Automated Test Data Generation for Aspect-Oriented Programs

Mark Harman[1] Fayezin Islam[2] Tao Xie[3] Stefan Wappler[4]
[1]Department of Computer Science, King's College London, Strand, London, WC2R 2LS, UK
[2]T-Zero Processing Services LLC, New York, NY 10011, USA
[3]Department of Computer Science, North Carolina State University, Raleigh, NC 27695-8206, USA
[4]Berner & Mattner Systemtechnik GmbH, Gutenbergstr. 15, D-10587 Berlin, Germany
mark.harman@kcl.ac.uk, fayezin.islam@tzero.com
xie@csc.ncsu.edu, stefan.wappler@berner-mattner.com

ABSTRACT

Despite the upsurge of interest in the Aspect-Oriented Programming (AOP) paradigm, there remain few results on test data generation techniques for AOP. Furthermore, there is no work on search-based optimization for test data generation, an approach that has been shown to be successful in other programming paradigms.

In this paper, we introduce a search-based optimization approach to automated test data generation for structural coverage of AOP systems. We present the results of an empirical study that demonstrates the effectiveness of the approach. We also introduce a domain reduction approach for AOP testing and show that this approach not only reduces test effort, but also increases test effectiveness. This finding is significant, because similar studies for non-AOP programming paradigms show no such improvement in effectiveness, merely a reduction in effort. We also present the results of an empirical study of the reduction in test effort achieved by focusing specifically on branches inside aspects.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Verification

Keywords

Test data generation, aspect-oriented software development, evolutionary testing, search-based software engineering

1. INTRODUCTION

Testing remains an important activity within the overall software development process. In 2002, the US National Institute for Standards in Technology (NIST) estimated the cost of software failures to the US economy at $60,000,000,000, being 0.6% of the entire GDP of the USA [30]. The same report found that more than one third of these costs attributed to software failures could be eliminated by improved testing. Testing remains one of the commonly used practices in assuring high quality of aspect-oriented systems developed through the aspect-oriented software development and Aspect-Oriented Programming (AOP) paradigm [23, 27, 36].

One widely adopted approach to testing concentrates upon the goal of coverage; the tester seeks to cover some set of programming features. One of the most widely used forms of coverage is branch coverage, a testing goal that forms the basis of several industry standards [10, 33]. These standards are applied to all software delivered, regardless of the programming paradigm adopted and so these standards simply cannot be ignored by practicing software engineers.

Manual test data generation for achieving coverage is tedious, error-prone and expensive. Therefore, for some time, automation has been considered to be the key to effective and efficient test data generation [24]. Because of the widespread practitioner usage of branch coverage, this testing goal has received a lot of attention from the software testing research community [14, 28]. Search-based optimization techniques have been shown to be very effective at automating the test data generation process for branch coverage [19,28,38,41]. However, search-based optimization techniques have not, hitherto, been applied in the AOP paradigm.

In this paper, we address this important gap in the existing literature. We introduce techniques for search-based test data generation [20, 28, 37] (e.g., evolutionary testing) and study their effect, in a detailed empirical study of 14 AOP programs. The results are very encouraging. They indicate that the search-based approach is at least as well suited to the AOP paradigm as it is to more conventional paradigms. Indeed, the results provide evidence to suggest that it may be even more effective and efficient.

Specifically, the new language constructs for AOP create both new challenges and optimization opportunities for software testing in general, and for automated software test data generation in particular. However, despite the recent upsurge in interest in AOP, there lacks sufficient work on testing of AOP, especially automated testing of AOP.

Xie et al. [43, 44] recently reduced the problem of automated test data generation for AOP to the problem of automated test data generation for object-oriented (OO) programs. They proposed a wrapper mechanism to address issues in leveraging an OO test data generation tool. However, they did not investigate specific test data generation techniques for AOP, but simply relied on reusing an OO test generation tool that adopts simplistic random test generation techniques.

Recently, there has been much interest in more advanced automated test data generation techniques for procedural or object-

AOSD'09, March 2–6, 2009, Charlottesville, Virginia, USA.

oriented programs, such as dynamic symbolic execution [14] and search-based test data generation [20, 28, 37], but none of this work has been applied to the AOP paradigm.

This lack of sufficient previous work leaves unanswered research questions concerning how well these techniques can be applied to AOP and what differences can be observed when automating test data generation for AOP compared with other more conventional programming paradigms. In this paper, we address these questions, providing the first automated, optimizing approach to test data generation for AOP. The approach exploits AOP-adapted versions of both search-based test data generation techniques and recent results on domain reduction optimizations [17].

The paper makes the following main contributions:

- The paper presents a system for Automated Test Data Generation (ATDG) for AOP. It is the first ATDG system for AOP (going beyond Xie et al.'s approach [43, 44] of simply leveraging an existing random OO ATDG system) and represents the first application of search-based testing techniques (e.g., evolutionary testing techniques) to the AOP paradigm.
- The paper presents the results of an empirical study that provides evidence to support the claim that search-based testing for AOP is effective.
- The paper introduces AOP domain reduction techniques to improve the performance of ATDG. The techniques use a dependence analysis based on slicing [42] to identify irrelevant parts of the test input that cannot affect the outcome of branch evaluation in aspects, presenting results on the effectiveness of these techniques. Specifically it presents the following main findings:
  - Domain reduction was applicable to many of the AOP benchmark programs being studied.
  - Test effort decreased when domains were reduced.
  - The number of covered branches was increased by domain reduction. This interesting finding was a pleasant surprise; no such increase in effectiveness was found for the imperative programming paradigm [17]. A further surprise was found in an effect that we call 'co-lateral coverage': as might be expected, domain reduction improves coverage of target branches, but, more interestingly, it also improves coverage of un-targeted branches. This paper is the first to report on this observed co-lateral coverage effect.
- The paper studies the efficiency gains obtained by focusing test effort on aspectual branches (branches inside aspects), rather than all branches. The results provide evidence that test effort can be reduced while achieving equivalent or better aspectual branch coverage.

Indeed, our approach can be generally applied to test object-oriented programs [8] beyond aspect-oriented programs, by focusing on selective elements (e.g., not-covered branches) of code in object-oriented programs. However, our approach is the first to be applied to test aspect-oriented programs, where the units under tests (aspects) cannot be directly invoked, posing stronger needs of our proposed approach than traditional problems of testing object-oriented programs. In addition, focusing on aspectual branches in a software system (which may include only a low percentage of branches being aspectual branches) offers unique optimization opportunities, as exploited by our approach.

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

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) {
   Customer customer = account.getCustomer();
*  if (customer == null) return;
   if (account.getAvailableBalance()< withdrawalAmount){
     float transferAmountNeeded = withdrawalAmount -
                          account.getAvailableBalance();
     ...
   } else System.out.println("I have enough money!");
  }
  ...
}
```

Figure 1: Sample aspect for the `Account` class

The rest of the paper is organized as follows. Section 2 uses examples to illustrate our approach to ATDG for aspect-oriented programs. Sections 3 and 4 introduce the approach and how it was implemented in order to produce the results reported in this paper. Section 5 describes the experimental setup: the subject programs being studied as well as the research questions posed by the research and addressed by the empirical studies. Sections 6, 7, and 8 present the results of the three empirical studies. These three studies are, respectively, concerned with (1) validating that the search-based testing approach is superior to pure random test data generation (a 'sanity check' to validate applicability), (2) showing that domain reduction can improve search-based test data generation for AOP, and (3) reporting on the differential effort required to test aspect-oriented features over-and-above non-aspect-oriented features.

The comparison with random test data generation is typically regarded as a sanity check for search-based test data generation. However, in the case of AOP test data generation, the only prior automated test data generation approach [43, 44] is random test data generation. Therefore, these results also concern the degree of improvement over the current state-of-the-art, which the paper shows to be statistically significant. Section 9 discusses threats to validity of the presented empirical results. Section 10 presents related work and Section 11 concludes.

2. EXAMPLE

We next introduce the background of AspectJ [22] programs and the overview of our approach through an example adapted from AspectJ examples by Laddad [26]. Figure 1 shows a sample aspect `ODRuleAspect` (for defining overdraft rules) for the `Account` class. In an AspectJ program, *pointcuts* specify where the crosscutting behavior applies within the program. For example, `ODRuleAspect` defines one pointcut `debitExecution`, which specifies the execution of `Account.debit*(float)`. The specific execution points that match a pointcut are known as *joinpoints*.

For example, the execution of `Account.debit(float)` is such a joinpoint for the pointcut. An *advice* contains a set of instructions

that specifies when it is invoked and what behavior it implements. There are three types of advices in AspectJ: before, around, and after. A *before* advice is executed before the joinpoint execution, an *around* advice is executed in place of the joinpoint execution, and an *after* advice is executed after the joinpoint execution. For example, in `ODRuleAspect`, one before advice is defined. The advice specifies that the behavior implemented in the before advice is to be executed before the execution of the `debit` method in the `Account` class.

After we use an AspectJ compiler to weave AspectJ code (defined in `ODRuleAspect`) with the Java code (defined in `Account`) to Java bytecode, our approach identifies all aspectual branches (i.e., branches within the advice in `ODRuleAspect` at the source code level) in the Java bytecode. Let us assume that our coverage target is the false branch of the predicate "`if (customer == null) return`" (highlighted with "*" in Figure 1) in `ODRuleAspect`.

Note that to cover this aspectual branch, we can treat the affected method `debit(float)` of `Account` as the method under test named as the target method, which eventually invokes the `before` advice defined in `ODRuleAspect` before the execution of the method `debit(float)`. Given the target method, 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 an approach of search-based test data generation [20, 28, 37] (e.g., evolutionary testing) based on the theory of evolution (more details are described in Section 3.2).

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 data 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 [42] (more details are described in Section 3.1). The slice criterion is the predicate that is involved with the target aspectual branch. Recall that our target aspectual branch is the false branch of the predicate "`if (customer == null) return`" (highlighted with "*" in Figure 1) in `ODRuleAspect`. 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:

```
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 testers not to try various data for these irrelevant parameters. By excluding the irrelevant parameters, we essentially reduce the search space for testing the target branch.

Note that our proposed approach has been applied to aspects involving static advice as shown in the experiments described in Section 5. But our approach is not limited to static advice and would be expected to work well with dynamic advice since our approach does not rely on specific characteristics of static advice. We plan to empirically validate our such hypothesis in future work.

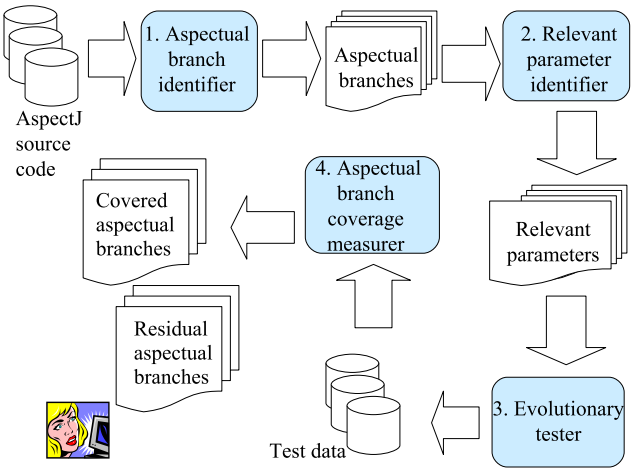

Figure 2: Overview of our approach

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 aspectual branch coverage [43]). The input to the framework includes aspects written in AspectJ as well as the Java classes (being named as *base classes*) where the aspects are woven. Following Aspectra [43], 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 tests 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*, which measures the branch coverage within aspect code.

An overview of our approach is presented in Figure 2. It consists of four major components:

Aspectual-branch identifier. 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. Aspectual branches include both branches from predicates in 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 [43]. 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 [17, 18] 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 (named as 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 (named as a target method) in a base class that invokes the aspect containing the target aspectual branch.

In existing approaches such as Aspectra [43], 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 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 test effort. However, it should be noted that even when all parameters are irrelevant, it does not mean no test effort, since there may be other factors such as state variables and infeasible paths that effect the attempts to cover branches.

Program slicing. To identify such irrelevant parameters, we use program slicing [42]. Program slicing is a static analysis technique that helps to create a reduced version of a program by placing 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 named as 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 statements 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 (ET) [28] is a search-based software testing approach based on the theory of evolution. It formulates the task to generate relevant test data (relevant in terms of the testing objective at hand, such as maximizing structural coverage) as one or several search problems. Each search problem consists of the definition of the search space based on the input domain of the target program (e.g., its relevant parameters), and a fitness function that ET constructs. In the case of structural testing, such a search problem aims at finding a test data leading to the coverage of a particular branch. Each search problem is tried to be solved using an evolutionary algorithm: a pool of candidate test data, the so-called individuals, is iteratively manipulated by applying fitness evaluation, selection, mutation, and crossover in order to eventually obtain a relevant test data. Such an iteration is named as a generation. For fitness evaluation, the candidate test data is executed. Better fitness values are assigned to individuals that are better able to solve the search problem at hand, e.g., coming closer to covering the target branch during execution. ET has been found to achieve better performance than random testing as it concentrates the search toward finding test data with high fitness values [41].

Structural testing. For structural testing, such as branch testing, the fitness value is usually determined based on how close the candidate test data comes to cover the target branch during execution. The closeness is typically expressed as the sum of the two metrics *approximation level* and *local distance*. Approximation level is defined in terms of the control flow graph of the target program and criticality of branching nodes. A branching node is critical if no path exists from one of its outgoing branches to the target. Approximation level is measured as the number of critical branching nodes between the path taken during execution and the target branch. Local distance is defined in terms of the condition of that critical branching node at which execution diverged away from the target branch. Its values are within $[0, 1]$. To determine the local distance, a condition-specific distance function is constructed by composing operator-specific atomic distance functions. For each relational operator of the programming language, an atomic distance function exists [6]. The lower its value, the closer the condition is to being evaluated in favor of the non-critical branch. Since both metrics are non-negative and 0 in the case of an optimal test data, the evolutionary searches aim at minimizing the fitness values of the candidate test data.

Class testing. In class testing (i.e., testing a class), evolutionary testing [40] 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 not-covered structural entity, 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 sequences. 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

Program	Whole program LoC	Test driver LoC	# Aspectual branches	# Aspectual branches from predicates	Brief description
Figure	147	147	1	0	Drawing program handling GUI updates using aspects
PushCount	119	119	1	0	Stack program counting the number of push operations using aspects
Instrumentation	96	96	2	0	Stack program counting number of stack operations using aspects
Hello	33	33	3	0	Introductory AspectJ program demonstrating use of advices in aspects
QuickSort	127	127	4	0	QuickSort program counting partitions and swaps using aspects
NonNegative	94	94	6	4	Stack program enforcing pushing only nonnegative values using aspects
NullCheck	3115	140	6	4	Stack program detecting null values during pop operations using aspects
NullChecker	70	70	7	6	Stack program finding null values in stack cells using aspects
Telecom	696	696	8	0	Telephone call management program maintaining billing using aspects
SavingsAccount	215	215	15	12	Bank account program enforcing policies using aspects
QueueState	545	545	15	12	Queue program maintaining various states of queues using aspects
ProdLine	1317	1317	25	8	Product lines with features enforced by aspects
DCM	3406	446	103	82	Program computing dynamic coupling metrics using aspects
Law of Demeter	3063	185	356	306	Policy enforcer program for the Law of Demeter using aspects

Figure 3: Subject programs being studied

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 named as EvolutionaryAspectTester (EAT). We next describe the implementation details of each tool component.

Aspectual-branch identifier. To identify aspectual branches and measure the coverage of aspectual branches, we modified the aspectual branch coverage measurement tool named as Jusc from the Aspectra approach [43]. In particular, based on Jusc, we identify branches from aspects by scanning and matching method names in the bytecode (produced by an AspectJ compiler) against predefined patterns related to aspects.

Relevant-parameter identifier. We used the Indus Java slicer [34] 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 [39] code, the format of slices generated by Indus. After slicing, line numbers of the slices are extracted from the Jimple output. Corresponding statements 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 [40] from Daimler Chrysler to implement the evolutionary testing technique as well as the random testing technique, which is used as the 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 where each test case covers at least one new target branch.

Aspectual-branch-coverage measurer. We used Jusc [43] 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 as 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 [22]. 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 from predicates in 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 [43]), the number of aspectual branches from predicates in aspects, and a brief description, respectively. Note that only the aspectual branches from predicates are used to conduct domain reduction in the empirical studies. These subject programs were previously used in the literature in testing and analyzing aspect-oriented programs [12, 35, 43].

Most of these programs were used by Xie and Zhao in evaluating Aspectra [43]. These programs include most of the programs used by Rinard et al. [35] to evaluate their classification system for aspect-oriented programs. The programs also include most of the programs[1] used by Dufour et al. [12] in measuring performance behavior of AspectJ programs. These programs also include one aspect-oriented design pattern implementation[2] by Hannemann and Kiczales [16].

Although the programs are relatively small, they represent a range of different popular 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 problem. The involved search space is large in many cases, leading to nontrivial search problems for search-based test data generation.

In the empirical studies, we compare evolutionary testing with random testing for testing aspect-oriented programs with two considerations. First, random testing is the test data generation technique used by existing test data generation approaches [43, 44] for aspect-oriented programs. Second, random testing is a popularly used testing approach [15] in practice.

[1]The AspectJ programs used by Dufour et al. [12] can be obtained from `http://www.sable.mcgill.ca/benchmarks/`.

[2]Hannemann and Kiczales's design pattern implementations can be obtained from `http://www.cs.ubc.ca/~jan/AODPs/`.

To overcome the inherent random effects present in both the evolutionary algorithm and the random search, the empirical results are averages of 30 trials, each of which applies the random and evolutionary testing techniques. On coverage improvement, we compare the average coverage achieved by the 30 trials for each optimization problem.

The effort calculations for the testing process are produced based on the used testing technique; these effort statistics are used in the experiments. For evolutionary testing, effort is calculated in terms of the number of evaluations. For random testing, it is calculated using the number of generations for random testing. As the effort for evolutionary and random testing is calculated using the same way, the results are directly comparable.

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

Assessment of evolutionary testing

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

Impact of domain reduction

- RQ 2.1. What is the number of branches in each program that have irrelevant parameters and how high percentage of 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?

Impact 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 on?

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. This metric was used in existing test data generation techniques [43] for aspect-oriented programs; it is stronger (arguably better) than aspectual statement coverage, whose counterpart in traditional program testing is statement coverage, one of the most popularly used code coverage metrics in practice.

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 data that covers the target branch. This measurement avoids implementation bias and the difficulties associated with reliable, robust, and replicatable assessment of computation effort; the evaluation of fitness is the core atomic unit of computational effort for search-based approaches.

In theory, if running forever, then random will eventually achieve coverage. In practice, testing researchers tend to give an upper bound. We set this bound at 10,000 runs. Effort is only measured for successful coverage. Fitness is evaluated in several places in the evolutionary algorithm. On each call, a static counter is increased to reflect the exact number of times the algorithm assesses fitness.

For the random test data generator, the 'algorithm' simply generates random test sequences and evaluates their fitness until either the target branch is covered (purely at random) or the 10,000 limit is reached without the target branch having been covered. In the case of random search, the number of fitness evaluations is therefore identical to the number of random test data generated.

To cater for the inherently stochastic nature of the search-based algorithms, each algorithm was executed 30 times on each subject program, 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.

The analysis cost of slicing and domain reduction is dwarfed by the cost of evolutionary testing; even in the best case, it takes two orders of magnitude more time to conduct evolutionary testing than it does to compute the reduced domain information. Therefore, the analysis time of slicing and domain reduction can be considered to be inconsequential in this study; the focus of the experiment is thus the difference in performance of the two test data generation problems (with and without domain reduction).

6. EMPIRICAL STUDY: ASSESSING EVOLUTIONARY TESTING

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

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 aspectual branch coverage[3] 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 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 a reduction in effort for evolutionary testing. In summary, evolutionary testing does not only achieve better branch coverage than random testing, it also does it with less effort. This study provides evidence that evolutionary testing is a better technique for testing aspect-oriented programs in comparison to random testing.

[3]Aspectual branch coverage is measured as the percentage of covered aspectual branches among all the aspectual branches.

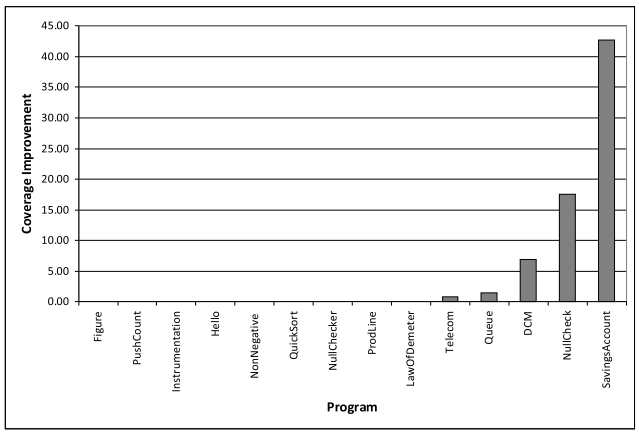

Figure 4: Coverage improvement of evolutionary testing over random testing

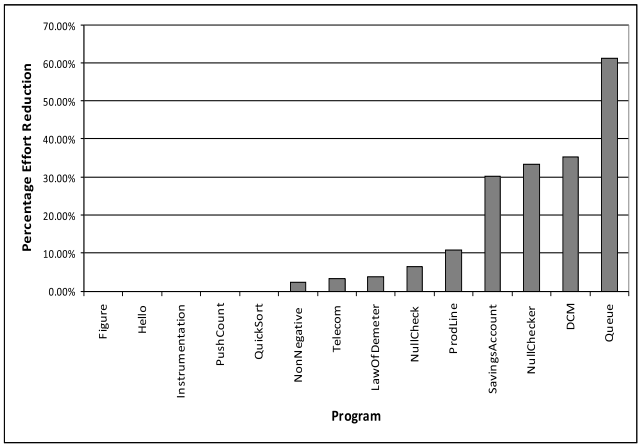

Figure 5: Effort reduction of evolutionary testing over random testing

7. EMPIRICAL STUDY: IMPACT OF DOMAIN REDUCTION

We applied evolutionary testing without and with domain reduction on the 14 programs to investigate research questions RQs 2.1, 2.2, 2.3, and 2.4.

Research Question RQ 2.1. What is the number of branches in each program that have irrelevant parameters and how high percentage of parameters are irrelevant for each of these branches?.

Figure 6 shows the results of applying domain reduction to 14 subject programs under study (the first 5 subject programs and the `Telecom` program have 0 values in Columns 2-5, not being shown in the figure for simplicity). Column 2 shows the total number of aspectual branches from predicates in each program. Column 3 shows the number of aspectual branches with irrelevant parameters among the branches listed in Column 2. Column 4 shows the number of aspectual branches with irrelevant parameters that were possible to be covered with the implemented testing tool. Column 5 shows the number of aspectual branches that were always covered during all 30 runs in testing. In other words, the aspectual branches counted in Column 5 do not include any aspectual branch that was not covered in some run during testing, even if they usually were covered.

We observed that for 6 of the programs there were no aspectual branches from predicates so that these programs are not qualified for domain reduction because we conduct domain reduction on only aspectual branches from predicates, although all aspectual branches have been used to guide the search. Indeed, we can similarly consider the entry of a method in aspects as the slicing criterion for conducting domain reduction, but in this empirical study, we focus on those branches from predicates in aspects.

Program	# Aspectual branches from predicates	# Aspectual branches with irrelevant parameters	# Aspectual branches with testable irrelevant parameters	# Testable aspectual branches covered in all 30 runs
NonNegative	4	0	0	0
NullCheck	4	4	4	3
NullChecker	6	0	0	0
SavingsAccount	12	2	2	2
Queue	12	12	12	12
ProdLine	8	2	2	0
DCM	82	46	42	30
Law of Demeter	306	24	4	1
Total	434	90	66	48

Figure 6: Domain reduction to remove irrelevant branches

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 (i.e., coverable) using the evolutionary tester. Untestable branches include those that make reference to interfaces, abstract classes and those that contain static global variables of primitive type. Of these 66 testable branches, it was possible to generate test data reliably (on all of the 30 runs of the test data generation system) for covering 48 branches.

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 the 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 (i.e., the set of relevant parameters is empty). 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. In summary, when the set of relevant parameters is empty, evolutionary testing can be still applied for searching relevant public fields.

Research Question RQ 2.2. What is the computational effort reduction for each branch that has irrelevant parameters and for how many of these branches is the improvement statistically significant?.

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

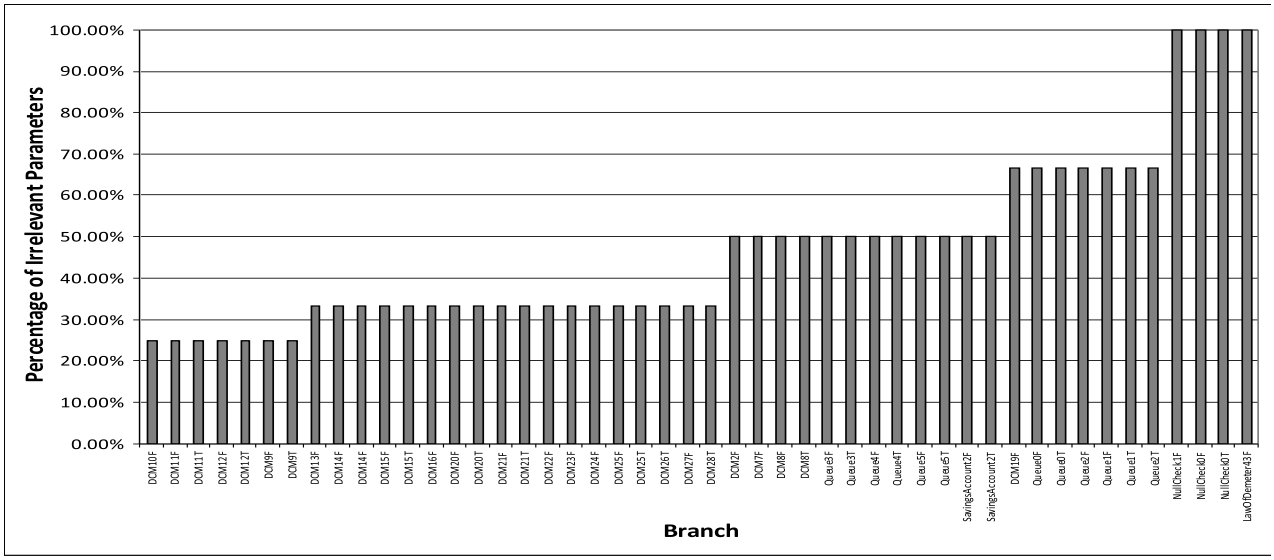

Figure 7: Input domain reduction for branches where some non-zero reduction was possible

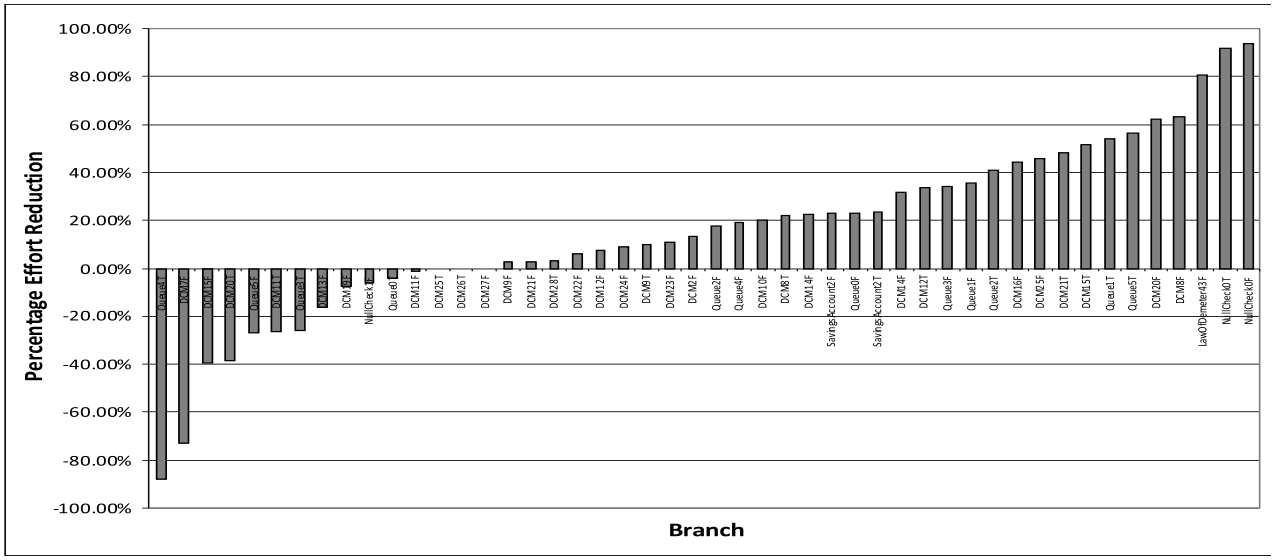

Figure 8: Effort reduction per branch of using domain reduction.

and the y axis shows the percentage reduction in effort after input domain reduction.

Branches covered by 15 evaluations or fewer have been considered to be trivial branches. 17 out of 33 branches fall into this 'trivial' category out of which, one is statistically significant. The majority of the branches (9 out of 11) with statistically significant changes in effort provides evidence that effort is reduced when domain reduction is applied.

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 a change in effort (before and after input domain reduction) is statistically significant. The results of the *t*-test 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.

Research Question 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?.

Figure 9 shows the effect of domain reduction (shown on the y axis) on each program (shown on the x axis) that contains branches with irrelevant parameters. Note that for some (comparatively triv-

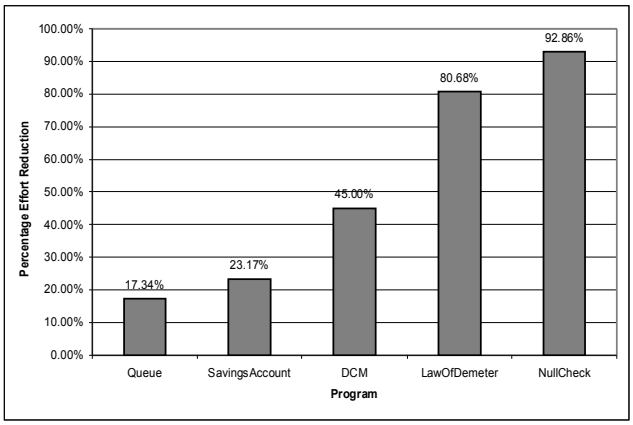

Figure 9: Effort reduction per program of using domain reduction.

ial) 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 parameters that were testable (i.e., coverable), 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 (three of which are testable, being considered in domain reduction), and yet enjoys a 92.86% reduction in test effort through domain reduction.

Research Question 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?.

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 test data generation [7, 25, 29, 32, 40, 41], it is common 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 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 often a 'positive effect' of domain reduction on other branch objectives; co-lateral coverage is more likely to increase with domain reduction. That is, a t-test revealed that the change in co-lateral coverage on 9 branches was statistically significant at the 95% level. Of these 9, only 2 branches had a reduction in coverage, whereas 7 branches had an increase in coverage. As Figure 10 shows, most branches experience an increase in co-lateral coverage, though this effect is not universal. More research is required in order to explore this co-lateral coverage effect in more detail.

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

Research Question 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 on?.

We compared the results of (1) our evolutionary testing approach (without domain reduction) that focuses on aspectual branches only and (2) the same approach that focuses on 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%. However, aspectual coverage improvement in the 4 programs was statistically significant at the 95% level.

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.

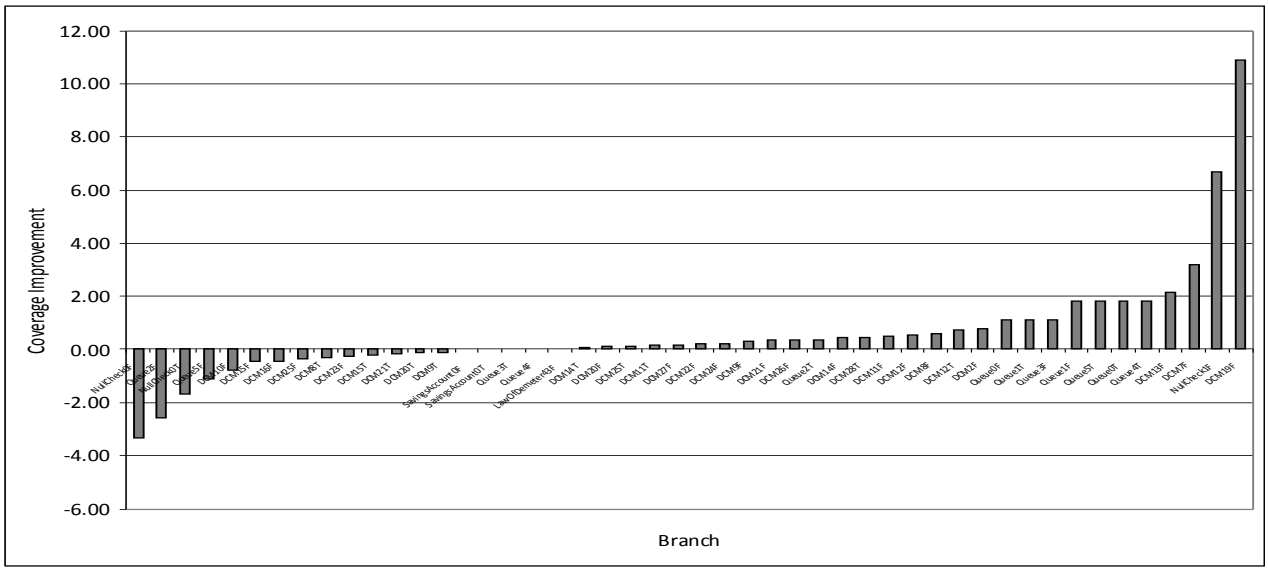

Figure 10: Co-lateral coverage improvement effect of domain reduction.

We performed a *t*-test independently on all 65 classes under test[4] (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 show that there are 47 classes where the reduction in effort is statistically significant. There were 13 classes 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. We collected AspectJ benchmarks from the web and reused benchmarks used in the literature in testing and analyzing aspect-oriented programs. Despite being of small size, the subject programs under study do represent types of aspects commonly used in practice. 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 [11, 21, 24]. These threats to external validity 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 [43], the underlying adapted Indus Java slicer [34], and the underlying adapted EvoUnit [40] might cause such effects. To reduce these threats, we manually inspected the intermediate results of each component for selected program subjects to confirm that these results are expected.

[4]We move down the abstraction chain to focus on an analysis 'per class' rather than 'per program' since the latter is too coarse-grained to provide an answer to the investigated question.

10. RELATED WORK

To support adequate testing of aspect-oriented programs, several approaches have been proposed recently, including fault models and coverage criteria [3, 5, 49], test selection [45, 48], model-based test data generation [46, 47], and white-box test data generation [43].

There exist neither previous approaches to optimization of test data generation nor empirical results on advanced test data generation (beyond random testing) for AOP. This current lack of AOP test automation progress and the associated empirical paucity poses barriers to increased uptake and practical application of AOP techniques [9]. This paper is the first to present evolutionary algorithms for AOP test data generation and the first to provide detailed empirical results concerning automated AOP test data generation. The remainder of this section surveys related work on proposals for AOP testing approaches and OO test data generation.

Xu et al. [46,47] presented specification-based testing approaches for aspect-oriented programs. The approaches create aspectual state models and then include two techniques for generating test data from the model. Their approaches require models to be specified whereas our approach does not. In addition, their approaches do not provide automation, implementation, or empirical results whereas our approach does.

Zhao [49] 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, this work does not provide any automation, implementation, or empirical results.

One of the few tools available that is applicable to aspect-oriented program test data generation is Aspectra [43]. It provides a wrapper

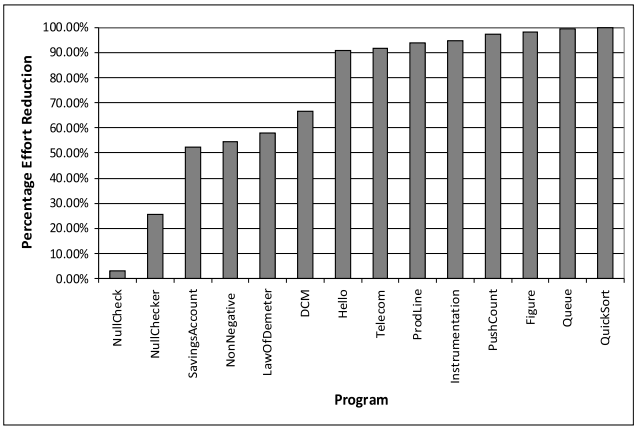

Figure 11: Effort reduction of focusing on aspectual behavior over all behavior

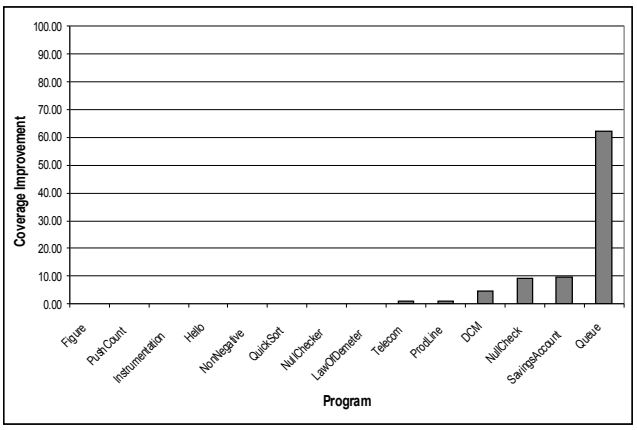

Figure 12: Coverage improvement of focusing on aspectual behavior over all behavior

mechanism to leverage the existing object-oriented test data generation tool Parasoft Jtest [31]. Parasoft Jtest generates default values for primitive-type arguments and generates random method sequences. However, the approach supports only the generation of default or random data. This testing strategy is known to be highly sub-optimal for testing non-aspect oriented programs [7, 25, 29, 32, 41], 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.

Other related work on the general area of testing aspect-oriented programs includes fault models for aspect-oriented programs [3–5, 13], 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 [45] 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 [48] 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

In this paper, we have introduced a novel approach to automated test data generation for AOP. The approach is based on evolutionary testing, which uses search-based optimization to target hard-to-cover branches. The results of empirical studies on several hundred search problems drawn from 14 AOP benchmark programs show that the evolutionary approach is capable of producing significantly better results than the current state of the art.

The results reported here are the first detailed empirical study of AOP testing. We have also adapted recent results on domain reduction, introducing an optimized tool to reduce the test data search space. The results provide evidence that this optimization increases both effectiveness and efficiency, an improvement over previous work [17], which was able to demonstrate efficiency improvement, but not effectiveness improvement. Finally, the paper presents the results of a study into the differential effects of AOP testing compared to traditional testing. The results quantify the improvement in efficiency that can be gained by focusing solely on aspectual behavior.

In future work, we plan to develop other more advanced test data generation techniques such as Dynamic Symbolic Execution [14] for AOP. We plan to evaluate various techniques on larger AOP systems such as AOP code in the AspectOptima framework [2] and the AOP demonstrators released by the AOSD-Europe project [1]. We also plan to conduct more experiments on comparing the fault detection capability of the test suites generated by various techniques, going beyond the structural coverage achieved by the test suites. Towards this aim, we plan to use mutation testing by seeding faults in aspects [13] or in pointcuts [4, 13]. Our current approach focuses on generating test data for achieving high aspectual branch coverage. In future work, we plan to extend the approach to generate test data for achieving other types of coverage in AOP systems such as data flow coverage [49] between aspect and base code, helping expose faults related to interactions of aspect and base code.

Acknowledgments

The anonymous referees provided extremely authoritative and technically detailed comments on an earlier version of the paper, which were very much valued and appreciated. Tao Xie's work is supported in part by NSF grant CCF-0725190. Mark Harman is partly supported by EPSRC grants EP/G04872X, EP/F059442, EP/F010443, and EP/D050863.

12. REFERENCES

[1] The AOSD-Europe project. `http://aosd-europe.net/`.

[2] The AspectOPTIMA aspect-oriented framework. `http://www.cs.mcgill.ca/~joerg/SEL/AspectOPTIMA.html`.

[3] R. T. Alexander, J. M. Bieman, and A. A. Andrews. Towards the systematic testing of aspect-oriented programs. Technical Report CS-4-105, Department of Computer Science, Colorado State University, Fort Collins, Colorado, 2004.

[4] P. Anbalagan and T. Xie. Automated generation of pointcut mutants for testing pointcuts in AspectJ programs. In *Proc. ISSRE*, pages 239–248, 2008.

[5] J. S. Bækken and R. T. Alexander. A candidate fault model for AspectJ pointcuts. In *Proc. ISSRE*, pages 169–178, 2006.

[6] A. Baresel, H. Sthamer, and M. Schmidt. Fitness function design to improve evolutionary structural testing. In *Proc. GECCO*, pages 1329–1336, 2002.

[7] A. Baresel, H. Sthamer, and M. Schmidt. Fitness function design to improve evolutionary structural testing. In *Proc. GECCO*, pages 1329–1336, 2002.

[8] R. V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley, 1999.

[9] L. C. Briand. A critical analysis of empirical research in software testing. In *Proc. ESEM*, pages 1–8, 2007.

[10] British Standards Institute. BS 7925-2 software component testing, 1998.

[11] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Software Eng.*, 2(3):215–222, 1976.

[12] B. Dufour, C. Goard, L. Hendren, O. de Moor, G. Sittampalam, and C. Verbrugge. Measuring the dynamic behaviour of AspectJ programs. In *Proc. OOPSLA*, pages 150–169, 2004.

[13] F. C. Ferrari, J. C. Maldonado, and A. Rashid. Mutation testing for aspect-oriented programs. In *Proc. ICST*, pages 52–61, 2008.

[14] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proc. PLDI*, pages 213–223, 2005.

[15] R. Hamlet. Random testing. In *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

[16] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proc. OOPSLA*, pages 161–173, 2002.

[17] M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, and J. Wegener. The impact of input domain reduction on search-based test data generation. In *Proc. ESEC/FSE*, pages 155–164, 2007.

[18] M. Harman, L. Hu, R. M. Hierons, C. Fox, S. Danicic, A. Baresel, H. Sthamer, and J. Wegener. Evolutionary testing supported by slicing and transformation. In *Proc. ICSM*, page 285, 2002.

[19] M. Harman and P. McMinn. A theoretical and empirical analysis of evolutionary testing and hill climbing for structural test data generation. In *Proc. ISSTA*, pages 73 – 83, 2007.

[20] K. Inkumsah and T. Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *Proc. ASE*, pages 297–306, 2008.

[21] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proc. TACAS*, pages 553–568, April 2003.

[22] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proc. ECOOP*, pages 327–353, 2001.

[23] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. ECOOP*, pages 220–242, 1997.

[24] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[25] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.

[26] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., 2003.

[27] K. Lieberherr, D. Orleans, and J. Ovlinger. Aspect-oriented programming with adaptive methods. *Commun. ACM*, 44(10):39–41, 2001.

[28] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, June 2004.

[29] P. McMinn, M. Harman, D. Binkley, and P. Tonella. The species per path approach to search-based test data generation. In *Proc. ISSTA*, pages 13–24, 2006.

[30] National Institute of Standards and Technology (NIST). The economic impacts of inadequate infrastructure for software testing, May 2002. Planning Report 02-3.

[31] Parasoft. Jtest manuals version 4.5. Online manual, April 2003. `http://www.parasoft.com/`.

[32] R. Pargas, M. Harrold, and R. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification and Reliability*, 9(4):263–282, 1999.

[33] Radio Technical Commission for Aeronautics. RTCA DO178-B Software considerations in airborne systems and equipment certification, 1992.

[34] V. P. Ranganath, T. Amtoft, A. Banerjee, J. Hatcliff, and M. B. Dwyer. A new foundation for control dependence and slicing for modern program structures. *ACM TOPLAS*, 29(5):1–43, 2007.

[35] M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In *Proc. FSE*, pages 147–158, 2004.

[36] P. Tarr, H. Ossher, W. Harrison, and J. Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *Proc. ICSE*, pages 107–119, 1999.

[37] P. Tonella. Evolutionary testing of classes. In *Proc. ISSTA*, pages 119–128, 2004.

[38] N. Tracey, J. Clark, and K. Mander. Automated program flaw finding using simulated annealing. In *Proc. ISSTA*, pages 73–81, 1998.

[39] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *Proc. CASCON*, pages 125–135, 1999.

[40] S. Wappler. *Automatic generation of object-oriented unit tests using genetic programming*. PhD thesis, Technical University of Berlin, 2008.

[41] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, 2001.

[42] M. Weiser. Program slicing. In *Proc. ICSE*, pages 439–449, 1981.

[43] T. Xie and J. Zhao. A framework and tool supports for generating test inputs of AspectJ programs. In *Proc. AOSD*, pages 190–201, 2006.

[44] T. Xie, J. Zhao, D. Marinov, and D. Notkin. Automated test generation for AspectJ program. In *Proc. WTAOP*, 2005.

[45] T. Xie, J. Zhao, D. Marinov, and D. Notkin. Detecting redundant unit tests for AspectJ programs. In *Proc. ISSRE*, pages 179–188, 2006.

[46] D. Xu and W. Xu. State-based incremental testing of aspect-oriented programs. In *Proc. AOSD*, pages 180–189, 2006.

[47] D. Xu, W. Xu, and K. Nygard. A state-based approach to testing aspect-oriented programs. In *Proc. SEKE*, pages 366–371, 2005.

[48] G. Xu and A. Rountev. Regression test selection for AspectJ software. In *Proc. ICSE*, pages 65–74, 2007.

[49] J. Zhao. Data-flow-based unit testing of aspect-oriented programs. In *Proc. COMPSAC*, pages 188–197, 2003.