testing context. Our approach complements these other approaches on testing aspect-oriented programs.

11. CONCLUSION

Aspect-Oriented Programming has recently received much interest and a growing amount of code (called aspect-oriented programs) is written in aspect-oriented programming languages such as AspectJ. However, there is a lack of algorithms and empirical results for testing these aspect-oriented programs, and particularly there exist few approaches to automated test data generation (ATDG) for aspect-oriented programs. It is questionable that programmers can gain high confidence on the correctness of their aspect-oriented programs if these programs do not get tested sufficiently. To offer ATDG for aspect-oriented programs and deepen our understanding of research questions in this area, we have developed a new approach and its supporting system to provide empirical evidence to validate its applicability. The new approach not only applies evolutionary testing, a common testing approach in testing procedural programs, on testing aspect-oriented programs but also applies domain reduction to reduce the test effort. With the application of the developed system on 14 aspect-oriented programs collected from the literature and web, we conducted intensive experiments to investigate six important research questions in ATDG of aspect-oriented programs. In particular, our studies in ATDG of aspect-oriented programs show that evolutionary testing provides better testing effectiveness than random testing, domain reduction can significantly reduce the computational expense of ATDG, and focusing on testing aspectual behavior can achieve significant effort reduction than focusing on testing all behavior in the whole program. Our research not only provides valuable insights for carrying out future research in this area but also offers a effective solution in ATDG of aspect-oriented programs.

12. REFERENCES


