A Survey on Aspect-Oriented Testing Approaches

Reza Meimandi Parizi
Department of Computer Science
University Putra Malaysia
43400 Serdang, Malaysia
reza.parizi@hotmail.com

Abdul Azim Ghani
Department of Computer Science
University Putra Malaysia
43400 Serdang, Malaysia
azim@fsktm.upm.edu.my

Abstract

Research on testing approaches for different programming paradigms has been done for several years. However, Aspect-Oriented Programming (AOP), a relatively new programming paradigm, has properties that other programming paradigms do not have. These characteristics bring new challenges and issues not present when testing other types of programs such as Object Oriented Programs (OOPs). So this new paradigm cannot provide correctness by itself, and thus it also, like any other programs, requires the use of systematic testing approaches to produce validated and high quality software. This paper surveys the research on testing AOPs, the testing approaches that are being proposed, and issues that arise when testing AOPs. The effectiveness of proposed approaches is compared in terms of their ability to find different kind of faults as described by [6]. By making comparison on AOPs testing approaches, a better understanding of these approaches and issues related in testing AOPs is provided, as well as conclusion about existing state of research in this area and future trends for testing of aspect-oriented is identified.

1. Introduction

Testing is an essential part of the software development process. One of the goals for testing programs is the detection of faults in the software. Testing is done throughout the process of software development. When developing programs, we want to detect faults as early as we can. There exist several types of testing techniques such as unit testing, integration testing, and black-box testing and white-box testing among others. These and other testing techniques have been developed, have been researched, and have been applied to different programming paradigms such as object oriented programming. Even after this research on testing for well known programming paradigms, testing software remains a challenging activity. Aspect-oriented programming is a relatively new programming paradigm that offers many advantages. AOP provides with means to improve modularity and separation of crosscutting concerns. Although Aspect-Oriented Programs (AOPs) have some characteristics that differ from object oriented programs or procedural oriented programs but it brings new challenges and questions due to characteristics of aspects. Some of these characteristics were presented in [6]. These include the following:

- Aspects are dependent on the context of other classes.
- Aspects are tightly coupled to the classes they are weaved in.
- When dealing with aspects, it is not clear the control and data dependencies when you analyze source code for either the aspects or the classes.
- There are many possible sources for a fault.

Due to these properties of aspects, testing AOPs brings interesting questions and issues such as, what are the criteria to test aspect-oriented programs? How different is this criteria from the one used in other programming paradigms? Do current testing techniques be applied to AOPs? Could they be modified so that they could test AOPs? Are there well defined heuristics for testing AOPs? In our opinion, the questions and issues regarding testing AOPs are very important to be addressed when dealing with aspects. Thus, by surveying, writing summaries, making comparisons, and analyzing current research on testing approaches for AOPs, we can get a better idea of the testing approaches and issues related in testing AOPs. This paper is aimed to explore the current research being done on testing AOPs, analyze and compare suggested approaches, and consider the issues that arise in testing AOPs.

The rest of the paper is organized as follows: Section 2 presents the background on software testing and
2. Background

This section provides the reader with information on software testing terminology, testing techniques and information on aspect-oriented programming.

2.1. Background on testing

Software testing has several definitions. As defined in [4], software testing is the process of executing a program or system with the goal of finding errors. Another definition that captures closely what is involved in testing is also given in [4]. Software testing is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results.

Software testing is used to achieve quality assurance, verification, validation, and reliability estimation. Quality assurance refers to ensuring that a program or system performs as required by specifications. Verification evaluates the accuracy of transforming specifications, flowcharts, and other documentation involved in the software development process to a program or system. In other words, verification deals with achieving the right model. Validation evaluates the accuracy of input-output from the model with respect to the input-output from the program or system. In other words, validation deals with building the right model. Reliability estimation refers to how probable is that an operation on the program or system is fault-free [4].

There are many testing techniques and testing methods used with different purposes. In other words, there are different classifications of testing methods or testing techniques, for example, [4] identifies three classes:

- By purpose: correctness testing, performance testing, reliability testing, and security testing.
- By life-cycle phase: requirements phase testing, design phase testing, program phase testing, evaluating test results, installation phase testing, acceptance testing, and maintenance testing.
- By scope: unit testing, component testing, integration testing, and system testing.

Also other classifications of testing methods such as: informal, static, dynamic, symbolic, constraint, and formal have been identified.

2.2. Aspect-oriented programming concepts

Aspect-oriented programming is a relatively new programming paradigm that allows for separation of concerns, especially separation of crosscutting concerns. AOP allows for better modularity, cohesion, understandability, maintainability, and evolvability of a program or system. Object oriented programming brought different concepts, ideas, and challenges with respect to procedural programming languages. Similarly, aspect-oriented programming brings new basic AOP concepts; Section 3 gives the comparison criteria; Section 4 describes the surveyed works on testing AOPs; Section 5 presents analysis and comparison on different approaches proposed by surveyed works; Section 6 reports the conclusion and future work on this area of research.
challenges, ideas, and concepts. Some of these concepts are:

- **Join point**: Is a well-defined point in the execution of a program. Examples of joint points are calling or execution of methods, access to an attribute, and initialization of an object.
- **Pointcut**: Is a set of patterns that are used to select join points.
- **Advice**: Is a method-like construct that contains additional behavior to be added at the matched joint point. The advice is what is woven into the join points when the pattern of a pointcut is matched. In other words, an advice is used to express the cross-cutting actions that must take place within the method body at the matched join point. Advice generally represents a fragment of control and data that must be added to the body of an existing method [6]. There are three kinds of advices: before advice, after advice, and around advice. A brief description of each type of advices are as follows:
  - **Before** advice: A before advice executes its body before executing the body of the matched join point.
  - **After** advice: An after advice executes its body after executing the body of the matched join point.
  - **Around** advice: An around advice body surrounds the match join point. Around advice may change the execution of the matched join point body, or may even replace the matched join point body. In other words, around advice can bypass the matched join point elements altogether, or make the matched join point's execution control dependent upon the around advice [6].
- **Aspect**: An aspect is a construct that encapsulates a cross-cutting concern. Aspects are similar to class in object oriented programming. Besides having the properties of a class in OOP, aspects encapsulate the behavior, and state of cross-cutting concerns. Aspects include pointcuts, advices, and intertw type declarations which are used to add a public or private method, field, or interface implementation declaration into a class [8].
- **Aspect weaving**: Is the process by which behavior on aspects are merged to the original code. In other words, weaving is the process of merging the cross-cutting concerns to the core concerns of a program or system. For example, in AspectJ aspect weaving composes the code of the base classes and the aspects to ensure that applicable advice runs at the appropriate join points [8].

### 3. Comparison criteria

In this study aspect-oriented testing approaches will be compared according to their ability to find the following classes of faults, as comparison criteria, which has been introduced in the fault model by [6].

#### 3.1. Incorrect strength in pointcut patterns

This is perhaps the most obvious category of faults. Pointcuts contain specifications that select join points of a particular type according to a signature that includes a pattern. For a pointcut \( p \), each matching join point \( j \) of a concern \( C \) will have the advice associated with \( p \) woven into \( j \). The strength of the pattern in the signature of \( p \) determines which join points are selected. If the pattern is too strong, some necessary join points will not be selected. If the pattern is too weak, additional join points will be selected that should be ignored. Either case is likely to cause incorrect behavior of the woven target. The statements in the woven advice and the statements that are executed after the woven join point determine whether a pattern strength error will introduce a fault [6].

#### 3.2. Incorrect aspect precedence

The order in which advice from multiple aspects is woven into a concern affects system behavior, especially when there are mutual interactions between aspects through state variables in the core concern. In AspectJ, weave order is determined by the specification of aspect precedence. For example, the aspect with the highest precedence executes its before advice on a join point prior to executing the before advice of the lower precedence aspect. If precedence is not specified, the order in which advice is applied remains undefined. Precedence is of no concern as long as the effects of woven advice are mutually independent.

#### 3.3. Failure to establish expected postconditions

Aspects can cause changes in the flow of control of class’s code. Such a change in flow of control can result in a class (core concern) not being able to fulfill the postconditions of its class contract. The clients of such core concerns expect those concerns to behave according to their contracts. Clients expect method post conditions to be satisfied regardless of whether or not aspects are woven into the concern. Hence the behavioral contracts of the concern should hold after the weaving process. Thus, for correct behavior, woven
advice must allow methods in core concerns to satisfy their post conditions [7].

3.4. Failure to preserve state invariants

A concern’s behavior is defined in terms of a physical representation of its state, and methods that act on that state. In addition to establishing their postconditions, methods must also ensure that state invariants are satisfied, or else it causes violations of state invariants.

3.5. Incorrect focus of control flow

A pointcut designator selects which of a method’s join points to capture. This selection is determined at weave time. However, there are often cases where the information needed to correctly make such a decision is available only at run time. Sometimes join points should only be selected in a particular execution context. This context could be within the control structure of a particular object, or within the control flow that occurs below a point in the execution [6].

3.6. Incorrect changes in control dependencies

The around advice can significantly alter the control flow of any method upon which it is applied [6]. New code is inserted, new branches appear that alter the dependencies among statements, and new data may also be inserted. Under these circumstances it is clear to see that control dependencies among statements have not only been represented in case of AOP, but they also have to be tested.

4. Surveyed works on testing AOPs

Since AOPs is a relatively new program paradigm; only a few approaches for testing of aspect-oriented programs have been proposed.

This section summarizes the surveyed works on testing approaches for aspect-oriented programs.

4.1. Data-flow-based unit testing of aspect-oriented programs

This paper presents an approach to testing AOPs using a data-flow-based approach for unit testing AOPs. The main contributions of this paper are that it was one of the first, if not the first, approach proposed for testing an aspect and that it describes how by combining unit testing and data-flow-based testing techniques aspects can be tested. The approach, which is taken in this paper tests aspects and classes that may be affected by one or more aspects. For each aspect or class, three levels of testing are performed:
- Intra-module: This level of testing is performed for an individual module (e.g. a piece of advice, a piece of introduction, and a method).
- Inter-module: This level of testing is performed for a public module along with other modules it calls in an aspect or class.
- Intra-aspect or intra-class testing: This level of testing is performed for modules that can be accessed outside the aspect or class and can be invoked in any order by users of the aspect or class.

Zhao’s proposed testing technique uses control flow graphs to compute def-use pairs of an aspect or class. The selection of tests for aspects or classes is based on the computation of these def-use pairs. Refer to [5] for the algorithm to build such control-flow graphs, for more information on the three levels of testing, and for computing def-use pairs that are used in this testing technique for AOPs. A main issue with this approach was identified in [6]. Around advices may change the behavior control or data dependences of a method, that is, it may change def-use pairs. This issue was analyzed on later works. Future work for this approach includes not only applying the data-flow testing technique to aspects or classes themselves but also analyze how to use this technique on testing the integration of aspects and classes.

Also, research needs to be conducted on how to test aspect or class inheritance in AOPs. The development of a testing tool based on the data-flow testing technique proposed is also part of future work. In conclusion, the authors of this paper propose a data-flow-based unit testing approach for aspect-oriented programs. This proposed approach handles testing problems unique to AOPs. Their technique uses control-flow graphs to calculate def-use pairs and use this information to select tests for aspects or classes [5]. This testing approach is capable of detecting the failure to preserve state invariants faults [7].

4.2. A state-based approach to testing aspect-oriented programs

This paper presents an approach to testing AOPs using a State-Based approach. The main contribution of this paper is that it shows the possibility to generate test suites for adequately testing object behavior and interaction between classes and aspects in terms of message sequences. The approach taken in this paper is based on FREE (Flattened Regular Expression)-based test design pattern for OOP. FREE is similar to the state model in UML, the authors of this paper also
extend FREE to ASM (Aspectual State Models) to specify classes and aspects. The main process for this approach was to transform an ASM to a transition tree, where each path from the root to a terminal leaf node becomes a test case. Refer to [3] for the algorithm for transforming an ASM to a transition tree. The main issue on this approach is that state-based approaches often suffer from the state explosion problem. This issue is to be analyzed for future work. That is, research needs to be conducted to identify the paths that are of most interest or importance. More future work is needed to investigate how this stated-based testing approach is adapted or can be used with other UML-based modeling methods for AOP. Future work includes also extending this approach to reveal more faults likely to occur in composition of the cross-cutting concerns. In conclusion, the authors of this paper show that a state-based testing approach is adapted or can be used with other UML-based modeling methods for AOP. Future work also includes extending the analysis of the class and transition tree refer to [10].

4.3. Aspect flow graph for testing aspect-oriented programs

This paper present an approach to testing AOPs using a hybrid testing model which is combination of a responsibility-based testing model and an implementation-based testing model. The main contribution of this paper is that it shows the test suits that are generated based on the hybrid testing model are manageable, code based and executable. The approach taken in this paper is based on AFG (Aspect Flow Graph). AFG is constructed by merging two models, the ASSM (Aspect Scope State Model) that is a state transition diagram for an aspect-oriented program and the flow graphs for methods and advices. The flow graphs for methods and advices are graphs representing the control flow within the methods and advices. Based on the AFG and the creation of transition tree, concert and executable code-based test suites can be derived in terms of coverage testing criteria. For more information about constructing AFG and transition tree refer to [10]. The main issue on this approach is that AFG can not handle the representation of dynamically applied advices (those that are applied depending on a particular context of execution). Future work also includes extending the analysis of the class and aspect behaviors to system architecture and abstract level. It has been claimed in [3] that the flow graph based testing approaches for AOPs can detect two types of faults in the candidate fault model [6]. These two classes are: incorrect focus of control flow and incorrect changes in control dependencies. But recently S.A.A. Naqvi, S. Ali, and M.U. Khan have proposed in [7], this testing approach is not be affective in detecting faults of incorrect focus of control flow because on the one hand, the Aspect Flow Graph is a model of static weaving and the other hand, faults of this kind arise due to the dynamic behavior of an AOP that consecutively is not handled by this approach, as mentioned above.

4.4. Unit-testing aspectual behavior

This paper focuses on testing aspectual behavior, i.e. behavior implemented in pieces of advice. The main contribution of this paper is that it shows the possibility and requirements for devising clean unit testing techniques for aspect-oriented programs using mock object mechanisms. The approach taken in this paper is based on the creation of JAML (Java Aspect Markup Language) and the development of JamlUnit (an extension to JUnit). JamlUnit is a framework for performing unit testing of aspects written in JAML. Additionally, JamlUnit uses mock objects to emulate the execution of matched join point. Refer to [1] for more information on JAML and JamlUnit. The main issues on this approach are that criteria to select meaningful mock points have not been developed and that creating mock execution context can become difficult for more complex AOPs. The difficulty is inherent to the nature of aspects, especially for those that interact with core concerns. These issues are to be analyzed for future work. That is, research needs to be conducted on developing criteria to select meaningful mock points. Also research needs to be conducted on the development of tools for writing mock execution points. Future work also includes creating techniques for testing other elements of JAML including: pointcut definitions, aspectual bindings, and introductions. In conclusion, the authors of this paper propose JamlUnit, a framework for performing unit testing of aspects written in JAML; they show that performing unit testing on aspectual behavior is possible if the AOP language meets some requirements as JAML does [1].

4.5. A model-based approach to test generation for aspect-oriented programs

This paper presents an approach to testing AOPs using a model-based approach for testing AOPs. The main contribution of this paper is that it shows the possibility to generate test suites for adequately testing interaction between classes and aspects in terms of message sequences. The approach taken in this technique is based on an extension to UML, that is, on
aspect-oriented UML models. These models consist of three kinds of diagrams: class diagrams, aspect diagrams, and sequence diagrams. The process in this approach is to weave advices on a particular method’s sequence diagram and get a new sequence diagram. Goal-directed reasoning is used on the sequence diagram to build a flow graph for certain coverage criteria. The authors of this paper use a combination of branch coverage and polymorphic coverage. Once the graph is built, it is transformed to a flow tree where each path becomes a test case [9]. For more information on the algorithms to build a sequence diagram, how to build a woven sequence diagram, how to construct a goal-directed flow graph, how to transform a goal-directed flow graph to a flow tree, and how to generate test data for a path in a flow tree refer to [9]. An issue that may arise on this approach is which paths are more important or which paths are of more interest. This issue is to be analyzed for future work. More future work would need to be done to see how this model-based testing approach for AOPs could be used with other extensions of UML that include aspect modeling. In conclusion, the authors of this paper show that a model-based testing approach is possible for AOPs and that is possible to develop testing techniques that address questions unique to AOP. These questions include: how to mock objects and aspects to handle tests? How to test for both, base elements and cross-cutting elements without modifying the original test cases for base elements?, and How to adequately test cross-cutting elements? [9].

5. Analysis and comparison on surveyed works

This section firstly gives a brief analysis on surveyed works and secondly provides a comparison among proposed testing AOPs approaches by these works.

One of the first works on testing AOPs is that Zaho proposed an approach to unit testing aspectual behavior [1]. This approach at the moment of the publication of the paper does not include support for testing some constructs of AOP such as testing the pointcuts. The ability of this approach in terms of detecting the class of faults is not still researched. Another approach for testing AOPs was proposed by W.Xu [10]. This AFG-based approach addressed its ability to reveal two classes of faults within the fault model (although its claim is questioned by [7]), as mentioned in section 4.3. One year after this W.Xu proposed also another approach in [9]. This model-based approach for testing AOPs addressed points also brought up in the fault model as well as issues brought up by Lopes for testing AOPs. W.Xu’s testing approach address issues on how to mock objects and aspects to handle tests, how to test for both, basee elements and cross-cutting elements, and how to adequately test cross-cutting elements?. Similar to the aspectual behavior testing approach, the ability of this approach in terms of detecting the class of faults is also not researched. Another testing approach was proposed by D.Xu in [3] and later refined in [2]. This testing approach for testing AOPs is a state-based testing approach. This testing approach addresses and provides a solution for issues brought up in [6] and finally the use of this testing approach reveals faults such as: incorrect strength of pointcut patterns and failure to preserve state invariants [3].

Based on the progress shown towards better testing approaches for AOPs in the analysis part and comparison criteria, as described in the section 3, the comparison among proposed approaches for testing AOPs in this study can be summarized in the table 1 (the results shown in this table have been analyzed and obtained by [7]), where rows represent the testing AOPs approaches and columns represent the comparison criteria.

The following information is derived from the table 1:

- The faults ‘Incorrect aspect precedence’, ‘Failure to establish expected postconditions’, and ‘Incorrect focus of control flow’ are not being found by any of the testing approaches.
- Except for fault ‘Failure to preserve state invariants’; none of the fault types can is being addressed by more than one testing approach.
- Except state-based approach; none of the testing approaches can reveal more than one type of fault.

With this information in mind, in addition to the information that was presented in the overall paper, the following conclusion can be drawn.

6. Conclusion and future work

In this paper an overall introduction to the problem of testing AOPs is given. Background and basic terminology on software testing and aspect-oriented programming is presented on this paper. Comparison
criteria are also given. A summary of surveyed works on testing approaches, including issues and future work, for AOPs is provided in this paper. Comparison on surveyed works, as well as conclusion and future work are included.

The conclusions that can be drawn from surveyed works and from this paper are the following:

- **Although AOP is relatively new, there is plenty of research on AOP. However, much of this research is being conducted towards problem analysis, software design, implementation techniques [5], and using aspect-oriented methods to build testing tools [6]. On the last three years there has been research on testing AOPs, this research include the following:**
  - Testing AOPs based on specifications.
  - Data-Flow-based approach for Unit testing AOPs.
  - Developing and improving fault models for testing AOPs.
  - Developing testing criteria for testing AOPs.
  - Developing frameworks, such as JamlUnit, for performing unit testing of aspects written in JAML.
  - State-based approach for testing AOPs.
  - Model-based approach to test generation for AOPs.
  - Control and data flow structural testing criteria for AOPs.

- **Current research on testing approaches for AOPs is similar to that of testing techniques for OOP when OOP was a new programming paradigm. Some of the first research on testing OOP can be mapped to research being done on AOP. Moreover, testing approaches such as in [3] are based on OOP testing techniques.**

- **Based on the comparison shown among testing approaches for AOPs and the fact that these testing approaches are improved from the issues brought up on previously testing approaches, at the moment and to the best of our knowledge and understanding of the surveyed testing approaches, the refined and revised state-based testing approach proposed by Xu on [3] and [2] shows to be one of the most complete testing approaches of AOPs at the present time.**

Future works for this study include surveying more testing approaches for AOPs and incorporate new testing approaches that are in the process of research.

### 7. Acknowledgment

The authors would like to acknowledge the suggestions by Carlos Acosta, University of Texas at El Paso, USA.

### 8. References


Table 1. Comparison table of testing AOPs approaches

<table>
<thead>
<tr>
<th></th>
<th>Incorrect strength in pointcut patterns</th>
<th>Incorrect aspect precedence</th>
<th>Failure to establish expected postconditions</th>
<th>Failure to preserve state invariants</th>
<th>Incorrect focus of control flow</th>
<th>Incorrect changes in control dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data flow based approach</strong></td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td><strong>State based approach</strong></td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td><strong>Aspect flow graph based approach</strong></td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Aspectual behavior approach</strong></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Model based approach</strong></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

✓ : Can be found by the approach.
× : Can not be found by the approach.
- : Not yet determined.